Título: Aceleración de Aplicaciones Científicas mediante GPUs.
Alumno: Diego Luis Caballero de Gea.
Directores: Xavier Martorell Bofill (UPC), Juan Fernández Peinador (UM) y Javier Cuenca Muñoz (UM).

Departamento: Arquitectura de Computadores (UPC) y Arquitectura y Tecnología de Computadores (UM).

Fecha: Junio de 2010.

Universidad de Murcia
Universidad Politécnica de Cataluña

Facultad de Informática

Aceleración de Aplicaciones Científicas mediante GPUs

Proyecto Fin de Carrera

Diego Luis Caballero de Gea

Junio de 2010

Dirigido por:

Xavier Martorell Bofill
Departamento de Arquitectura de Computadores.
Universidad Politécnica de Cataluña.

Juan Fernández Peinador
Departamento de Arquitectura y Tecnología de Computadores.
Universidad de Murcia.

Javier Cuenca Muñoz
Departamento de Arquitectura y Tecnología de Computadores.
Universidad de Murcia.

Índice General

Índice de Figuras

Índice de Tablas

A mis padres, Felipe Caballero y María Josefa de Gea.

Agradecimientos

Quería mostrar mis agradecimientos a Xavier Martorell, uno de mis directores de proyecto, por su acogida en la Universidad Politécnica de Cataluña, su ayuda e implicación en este proyecto y por haber puesto a mi disposición todos los recursos que he necesitado para el desarrollo del mismo, además de haber realizado la tarea extra oficial de tutor de estudios.

A Juan Fernández Peinador y Javier Cuenca, también directores de proyecto, por abrirme las puertas del mundo de la programación paralela y la investigación, soportarme dos años como alumno interno y por haber hecho posible este proyecto final de carrera. He aprendido mucho con vosotros.

A Daniel Jiménez, profesor de la UPC, por su colaboración en este proyecto con su experiencia sobre una de las aplicaciones.

A mi familia, muy especialmente a mis padres, Felipe Caballero y María Josefa de Gea, a mi abuela Isabel y a Antonio y Manolita, por sacrificarse tantísimo porque este proyecto de vida salga adelante, por confiar en mí incondicionalmente y por su incesante apoyo. Sin vosotros esto no hubiera sido posible.

A mis amigos, porque el esfuerzo ha sido grande para ellos también, y los buenos de verdad han sido estando ahí, aún en la distancia.

A mis profesores de la Universidad de Murcia y de la Universidad Politécnica de Cataluña, por vuestro alto nivel de exigencia, vuestra calidad de enseñanza, las lecciones dadas y los golpes que día a día me ha ido constituyendo como la persona que hoy en día soy.

Gracias a todos.

Abstract

Los cambios en las tendencias de la computación de propósito general, hacia sistemas heterogéneos conformados por CPUs *multicore* y arquitecturas aceleradoras, GPUs en particular, han motivado la realización de este proyecto. En concreto, NVIDIA y sus dispositivos, en los cuales nos centraremos, se han establecido como líderes en el sector, y su modelo de programación CUDA, basado en la utilización de GPUs para el cómputo de propósito general, es ampliamente usado.

Este proyecto fin de carrera aborda la aceleración y optimización de tres aplicaciones científicas mediante el modelo de programación CUDA. Las seleccionadas han sido EP y CG, pertenecientes a los *NAS Parallel Benchmarks*, y FTDock, que simula el acoplamiento de dos proteínas.

Con las dos primeras, se pretende estudiar el rendimiento que este tipo de dispositivos es capaz de obtener trabajando con datos en punto flotante de doble precisión, cuyo soporte ha sido recientemente incorporado a esta tecnología. La tercera es una aplicación mucho más realista y bastante irregular, por lo que veremos qué tal se comporta esta arquitectura bajo situaciones no hechas específicamente a su medida.

Se han aplicado todo tipo de técnicas y optimizaciones, buscando el máximo aprovechamiento de los recursos que las tarjetas nos ofrecen, y su interrelación con el resto del sistema. Podríamos destacar el uso de memoria compartida, constante y de texturas, diferentes estrategias de mapeo de *threads* para conseguir accesos *coalesced*, maximización de la ocupación del dispositivo, uso de memoria *pinned* y técnicas para evitar saltos divergentes, entre otras.

Uno de los aspectos más importantes del proyecto ha sido la expansión de las aplicaciones a múltiples GPUs dentro de un mismo nodo, que ha requerido de la incorporación de OpenMP. FTDock, además, se ha desplegado sobre distintos nodos con modelos diferentes de GPUs, lo que ha dado lugar a tener que lidiar con un balanceo de carga no homogéneo. En esta última versión, resultó imprescindible hacer uso de MPI para llevar a cabo la comunicación entre los nodos. Como resultado, encontramos a tres modelos de programación conviviendo bajo el mismo ejecutable.

Los resultados en cuanto a rendimiento han sido bastante satisfactorios, sobre todo para tamaños de problemas grandes que permitían alimentar lo suficiente a las arquitecturas aceleradoras. Con respecto a EP, conseguimos hasta 58X de *speedup* con una sola GPU, y 114X haciendo uso de dos dispositivos. CG, con mayores necesidades de comunicación y sincronización que el anterior, alcanzaba hasta 29X y 41X, respectivamente, mientras que FTDock fue acelerada hasta 11X y 22X, con uno y dos dispositivos, y hasta 35X haciendo uso de tres nodos multi-dispositivo heterogéneos.

Capítulo 1

Introducción y Contexto del Proyecto

1.1 Introducción

Hoy en día, el mundo de la computación de propósito general se encuentra en un estado de continua evolución y está sufriendo cambios en su estructura interna. Cada vez son más los casos en los que supercomputadores de gran escala utilizan aceleradores como uno de los pilares de su potencia de cálculo. Como ejemplo, podemos destacar el caso del supercomputador Tianhe-1, compuesto por 2560 tarjetas gráficas ATI dobles, como complemento a sus 5120 CPUs Intel Xeon, que es capaz de superar la barrera del *petaflop* (1000 trillones de operaciones en punto flotante por segundo), o de Roadrunner, que gracias a sus Power Cells, hace tan sólo unos meses se consagraba como el supercomputador más potente del mundo [39].

Esta tendencia está dando lugar a sistemas más potentes y sofisticados, pero también más difíciles de utilizar y aprovechar toda su potencia de cómputo debido a su heterogeneidad. Ya no es suficiente con llevar a cabo la laboriosa tarea de paralelizar una aplicación, sino que además, debemos tener en cuenta que existen diferentes unidades de cómputo que tendremos que comunicar entre sí, sincronizar, y balancear la carga de trabajo en función de si las características del código se adecúan mejor a las capacidades de una u otra. Por si eso no fuera suficiente, en la mayoría de los casos se hace imprescindible hacer converger diferentes modelos de programación en un mismo código fuente. Este hecho conlleva que la complejidad de desarrollo crezca exponencialmente y esta tenga que ser realizada por verdaderos especialistas.

La aparición del estándar OpenCL [31][42] pretende marcar un antes y un después en la programación de arquitecturas heterogéneas. La implementación del mismo no busca otra cosa que ofrecer la posibilidad de programar un entorno heterogéneo, como el descrito anteriormente, utilizando un único lenguaje. Sin embargo, el excesivo bajo nivel de abstracción y la escasa madurez que lo caracterizan ha generado bastante controversia. Son necesarios mayores conocimientos para su utilización, además de que ofrece algunas diferencias en cuanto a rendimiento respecto de los modelos nativos de cada arquitectura, por lo que actualmente, en ningún caso

podríamos hablar de un claro sustitutivo de CUDA si el objetivo es la programación de la GPUs de NVIDIA.

El mayor aspecto positivo de la aparición de estas nuevas alternativas para el cómputo de carácter general, es que una gran potencia de cálculo se encuentra al alcance de cualquiera. El *Home Supercomputing* ligado al *Green Supercomputing* [40] están de moda. Y es que no sólo podemos construir nuestro particular supercomputador por muy poco dinero, sino que además, el consumo eléctrico de la misma solución basada en CPUs sería decenas de veces superior. Sin ir más lejos, en la Universidad de Amberes, Bélgica, podemos encontrar un sistema basado en GPUs de NVIDIA. Este supercomputador de "andar por casa", llamado Fastra II, es capaz de procesar hasta 12 trillones de operaciones en punto flotante por segundo (*teraflops*), consumiendo 100 veces menos energía que su alternativa con CPUs, y por un coste de fabricación de aproximadamente 6000 euros [30][41].

Pero esta revolución no sólo se reduce al mundo de las tarjetas gráficas. El salto radical hacia el *multicore*, que se produjo hace unos años en el diseño de microprocesadores, resulta ser, a día de hoy, una realidad totalmente afianzada. Ejemplo de ello son los prototipos de 80 *cores* en los que Intel ha estado investigando [29], o el anuncio de la salida al mercado de sus multiprocesadores de seis y ocho núcleos.

1.2 Contexto del Proyecto

El curso 2009/2010 lo he desarrollado en la Universidad Politécnica de Cataluña como participante en el Sistema de Intercambio entre Centros Universitarios Españoles (SICUE), y beneficiario de la beca Séneca promovida por el Ministerio de Educación. Esta circunstancia me ha proporcionado la posibilidad de llevar a cabo un proyecto fin de carrera conjuntamente con personal de la Universidad Politécnica de Cataluña y de la Universidad de Murcia.

Tras dos años como alumno interno del departamento de Arquitectura y Tecnología de Computadores de la Universidad de Murcia, he podido conocer las diferentes técnicas y modelos de programación paralela que se utilizan actualmente en el cómputo científico. Esto también me permitió vislumbrar las tendencias futuras que podrían seguirse en este sentido, donde el uso de tarjetas gráficas como aceleradores, y CUDA, en concreto, me parecieron de lo más prometedoras. Estas circunstancias, junto con la fuerte evolución de CUDA y las GPUs de NVIDIA, me movieron a desarrollar el proyecto fin de carrera aplicando a fondo esta tecnología en un campo tan interesante como el de la ciencia.

1.3 Estructura del Documento

En los siguientes apartados de este Capítulo queda reflejada la planificación inicial del proyecto, así como los costes estimados del mismo. El Capítulo 2 contempla todo lo relacionado con el estado del arte, desde la evolución de las CPUs y GPUs hasta la tecnología y modelo de programación CUDA. Los objetivos marcados por el proyecto, y la metodología seguida para alcanzarlos, se describen en el Capítulo 3. En el Capítulo 4 podemos encontrar la descripción de las aplicaciones utilizadas para el proyecto y su correspondiente aceleración mediante CUDA. El Capítulo 5 muestra la planificación real del proyecto y el verdadero coste del mismo, dando paso al Capítulo 6, donde se comentan las conclusiones finales alcanzadas, junto con la enumeración de posibles vías futuras. Por último, se incluye un apartado de bibliografía y referencias, seguido de una serie de anexos que pueden resultar necesarios para la correcta comprensión del proyecto, como son el desarrollo de algunos scripts y herramientas que se han necesitado, los *profilings* de algunas versiones, la guía de ejecución del código fuente, etc.

1.4 Planificación Inicial

La planificación inicial del proyecto es la mostrada en la Figura 1.1*:*

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	**Búsqueda, Análisis y Estudio Bibliográfico**	01/09/2009	01/06/2010	35h
2	**EP**	01/11/2009	31/12/2009	70h
3	**CG**	01/02/2010	31/03/2010	50h
4	**FTDock**	15/04/2010	25/05/2010	50h
5	**Memoria del Proyecto**	01/01/2010	01/06/2010	120h
6	**Total**	**01/09/2009**	**01/06/2010**	**325h**

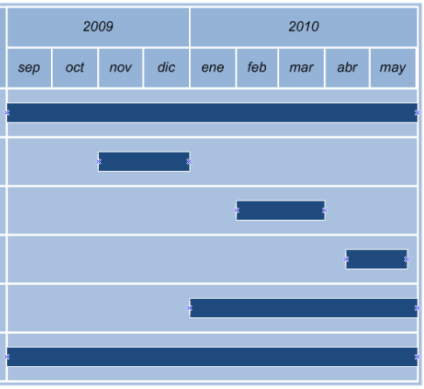


Figura 1.1. Planificación Inicial del Proyecto

1.5 Estimación de Costes

Según las horas estimadas en el apartado 1.4, suponiendo que el salario mensual de un Ingeniero en Informática recién matriculado sea de 1200 euros al mes, con un total de 160 horas mensuales, la hora de trabajo tendría un coste de 7,5 euros. Por tanto, el coste estimado en mano de obra del proyecto será de:

2437,5 €

Capítulo 2

Estado del Arte

2.1 Evolución de las CPUs y las GPUs

En los últimos años, se ha abierto un nuevo campo en el ámbito de la computación de propósito general, y los ya existentes han comenzado a evolucionar en nuevas direcciones. Las Unidades Centrales de Procesamiento – en inglés CPUs – han mejorado su rendimiento de manera sustancial, pero lejos quedan las dos pasadas décadas, en las que se producían enormes mejoras en el rendimiento de procesadores capaces de ejecutar un solo *thread* simultáneamente. En dicha época, los desarrolladores de software simplemente tenían que esperar a que saliera a la luz la próxima generación de procesadores, para ver incrementado en gran medida el rendimiento de sus aplicaciones secuenciales. La funcionalidad de las aplicaciones se limitaba a explotar al máximo la potencia que estas unidades secuenciales nos ofrecían, por lo que las características de dicho software iban creciendo y evolucionando a la par que lo hacían estas últimas. Sin embargo, llegó un momento en el que aparecieron problemas que afectaron a esta evolución: la conocida *power wall* – dificultad para disipar el calor producido por los circuitos integrados al trabajar a frecuencias de reloj muy altas – y la *memory wall* – creciente disparidad entre la velocidad de cómputo de una CPU y el tiempo de acceso a memoria fuera del chip –, junto con la incapacidad de mejorar el IPC – número de instrucciones ejecutadas por ciclo de procesador – de los programas que utilizaban un solo *thread*, hacían imposible el aprovechamiento del número de transistores que la propia Ley de Moore ofrecía para incrementar el rendimiento de los microprocesadores. Por estas razones, ha sido necesario llevar a cabo una modificación en la dirección en la que este mundo estaba evolucionando, para encaminarse hacia los actuales *Chip-level Multi-Processors* (CMPs), más comúnmente conocidos como procesadores *multicore*.

El salto hacia esta tecnología en el ámbito de las CPUs ha supuesto un cambio drástico en el modo de concebir el software desde el punto de vista de los desarrolladores. Ya no es posible incrementar el rendimiento y la funcionalidad de las aplicaciones de una manera tan sustancial como hace unos años, simplemente esperando a que una nueva gama de procesadores aparezca en el mercado. Ahora, es el propio

programador el que debe hacer el esfuerzo para que su aplicación se ejecute lo mejor posible y utilice todos los recursos, núcleos en este caso, que las nuevas CPUs ofrecen. Sino, por el contrario, el rendimiento de la misma se verá mermado con el limitado aumento en la capacidad de procesamiento de uno solo de los *cores*, los cuales pasarán a ser cada vez más insignificantes si se observan desde un punto de vista individual.

Paralelamente, las Unidades Gráficas de Procesamiento – GPUs en inglés – se desenvolvían en un ámbito diferente. Centradas en el mundo de los gráficos, su arquitectura se componía, principalmente, de dos tipos de unidades funcionales replicadas: los *pixel shaders* y los *vertex shaders*. Estas unidades tenían asignada una función específica, limitando así su uso para otros fines. Al contrario que ocurría con las CPUs, el avance en la tecnología de transistores permitía construir GPUs mucho más potentes, ya que estos eran utilizados, a grandes rasgos, para aumentar el número de unidades funcionales replicadas en cada dispositivo. Fueron varios los intentos que tuvieron lugar para aprovechar esta potencia de cómputo sobre problemas de propósito general. Lenguajes de programación como Cg [44]– *C for Graphics* – o HLSL [45] – *High Level Shader Language* – tuvieron su momento de éxito, pero la gran dificultad que suponía su uso acotó su expansión de manera significativa.

Con la llegada de DirectX[1] 9 [43], fueron incluidas unidades funcionales para operaciones en punto flotante y, además, se eliminaron ciertas restricciones sobre los *shaders* de píxeles y vértices, permitiendo que estos fueran programables. Sin embargo, no fue hasta la versión 10 de esta interfaz donde se introdujeron unidades funcionales unificadas, que permitían realizar distintas operaciones sobre tipos de datos diferentes, píxeles y vértices, entre otros. Este nuevo tipo de unidad funcional abrió las puertas del cómputo general sobre GPUs – GPGPU [46], *General-Purpose computation on Graphics Proccesing Units* – brindando la posibilidad de ejecutar programas en estos potentes dispositivos que antes solamente se podían ejecutar sobre CPUs.

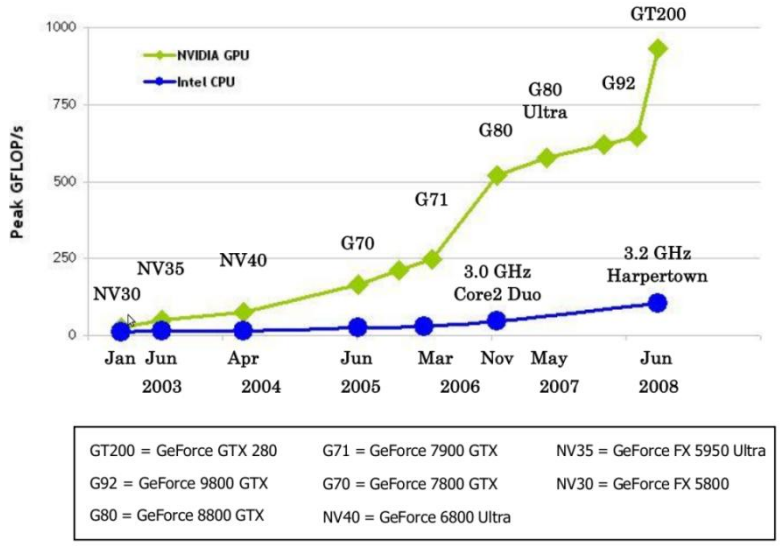


Figura 2.1. Comparación de la evolución de la potencia de CPUs y GPUs [1]

[1] Microsoft DirectX es una interfaz de programación de aplicaciones multimedia, especialmente juegos.

Como podemos apreciar en la Figura 2.1, se ha producido un enorme desfase entre la potencia de cálculo de las GPUs y las CPUs, a pesar de que estas últimas han comenzado a desviarse en una dirección ligeramente similar a las primeras, en cuanto a implementación. No obstante, aunque son muchos años de investigación y desarrollo los que las GPUs llevan en este sentido, las CPUs distan de estas en la finalidad de uso, por lo que difícilmente terminarían convergiendo en el mismo punto si se encontraran a idéntico nivel. Mientras que las CPUs pueden ejecutar programas de cualquier naturaleza, gracias a que cuentan con *cores* menos numerosos pero más agresivos, sofisticados e independientes que los de una GPU, estas últimas no pueden hacerlo, o al menos no con buen rendimiento, debido a que sus cientos de "*cores*" son muy simples, menos agresivos y más limitados. Una comparación en este sentido la encontramos en la Figura 2.2, donde puede observarse la diferencia en cuanto al número de transistores dedicados a constituir unidades de cómputo (color verde) en ambas tecnologías. En definitiva, a nivel de general, actualmente deberíamos considerar a estos dispositivos gráficos como complementos o aceleradores de las CPUs.

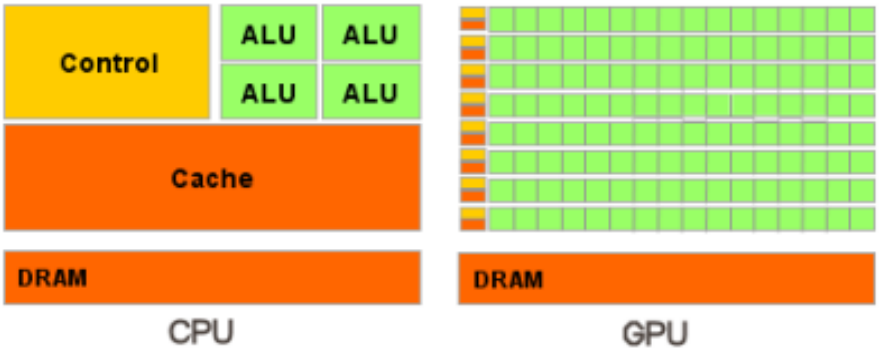


Figura 2.2. Arquitectura CPU vs GPU [1]

2.2 CUDA: Compute Unified Device Architecture

Compute Unified Device Architecture es la apuesta más fuerte que ha realizado la corporación NVIDIA sobre el uso de GPUs en el ámbito de la computación de propósito general. Además de la arquitectura y su juego de instrucciones asociado, CUDA abarca desde un nuevo modelo de programación, cuya base se fundamenta en la extensión del lenguaje C/C++, hasta un compilador y una serie de herramientas y librerías que ayudan en el desarrollo de aplicaciones que siguen esta filosofía.

La idea básica es tratar de aprovechar al máximo la capacidad de los cientos de núcleos de cómputo que poseen las GPUs, y lanzar, así, un altísimo número de *threads* en paralelo que lleven a cabo las mismas operaciones sobre un conjunto de datos diferente. Este tipo de paralelismo es conocido como paralelismo SIMD – *Single Instruction Multiple Data* – y es el que encontramos en las aplicaciones que nativamente hacen uso de estos dispositivos: aplicar las mismas operaciones sobre un

elevadísimo número de píxeles (datos). De esta manera, si la aplicación posee las citadas características, podrá lanzarse a la GPU, obteniendo, normalmente, mejores resultados de rendimiento que si la ejecución tuviera lugar en la CPU.

Es evidente que ni todas las aplicaciones van a caracterizarse por este tipo de paralelismo, ni este tipo de paralelismo va a conformar el 100% de una aplicación, por lo que debemos disponer de una CPU para dichos casos, y permitir que ambas tecnologías se complementen. Por tanto, debemos ver a CUDA como la base para un entorno heterogéneo compuesto de CPUs y GPUs, donde cada componente tiene un papel imprescindible en el rendimiento.

Con respecto a la programación, tenemos a nuestra disposición dos APIs diferentes. La primera de ellas, la API Runtime, proporciona un nivel más alto de abstracción al programador, lo que facilita que este no tenga que ser consciente de algunas cuestiones de bajo nivel. La alternativa la ofrece la API Driver, donde el nivel de abstracción es mucho más bajo, siendo así más complicada la programación, pero permitiendo una mayor versatilidad en la toma de ciertas decisiones.

Podemos encontrar dispositivos NVIDIA compatibles a partir de la serie 8 de la familia GeForce y de la serie FX y Plex de la familia Quadro. Conjuntamente, la corporación lanzó al mercado los dispositivos Tesla, destinados sólo al cómputo de carácter general, que se caracterizan por poseer una mayor frecuencia de procesamiento y tamaño de memoria, junto con la ausencia de salida de video.

Por medio de *wrappers*, en lugar de C/C++, CUDA también puede utilizarse mediante distintos lenguajes de programación, como Python, Fortran y Java, entre otros.

2.2.1 Visión General de la Arquitectura de una GPU

En este apartado vamos a describir algunos detalles básicos de la arquitectura de una GPU de NVIDIA. El tema no será tratado con excesiva profundidad, ya que el objetivo no es más que introducir una serie de conceptos a los que se hará referencia a lo largo de este documento.

Arquitectónicamente, una GPU podría encasillarse dentro del grupo de procesadores vectoriales. Este tipo de procesadores se caracteriza por ser capaz de ejecutar una misma operación sobre múltiples datos (vectores de datos) de manera simultánea, consiguiendo, así, un alto rendimiento sobre estas situaciones.

Un esquema general de la arquitectura de una GPU de la serie NVIDIA GT200 puede observarse en la Figura 2.3. Existen diez unidades de procesamiento compuestas, denominadas *Streaming Multiprocessors* o SMs, que están formadas por ocho núcleos más simples, capaces de trabajar sobre varios vectores de datos simultáneamente. Cada uno de estos SMs también posee una pequeña memoria caché (memoria constante), que permite acelerar los accesos de sólo lectura a la memoria global del dispositivo. A su vez, varios SMs se agrupan en *Thread Processing Clusters* o TCPs, donde podemos encontrar un primer nivel de memoria caché de texturas privado, a compartir por los

distintos multiprocesadores del conjunto. Un segundo nivel de este tipo de caché será compartido por todos los TCPs.

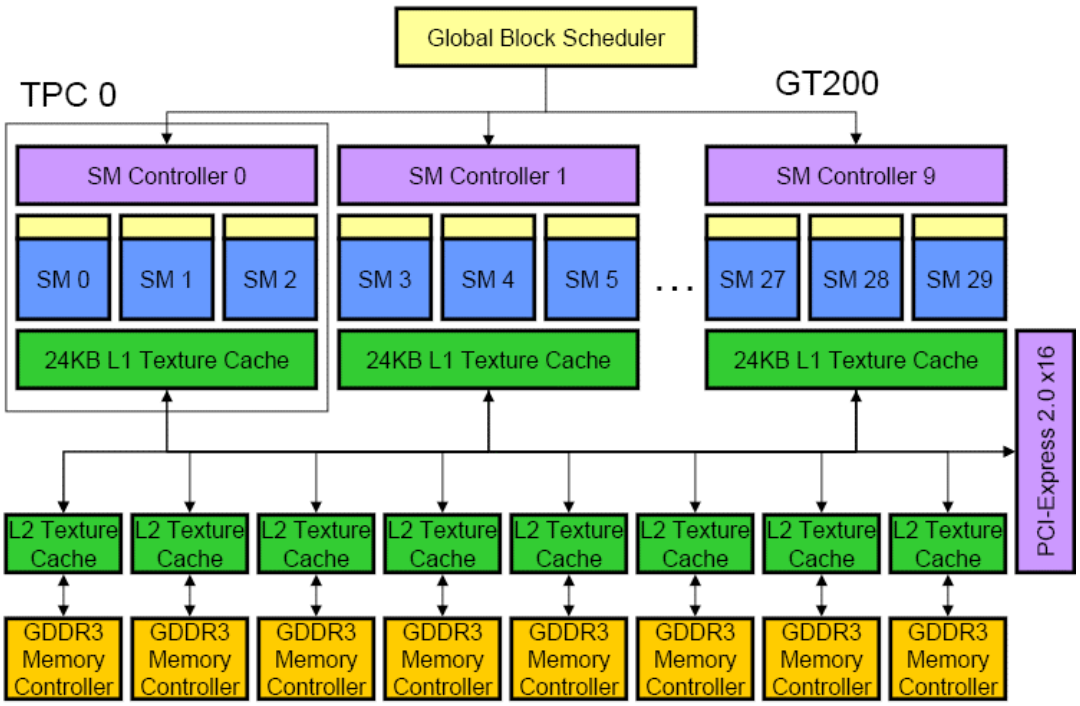


Figura 2.3. Esquema de la arquitectura de una GPU GT200 [8]

Cada uno de los 240 *cores* puede realizar operaciones sobre datos de tipo entero o en punto flotante de simple precisión. Para realizar operaciones avanzadas, como una raíz cuadrada, cada SM incorpora dos unidades funcionales a repartir entre sus ocho núcleos. También es el caso de las operaciones en punto flotante de doble precisión, donde encontramos una sola unidad funcional compartida de la misma forma. Dichas características originarán un rendimiento más bajo en este tipo de operaciones.

Por último, cabría destacar que, al igual que las CPUs, las instrucciones recorren un cauce de varias etapas para ser ejecutadas. Sin embargo, a diferencia de estas, las GPUs no poseen mecanismos de predicción de saltos, la ejecución tiene lugar en orden, y no se llevan a cabo ningún tipo de ejecución de instrucciones de manera especulativa [33].

2.2.2 Modelo de Ejecución

En un programa en CUDA existen dos partes muy bien diferenciadas: el ámbito del ***host***, referente a código dirigido a las CPUs, y el ámbito del dispositivo o ***device***, relativo al código que se lanzará en al menos una GPU. De esta forma, un programa en CUDA consiste en la ejecución de diferentes fases de código en la CPU o/y en la GPU, dependiendo de sus características. Si una de estas fases porta un considerable

paralelismo como el descrito anteriormente, este debería lanzarse al dispositivo en busca de un mayor rendimiento. En caso contrario, el código deberíamos ejecutarlo en la CPU.

Como dispositivo independiente de la jerarquía de memoria principal del sistema, la GPU necesitará de transferencias explícitas de datos entre esta y su espacio de memoria, conllevando esta acción una gran cantidad de tiempo adicional que afectará, en gran medida, al rendimiento de la aplicación.

Además de su correspondiente fase de ejecución, el *host* jugará un papel determinante en este entorno. Se verá implicado en la reserva y liberación de memoria en el dispositivo, así como en la realización de las transferencias de datos entre ambos ámbitos, y determinará cuándo el dispositivo comenzará a ejecutar cada uno de sus *kernels*. Podemos ver un esquema de las distintas fases en la Figura 2.4.

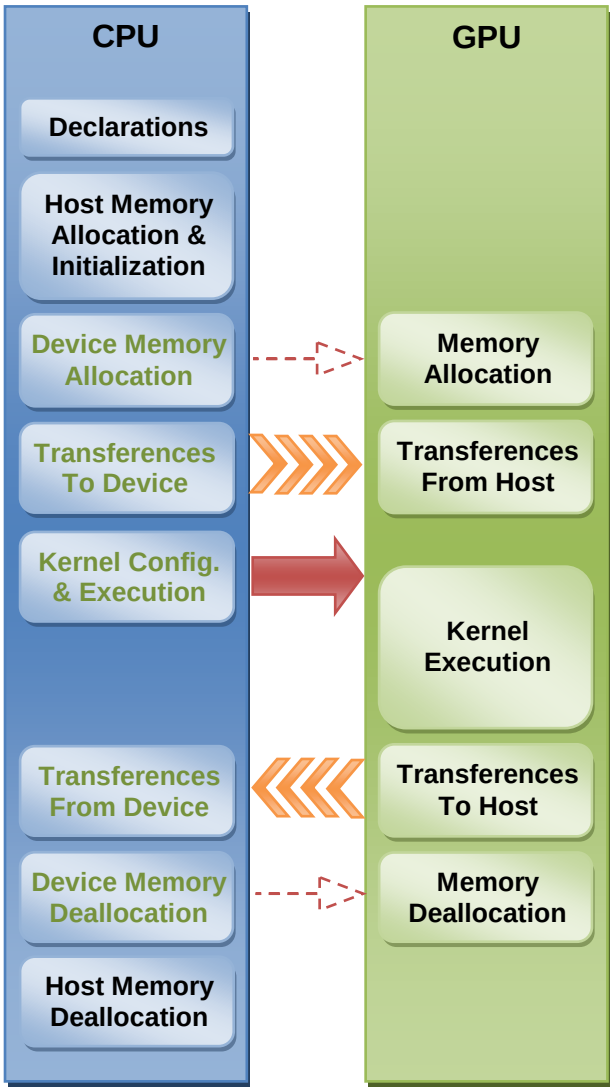


Figura 2.4. Fases del Modelo de Ejecución de CUDA

Entrando en un mayor nivel de detalle, la Figura 2.5 muestra un esquema del modelo de ejecución. Cada fase del código que está destinado a ejecutarse en una o varias GPUs recibe en nombre de ***kernel***. Este código será ejecutado por todos y cada

uno de los *threads* del/los dispositivo/s, por lo que estará escrito en función del identificador de estos *threads*, entre otros factores.

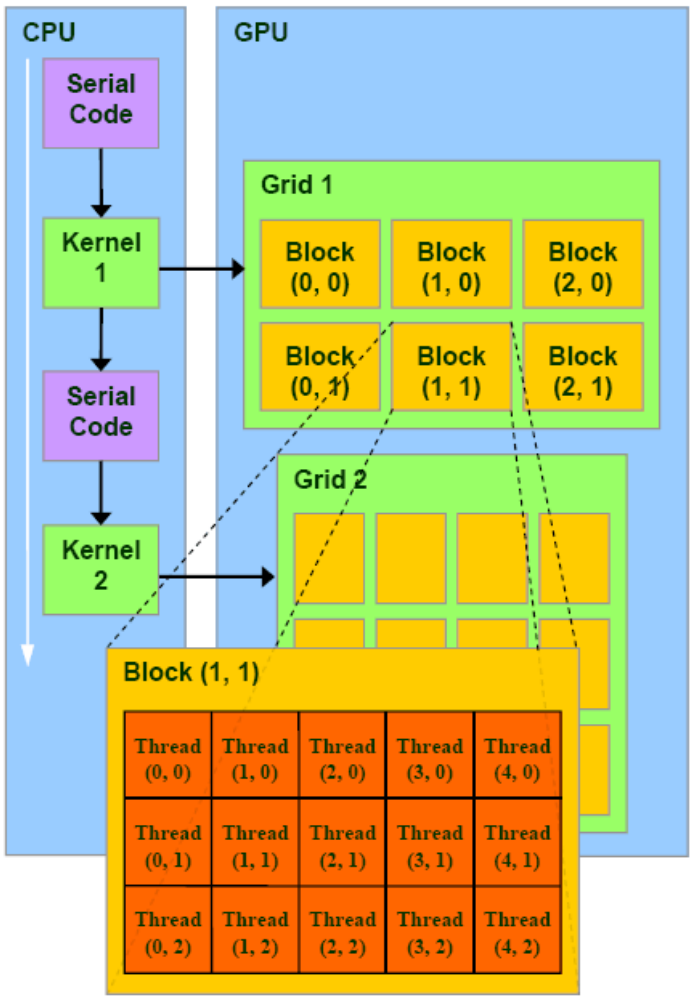


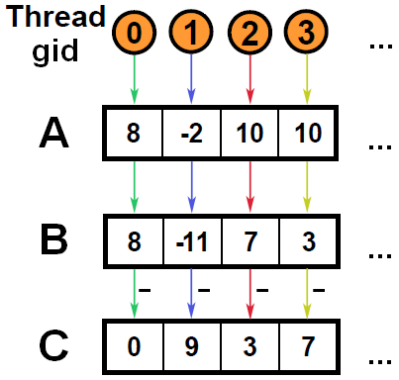
Figura 2.5. Modelo de Ejecución de CUDA. *Kernels*, *Grids*, Bloques y *Threads* [8]

Un **bloque** es una agrupación de *threads* y viene a representar la unidad mínima de asignación entre dichos *threads* y un multiprocesador del dispositivo. De esta forma, si el número de *threads* por bloque es demasiado alto, tendremos menos bloques por multiprocesador ejecutándose en paralelo, y si es demasiado bajo, justo lo contrario. Existe un máximo de bloques y *threads* que un multiprocesador puede manejar físicamente en paralelo. Para un aprovechamiento óptimo de los recursos, debemos tratar de que el número de *threads* por bloque sea múltiplo del máximo de *threads* que soporta cada multiprocesador, y sea lo suficientemente grande como para que los bloques puedan cubrir todos los recursos disponibles. Se recomienda que este valor coincida con una potencia de dos no excesivamente pequeña. También debemos intentar disponer de un número de bloques elevado para ocultar tiempos de latencia, y permitir que los multiprocesadores sigan trabajando cuando un bloque queda a la espera por algún motivo. A ser posible, esta cifra debe ser mayor al doble del número de multiprocesadores del dispositivo.

Los bloques y *threads* de cada *kernel* vienen englobados en una estructura denominada ***grid***. Esta estructura representa la materialización de cada *kernel* en un dispositivo, y es en la cual el programador establece el número de *threads* por bloque y número total de bloques con los que se lanzará dicho código a la GPU. Así, podremos ejecutar el mismo *kernel* con la misma o diferente configuración de *threads* y bloques sin que sea necesaria ningún tipo de modificación de su código.

Si bajamos un nivel más de abstracción, por debajo de los bloques nos encontramos a los ***warps***, que no son más que una nueva agrupación de *threads* que vienen a constituir la unidad básica de ejecución. Actualmente, los *warps* están formados por 32 *threads*, lo que quiere decir que una misma instrucción será ejecutada a la vez por grupos de este tamaño. Cuando esto no sea posible porque distintos *threads* dentro del mismo *warp* diverjan en instrucciones diferentes, cada *threads* ejecutará su instrucción en serie con respecto a las demás distintas, disminuyendo, así, el paralelismo. También encontraremos referencias a mitades de un *warp* o ***half-warp***, ya que es la unidad de acceso a la memoria del dispositivo.

Volviendo a la Figura 2.5, vemos como es posible disponer de diferentes dimensiones a la hora de desplegar los *threads* y los bloques dentro de un *grid*. Dichas dimensiones nos permiten mapear los *threads* a estructuras multidimensionales de una manera más sencilla. Tendremos hasta tres dimensiones para indexar un *thread* (variables `threadIdx.[x,y,z]`) y dos para los bloques (variables `blockIdx.[x,y]`). Además, `gridDim.[x,y,z]` y `blockDim.[x,y,z]` describen el tamaño de un *grid* (en bloques) y un bloque (en *thread*) respectivamente. Todas estas variables son preestablecidas y accesibles desde cualquier *kernel*.




```
__global__ void my_kernel( int* A,
                           int* B,
                           int* C)
{
  int lid = threadIdx.x;
  int gid = blockIdx.x*blockDim.x + lid;

  C[gid] = A[gid] - B[gid];
}
```

Figura 2.6. Ejemplo de *kernel*. Resta de dos vectores

Si pasamos a la práctica, la Figura 2.6 muestra cómo sería la declaración de un *kernel* que vendría a restar dos vectores y escribir la diferencia en un tercero. Cada *thread* que ejecute dicho *kernel* tendrá un identificador global, `gid`, que le servirá para acceder a las posición de los vectores que le toque computar. Este identificador será calculado como el desplazamiento global del *thread* con respecto al resto de bloques, más el identificador local del *thread* dentro del bloque en el que se encuentra (`lid` o

`threadIdx.x)`. Este *kernel* es caracterizado como `__global__`, lo que significa que podrá ser invocado tanto desde el *host* como desde otro *kernel*.

Suponiendo que el tamaño de los vectores es N (potencia de dos mayor o igual a 128), la Figura 2.7 describe dicha invocación con una configuración de 128 *threads* por bloque y un *grid* de (N/128) bloques. A, B y C representan a los vectores, donde los sufijos "_h" y "_d" indican si ese puntero hace referencia a la estructura que se encuentra en el *host* o en la memoria global del dispositivo, respectivamente.

```
//Declaration
int * A_h, *B_h, *C_h;  //Host pointers
int * A_d, *B_d, *C_d;  //Device pointers
//Host Memory Allocation
A_h = (int *) malloc (N * sizeof(int));
B_h = (int *) malloc (N * sizeof(int));
C_h = (int *) malloc (N * sizeof(int));

//Host structures initialization
...

//Device Memory Allocation
cudaMalloc((void **)&A_d, N * sizeof(int));
cudaMalloc((void **)&B_d, N * sizeof(int));
cudaMalloc((void **)&C_d, N * sizeof(int));

//Host To Device Transferences
cudaMemcpy(A_d, A_h, N*sizeof(int),cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, N*sizeof(int),cudaMemcpyHostToDevice);

//Kernel Configuration
dim3 block(128);
dim3 grid(N/128);

//Kernel Launching (GPU Execution)
my_kernel<<<grid, block>>>(A_d, B_d, C_d);

//Device To Host Transferences
cudaMemcpy(C_h, C_d, N*sizeof(int),cudaMemcpyDeviceToHost);

//Host Memory Deallocation
free(A_h);
free(B_h);
free(C_h);
//Device Memory Deallocation
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
```

Figura 2.7. Ejemplo de invocación a un *kernel*

La Figura también deja constancia de las distintas fases que podemos encontrar en la invocación de un *kernel* habitualmente, aunque algunas de ellas podrían suprimirse o encontrarse en distinto orden, dependiendo de las necesidades del programa.

Debemos destacar que la invocación a un *kernel* es asíncrona, lo que quiere decir que el control de ejecución se devuelve al *host* cuando este ha terminado de lanzar el código a la GPU. Para conseguir que el *host* espere a que el *kernel* termine su ejecución,

se puede hacer uso de la función `cudaThreadSynchronize()`, que actuará de *barrier* hasta que todos los *threads* del *kernel* hayan finalizado su trabajo. El uso de esta función puede suprimirse si a continuación vamos a realizar una transferencia de memoria, ya que la función `cudaMemcpy()`, además de ser bloqueante, espera a realizar su cometido hasta que el *kernel* no ha terminado de utilizar la GPU.

En definitiva, para aprovechar la potencia de estos dispositivos deberemos familiarizarnos, como mínimo, con los conceptos de *kernel*, *grid*, bloque y *thread*, sin detenernos en demasiados detalles de bajo nivel. Desde este punto, podremos desarrollar aplicaciones con un rendimiento más o menos aceptable, llevando a cabo una programación con granularidad de *thread*.

El hecho de que una arquitectura de carácter vectorial permita el desarrollo de aplicaciones sin tener en cuenta, inicialmente, el tamaño de los vectores o unidades de cómputo, *warps* en este ámbito, la dotan de cierta versatilidad sobre arquitecturas y/o modelos de programación similares. Tanto es así, que NVIDIA introduce el concepto de paralelismo SIMT – *Single Instruction Multiple Thread* – como evolución al paralelismo SIMD. En este nuevo término se destaca esta granularidad de *thread*, que no es posible explotar directamente en otras arquitecturas. Sin embargo, no debemos olvidar que deberíamos ahondar más profundamente en detalles de bajo nivel de la arquitectura (*warps*, diferentes tipos de memoria, etc.) si realmente queremos extraer el máximo rendimiento que esta nos puede ofrecer.

2.2.3 Modelo de Memoria

Es, la memoria, uno de los aspectos de las GPUs donde podemos encontrar más lagunas de información, quizás, debido a temas de confidencialidad o a que los detalles existentes son de tan bajo nivel que resulta imposible su inclusión detallada en los manuales. Sin embargo, el correcto uso de los diferentes tipos de memoria que un dispositivo nos brinda será nuestro mejor aliado si queremos conseguir un mayor rendimiento.

La Figura 2.8 nos muestra los seis tipos de recursos que los dispositivos nos ofrecen para almacenar información, y el ámbito de visibilidad de los mismos: registros (*registers*), memoria local (*local memory*), memoria compartida (*shared memory*), memoria constante (*constant memory*), memoria de texturas (*texture memory*) y memoria global (*global memory*). Por defecto, la coherencia entre los recursos que requieren ser manipulados manualmente, en caso de ser necesaria, deberá ser mantenida por el programador. Este hecho debe ser aún más tenido en cuenta cuando nos referimos a la interacción entre la memoria del *host* y la memoria del dispositivo. No obstante, existen formas de que ambos compartan información de manera automática, como veremos más adelante.

La **memoria global** es la memoria más abundante, pero también la que se caracteriza por una mayor latencia y menor ancho de banda. Si la comparamos con una

CPU, sería equivalente a la memoria RAM. Se encuentra fuera del chip de procesamiento y es el medio principal para comunicar datos entre el *host* y el dispositivo. Esta comunicación, por defecto, debe ser manualmente especificada por el programador. El tiempo de vida de los datos es de todo el programa, y estos son visibles por cualquier *thread* de cualquier bloque que se encuentre en ejecución en el dispositivo.

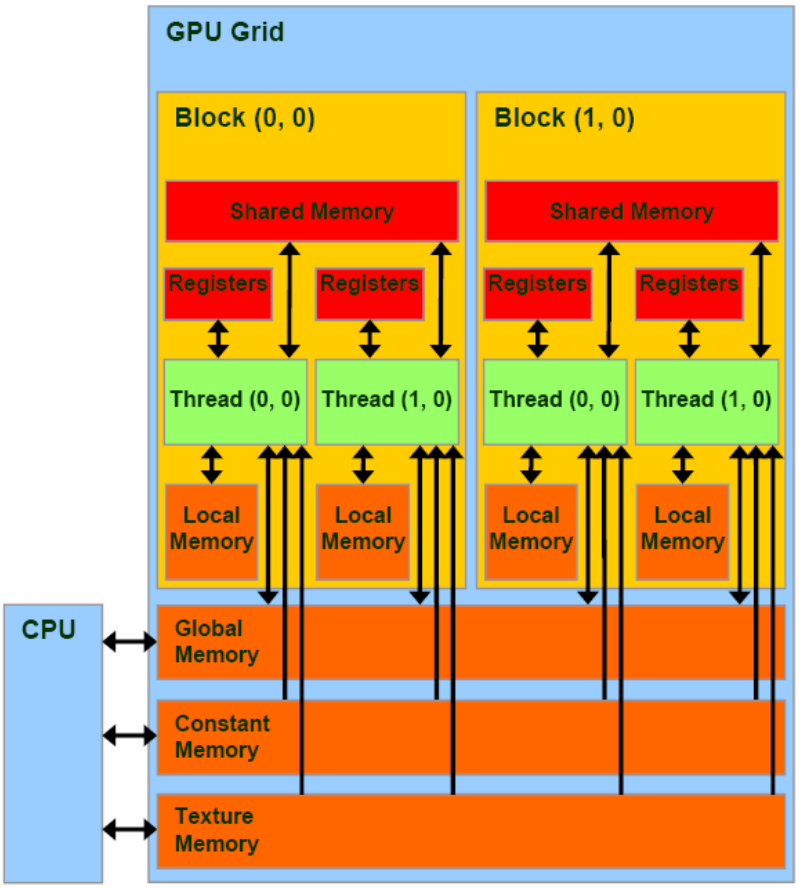


Figura 2.8. Modelo de Memoria de CUDA [8]

Los *threads* acceden a ella en grupos de *half-warps*. Para obtener el mejor rendimiento, los accesos deben ser ***coalesced***. Este concepto ha ido evolucionando a la vez que lo han hecho las GPUs de propósito general, siendo cada vez menos restrictivo. Para dispositivos con *Compute Capability* 1.0 y 1.1 (véase apartado 2.2.4), un acceso será *coalesced* cuando en cada *half-warp*:

i. El tamaño de palabra accedido por *thread* sea de 4, 8 o 16 bytes.
ii. El primer elemento de cada *half-warp* debe estar alineado a un múltiplo de 16 veces su tamaño, para que todos los elementos se encuentre dentro del mismo segmento de memoria.
iii. Los *threads* deben acceder a los datos secuencialmente: el *thread k* debe acceder a la palabra *k*. Pueden existir *threads* que no soliciten acceso a ningún dato, siempre que esto no altere el orden para los *threads* siguientes.

Si alguna de las restricciones no se cumple, la penalización será máxima, y se llevarán a cabo accesos secuenciales independientes de 32 bytes por cada *thread* del *half-warp*.

En dispositivos con *Compute Capability* 1.2 y 1.3, las restricciones anteriores desaparecen. En este caso, para cada *half-warp*, se busca hacer el menor número de transacciones posibles de segmentos de 32, 64 y 128 bytes, según corresponda. Resulta independientemente el orden de acceso de cada *thread* (iii) o si el primer elemento accedido está alineado al principio del segmento o no (ii). Tal diferencia supone una enorme ventaja con respecto a la definición anterior, ya que ahora los accesos no se serializarán si no cumplimos las restricciones.

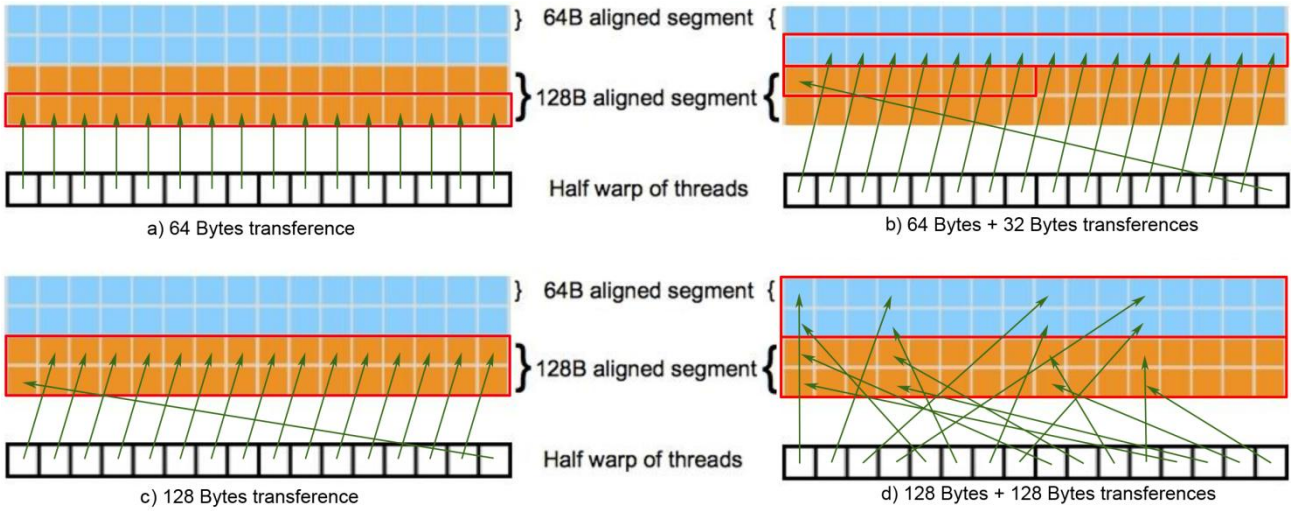


Figura 2.9. Ejemplos de Accesos a Memoria Global para *Comp. Cap.* 1.2 y 1.3

La Figura 2.9 muestra cuatro ejemplos de accesos a memoria global, realizados por un *half-warp*. El patrón a) sería un acceso completamente *coalesced* para cualquier *Compute Capability*, resultando, así, en una sola transferencia de 64 Bytes. Si se accediera a las mismas posiciones de memoria, pero en un orden desordenado, la transferencia se llevaría a cabo de igual forma para la versión 1.2 y 1.3, mientras que todos los accesos serían serializados para las dos primeras versiones. De igual modo para ellas, el resto de ejemplos desencadenarán la serialización completa de los accesos.

Los casos b) y c) se corresponden con el patrón de acceso anterior, desplazado un elemento. Resulta interesante ver como un mismo patrón de acceso puede desencadenar una o dos transferencias de memoria, dependiendo de la alineación con respecto al segmento, en la versión 1.2 y 1.3. Sobre las mismas versiones, respecto al patrón d), pese al desorden que podemos encontrar, solamente desencadenará dos transferencias de 128 Bytes.

Como podemos comprobar, la diferencia de funcionamiento entre las dos primeras versiones y las dos últimas es abismal, lo cual se traducirá de igual forma en el rendimiento cuando las restricciones del primer caso no se satisfagan. Si nos centramos en el segundo caso, para optimizar al máximo el ancho de banda útil, todos los *threads*

del mismo *half-warp* deben solicitar datos dentro del mismo segmento, y todos los datos del segmento tendrían que ser utilizados, aunque en este caso puedan serlo de manera desordenada.

La **memoria local** es una manera alternativa de utilizar memoria global, acotando la visibilidad y el tiempo de vida de los datos a nivel de *thread* y de *kernel* respectivamente. Por ende, no es otro tipo diferente de memoria y podemos asumir todo lo dicho anteriormente en lo que a características se refiere, con algunas salvedades. Normalmente, esta memoria es utilizada como área de *spilling* de los registros o para almacenar estructuras de datos de varios elementos, locales a cada *thread*. No es una decisión directa del programador el usarla o no, sino que es el compilador (estructuras de datos) y el *runtime* (*spilling*) los que arbitrarán a su antojo su utilización. Esto supone que el hecho de realizar los accesos *coalesced* no esté en nuestra mano, por lo que serán estas entidades las encargadas de situar los datos en memoria global, de manera que se obtengan los mejores resultados de acceso.

Los **registros** son el recurso de almacenamiento más rápido de las GPUs, junto con la memoria compartida. Se encuentran dentro del chip de procesamiento y su visibilidad y tiempo de vida son similares a los de la memoria local. Como ya hemos comentado, al tratarse de un recurso limitado, cuando no quedan registros disponibles se utiliza la memoria local como área de *spilling.* Al igual que esta, el programador no puede controlar directamente su uso, aunque sí establecer algunas limitaciones.

La **memoria compartida** es uno de los recursos más valiosos del dispositivo. Se encuentra dentro del chip y, a diferencia de los registros y la memoria local, su visibilidad es a nivel de bloque. Esto quiere decir que todos los *threads* del mismo bloque pueden compartir información utilizando este medio. Es un tipo de memoria bastante limitado, y será el programador el encargado de mover los datos entre esta y el resto de recursos manualmente.

Además, su tiempo de vida es a nivel de *kernel* y su latencia puede equipararse a la de los registros, en el mejor de los casos. Sin embargo, esto no siempre será así. Los accesos a la memoria se llevan a cabo en grupos de *half-warps*, por lo que se dispone de 16 bancos con tamaño de palabra de 4 bytes. Cuando varios *threads*, pertenecientes al mismo *half-warp*, intenten acceder al mismo banco, ya sea para alcanzar el mismo elemento o elementos diferentes, dichos accesos serán atendidos en serie, y por tanto, la latencia será mayor.

La Figura 2.10 muestra algunos patrones de acceso que ilustran lo descrito anteriormente. En los ejemplos a) y b), los accesos tendrán lugar en paralelo a pesar de la diferencia de orden, mientras que en el caso de c) y d), tendríamos una serialización de factor dos y ocho, respectivamente. La penalización es tal, porque se producen conflictos en los bancos de memoria, ya que varios *threads*, grupos de dos y ocho para cada ejemplo, intentan conseguir datos a través del mismo puerto, y este solamente puede atender una petición a la vez. Sólo cuando todos los *threads* del mismo *half-warp* soliciten el mismo elemento, se producirá un solo acceso y el dato será repartido a todos los *threads* (*broadcast*).

El patrón c) ocurrirá de manera natural cuando utilicemos arrays de datos en punto flotante de doble precisión, por tratarse de elementos de 8 bytes uniformemente distribuidos. Estos provocarán que la mitad del *half-warp* entre en conflicto con la segunda mitad, en las dos lecturas de 4 bytes que necesita cada *thread* para leer un elemento.

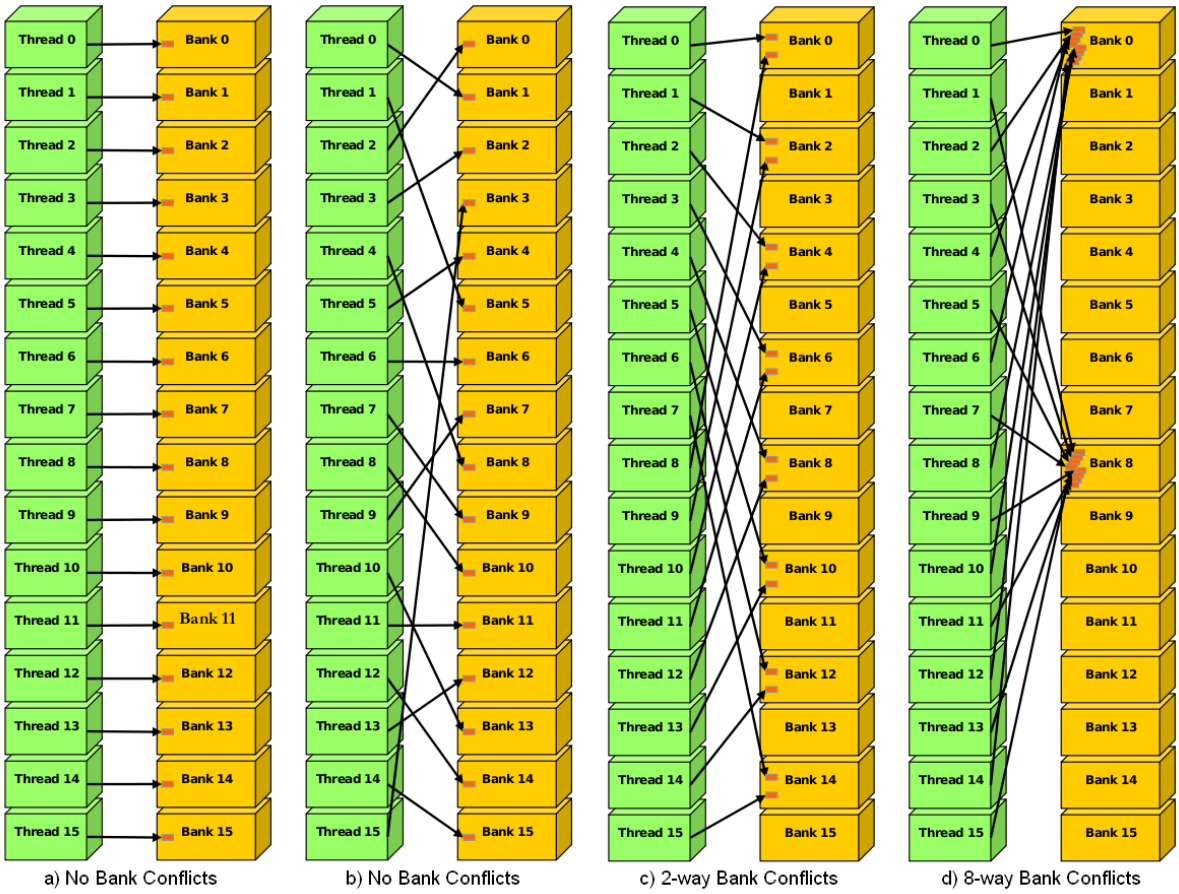


Figura 2.10. Ejemplos de Accesos a Memoria Compartida [1]

La **memoria constante** es un tipo de memoria dentro del dispositivo, cuya función es ofrecer un nivel de caché por encima de la memoria global. Como su nombre indica, se trata de una memoria de sólo lectura a nivel de GPU, por lo que los datos deberán ser transferidos/asociados a ella desde el *host*, antes de proceder con la ejecución de un *kernel* que acceda a esos elementos. Su visibilidad y tiempo de vida son similares a los de la memoria global, pero hay que destacar que no guarda coherencia con la misma.

Una lectura sobre la memoria constante resultará en un acceso a la memoria global del dispositivo en caso de fallo de caché, y un acceso a la memoria constante en caso de acierto. Dicho acierto, en condiciones óptimas, será tan rápido como un acceso a un registro. Para que esto sea así, todos los *threads* de un mismo *half-warp* deberán leer la misma posición de memoria. En caso contrario, los accesos a memoria serán serializados, y el coste del acceso se incrementará linealmente, en función del número de posiciones de memoria diferentes solicitadas dentro del mismo *half-warp*.

Actualmente, los dispositivos de NVIDIA cuentan con una memoria constante total de 64 KB. Este tamaño podría resultar interesante si lo contemplamos desde el punto de vista de una memoria caché convencional de una CPU, donde es posible trabajar con un conjunto de datos de mayor tamaño al de su capacidad total, de manera transparente al usuario. Sin embargo, en este caso nos encontramos con la limitación de no poder utilizar ni un solo byte más por encima de la capacidad de esta memoria, al menos de manera automática, por lo que el tamaño podría resultar algo escaso para objetivos de carácter general.

La **memoria de texturas** es utilizada nativamente para leer texturas de imágenes en aplicaciones gráficas. Podemos decir que esta memoria pretende suplir las mismas funcionalidades que la memoria constante (nivel de caché sobre memoria global), pero con restricciones de acceso diferentes. Además, esta caché está organizada en dos niveles y diseñada para optimizar accesos con localidad de datos en dos dimensiones (apartados 2.2.1 y 2.2.3). Con ello, se consigue que los *threads* de un mismo *half-warp* que lean datos próximos en alguna de las dimensiones obtengan un mejor rendimiento. Si los datos resultan no estar en esta caché, será necesario leer de memoria global.

Esta memoria es menos restrictiva en cuanto a patrones de acceso, por lo que resulta de gran utilidad cuando nuestro programa no es capaz de adaptarse a las restricciones de la memoria global o constante, siempre que no se requieran escrituras.

Memoria	Localización	Cacheada	Accesos GPU	Visibilidad	Tiempo de Vida
Registros	Dentro del chip	-	R/W	Thread	Kernel (Thread)
Local	Fuera del chip	No	R/W	Thread	Kernel (Thread)
Compartida	Dentro del chip	No	R/W	Bloque	Kernel (Bloque)
Global	Fuera del chip	No	R/W	host+device	Programa
Constante	Caché dentro del chip	Sí	R	host+device	Programa
Texturas	Cachés fuera del chip	Sí	R	host+ device	Programa

Tabla 2.1. Resumen de Memorias de una GPU

La Tabla 2.1 muestra un resumen de lo descrito hasta sobre las memorias de dispositivos de hasta la *Compute Capability* 1.3.

2.2.4 Compute Capabilities

Bajo el concepto de *Compute Capability*, se esconde la descripción de las capacidades y características que posee cada determinada versión de la arquitectura de una GPU, en lo referente a CUDA. Existen diferentes versiones que, poco a poco, van haciendo de estos dispositivos de NVIDIA unidades de cómputo más potentes y versátiles.

La Tabla 2.2 y Tabla 2.3 muestran las especificaciones y funcionalidades más importantes que soporta cada una de las versiones.

Dispositivos con *Compute Capability* 1.0 podemos encontrar en la mayoría de las tarjetas gráficas de la serie G80 de NVIDIA. Algunas versiones de esta serie poseen *Compute Capability* 1.1, al igual que la serie G90. La versión 1.2 se encuentra en algunos dispositivos de la serie GT200, aunque los de gama alta ya incorporan la *Compute Capability* 1.3, con soporte para realizar operaciones en punto flotante de doble precisión. La versión 2.0 hace referencia a la nueva generación de dispositivos que acaba de aparecer en el mercado (GF100 y Fermi). No se ha contemplado esta versión para el desarrollo del proyecto.

De igual forma, existen dispositivos con distintas capacidades en la familia Tesla y Quadro.

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Feature Support
(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th></tr>
<tr><td>Integer atomic functions operating on 32-bit words in global memory (Section B.10)</td><td>No</td><td colspan="4">yes</td></tr>
<tr><td>Integer atomic functions operating on 64-bit words in global memory (Section B.10)</td><td colspan="2" rowspan="3">No</td><td colspan="3" rowspan="3">Yes</td></tr>
<tr><td>Integer atomic functions operating on 32-bit words in shared memory (Section B.10)</td></tr>
<tr><td>Warp vote functions (Section B.11)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.10)</td><td colspan="4" rowspan="4">No</td><td rowspan="4">Yes</td></tr>
<tr><td>__ballot() (Section B.11)</td></tr>
<tr><td>__threadfence_system() (Section B.5)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Section B.6)</td></tr>
</table>

Tabla 2.2. Características soportadas por cada *Compute Capability* [1]

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				

Tabla 2.3. Especificaciones técnicas de cada *Compute Capability* [1]

Algunas de las características de cada versión ya han sido descritas en las correspondientes secciones a las que hacen mención, por lo que no se incluirán de nuevo en este apartado. (Véase apartado 2.2.3).

Capítulo 3

Objetivos y Metodología

3.1 Análisis de Objetivos

El principal objetivo de este proyecto es llevar a cabo la aceleración de aplicaciones científicas de distinta naturaleza, a través del uso de GPUs. Esto nos permitirá determinar qué rendimiento podemos esperar de dichos dispositivos en distintas condiciones, y además, servirá de base para el estudio de diferentes técnicas de optimización relacionadas con el entorno. De ellas, será posible evaluar la dificultad que conlleva su aplicación, y determinar si merece la pena el uso de las mismas en función de la ganancia conseguida.

Los resultados deberán observarse dentro del marco de aprendizaje de una persona inexperta en este modelo de programación, donde las actividades de estudio y adquisición de experiencia serán parte importante del tiempo de trabajo. Así, el tiempo obtenido en las distintas fases de optimización será variable, conforme vaya en aumento la destreza y la práctica conseguida.

No se pretende abordar un volumen de aplicaciones elevado, sino, más bien, tratar de estudiar, aplicar y valorar el mayor número de optimizaciones posibles sobre diferentes situaciones. Indirectamente, estaremos generando un *background* basado en la experiencia, que nos permitirá determinar en el futuro cuándo una optimización es recomendable que sea utilizada en una situación concreta, o cuándo esta debe ser descartada por su baja relación beneficio/esfuerzo.

Nos centraremos en optimizaciones aplicables al tipo de dispositivos que se encuentran instalados en el sistema que utilizaremos (véase apartado 3.3 y 3.4), pudiendo estas no funcionar correctamente, u obtener rendimiento diferente, si se ejecutan en dispositivos anteriores o posteriores (véase apartado 2.2.4).

3.2 Metodología

La metodología empleada para la aceleración de aplicaciones será la propia de este tipo de tareas. A continuación, se describen los diferentes pasos a seguir:

1. Se llevará a cabo un estudio del código en busca de posibles zonas con un potencial paralelismo de datos que pueda explotarse mediante CUDA, esto es, bucles de gran tamaño mayormente regulares, operaciones ya estudiadas en la bibliografía con las que se obtenga un buen rendimiento en el dispositivo (reducciones, ordenaciones…), uso de librerías que posean una implementación equivalente para GPUs (CUFFT, CUBLAS), etc.
2. En caso de que la aplicación sea lo suficientemente grande como para que existan diferentes fases del código, de las que se pudiera dudar si migrar o no al dispositivo, se realizará un estudio de las mismas con la herramienta *gprof* (apartado 3.5.5). Con ella, determinaremos cuál es el peso de esas fases sobre el tiempo total del programa, y si merece la pena emplear esfuerzo en su aceleración.
3. Procederemos con a la obtención de una primera versión lo más sencilla posible, que utilice la GPU para las zonas de código seleccionadas.
4. La versión será evaluada según lo descrito en el apartado 3.4.
5. Estudiaremos diferentes posibilidades de optimización haciendo uso de herramientas como *CUDA Visual Profiler* (apartado 3.5.1), *CUDA Occupancy Calculator* (apartado 3.5.2) o *gprof*.
6. Se seleccionará una de las optimizaciones y se aplicará individualmente. Basándonos en la experiencia o en determinadas dependencias, podrían aplicarse optimizaciones de manera conjunta.
7. La nueva versión obtenida será evaluada de igual forma que en el punto 4, comparando los resultados con las versiones anteriores para determinar si la optimización debe conservarse o eliminarse.
8. Se repetirán los pasos 5-7 hasta que ya no queden optimizaciones razonables para ser aplicadas, alcanzando así la versión final de la aplicación.
9. Se recogerá un *feedback* de la experiencia obtenida del que haremos uso sobre las siguientes aplicaciones, que nos permitirá ahorrar tiempo y esfuerzo.

3.3 Sistemas Utilizados

En este apartado, se describen las características más importantes de los sistemas sobre los que se ha trabajado y obtenido resultados de todas o alguna de las aplicaciones.

Cell

Arquitectura	**Blade QS20**
Procesadores	2 x Cell BE, 3.2 GHz
Memoria RAM	1 GB
SS.OO	Fedora 7
Kernel	Linux 2.6.22-5.fc7 SMP ppc64 GNU/Linux
GCC	4.1.2

Tabla 3.1. Características del Servidor Cell

Altix

Arquitectura	**SGI Altix 4700 (NUMA)**
Procesadores	128 x Dual-Core Intel Itanium Montecito, 1.6 GHz
Total Cores	256 (2x128)
Caché (por MP)	32 KB L1 (2x), 1.25 MB L2 (2x), 12 MB L3 (2x)
Memoria RAM	2.5 TB
SS.OO	SUSE Linux Enterprise Server 10 (IA64)
Kernel	Linux 2.6.16.60-0.42.5-default SMP ia64
GCC	4.1.2
Red de Interconexión	Custom SMP NUMA

Tabla 3.2. Características del Servidor Altix

Obelix

Procesadores	**2x Dual-Core AMD Opteron 2222 @ 3GHz**
Total Cores	4 (2x2)
Caché (por MP)	128 KB L1 (2x), 2 MB L2
Memoria RAM	8 GB
GPUs	2x NVIDIA GeForce 285
Memoria	1 GB
Compute Capability	1.3
Driver	Versión 3.0, 195.36.15
CUDA Toolkit y SDK	Versión 3.0
SS.OO	Linux Debian Squeeze
Kernel	Linux obelix 2.6.32-trunk-amd64 #1 SMP x86_64
GCC	4.3.4

Tabla 3.3. Características del Servidor Obelix

Praliné y Gelatine

Procesadores	**2x Intel Xeon E5420 @ 2.50GHz**
Total Cores	8 (2x4)
Caché (por MP)	256 KB L1 (2x), 12 MB L2
Memoria RAM	8 GB
GPUs	2x NVIDIA Tesla C870
Memoria	1.5 GB
Compute Capability	1.0
Driver	Versión 3.0, 195.36.15
CUDA Toolkit y SDK	Versión 3.0
SS.OO	Linux Debian Squeeze
Kernel	Linux 2.6.32-3-amd64 SMP x86_64
GCC	4.3.4

Tabla 3.4. Características de los Servidores Praliné y Gelatine

Red MPI

Servidores	Obelix, Praliné y Gelatine
Red de Interconexión	Gigabit Ethernet

Tabla 3.5. Características de la Red MPI

3.4 Línea Base, Medidas y Métricas

La medida utilizada para caracterizar y comparar el rendimiento de cada versión será el *speedup* calculado a partir de los tiempos de ejecución, en segundos. Se tomará como **línea base** para este cálculo, el tiempo de la versión serie de cada aplicación ejecutada sobre el servidor **Obelix**. Este también será utilizado para la ejecución de las versiones en CUDA, y solamente dispondremos del resto cuando sea necesario para versiones específicas que requieran múltiples nodos de ejecución. Así, las optimizaciones implementadas se ajustarán a la *Compute Capability* 1.3 que poseen los dispositivos del mismo.

Los tiempos de ejecución de cada versión serán calculados como la media aritmética de 100 ejecuciones para las aplicaciones EP y CG. En el caso de FTDock, se procederá de igual forma, reduciendo el número de repeticiones a cinco por sus elevadísimos tiempos de ejecución individual. Con motivo de la escasa varianza existente entre ejecución y ejecución, han sido descartadas aquellas instancias muy puntuales que no se ajustaban al resto.

Las medidas de las versiones en CUDA incluyen el 100% de las reservas de memoria del dispositivo y las transferencias de datos que tienen lugar entre este y el *host*. En aquellos casos en los que se ejecutan fragmentos de código fuera de tiempo para poner a punto las cachés, como en CG, la memoria reservada en el dispositivo será liberada y se procederá de nuevo con ella y las transferencias dentro del tiempo.

3.5 Herramientas

Para el desarrollo del proyecto, ha sido necesario contar con una serie de herramientas de diferente naturaleza, que van desde *profilers*, para obtener un perfil de la ejecución de las aplicaciones, hasta herramientas para depuración y verificación de accesos a la memoria de la GPU. También se han desarrollado algunas macros y scripts (ver Anexo I) que han resultado necesarios.

3.5.1 CUDA Visual Profiler

CUDA Visual Profiler es una herramienta gráfica que permite realizar un perfil de la ejecución de aquellas partes de código destinadas a la GPU. Es capaz de brindarnos gran cantidad de información, que nos será muy útil para saber cuáles son los recursos

del dispositivo que se están utilizando, de qué manera, detectar cuellos de botella, plantear estrategias de optimización, así como descubrir el por qué de algunos errores.

Como podemos ver en la Figura 3.1, los datos van desde los tiempos de ejecución y número de llamadas de cada *kernel*, hasta el número de *warps* serializados, la configuración de bloques y *threads* de cada *grid*, el número y tipo de accesos a la memoria global, cantidad de memoria local y compartida utilizada, tamaño de las transferencias de memoria, saltos divergentes, etc.

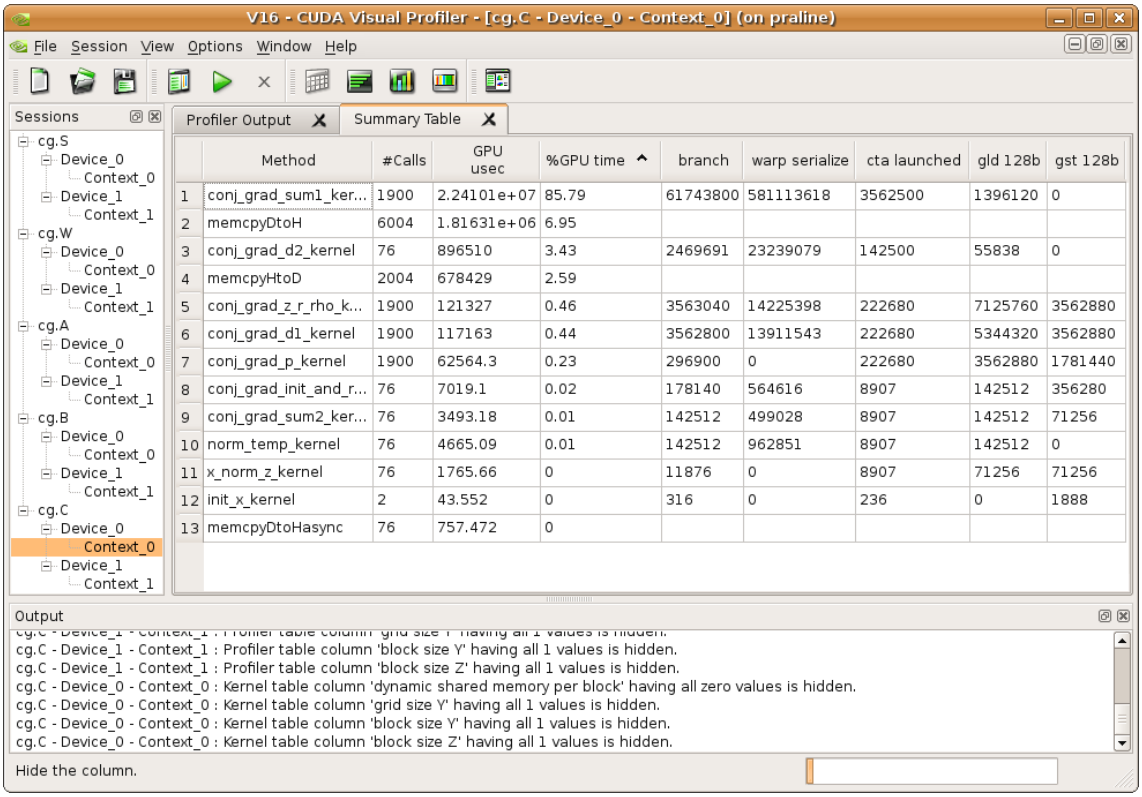


Figura 3.1. *CUDA Visual Profiler*

Sin embargo, la interpretación de estos datos debe hacerse con cautela, pues son obtenidos de solamente uno de los varios multiprocesadores que posee el dispositivo. Aparte de que no todos tienen por qué ejecutar exactamente el mismo código (saltos divergentes) es posible que el total de bloques de un *kernel* no quede repartido de manera equitativa entre ellos. Esta peculiaridad, es algo que debemos tener en cuenta si pretendemos hacer coincidir cierta información del perfil obtenido con el código de nuestra aplicación, porque los datos podrían no ser precisos sin explicación aparente. No obstante, en el peor de los casos, la herramienta nos servirá para vislumbrar que quizás existe la posibilidad de mejorar nuestro código, o que la modificación realizada sobre el mismo ha tenido sus consecuencias.

Además de mostrar las cifras en formato de tabla, *CUDA Visual Profiler* permite representar diferentes gráficas muy ilustrativas, relacionadas con el tiempo y el orden de ejecución de los diferentes *kernels* de la aplicación. También posee la capacidad de analizar los datos obtenidos y sugerir qué factores son los más perjudiciales para el

rendimiento, y por tanto, los que sería prioritario mejorar. No obstante, la utilidad de este último punto es más bien reducida cuando se posee algo de experiencia.

Una información que se echa muy en falta es la referente al código ejecutado en la CPU. Esto ocasiona que no puedan valorarse cuán influyentes son las partes de código que se ejecutan en el *host*, y compararlas con las que se ejecutan en el dispositivo, lo cual es algo imprescindible a la hora de decidir si una determinada parte del código debe migrarse a la GPU o no.

3.5.2 CUDA Occupancy Calculator

CUDA Occupancy Calculator es una hoja de Excel que nos ayuda a analizar la ocupación de multiprocesadores que nuestro *kernel*, lanzado al dispositivo, está realizando en función de los recursos que utiliza.

Just follow steps 1, 2, and 3 below! (or click here for help)	
1.) Select Compute Capability (click):	**1,3**
2.) Enter your resource usage:	
Threads Per Block	256
Registers Per Thread	20
Shared Memory Per Block (bytes)	8192
(Don't edit anything below this line)	
3.) GPU Occupancy Data is displayed here and in the graphs:	
Active Threads per Multiprocessor	**512**
Active Warps per Multiprocessor	**16**
Active Thread Blocks per Multiprocessor	**2**
Occupancy of each Multiprocessor	**50%**
Physical Limits for GPU:	**1,3**
Threads / Warp	32
Warps / Multiprocessor	32
Threads / Multiprocessor	1024
Thread Blocks / Multiprocessor	8
Total # of 32-bit registers / Multiprocessor	16384
Register allocation unit size	512
Shared Memory / Multiprocessor (bytes)	16384
Warp allocation granularity (for register allocation)	2
Allocation Per Thread Block	
Warps	8
Registers	5120
Shared Memory	8192
These data are used in computing the occupancy data in blue	
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Multiprocessor	4
Limited by Registers / Multiprocessor	3
Limited by Shared Memory / Multiprocessor	2
Thread Block Limit Per Multiprocessor highlighted	RED
CUDA Occupancy Calculator	
Version:	1,5
Copyright and License	

Figura 3.2. *CUDA Occupancy Calculator*

La herramienta nos informa de qué recursos están limitando que no se utilice el resto de multiprocesadores que quedan disponibles. Así, podremos valorar si obtenemos un mejor rendimiento reduciendo el uso de dicho recurso (en caso de ser posible) y aumentando la ocupación de multiprocesadores, o por el contrario, el uso actual del recurso proporciona mejor rendimiento, a pesar de que no se utilicen todas las unidades de cómputo disponibles.

Como podemos ver en el ejemplo de la Figura 3.2, la herramienta solamente necesita disponer del número de *threads* por bloque de nuestro *kernel* (256), del número de registros por *thread* (20) y del tamaño usado de memoria compartida por bloque (8192). Estos datos pueden obtenerse directamente con *CUDA Visual Profiler*, entre otras opciones. Como resultado, la hoja de cálculo nos muestra la ocupación que estamos haciendo de la tarjeta (50%), y el recurso que está limitando ese valor (la memoria compartida).

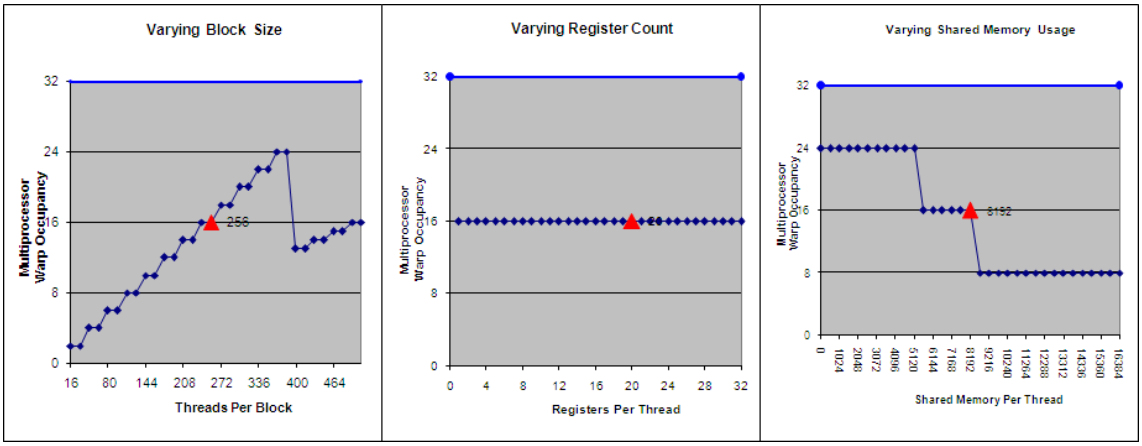


Figura 3.3. Gráficas de predicción de la ocupación

Las gráficas generadas por la utilidad (Figura 3.3) resultan muy interesantes para saber cómo va a evolucionar la ocupación de la GPU, en función de los tres parámetros que afectan a su valor. De esta manera, podremos planificar las modificaciones antes de llevarlas a cabo. En el ejemplo, según podemos observar, si reducimos el uso de memoria compartida podríamos alcanzar hasta un 75% de la ocupación (24). También podemos conseguir el mismo objetivo si aumentamos el número de *threads* por bloque hasta 384.

3.5.3 CUDA GDB

CUDA-GDB es una extensión del famoso *debugger GDB*, diseñada para poder depurar de manera completa aplicaciones CUDA, tanto a nivel de código en el *host* como de código en el dispositivo. La herramienta hereda la funcionalidad estándar que se utiliza para trabajar sobre códigos que se ejecutan en la CPU, y, además, añade nuevas funciones para facilitar la depuración de códigos en el dispositivo.

Esta extensión, básicamente, considera que la memoria y los *threads* del dispositivo pertenecen realmente al *host* y son tratados como tal, con la salvedad de que en ellos se establece el *warp* como granularidad de ejecución. Las nuevas funciones permiten explorar todo tipo de variables alojadas en las distintas memorias que ofrece el dispositivo, así como navegar por las coordenadas lógicas (*grid*, bloque y *thread*) y físicas (dispositivo, multiprocesadores, *warps* y *lanes* – identificador del *thread* dentro del *warp* –). Sin embargo, aún tendremos que esperar a nuevas versiones en las que se pueda trabajar con aplicaciones que utilicen más de una GPU, pues esta funcionalidad aún no está disponible.

Al igual que g*prof* y la versión original de *GDB*, es necesaria la instrumentación del código fuente, la cual se llevará a cabo compilando con las opciones -g -G.

Un aspecto negativo es que sólo podremos hacer uso de *CUDA-GDB* sobre dispositivos con *Compute Capability* superior a la 1.0 (consúltese apartado 2.2.4).

3.5.4 CUDA Memcheck

CUDA Memcheck es una herramienta sencilla de usar e imprescindible en el desarrollo de aplicaciones sobre las GPUs actuales. Debido a que no existe ningún tipo de mecanismo de protección en la memoria de los dispositivos, un puntero mal configurado puede producir fallos catastróficos en la tarjeta. El *driver* detectará estos fallos y tratará de corregirlos, muchas veces sin éxito, llegando, en el intento, incluso a bloquear el equipo por completo de manera muy habitual.

Anteponiendo `cuda-Memcheck` en la ejecución de nuestro programa, los accesos a direcciones de memoria global del dispositivo serán capturados y se comprobará que las posiciones a las que se intenta acceder son válidas. En caso de no serlo, se abortará la ejecución para no corromper el estado del dispositivo y/o el sistema. También obtendremos información sobre los accesos a memoria global no alineados. Tras la ejecución nos encontraremos un resumen similar al de la Figura 3.4.

```
========= CUDA-MEMCHECK
========= Invalid write of size 4
=========     at 0x00000350 in get_best_scores_kernel
=========     by thread 33 in block 89
========= Address 0x03c54b3c is out of bounds
=========
========= ERROR SUMMARY: 1 errors
```

Figura 3.4. Ejemplo de salida de *CUDA Memcheck*

3.5.5 Gprof

Gprof es un conocido *profiler* de Linux que se utiliza en el ámbito de las CPUs. Se caracteriza por proporcionar información relacionada con las diferentes funciones del programa o incluso a nivel de línea de código. Este lleva a cabo la instrumentación de la aplicación para obtener sus resultados, por lo que la recompilación con las opciones –g –pg es imprescindible.

Entre los datos que la herramienta puede ofrecernos, encontramos el tiempo de ejecución a nivel de función o línea de código (en segundos y en porcentaje con respecto al total) las veces que una función es invocada, el tiempo acumulado de una función y el resto de funciones a las que esta llama, etc.

Capítulo 4

Aceleración de Aplicaciones mediante GPUs

En este capítulo, veremos las transformaciones que se han realizado sobre una serie de aplicaciones, seleccionadas con el objetivo de conseguir su aceleración mediante el uso de GPUs de NVIDIA. En primer lugar, comenzaremos describiendo de manera genérica diferentes técnicas de optimización, para, posteriormente, ver su aplicación y el rendimiento obtenido sobre cada caso particular. Por último, analizaremos la evolución del rendimiento que se ha ido consiguiendo tras las diferentes fases de optimización, y también buscaremos un punto de referencia en otras arquitecturas ajenas al ámbito de las GPUs.

La selección de aplicaciones se ha realizado buscando la diversidad sobre criterios como la estructura, complejidad, comunicaciones, extensión de código, tipos de datos que se utilizan y posibilidades de optimización. Las elegidas han sido EP, por el uso de datos en punto flotante de doble precisión, su regularidad y bajo nivel de comunicaciones; CG, por usar el mismo tipo de datos que la anterior, su necesidad en cuanto a comunicaciones y sus estructuras de datos irregulares; y FTDock, por utilizar datos en simple precisión, su densidad computacional, hacer uso de transformadas de Fourier y su envergadura más consolidada como aplicación independiente con una finalidad específica.

Las dos primeras pertenecen al conjunto de *benchmarks* desarrollados por el departamento de supercomputación de NASA: *The NAS Parallel Benchmarks* (NPB) [35]. Estos *benchmarks* derivan de aplicaciones reales de simulación de dinámica de fluidos, que fueron diseñados "para proveer a la comunidad de desarrollo e investigación aeroespacial de Estados Unidos, supercomputadores de alto rendimiento capaces de simular un completo sistema de transporte aeroespacial en tan sólo varias horas" [22]. NPB está disponible en diferentes lenguajes y modelos de programación paralela, como OpenMP y MPI. Son usados, en su mayoría, para cuantificar el rendimiento de supercomputadores, utilizando como medida la aritmética en punto flotante de doble precisión. Esto resulta especialmente interesante para el proyecto, ya que la capacidad de realizar operaciones con datos de doble precisión ha sido recientemente incorporada a la arquitectura de las tarjetas gráficas de NVIDIA

(*Compute Capability* 1.3). Todos los *benchmarks* de este conjunto tienen predefinidos cinco tamaños del problema que resuelven, los cuales vienen representados, de menor a mayor, por las letras S, W, A, B, y C.

Por otro lado, FTDock es una aplicación científica que simula el acoplamiento de dos proteínas, haciendo uso de datos de simple precisión. Se emplea en la investigación de medicamentos contra el Cáncer y pertenece al *Biomolecular Modelling Laboratory* del Reino Unido [36]. Actualmente, es utilizada por el departamento *Life Science* de *Barcelona Supercomputing Center* [52]. Para el proyecto solamente hemos dispuesto de su versión C en serie.

4.1 Descripción de las Técnicas de Optimización

En este apartado, llevaremos a cabo la explicación de las diferentes técnicas de optimización que vamos a utilizar, para continuar con su aplicación específica sobre cada aplicación que así lo requiera.

4.1.1 Memoria Compartida

Para que una variable sea alojada en la memoria compartida del dispositivo, dentro del *kernel* debemos declararla precedida de la palabra clave `__shared__`. Con esto, conseguiremos que exista una copia privada de la variable en cada uno de los bloques de ejecución, que será compartida por todos los *threads* dentro de cada bloque. Si por el contrario, deseamos tener una variable privada para cada uno de los *threads* de un bloque, declararemos un array del estilo `__shared__ array[blockSize]`, donde `blockSize` se corresponde con una constante con el número de *threads* por bloque con el que se está ejecutando dicho *kernel*. Un ejemplo de este caso puede verse en la Figura 4.1.

```
__global__ void sm_example_kernel ()
{
    unsigned int lid = threadIdx.x;
    __shared__ double sx[blockSize];

    sx[lid] = 0.0;
    __syncthreads();
    sx[(lid+1)%blockDim.x] += lid;

    …
}
```

Figura 4.1. Ejemplo de uso de memoria compartida

Utilizaremos el identificador local al bloque de cada *thread*, `lid`, para que cada uno de ellos acceda a su variable correspondiente. Si es necesario que cada *thread* comparta su información con el resto de *threads* del mismo bloque, deberemos utilizar la función de sincronización `__syncthreads()`. Tras la llamada a esta función,

podemos tener la seguridad de que cada *thread* de un mismo bloque ha terminado de utilizar su variable compartida, en lo referente al código anterior a esta llamada, y por tanto, otro *thread* de dicho bloque puede pasar a leer o a escribir estas posiciones con total seguridad. Para que todo funcione correctamente, todos los *threads* de un bloque deben ejecutar la función anterior, por lo que debemos asegurarnos de que el flujo de control no da lugar a que una parte del bloque no la ejecute. Debemos destacar que esta función no permite llevar a cabo una sincronización global entre todos los bloques de ejecución, sino solamente a nivel interno de cada bloque. Para ese cometido existe la función `cudaThreadSynchronize()`, de la que ya hablamos en el Modelo de Ejecución.

Puesto que esta memoria es muy limitada, debemos cuidar no exceder el tamaño máximo de la misma, aunque es recomendable realizar un uso por bloque bastante inferior a este. En caso contrario, estaremos afectando a la ocupación de los multiprocesadores de la tarjeta con el excesivo uso por bloque de este recurso. Por este mismo motivo, debemos de saber que los parámetros que se le pasan a un *kernel* son almacenados en la memoria compartida de todos los bloques, conjuntamente con algunas variables por defecto, como las que describen la configuración del *grid* y de los bloques (`gridDim`, `blockIdx`, `blockDim` y `threadIdx`).

También es posible hacer uso de memoria compartida de manera dinámica, indicando el tamaño de esta cuando el *kernel* es invocado. Puesto que no ha sido requerido su uso, prescindiremos de dar detalles al respecto.

4.1.2 Algoritmo de Reducción Parcial en GPU

El algoritmo de reducción de un conjunto de valores, aplicando sobre ellos una operación determinada (suma, resta, máximo, mínimo, etc.) es algo que ha sido especialmente estudiado en la corta pero intensa historia de estos dispositivos.

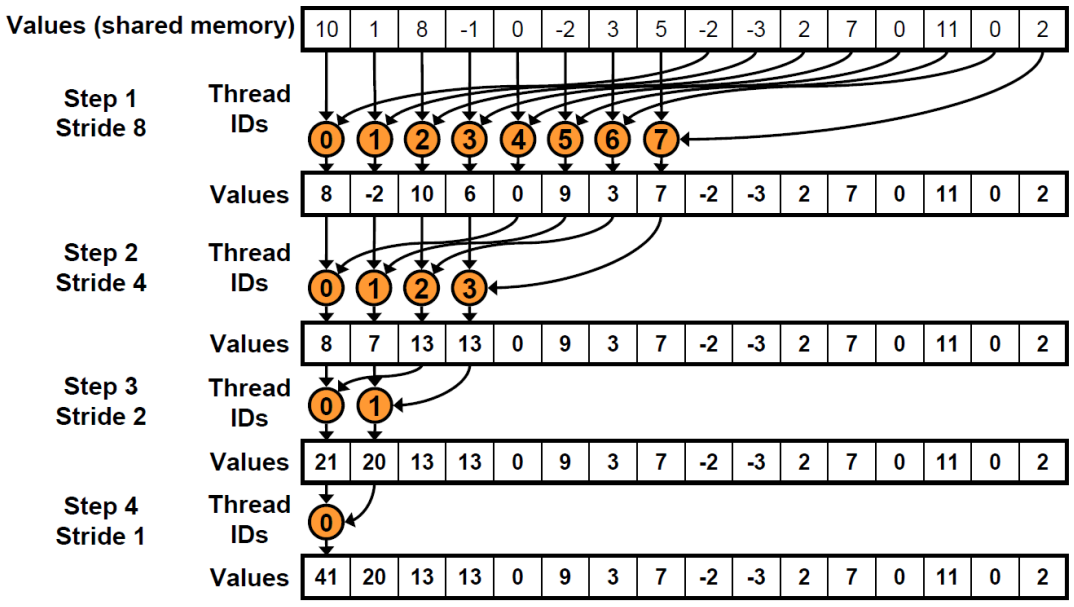


Figura 4.2. Esquema (por bloque) del algoritmo de reducción en la GPU [13]

Si nos referimos al SDK de NVIDIA y a la referencia [13], podemos encontrar varios algoritmos de reducción con rendimiento diferente. Tras llevar a cabo diversas pruebas, se seleccionó aquel con el que se obtenía el mejor resultado. Está basado en el uso de memoria compartida, y ha sido adaptado a cada caso de aplicación en este proyecto.

En la Figura 4.2, podemos ver un ilustrativo esquema del algoritmo. Este está optimizado para reducir cantidades de elementos que sean potencia de dos. Por tanto, siendo N el número de elementos a reducir, ya almacenados en memoria compartida, los primeros N/2 *threads* se encargarán de operar sobre sus homónimos de la segunda y primera mitad de N, depositando el resultado en esta última, y reduciendo, así, N elementos a N/2. Esta fase se repetirá hasta conseguir un solo elemento por bloque que se almacenará en memoria global. Podemos observar la implementación de lo descrito en la Figura 4.3.

```
// do reduction in shared mem
if (blockSize >= 512) {
  if (lid < 256) { sdata[lid] += sdata[lid + 256]; } __syncthreads();
}
if (blockSize >= 256) {
  if (lid < 128) { sdata[lid] += sdata[lid + 128]; } __syncthreads();
}
if (blockSize >= 128) {
  if (lid <  64) { sdata[lid] += sdata[lid +  64]; } __syncthreads();
}
if (lid < 32)
{
  if (blockSize>=64){sdata[lid]+=sdata[lid + 32];__syncthreads();}
  if (blockSize>=32){sdata[lid]+=sdata[lid + 16];__syncthreads();}
  if (blockSize>=16){sdata[lid]+=sdata[lid +  8];__syncthreads();}
  if (blockSize>= 8){sdata[lid]+=sdata[lid +  4];__syncthreads();}
  if (blockSize>= 4){sdata[lid]+=sdata[lid +  2];__syncthreads();}
  if (blockSize>= 2){sdata[lid]+=sdata[lid +  1];__syncthreads();}
}

// write result for this block to global mem
if (lid == 0) g_odata[blockIdx.x] = sdata[0];
```

Figura 4.3. Fragmento del algoritmo de reducción parcial en la GPU [7]

Una vez alcanzado este punto, existen dos opciones:

a) Realizar una sincronización global (`cudaThreadSynchronize()`) desde el *host* y lanzar otro *kernel* idéntico al anterior, que realice una última fase de reducción con una configuración de sólo un bloque.
b) Mover los datos al *host* y terminar la reducción allí.
c) Usar operaciones atómicas en el *kernel*.

Tras diversas pruebas, se obtuvo un mejor rendimiento utilizando la segunda de las opciones. No obstante, no tiene por qué ser siempre la mejor de las opciones en otro tipo de problemas.

Además, en algunas situaciones, el algoritmo de reducción ha sido modificado para que lleve a cabo más de una operación de reducción a la vez sobre diferentes datos, o incluso, varias reducciones en diferentes fases para, así, reutilizar la memoria compartida.

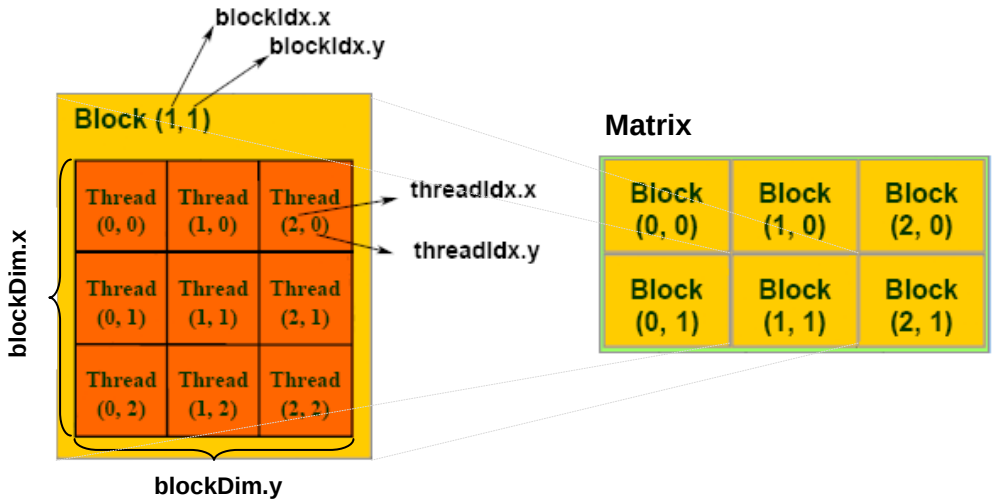
Si deseamos reducir un número de elementos muy elevado, podemos encontrarnos con que no disponemos de tanta memoria compartida, o con que no podemos tener un *thread* por cada dos elementos. Esta situación requerirá de una fase previa de reducción en memoria global, donde cada *thread* procesará tantos elementos como sea necesario para alcanzar un número de estos que cumpla las restricciones de las fases descritas anteriormente, y proseguir, entonces, con ellas. Lo mismo será aplicado cuando no se cuente con un número de elementos potencia de dos.

4.1.3 Mejorando el Coalecing mediante Estrategias de Mapeo de Threads

Cuando se va a trabajar con un vector de datos, es muy habitual que cada *thread* del dispositivo, individualmente, quede mapeado a uno o varios elementos del mismo. Para ello, basta con que cada *thread* acceda a la estructura en función de su identificador global o `gid`, como ya hemos visto. Esta estrategia puede extenderse también a matrices de varias dimensiones, donde CUDA ofrece la posibilidad de configurar bloques y *grids* conformados con *threads* y bloques multidimensionales, respectivamente, para así facilitar dicha tarea. En este caso, una de las formas con la que podremos acceder a la estructura, será empleando diferentes identificadores globales para cada una de las dimensiones de los *threads* y los bloques. Un ejemplo que ilustra esta idea sobre una matriz de dos dimensiones puede observarse en la Figura 4.4. En ella, los identificadores globales se corresponden con las variables `i` y `j`.

Si en lugar del mapeo de *threads* anterior, hubiéramos asignado un *thread* por fila de la matriz, los accesos a memoria serían no *coalesced*, pues entre el elemento de cada *thread* tendríamos una fila de distancia. Sin embargo, como muestra el ejemplo, con un mapeo multidimensional se llevarían a cabo accesos secuenciales y ordenados, satisfaciendo así las restricciones de coalescing.

Por otro lado, existen situaciones en las que se trabaja con estructuras de datos irregulares, donde este mapeo no es tan sencillo y directo. Para el caso concreto del proyecto, nos encontraremos con una matriz dispersa compactada, donde el número de elementos por fila es variable. Como hemos visto, la primera idea que podría surgir sería la de mapear un *thread* multidimensional por cada elemento de la fila. Sin embargo, nos encontramos ante una estructura con un número de elementos por fila variable y no resultaría nada sencillo. La opción de un *thread* por fila ya hemos visto que provocaría accesos no *coalesced*, penalizando en gran medida el rendimiento de la aplicación.




```
__global__ void kernel (int matrix[][])
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;

  matrix[i][j] = i+j;
}
```

Figura 4.4. Ejemplo de Mapeo de Threads sobre una Matriz 2D

Un alternativa interesante que puede solventar el problema es la de mapear un *warp* por fila de la matriz irregular [11]. De esta forma, los *threads* dentro de un mismo *warp* estarían trabajando con posiciones contiguas en memoria, cumpliendo así las restricciones para que los accesos sean *coalesced*.

Como punto negativo, esta configuración introduce un segundo problema de menor índole, aunque también importante. Se trata de los saltos divergentes que tendrían lugar cuando el número de elementos de cada fila no fuera múltiplo del tamaño del *warp*, ya que existirían *threads* en el mismo que tendrían elementos que procesar y otros que no, dando lugar a una bifurcación en el flujo de ejecución (serialización) que desencadenaría cierta, aunque menor, penalización (saltos divergentes).

La estrategia de mapeo de *threads* no es la única que puede afectar a que los accesos sean o no *coalesced*. Sin embargo, no han sido necesarias otras técnicas para el proyecto, por lo que se prescinde de su descripción.

4.1.4 Saltos Divergentes y Predicación de Saltos

La ausencia de hardware de predicción de saltos en las GPUs provoca que los saltos, en general, y los divergentes, en particular, puedan resultar catastróficos en la ejecución de código en el dispositivo, llegando incluso a neutralizar en gran medida el paralelismo.

Los saltos divergentes tienen lugar cuando el flujo de ejecución, dentro del mismo *warp*, se bifurca en más de un camino. En tal caso, la ejecución del *warp* se serializará en función del número de caminos alternativos, pudiendo darse, como máximo, uno diferente por cada *thread*. Por este motivo debemos tener cuidado con

aquellas instrucciones de control que pueden alterar al flujo de ejecución (*for*, *if*, *if-else*, *switch*, etc.), para que si lo hacen, lo hagan en conjuntos del tamaño de un *warp*. Por desgracia, muchas veces esto es imposible, y debemos aceptar la penalización que estos saltos provocan o valorar la opción de ejecutar ese fragmento de código en el *host.*

Existen técnicas que permiten eliminar ciertos saltos divergentes, como cambiar la estrategia del algoritmo, cambiar el mapeo de *threads* sobre los datos, o intentar eliminar saltos condicionales mediante *bithacks* – aplicando operaciones a nivel de bits (lógicas) –.

El compilador de CUDA también está dotado de un mecanismo que permite paliar el efecto de los saltos en general, e incluso eliminar los divergentes en algunas situaciones. Es denominado *branch predication* y permite que algunos de estos saltos sean compilados de manera especial, para reducir sus efectos en el tiempo de ejecución. Esta técnica será aplicada solamente cuando el compilador pueda determinar qué *threads* son aquellos que divergirán en un salto, y el cuerpo de éste no supere cierto umbral de longitud. Por este motivo, debemos fomentar el uso de constantes cuando sea posible, por ejemplo mediante *templates* o plantillas de C/C++ a la hora de invocar a un *kernel*. El compilador también pone a nuestra disposición la directiva `#pragma unroll`, que nos da la posibilidad de intervenir en el desenrollado (o no) de los bucles que se escapan de la técnica anterior.

4.1.5 Maximizando la ocupación de la GPU

Los dispositivos actuales cuentan con una serie de recursos limitados dentro de cada multiprocesador, que son compartidos por los distintos *threads* y bloques que se encuentran en ejecución. Principalmente, nos referimos a los registros y la memoria compartida. Al ser compartidos, cada bloque tiene la posibilidad de acceder a cualquiera de estos recursos, pudiendo, perfectamente, acapararlos por completo para su uso privado. En esta situación, ningún otro bloque podría pasar a ejecutarse en paralelo en un mismo multiprocesador, mientras ya exista un bloque activo haciendo uso de ese nivel de recursos. Si suponemos que el número de *threads* por bloque es de 128 y cada multiprocesador puede mantener activos hasta 1024 *threads*, estamos desperdiciando el 87,5% de la capacidad de cómputo de la GPU.

Buscando hacer un mejor aprovechamiento, debemos reducir el uso de registros por *thread*, memoria compartida por bloque, o *threads* por bloque, para que sea posible mantener varios bloques en ejecución al mismo tiempo, dentro del mismo multiprocesador. Mientras que el uso de memoria compartida por bloque o el número de *threads* por bloque pueden controlarse modificando el código fuente, la limitación de los registros por *thread*, si las operaciones de nuestro código son inamovibles, sólo puede realizarse en tiempo de compilación, utilizando el parámetro `–maxrregcount num_registros`. Si se establece un límite inferior al inicialmente necesario, se

llevarán a cabo operaciones de *spilling* entre los registros y la memoria local, como ya hemos descrito anteriormente.

Sin embargo, una mejora de la ocupación de la tarjeta no tiene por qué traducirse en un aumento del rendimiento. El hecho de utilizar ingentes cantidades de memoria compartida y un elevado número de registros pueden traer unas consecuencias en cuanto a rendimiento que perderíamos al reducir dicho volumen, y que no se verían compensadas por el aumento de bloques ejecutándose en paralelo. No obstante, siempre se recomienda que el porcentaje de ocupación de la tarjeta no sea excesivamente bajo, para que puedan ocultarse ciertos tiempos de latencia relacionados con conflictos en los bancos de registros y otros menesteres de bajo nivel.

Sin duda, la mejor y única regla válida al respecto es que debemos probar las distintas situaciones posibles, pues el rendimiento no puede derivarse de manera genérica en función de los parámetros mencionados.

Para ayudarnos a calcular y a realizar estimaciones sobre la ocupación utilizaremos la herramienta *Cuda Occupancy Calculator* (3.5.2).

4.1.6 Memoria Pinned y Zero Copy

La memoria *pinned*, es un tipo de memoria del *host* que se marca como no paginable. Esto quiere decir que no puede ser paginada a disco. Al dotarla de esta característica, las transferencias que se llevan acabo entre el *host* y el dispositivo se aceleran en gran medida, debido a que ciertas comprobaciones son desactivadas al poder asegurar que los datos no serán enviados a disco. Por tanto, en aquellas situaciones en las que se realicen transferencias de gran tamaño, será recomendable utilizar este tipo de memoria, siempre que no se supere el máximo permitido en el sistema.

```
//Host Memory Allocation
cudaHostAlloc((void **)&sx_h, GRID_SIZE * sizeof(double),
              cudaHostAllocDefault);
//Device Memory Allocation
cudaMalloc((void **)&sx_d, GRID_SIZE * sizeof(double));

//Host To Devices Transferences
cudaMemcpy(sx_d, sx_h, GRID_SIZE * sizeof(double),
           cudaMemcpyHostToDevice));

//Kernel Execution
kernel<<<grid, block>>>(sx_d);

//Device To Host Transferences
cudaMemcpy(sx_h, sx_d, GRID_SIZE * sizeof(double),
           cudaMemcpyDeviceToHost));

//Free Device Memory
cudaFree(sx_d);
//Free Host Memory
cudaFreeHost(sx_h);
```

Figura 4.5. Ejemplo de uso de memoria no paginable (Código *host*)

Para su uso, la reserva y liberación de memoria no paginable se llevará a cabo mediante la función `cudaAllocHost()`, con el parámetro `cudaHostAllocDefault`, y `cudaFreeHost()` respectivamente, en lugar de los habituales `malloc()` y `free()` a los que estamos acostumbrados. Un ejemplo puede verse en la Figura 4.5.

Además de no paginable, esta función permite hacer reservas de memoria con cierto tipo de características. Si en lugar de `cudaHostAllocDefault` utilizamos el parámetro `cudaHostAllocWriteCombined`, el área de memoria reservada nos permitirá optimizar aún más las transferencias, al eliminarse también el paso de los datos por los niveles de caché de la CPU (memoria no cacheable), y con ello, todo el mecanismo de mantenimiento de la coherencia. Como es de imaginar, esta opción sólo proporcionará buenos resultados cuando el *host* se limite exclusivamente a leer información de dicha región, pues las escrituras resultarían tremendamente lentas.

```
//Host Allocation
cudaHostAlloc((void **)&sx_h, GRID_SIZE * sizeof(double),
                cudaHostAllocMapped);
//Device Allocation -> Not Necessary
cudaMalloc((void **)&sx_d, GRID_SIZE * sizeof(double));

//Get Device Pointer
cudaHostGetDevicePointer((void **)&sx_d, (void *)sx_h, 0);

//Copy Data from Host to Device → Not Necessary
cudaMemcpy(sx_d, sx_h, GRID_SIZE * sizeof(double),
            cudaMemcpyHostToDevice));

kernel<<<grid, block>>>(sx_d);

//Copy Data from Device to Host → Not Necessary
cudaMemcpy(sx_h, sx_d, GRID_SIZE * sizeof(double),
            cudaMemcpyDeviceToHost));
```

Figura 4.6. Ejemplo de uso de la técnica *Zero Copy* (Código *host*)

Otra posibilidad es el uso del parámetro `cudaHostAllocMapped()` (Figura 4.6). Este permite mapear a un área de la memoria del dispositivo una zona de memoria no paginable alojada en el *host.* Así, se lleva a cabo la apertura de una ventana en la memoria del dispositivo, que es compartida automáticamente con la CPU, evitando las transferencias de memoria explícitas. Con ello, podremos conseguir que se solapen partes de la ejecución de un *kernel* con las transferencias de datos, aunque dependerá de la complejidad del código a ejecutar. A esta técnica se la denomina *Zero Copy*, y la Figura 4.7 muestra lo que podríamos conseguir con respecto a las transferencias de memoria tradicionales.

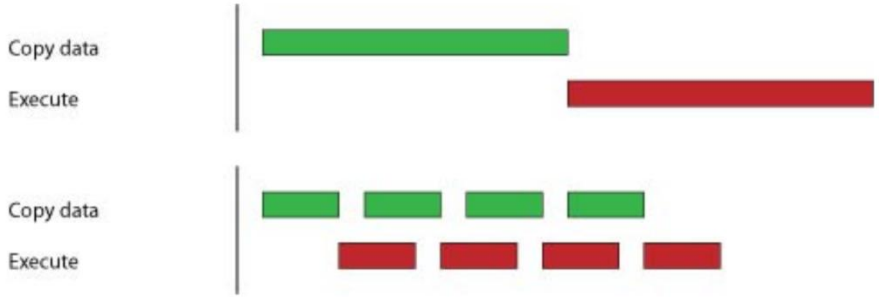


Figura 4.7. Transferencias estándar (arriba) vs. *Zero Copy* (abajo) [3]

4.1.7 Memoria Constante

Como ya hemos visto en el apartado 2.2.3, la memoria constante es una memoria especialmente limitada en cuanto a tamaño y forma de acceder a ella para obtener un buen rendimiento. Para usarla tendremos que utilizar la palabra clave `__constant__` delante del tipo y nombre de la variable. Estas declaraciones deben realizarse en el espacio de direcciones global del *host*, lo que dará lugar a que, con una sola declaración, se cree una copia privada sobre cada dispositivo que haga uso de la misma. Normalmente, estas declaraciones se realizan en el fichero donde se encuentra el código fuente de los *kernels* para que dicha variable sea visible desde estos (Figura 4.8).

```
__constant__ double * x_c;

__global__ void init_x_kernel(){

    unsigned int gid = (blockIdx.x * gridDim.x + threadIdx.x);

    if (gid < NA+1){
        x_c[gid] = 1.0;
    }
}
```

Figura 4.8. Ejemplo de uso de Memoria Constante. (Código *device*)

Además, se deben copiar los datos a la mencionada memoria antes de utilizaros en el código del *kernel*. Dicha operación tendrá lugar en el código del *host*, y cualquier modificación sobre los datos después de la copia no se verá reflejada en la memoria constante, puesto que no existe coherencia. La Figura 4.9 ilustra como se lleva a cabo la copia descrita.

```
  cudaMemcpyToSymbol(x_c, &x_d, sizeof(double *));

  init_x_kernel<<<grid, block>>>();
```

Figura 4.9. Ejemplo de uso de Memoria Constante. (Código *host*)

Como vemos en el ejemplo, el uso de esta memoria es aconsejable para el paso de parámetros a los *kernels* cuando estos son muy abundantes y se necesita liberar

memoria compartida. Para este caso, el acceso a los mismos será eficiente debido a que todos los *threads* leen la misma posición de memoria, satisfaciendo así su patrón de acceso óptimo.

4.1.8 Memoria de Texturas

La memoria de texturas es, quizás, la más compleja de utilizar, ya que, como hemos visto en el apartado 2.2.3, su uso nativo tiene relación con las texturas de los gráficos y está adaptada a las características de los mismos.

En primer paso que debemos dar para hacer uso de esta memoria es declarar variables globales de tipo `__texture__`. Este tipo es un *wrapper* de una estructura, y requiere de la caracterización de una serie de factores a través de una plantilla. En concreto, será necesario indicar el tipo de datos que contendrá la textura (`int` o `float` entre otros), la dimensión de la misma (1, 2 ó 3) y el modo de lectura (`cudaReadModeNormalizedFloat`, que permite que ciertos tipos de datos sean mapeados al rango [-1.0, 1.0] automáticamente, o `cudaReadModeElementType`, para lectura estándar), tal y como podemos ver en la Figura 4.10.

```
texture<int, 1, cudaReadModeElementType> pTextRef;

__global__ void kernel(int * sum){

    sum[gid] = 4 * tex1Dfetch(pTextRef, gid),
}
```

Figura 4.10. Ejemplo de uso de Memoria de Texturas. (Código *device*)

Tras su declaración, debemos de inicializar algunos valores de la textura en el código del *host*, antes de que esta sea utilizada. Podemos ver un ejemplo en la Figura 4.11. El primero de ellos hace referencia a si accederemos a la textura mediante coordenadas normalizadas en el rango [0,1] o lo haremos mediante índices sin normalizar. El segundo se trata de un array de tres dimensiones donde se nos permite especificar cómo serán tratados los accesos fuera del rango de la textura. Las opciones disponibles son `cudaAddressModeClamp`, que devolverá el valor del extremo más próximo del array, o `cudaAddressModeWrap`, que aplicará la operación de módulo al índice de acceso, para que si se sale del rango del array sea mapeado a elementos válidos. Por último, debe especificarse el tipo de filtro aplicado para devolvernos el valor de una posición de la textura. La opción `cudaFilterModePoint` devuelve el valor de la posición indicada por los índices sin más, y `cudaFilterModeLinear` realiza la "interpolación" constituida por los valores que se encuentren junto a esa coordenada.

```
cudaChannelFormatDesc pChannel = cudaCreateChannelDesc<int>();

pTextRef.normalized     = 0;
pTextRef.addressMode[0] = cudaAddressModeClamp;
pTextRef.filterMode     = cudaFilterModePoint;

cudaMalloc((void **)&p_d, num_threads_gpu * sizeof(int));
cudaMemcpy(rowstr_d, rowstr, num_threads_gpu * sizeof(int),
           cudaMemcpyHostToDevice));

cudaBindTexture (NULL, pTextRef, (void *)p_d, pChannel,
                 num_threads_gpu * sizeof(int));

kernel<<<grid, block>>>(sum);
```

Figura 4.11. Ejemplo de uso de Memoria de Texturas. (Código *host*)

Antes de pasar al uso de la textura en el código del dispositivo, debemos asociar los datos de memoria global a la misma. Para ello, utilizaremos la función `cudaBindTexture`, que requiere como parámetros más importantes la referencia a la propia textura, el puntero hacia la región de memoria global que queremos asociar, el tamaño de los datos, y un canal de comunicación que debemos haber creado previamente, mediante la función `cudaCreateChannelDesc()`.

Si los datos de memoria global son modificados tras la asociación con la textura, habrá que volver a realizar dicha asociación para que se actualicen en esta última.

Por último, para leer los valores a través de la textura desde el dispositivo, haremos uso de la función `tex1Dfetch()`, indicándole la referencia de la textura y el índice de la posición que queremos leer.

Otras funciones distintas serán necesarias para asociar y acceder a texturas de varias dimensiones.

Puesto que el tipo de datos `double` no es soportado en este tipo de memoria, deberemos utilizar texturas de enteros y almacenar los `doubles` descomponiéndolos en dos fragmentos de 32 bits. Para esto, CUDA pone a nuestra disposición las funciones `__hiloint2double()`, `__double2hiint()` y `__double2loint()`.

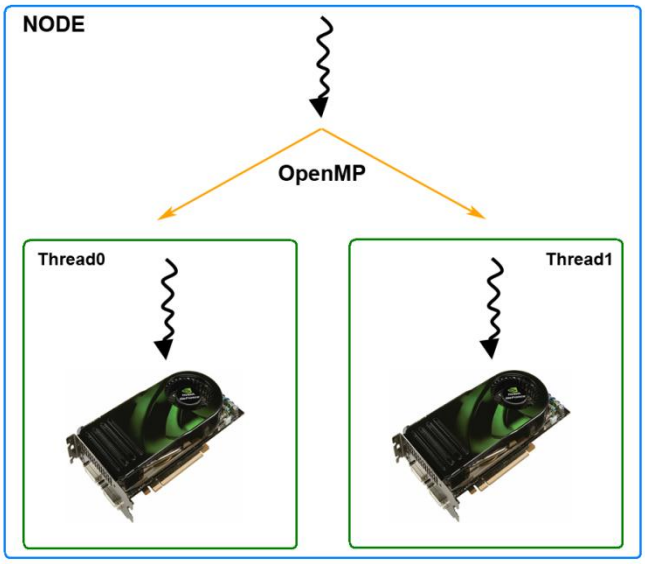
Como puede percibirse, el uso de texturas puede resultar una tarea laboriosa, en la que tendremos que lidiar con atributos poco habituales en la computación de propósito general.

4.1.9 Enfoque Multi-GPU (CUDA + OpenMP)

Actualmente es muy normal que encontremos sistemas con más de una GPU dedicada al cómputo de propósito general. Sin embargo, CUDA no proporciona un mecanismo automatizado y sencillo para hacer uso de varias de ellas en el mismo programa. La principal limitación que el modelo impone en este sentido es que un *thread* de *host* solamente puede estar asociado a una GPU al mismo tiempo.

No conocemos realmente la necesidad de dicha restricción, pero como consecuencia, nos veremos obligados a crear al menos tantos *threads* como dispositivos queramos utilizar. Así, será necesario el uso de Pthreads [47] , OpenMP [48], MPI [49]o cualquier otro modelo que no interfiera en el funcionamiento de CUDA. En lo referente a este proyecto haremos uso de OpenMP para este cometido.

La Figura 4.12 muestra uno de los esquemas posibles para repartir el trabajo entre dos GPUs. En este caso, tenemos un solo *kernel* que procesa datos que son independientes entre sí, y que ha de ser modificado con respecto al original, para que sea capaz de recibir los límites de su procesamiento. Dentro de la región paralela, cada *thread* OpenMP será asociado a su correspondiente dispositivo (`cudaSetDevice()`) y calculará la parte de trabajo que le toca (`my_workload()`) en función del total de GPUs existentes.




```
#pragma omp parallel num_threads(num_gpus)if(num_gpus>1)
{
  int my_device, my_chunk, from_workload, to_workload;

  cudaSetDevice(omp_get_thread_num());
  cudaGetDevice(&my_device);

  my_workload(total_workload, num_gpus, &my_chunk,
              &from_workload, &to_workload);

  kernel<<<grid,block>>>(from_workload, to_workload, my_chunk);
}
```

Figura 4.12. Esquema de cómputo simple Multi-GPU. Mismo nodo

Otro de los posibles esquemas puede contemplarse en la Figura 4.13. Al igual que el anterior cada *thread* se asocia a su dispositivo y calcula su trabajo a realizar, pero en este caso el *kernel* no requiere de modificación alguna con respecto a su versión con una sola GPU, ya que el control del trabajo se lleva a cabo en el código del *host* (bucle *for* de iteraciones interdependientes).

```
#pragma omp parallel num_threads(num_gpus) if(num_gpus>1)
{
  int my_device, my_chunk, from_workload, to_workload;

  cudaSetDevice(omp_get_thread_num());
  cudaGetDevice(&my_device);

  my_workload(total_workload, num_gpus, &my_chunk,
              &from_workload, &to_workload);

  for (i = from_workload; i < to_workload; i++){
    kernel<<<grid,block>>>(i);
  }
}
```

Figura 4.13. Esquema de cómputo Multi-GPU. *Kernel* en bucle. Mismo nodo

Hasta ahora hemos supuesto que no era necesaria ningún tipo de comunicación entre los diferentes dispositivos. Sin embargo, esto no será así en la mayoría de las ocasiones y seremos nosotros los encargados de desarrollar manualmente dicho esquema de comunicación. En la Figura 4.14 mostramos un esquema que implementa una comunicación entre GPUs equivalente al esquema MPI *all to all* (cada proceso envía sus datos al resto y recibe los datos enviados del resto). Como vemos, cada dispositivo calcula unos datos en `Kernel1`, copia el resultado al *host*, se sincronizan todos los *threads* del *host* para asegurar que todas las GPUs hayan llegado hasta ese punto, y cada una de ellas vuelve a copiar los datos del resto a su área de memoria global para proseguir con el cómputo del `Kernel2`.

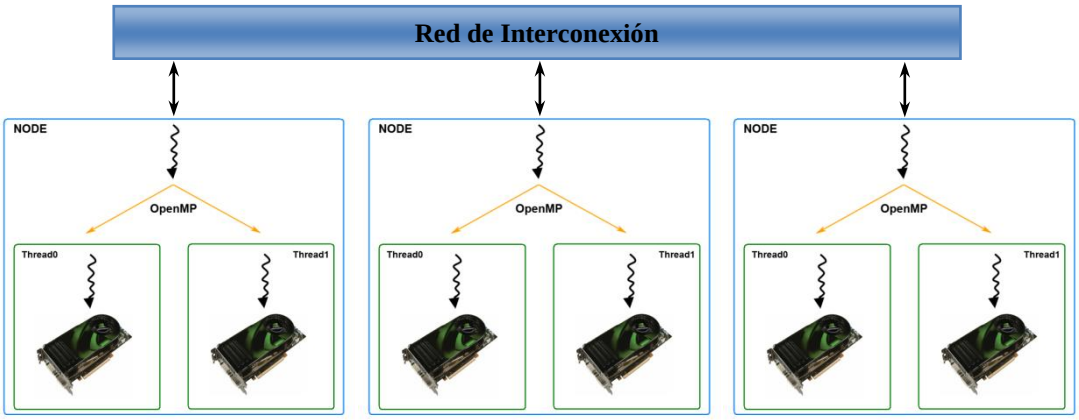
```
#pragma omp parallel num_threads(num_gpus) if(num_gpus>1)
{
  int my_device, my_chunk, from_workload, to_workload;

  cudaSetDevice(omp_get_thread_num());
  cudaGetDevice(&my_device);

  my_workload(total_workload, num_gpus, &my_chunk,
              &from_workload, &to_workload);

  Kernel1<<<grid,block>>>(from_workload, to_workload, my_chunk);

  //Each GPU copies data to share to global space host
  cudaMemcpy(cudaMemcpyDeviceToHost);

  #pragma omp barrier          //GPUs Sync.

  //Each GPU copies shared data from other GPUs,
  //from host global space memory to its memory.
  cudaMemcpy(cudaMemcpyHostToDevice);

  Kernel2<<<grid,block>>>(from_workload, to_workload, my_chunk);
}
```

Figura 4.14. Esquema de cómputo Multi-GPU.
Mismo nodo. Comunicación All2All

Otros muchos esquemas de comunicación son posibles, como los conocidos Gather, Scatter, Broadcast, etc. del estándar MPI. Todos ellos requieren de una implementación manual.

4.1.10 Enfoque Multi-GPU Multi-Nodo (CUDA + OpenMP + MPI)

Si queremos ir un paso más allá, podemos encontrarnos con un sistema conformado por distintos nodos como los del apartado anterior, que estén comunicados por red y pretendan colaborar en el mismo fin computacional. En este caso se hace imprescindible el uso de un modelo de paso de mensajes como MPI, para distribuir, sincronizar y comunicar a los diferentes procesos que se lanzarán en total, uno por cada nodo del sistema. Si cada uno de dichos nodos posee más de una GPU, además procederemos de similar manera a la descrita en el apartado anterior.




```
MPI_Init(&argc, &argv);
MPI_Comm_rank( MPI_COMM_WORLD, &my_node_id );
MPI_Comm_size( MPI_COMM_WORLD, &num_nodes );

my_node_workload(total_workload, num_nodes, &my_node_chunk,
                 &from_node_workload, &to_node_workload);

#pragma omp parallel num_threads(num_node_gpus) if(num_node_gpus>1)
{
  int my_device, my_chunk, from_workload, to_workload;

  cudaSetDevice(omp_get_thread_num());
  cudaGetDevice(&my_device);

  my_gpu_workload(my_node_chunk, from_node_workload,
                  to_node_workload, num_node_gpus, &my_chunk,
                  &from_workload, &to_workload);

  Kernel<<<grid,block>>>(from_workload, to_workload);

  gpu_data_to_MPI_global_node_space();
}

MPI_Gather();
MPI_Finalize();
```

Figura 4.15. Esquema de cómputo independiente Multi-GPU. Multi-Nodo

Uno de los posibles esquemas de cómputo puede consultarse en la Figura 4.15. Se comienza por las operaciones típicas de un proceso MPI: inicialización y obtención de la información de identificación y número de procesos. A continuación se lleva a cabo un primer reparto de trabajo entre los distintos procesos que han sido lanzados (`my_node_workload()`), y posteriormente, el resultado del reparto anterior vuelve a ser dividido entre los dispositivos del nodo local de igual manera que en el apartado anterior. Tras el cómputo del trabajo en cada GPU, los datos deben ser devueltos a la memoria global del proceso (*host*) para que sea accesible por la llamada MPI que enviará los datos a su destino correspondiente. Como ejemplo, en este caso se utiliza la llamada `MPI_Gather()`. De igual forma, muchos otros esquemas de comunicación pueden ser implementados.

4.2 Embarrassingly Parallel (NAS Parallel Benchmarks)

EP es el primer *benchmark* de NPB. Calcula masivamente pares de números pseudo-aleatorios Gaussianos (desviaciones) siguiendo un esquema determinado, y los tabula en anillos cuadrados sucesivos.

Esta operación se caracteriza por una densidad computacional relativamente elevada, por encima de las necesidades de memoria. Se suele emplear en simulaciones con algoritmos Monte Carlo – algoritmos que utilizan muestreo aleatorio para computar sus resultados –, tan utilizados en dinámica de fluidos. Debe su nombre al tipo de paralelismo que lo caracteriza, *Embarrassing Parallelism*, donde las comunicaciones en las zonas paralelas son prácticamente nulas. Debido a esta cualidad, este *benchmark* sirve para medir el rendimiento pico de operaciones en punto flotante de doble precisión de un sistema.

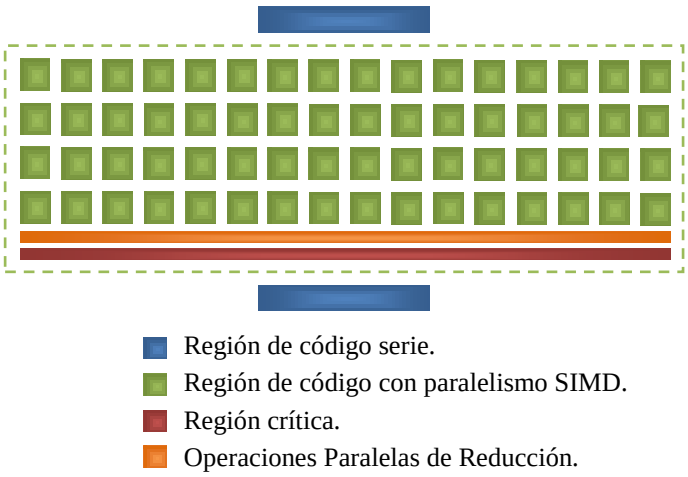


Figura 4.16. Estructura del *benchmark* EP

La Figura 4.16 muestra las diferentes fases del *benchmark* con la finalidad de ilustrar las distintas partes del código que podemos encontrar en cuanto a paralelismo. No se pretende relacionar el tamaño de las mismas con su influencia en el tiempo de

ejecución, aunque sí destacar la existencia de la región predominante con paralelismo SIMD, que finaliza con varias operaciones de reducción y una región crítica. Esta última se corresponde también con operaciones de reducción mapeadas a distintas posiciones de un array, que OpenMP no permite hacer de manera automática.

Con respecto al tiempo de ejecución debemos resaltar que la región paralela supone un porcentaje superior al 99% del total (medida experimental), además de que la aplicación, por defecto, solamente mide el tiempo de esta zona. Nos basaremos en esta medida para calcular el *speedup*.

La Figura 4.17 mapea las regiones anteriores sobre las líneas de código de EP más importantes. Podemos ver como por cada iteración del bucle principal se genera un vector de números aleatorios `x` (`vranlc()`), y con estos se llevan a cabo una serie de operaciones dentro del segundo bucle. La reducción se aplica sobre dos variables `sx` y `sy`, y la región crítica corresponde a realizar una operación de reducción sobre cada una de las posiciones de un array, del que cada *thread* posee una copia privada. El tamaño del problema viene determinado por la variable `np`, que indica el número de veces que iteraremos sobre el bucle principal. Este número será de 2^{15}, 2^{16}, 2^{19}, 2^{21} y 2^{23} para S, W, A, B y C respectivamente.

```
#pragma omp parallel copyin(x)
{
  double qq[NQ]; /* Private thread copy of q[0:NQ-1] */
  ...
  #pragma omp for reduction(+:sx,sy) schedule(static)
  for (k = 1; k <= np; k++) {
  ...
    vranlc(2*NK, &t1, A, x-1);

    for ( i = 0; i < NK; i++) {
      x1 = 2.0 * x[2*i] - 1.0;
      x2 = 2.0 * x[2*i+1] - 1.0;
      t1 = pow2(x1) + pow2(x2);
      if (t1 <= 1.0) {
        t2 = sqrt(-2.0 * log(t1) / t1);
        t3 = (x1 * t2);
        t4 = (x2 * t2);
        l = max(fabs(t3), fabs(t4));
        qq[l] += 1.0;
        sx = sx + t3;
        sy = sy + t4;
      }
    }
  }
  #pragma omp critical
  {
    for (i = 0; i <= NQ - 1; i++) q[i] += qq[i];
  }
}
```

Figura 4.17. EP: Resumen de código base para la CPU

El *speedup* conseguido por las versiones paralelas de 4 y 8 *threads* (Tabla 4.1) demuestra que este problema escala perfectamente en función del número de *cores* del

sistema. La versión sobre Opterons (Obelix) alcanza prácticamente las 4X, y la versión sobre Xeons (Praliné) supera las 8X, lo que pone de manifiesto una ligera superioridad entre el multiprocesador Xeon y el Opteron a nivel de cada *core* en particular.

	S	W	A	B	C
Opteron Serie	1,00 X	1,00 X	1,00 X	1,00 X	1,00 X
Opteron 4 Cores	3,96 X	3,97 X	3,97 X	3,94 X	4,03 X
Xeon 8 Cores	8,62 X	8,73 X	8,79 X	8,81 X	8,97 X

Tabla 4.1. EP *Speedups*: CPUs *Multicores*

4.2.1 Versión Inicial en CUDA

Nuestra primera versión en CUDA pretende conseguir una primera aproximación que haga uso de la GPU para la parte paralela del algoritmo. Para ello se implementará un *kernel* que englobe todos los cálculos realizados dentro del bucle más externo, mapeando cada una de las iteraciones del mismo a un *thread*. Como este número de iteraciones depende de la variable `np`, la configuración de *threads* y bloques del *kernel* también lo harán.

```
int block_size = 256;
dim3 grid(np/block_size);
dim3 block(block_size);
int nThreads = block.x * grid.x;
...
double * sx_h = (double *) malloc(nThreads*sizeof(double));
double * sy_h = (double *) malloc(nThreads*sizeof(double));
...
cudaMalloc((void **)&sx_d, nThreads * sizeof(double));
cudaMalloc((void **)&sy_d, nThreads * sizeof(double));
cudaMalloc((void **)&q_d, NQ * sizeof(unsigned int));
cudaMemcpy(q_d,q_h,NQ*sizeof(unsignedint),cudaMemcpyHostToDevice);

ep_kernel_double<<<grid, block>>>(sx_d,sy_d,k_offset,an,q_d);

cudaMemcpy(sx_h, sx_d, nThreads * sizeof(double),
           cudaMemcpyDeviceToHost);
cudaMemcpy(sy_h, sy_d, nThreads * sizeof(double),
           cudaMemcpyDeviceToHost);
cudaMemcpy(q_h, q_d, NQ * sizeof(unsigned int),
           cudaMemcpyDeviceToHost);
...
for (i = 1; i < nThreads ; i++) {
  sx_h[0] += sx_h[i];
  sy_h[0] += sy_h[i];
}
```

Figura 4.18. EP *host*: CUDA V1

Se procederá fijando un número de *threads* por bloque, 256 en este caso, para calcular con él el número de bloques que necesitamos para alcanzar tantos *threads* como

iteraciones. Según esta estrategia, no podremos fijar un número de *threads* por bloque inferior al establecido, ya que si lo hacemos, el número de bloques sobrepasaría el máximo que es capaz de manejar nuestro dispositivo.

Tras la reserva de datos y transferencias necesarias, se continuará con la ejecución de dicho *kernel* y se recuperarán los datos computados para proceder con las operaciones de reducción de `sx` y `sy` en el *host*, por simplicidad. La Figura 4.18 muestra el esquema descrito.

Respecto al código que ejecutará la GPU, básicamente será similar al que podemos encontrar en la versión original, con la salvedad de que el bucle más externo ha sido mapeado a una iteración por *thread*. Además, el acceso a las estructuras que dependían del número de iteración de dicho bucle se verá modificado, utilizando ahora el identificador global de cada *thread* (`gid`), tal y como podemos ver en la Figura 4.19. También dispondremos de varias estructuras de datos privados a cada uno de los *threads*, que se alojarán en memoria local del dispositivo debido a su tamaño (`x` y `qq`). Para atacar el problema de la región crítica dentro la GPU se han utilizado las operaciones atómicas que CUDA nos proporciona. En concreto, con la operación de suma llevamos a cabo las `NQ` reducciones sobre la estructura `qq`.

```
__global__ void ep_kernel(double * sx_d, double * sy_d,
                          int k_offset, double an,
                          unsigned int * q_d)
{
  unsigned int gid = (blockIdx.x * blockDim.x + threadIdx.x);
  double x[2*NK];
  unsigned int qq[NQ];    /* Private thread copy of q[0:NQ-1] */
  ...
  sx_d[gid] = 0.0;
  sy_d[gid] = 0.0;
  ...
  for ( i = 0; i < NK; i++) {
    x1 = 2.0 * x[2*i] - 1.0;
    x2 = 2.0 * x[2*i+1] - 1.0;
    t1 = pow2(x1) + pow2(x2);
    if (t1 <= 1.0) {
      t2 = sqrt(-2.0 * log(t1) / t1);
      t3 = (x1 * t2);
      t4 = (x2 * t2);
      l = max(fabs(t3), fabs(t4));
      qq[l] += 1;
      sx_d[gid] = sx_d[gid] + t3;
      sy_d[gid] = sy_d[gid] + t4;
    }
  }
  for (i = 0; i < NQ; i++) atomicAdd(&q_d[i], qq[i]);
}
```

Figura 4.19. EP *kernel*: CUDA V1

Los resultados de rendimiento conseguidos (Tabla 4.2) son muy superiores a los obtenidos con las CPUs *multicore*. Además, podemos observar como el *speedup*

aumenta con el tamaño del problema, ya que existen más datos que pueden alimentar a la GPU. Sin embargo, nos consta que nuestro dispositivo aún es capaz de sacar un mayor rendimiento del algoritmo según las características que este posee. Aún no hemos abordado la reducción desde la GPU, y las operaciones atómicas se caracterizan por ser excesivamente lentas, por lo que debería estudiarse una versión alternativa que no las utilice.

	S	W	A	B	C
Opteron 4 Cores	3,96 X	3,97 X	3,97 X	3,94 X	4,03 X
Xeon 8 Cores	8,62 X	8,73 X	8,79 X	8,81 X	8,97 X
CUDA V1	14,43 X	19,10 X	29,81 X	32,33 X	33,64 X

Tabla 4.2. EP *Speedups*: CUDA V1

4.2.2 Memoria Compartida + Reducción Parcial en GPU

En la segunda versión implementada, se introduce el uso de memoria compartida en conjunción con la operación de reducción, según lo descrito en los apartados 4.1.1 y 4.1.2, que también utilizaremos para prescindir de las operaciones atómicas que realizan las `NQ` reducciones sobre `qq`. Incluimos, además, el uso de *templates* en la invocación del *kernel* (4.1.4), para ayudar al compilador con la predicación de saltos como los que introduce la operación de reducción. Esta modificación conllevará trasladar los datos referentes a la configuración del *kernel* a macros que los transformen en constantes. Por último, la transferencia de memoria desde el *host* al dispositivo referente a la estructura `qq` ha sido eliminada, ya que esta puede inicializarse dentro del código del dispositivo.

```
template <unsigned int gridSize, unsigned int blockSize, unsigned int nThreads>
__global__ void ep_kernel_double(double * sx_d, double * sy_d, int k_offset,
                                 double an, unsigned int * qqs_d){
  ...
  __shared__ double sx[blockSize];
  __shared__ double sy[blockSize];
  __shared__ unsigned int my_block_qqs[blockSize * NQ];
  ...
  sx[lid] = 0.0;  sy[lid] = 0.0;
  ...
  for ( i = 0; i < NK; i++) {
    ...
    if (t1 <= 1.0) {
      ...
      my_block_qqs[lid + (l*blockSize)] += 1;
      sx[lid] = sx[lid] + t3; sy[lid] = sy[lid] + t4;
    }
  }
  ...
}
```

Figura 4.20. EP *kernel*: V2-Shared Memory

En la Figura 4.20 se encuentra descrita la primera parte del *kernel* en la que debemos destacar la declaración en memoria compartida de las estructuras `sx`, `sy` y `my_block_qq`, que contendrán una copia privada a cada *thread* de los datos a los que sus nombres hacen referencia. En este caso la memoria compartida no se utiliza para compartir información, sino por su bajo tiempo de acceso. Posteriormente serán utilizadas en la fase de reducción.

```
...
//sx, sy & my_block_qqs reduction in shared memory

__syncthreads();
if (lid < 128) {
  sx[lid] += sx[lid + 128];
  sy[lid] += sy[lid + 128];
  my_block_qqs[lid] += my_block_qqs[lid + 128];
  my_block_qqs[lid+blockSize]+=my_block_qqs[lid+blockSize+128];
  my_block_qqs[lid+(blockSize*2)]+=my_block_qqs[lid+(blockSize*2)+128];
  my_block_qqs[lid+(blockSize*3)]+=my_block_qqs[lid+(blockSize*3)+128];
  my_block_qqs[lid+(blockSize*4)]+=my_block_qqs[lid+(blockSize*4)+128];
  my_block_qqs[lid+(blockSize*5)]+=my_block_qqs[lid+(blockSize*5)+128];
  my_block_qqs[lid+(blockSize*6)]+=my_block_qqs[lid+(blockSize*6)+128];
  my_block_qqs[lid+(blockSize*7)]+=my_block_qqs[lid+(blockSize*7)+128];
  my_block_qqs[lid+(blockSize*8)]+=my_block_qqs[lid+(blockSize*8)+128];
  my_block_qqs[lid+(blockSize*9)]+=my_block_qqs[lid+(blockSize*9)+128];
}
__syncthreads();
if (lid <  64) {
  sx[lid] += sx[lid +  64];
  ...                       //The same for sy & my_block_qqs[]
}
__syncthreads();
if (lid < 32){
  sx[lid] += sx[lid + 32];
  ...                       //The same for sy & my_block_qqs[]
  sx[lid] += sx[lid + 16];
  ...                       //The same for sy & my_block_qqs[]
  sx[lid] += sx[lid + 8];
  ...                       //The same for sy & my_block_qqs[]
  sx[lid] += sx[lid + 4];
  ...                       //The same for sy & my_block_qqs[]
}

// write result for this block to global mem
if (lid == 0){
  sx_d[blockIdx.x] = sx[0];
  sy_d[blockIdx.x] = sy[0];
  qqs_d[blockIdx.x] = my_block_qqs[0];
  qqs_d[blockIdx.x + nThreads]= my_block_qqs[blockSize];
  qqs_d[blockIdx.x + (nThreads*2)] = my_block_qqs[blockSize*2];
  ...                       //The same for the rest my_block_qqs[]
}
```

Figura 4.21. EP *kernel*: V2-Reducción

En la fase de reducción parcial en el dispositivo (Figura 4.21), el algoritmo original ha sido adaptado para que lleve a cabo múltiples operaciones al mismo tiempo, consiguiendo eliminar un número de saltos importante. Además, se han suprimido aquellas partes del algoritmo de las que tenemos constancia que nunca se ejecutarán con la actual configuración de *kernel*.

Como consecuencia de esta última modificación, la transferencia de datos desde el dispositivo hacia el *host* también se verá reducida en gran medida, ya que solamente enviamos los datos resultantes de las reducciones realizadas por cada uno de los bloques, para que se les aplique la última fase de reducción en el *host*.

El *speedup* de esta versión ha superado con creces nuestras expectativas iniciales (Tabla 4.3). Podemos confirmar, entonces, que el uso de memoria compartida es un factor importante a tener en cuenta en el rendimiento, junto con la realización de la reducción parcial en la GPU. Además, la supresión de las operaciones atómicas ha desencadenado la eliminación de gran parte de los *warps* serializados que podíamos encontrar en la versión anterior (véanse *profilings* en Tabla II.1 y Tabla II.2) que estas operaciones provocaban dada su naturaleza. El hecho de hacer un menor uso de las transferencias también ha influido en gran medida en el resultado.

	S	**W**	**A**	**B**	**C**
CUDA V1	14,43 X	19,10 X	29,81 X	32,33 X	33,64 X
CUDA V2-SM	21,08 X	31,34 X	51,18 X	55,12 X	57,19 X

Tabla 4.3. EP *Speedups*: CUDA V2-SM

4.2.3 Mejorando la Ocupación

Si observamos el *profiling* de la versión anterior (Tabla II.2) descubriremos que las modificaciones realizadas han tenido una influencia nefasta en la ocupación de los multiprocesadores del dispositivo, dejando esta sólo al 25%.

Haciendo uso de la herramienta *Cuda Occupancy Calculator* y de los conocimientos anteriormente descritos sobre la ocupación (punto 4.1.5), llegamos a la conclusión de que, para mejorar el uso que nuestra aplicación hace de los multiprocesadores, debemos adaptar nuestro programa para que tan sólo utilice un máximo de 5 KB por bloque de memoria compartida y 20 registros a la vez por *thread*, frente a los 14 KB y 40 registros que utiliza actualmente. Con ello alcanzaremos una ocupación del 75%.

La primera de las restricciones se satisface exiliando la estructura `qq` de cada *thread* de memoria compartida a la memoria global. Con esto, dejaremos libres 10 KB de los 14 ocupados. Para continuar aplicando la operación de reducción sobre las distintas posiciones de `qq` declararemos un par de punteros que nos permitirán reutilizar las estructuras en memoria compartida de `sx` y `sy`, una vez que estas ya hayan terminado de reducirse. Al no poder albergar dichas estructuras todos los datos de las

diferentes `qq`, para realizar las distintas reducciones tendremos que aplicar el algoritmo varias veces para que realice reducciones solamente de dos en dos. El código resultante puede verse en la Figura 4.22.

```
  __shared__ double sx[blockSize];
  __shared__ double sy[blockSize];
  unsigned int * qqs_1 = (unsigned int *) sx;
  unsigned int * qqs_2 = (unsigned int *) sy;
  ...
/* sx, sy reduction */
  __syncthreads();

  // do reduction in shared mem
  if (lid < 128) {
    sx[lid] += sx[lid + 128];
    sy[lid] += sy[lid + 128];
  }
  ...
  // write result for this block to global mem
  ...
/* qqs_d reduction */
  for (i=0; i<NQ; i+=2){
    __syncthreads();

    if (lid < 128) {
      qqs_1[lid] = qqs_d[( i    * nThreads) + gid] +
                   qqs_d[( i    * nThreads) + gid + 128];
      qqs_2[lid] = qqs_d[((i+1) * nThreads) + gid] +
                   qqs_d[((i+1) * nThreads) + gid + 128];
    }
    ...

    // write result for this block to global mem
    if (lid == 0){
      qqs_reduction_d[( i   *gridSize) + blockIdx.x] = qs_1[0];
      qqs_reduction_d[((i+1)*gridSize) + blockIdx.x] = qqs_2[0];
    }
  }
```

Figura 4.22. EP *kernel*: V3-Occupancy 0.75

Tras el gran esfuerzo necesario para llevar a cabo las modificaciones, solamente se produce una ligera mejora de rendimiento para grandes tamaños del problema (Tabla 4.4). Sin embargo, deberíamos considerarlo como algo positivo si tenemos en cuenta que hemos pasado una gran estructura (`qq`'s) de memoria compartida a memoria global. Por tanto, la mejora conseguida por ocupar un mayor número de multiprocesadores sobrepasa la enorme diferencia entre las latencias de ambos tipos de memorias.

	S	W	A	B	C
CUDA V2-SM	21,08 X	31,34 X	51,18 X	55,12 X	57,19 X
CUDA V3-Occup	21,02 X	31,38 X	51,59 X	55,65 X	57,80 X

Tabla 4.4. EP *Speedups*: CUDA V3-Occupancy 0.75

4.2.4 Uso de Memoria Pinned

Como hemos descrito en el apartado 4.1.6, para hacer uso de la memoria *pinned* solamente necesitaremos de la función `cudaHostAlloc()` para reservar la memoria en el *host*. El volumen de datos que se transfiere hacia el dispositivo es menor a 2 megabytes para el tamaño C según el *profiling* de la versión anterior (Tabla II.3), por lo que no deberíamos esperar una gran mejora con la aplicación de esta técnica.

	S	W	A	B	C
CUDA V3-Occup	21,02 X	31,38 X	51,59 X	55,65 X	57,80 X
CUDA V4-Pinned	48,05 X	53,94 X	56,47 X	56,97 X	58,15 X

Tabla 4.5. EP *Speedups*: CUDA V4-Pinned

Sin embargo, tal y como podemos ver en la Tabla 4.5, el rendimiento se dispara para los tamaños más pequeños del problema. La justificación de tal mejora radica en que, para tamaños pequeños, las transferencias de datos suponen más tiempo que el procesamiento de los propios datos, por lo que, cuando el volumen de cómputo aumenta, la diferencia de rendimiento se reduce enormemente.

4.2.5 Zero Copy

Como caso particular del anterior, la técnica *Zero Copy* no consigue un aumento de rendimiento significativo (Tabla 4.6) a pesar de que, en teoría, debería producirse cierto solapamiento entre la transferencia de datos y su cómputo. Como consecuencia, esta optimización es desechada.

	S	W	A	B	C
CUDA V4-Pinned	48,05 X	53,94 X	56,47 X	56,97 X	58,15 X
CUDA V5-Zero	48,20 X	54,03 X	56,47 X	56,96 X	58,13 X

Tabla 4.6. EP *Speedups*: CUDA V5-Zero

4.2.6 Memoria Local vs Memoria Global

La siguiente versión viene a recuperar la ubicación original de la estructura `qq`. En la tercera versión había sido movida de memoria compartida a memoria global para aumentar la ocupación del dispositivo. Sin embargo, al tratarse de una estructura privada a cada *thread*, lo más adecuado es que se encuentre en la memoria que existe para tal cometido, es decir, la memoria local.

Como ya hemos comentado en el apartado de Modelo de Memoria, la memoria local no es más que memoria global con visibilidad a nivel de *thread*, cuyo uso queda en manos de compilador. Sin embargo, podemos asegurar que la estructura quedará almacenada en esta debido a su tamaño.

	S	W	A	B	C
CUDA V5-Zero	48,20 X	54,03 X	56,47 X	56,96 X	58,13 X
CUDA V6-QQsLM	48,48 X	54,35 X	56,83 X	57,34 X	58,52 X

Tabla 4.7. EP *Speedups*: CUDA V6-QQsLM

En cuanto al rendimiento, no deberíamos encontrarnos con una diferencia significativa. El pequeño aumento del *speedup* que podemos observar en la Tabla 4.7 puede deberse a las operaciones que se realizaban en la versión anterior para calcular el índice de acceso a la estructura en memoria global, ya que estas se ven reducidas, al menos en tiempo de ejecución, al usar la memoria local (Figura 4.23).

```
Global Memory:
      my_block_qqs[lid + (l*blockSize)] += 1;
Local Memory:
      qq[l] += 1;
```

Figura 4.23. Acceso a Memoria Local vs. Memoria Global

4.2.7 Unrolling

Recuperando la importancia de los saltos condicionales descrita en el punto 4.1.4, y la posibilidad de usar la directiva *#pragma unroll* para controlar el nivel de desenrollado de los bucles del *kernel*, se han llevado a cabo distintas pruebas para conseguir el nivel más adecuado de este para cada uno de los distintos.

```
   //Full unrolled. NQ=10
   #pragma unroll
   for (i = 0; i < NQ; i++){
       my_qqs[i] = 0;
   }
   ...

   //Not Unrolled
   for (i = 1; i <= 100; i++) {
     ...
   }

   //Unroll 4 by hand
   for ( i = 0; i < NK; i+=4) {
     ...
   }

/* qqs_d reduction */
   //Not unrolled. NQ=10
   #pragma unroll 1
   for (i=0; i<NQ; i+=2){
     ...
   }
```

Figura 4.24. Nivel de desenrollado en los bucles del *kernel*

Como resultado, hemos podido comprobar como, en algunos casos, el compilador no es capaz de aplicar el grado de desenrollado óptimo para conseguir un mayor rendimiento. Tampoco es capaz de aplicar el mencionado desenrollado aún forzándolo con la directiva `#pragma unroll`, siendo necesario intervenir manualmente para lograr el objetivo perseguido. La Figura 4.24 muestra la mejor configuración conseguida tras diferentes pruebas y análisis del código ensamblador.

	S	W	A	B	C
CUDA V6-QQsLM	48,48 X	54,35 X	56,83 X	57,34 X	58,52 X
CUDA V7-Unroll	48,56 X	54,45 X	56,93 X	57,44 X	58,62 X

Tabla 4.8. EP *Speedups*: CUDA V7-Unroll

El *speedup* conseguido no dista demasiado del resultante de la versión anterior (Tabla 4.8). Sin embargo, nos encontramos al final de la fase de optimización, donde ya se pretenden mejorar pequeños detalles.

4.2.8 Enfoque Multi-GPU

Para conseguir extender el algoritmo a varias GPUs dentro de un mismo nodo, seguiremos el primer esquema descrito en el punto 4.1.9 con ciertas adaptaciones. En este caso, el reparto de trabajo entre varios dispositivos se llevará a cabo en tiempo de compilación, mediante las diferentes directivas que definen la configuración de *threads* y bloques con los que será ejecutado el *kernel*. En concreto, el número de bloques necesarios para computar el problema será repartido equitativamente en función del número de GPUs existentes. El código utilizado para la distribución del trabajo entre dos dispositivos puede verse en la Figura 4.25. Pueden ser necesarias algunas modificaciones si la carga de trabajo no es divisible entre el número de dispositivos utilizado, como ocurre en este caso.

```
#define NUM_GPUS        2
#define BLOCK_SIZE      256
#if (NUM_GPUS > 1)
  #define GRID_SIZE     ((NN/BLOCK_SIZE)/NUM_GPUS)

#else
  #define GRID_SIZE     (NN/BLOCK_SIZE)
#endif

#define NTHREADS        GRID_SIZE*BLOCK_SIZE
```

Figura 4.25. EP *host*: V8-Multi-GPU. División de trabajo

Las modificaciones que serán necesarias en el *kernel* son mínimas, ya que este solamente requerirá el paso de un nuevo parámetro que hará las funciones de `my_chunk`, según el esquema. En concreto, este parámetro será el desplazamiento necesario para calcular un identificador global de los *threads* a través de las distintas

tarjetas utilizadas, ya que es preciso para el cálculo de la semilla de los números aleatorios (Figura 4.26).

```
__global__ void ep_kernel_double(..., unsigned int device_disp)

  unsigned int inter_gid = (blockIdx.x * blockSize + threadIdx.x)
                           + device_disp;
  ...
  kk = k_offset + inter_gid + 1;
  ...
}
```

Figura 4.26. EP *kernel*: V8-Multi-GPU

Al tener que introducir una región paralela dentro del *host*, será necesario tener en cuenta algunas consideraciones en cuanto a la fase de reducción. En primer lugar habrá que realizar una primera reducción similar a la que se llevaba a cabo en versiones anteriores, pero en este caso la ejecutará cada uno de los *threads* del *host*. Para concluir la fase de reducción le indicaremos a OpenMP que aúne los resultados obtenidos por cada *thread* mediante el la opción `reduction` en la directiva `parallel`. El esquema concreto de los cambios necesarios en el *host* puede consultarse en la Figura 4.27.

```
#pragma omp parallel reduction(+:sx,sy,q0,q1,q2,q3,q4,q5,q6,q7,q8,q9)
num_threads(num_gpus) if (num_gpus > 1)
{
  ... //Local thread/gpu declarations

  cudaSetDevice(omp_get_thread_num());
  cudaGetDevice(&my_device);

  ... //Device Memory Allocation

  ep_kernel_double<<<grid, block>>>
      (sx_d,sy_d,k_offset,an,qqs_reduced_d,my_device*NTHREADS);

  ... //Device To Host Transferences

  /*sx , sy & qqs final reduction. */
  for (i = 0; i < GRID_SIZE; i++) {
    sx += sx_h[i]; sy += sy_h[i];
    q0 += qqs_reduced_h[i];
    q1 += qqs_reduced_h[GRID_SIZE     + i];
    q2 += qqs_reduced_h[(GRID_SIZE*2) + i];
    q3 += qqs_reduced_h[(GRID_SIZE*3) + i];
    q4 += qqs_reduced_h[(GRID_SIZE*4) + i];
    q5 += qqs_reduced_h[(GRID_SIZE*5) + i];
    q6 += qqs_reduced_h[(GRID_SIZE*6) + i];
    q7 += qqs_reduced_h[(GRID_SIZE*7) + i];
    q8 += qqs_reduced_h[(GRID_SIZE*8) + i];
    q9 += qqs_reduced_h[(GRID_SIZE*9) + i];
  }
  ...//Device Memory Free
}
```

Figura 4.27. EP *host*: V8-Enfoque Multi-GPU

Los resultados de rendimiento son satisfactorios para tamaños grandes del problema, tal y como se muestra en la Tabla 4.9. Podemos ver como el programa escala perfectamente al duplicar el número de GPUs. Obviamente, la mejora se ve mermada cuando el problema posee un tamaño relativamente pequeño, e incluso obtiene un peor resultado que cuando utilizamos solamente un dispositivo. Esto es debido a que en estos casos, el overhead introducido por tener que crear una región paralela, repartir el trabajo, utilizar múltiples dispositivos y sincronizar los resultados, no compensa con respecto a lo mejora obtenida por disponer de más capacidad de cómputo. El volumen de trabajo es tan pequeño que no llega ni a cubrir el 100% de los recursos de cómputo disponibles en una sola de las tarjetas, por lo que la distribución de trabajo no consigue más que introducir sobrecarga que antes no existía.

	S	W	A	B	C
CUDA V7-Unroll	48,56 X	54,45 X	56,93 X	57,44 X	58,62 X
CUDA V8-MultiGPU (2x)	15,65 X	28,19 X	82,73 X	105,28 X	114,48 X

Tabla 4.9. EP *Speedups*: CUDA V8-MultiGPU (2x)

4.2.9 Conclusiones

Como podemos apreciar en la Tabla 4.10, de las siete versiones finales desarrolladas en CUDA para una sola GPU, debemos destacar que, sin lugar a dudas, la optimización más relevante ha sido la referente al uso de memoria compartida y a la introducción de las operaciones de reducción en el dispositivo (V2-SM). Esta versión es en la que más cambios se han aplicado, eliminando gran parte de las transferencias de memoria y operaciones atómicas, entre otros. También es de las versiones más costosas, pero el aumento de casi 24 X conseguido no ha sido equiparable ni al de la mejora promovida por el resto de optimizaciones juntas, basadas en una sola GPU.

	S	W	A	B	C
Opteron Serie	1,00 X	1,00 X	1,00 X	1,00 X	1,00 X
Opteron 4 Cores	3,96 X	3,97 X	3,97 X	3,94 X	4,03 X
Xeon 8 Cores	8,62 X	8,73 X	8,79 X	8,81 X	8,97 X
CUDA V1	14,43 X	19,10 X	29,81 X	32,33 X	33,64 X
CUDA V2-SM	**21,08 X**	**31,34 X**	**51,18 X**	**55,12 X**	**57,19 X**
CUDA V3-Occup	21,02 X	31,38 X	51,59 X	55,65 X	57,80 X
CUDA V4-Pinned	**48,05 X**	**53,94 X**	56,47 X	56,97 X	58,15 X
CUDA V5-Zero	48,20 X	54,03 X	56,47 X	56,96 X	58,13 X
CUDA V6-QQsLM	48,48 X	54,35 X	56,83 X	57,34 X	58,52 X
CUDA V7-Unroll	48,56 X	54,45 X	56,93 X	57,44 X	58,62 X
CUDA V8-MultiGPU (2x)	15,65 X	28,19 X	82,73 X	105,28 X	114,48 X

Tabla 4.10. EP: Resumen *Speedups*

Sin embargo, es interesante observar como una implementación simple, aunque bastante costosa, del algoritmo en CUDA ya supone una diferencia de más de 24 X con respecto a la ejecución paralela del algoritmo en una CPU de ocho *cores*.

También debemos destacar el papel que ha tenido el uso de memoria no paginable sobre tamaños pequeños del problema, permitiendo nivelar el rendimiento obtenido por la GPU sobre los mismos con respecto a los de mayor envergadura.

El resto de optimizaciones, aunque costosas, han resultado ser menos relevantes, al menos sobre este tipo de problema. Por tanto, debemos concluir no recomendando su aplicación en situaciones similares, haciendo referencia a la bajísima relación $\frac{rendimiento}{tiempo\ de\ implementación}$.

Respecto a la versión Multi-GPU, ha sido interesante comprobar como el algoritmo es capaz de escalar adecuadamente al utilizar dos dispositivos, siempre que el tamaño sea suficiente para cubrir las capacidades de cómputo que estos nos ofrecen. Como consecuencia, con un relativamente bajo tiempo de implementación hemos podido multiplicar casi por dos el rendimiento de la aplicación, y posiblemente, con la escasa comunicación necesaria, el escalado mantendrá una tendencia similar para un número de dispositivos más elevado siempre que la carga siga siendo suficiente. Por todo ello, se recomienda encarecidamente aplicar la mencionada estrategia en problemas similares.

La representación gráfica de los resultados se muestra a continuación en la Figura 4.28.

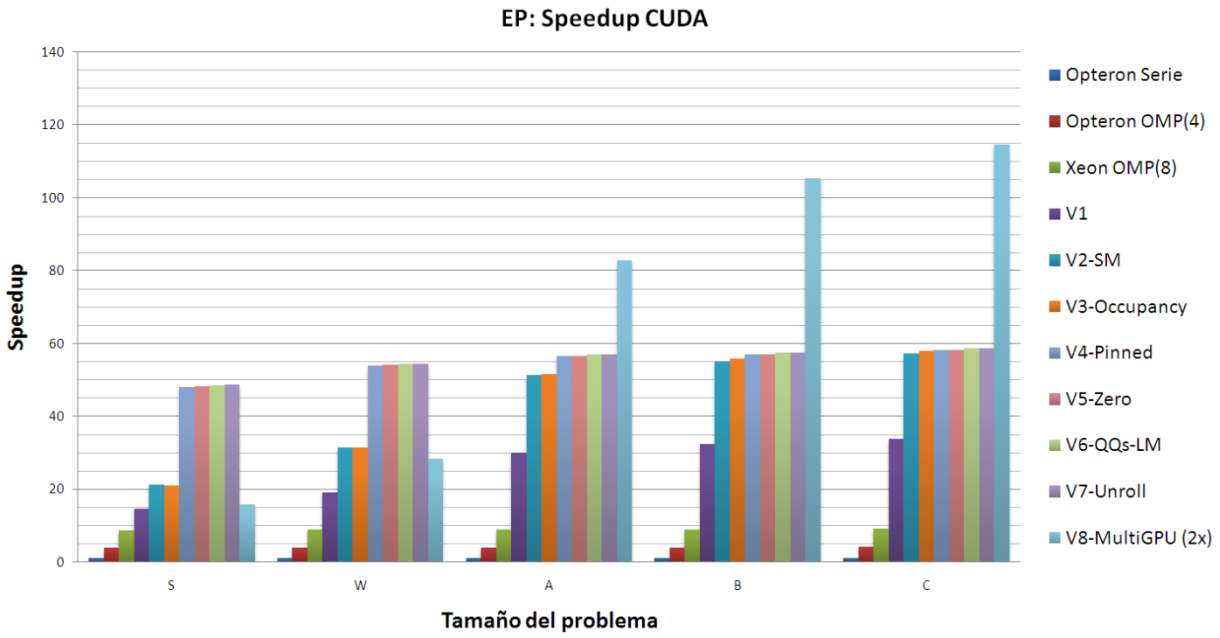


Figura 4.28. EP: Gráfica de *Speedups*

4.3 Conjugate Gradient (NAS Parallel Benchmarks)

El *benchmark* CG utiliza el método del gradiente conjugado para aproximar la solución de un sistema de ecuaciones lineal disperso de gran tamaño, donde los coeficientes de dicho sistema quedan representados en una matriz dispersa compactada en formato CSR – *Compressed Sparse Row Format* – (Véase Figura 4.29). Este formato se caracteriza por agrupar a los elementos distintos de cero de toda la matriz en un vector de una sola dimensión (`a`), conservando en vectores auxiliares las posiciones de la matriz compactada que se corresponden con el inicio de cada fila de la estructura en su formato original (`rowstr`), y los índices de las columnas referentes a posiciones distintas de cero (`colidx`)[11].

$$\mathbf{A} = \begin{pmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{pmatrix}$$

$$\mathbf{rowstr} = \begin{pmatrix} 0 & 2 & 4 & 7 & 9 \end{pmatrix}$$

$$\mathbf{colidx} = \begin{pmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{pmatrix}$$

$$\mathbf{a} = \begin{pmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{pmatrix}$$

Figura 4.29. Ejemplo de representación CSR de una matriz

El paralelismo de este *benchmarks* se encuentra repartido por diferentes bucles internos a las iteraciones que computan el gradiente. La densidad computacional no es muy elevada, frente a la gran cantidad de estructuras de datos que se utilizan y la dispersión de datos originada por algunas operaciones. Por tanto, con CG se pretende valorar el rendimiento que obtiene un sistema bajo un perfil de ejecución donde los accesos a memoria no se encuentran próximos entre sí, y las comunicaciones juegan un papel importante.

La Figura 4.30 muestra un resumen aproximado de la estructura del *benchmark*, donde la parte paralela supone el 99% del tiempo de ejecución inicial (medido experimentalmente). Por esta razón, al igual que en EP, seguiremos la filosofía de NAS y tomaremos las medidas de tiempo que el propio *benchmark* proporciona, referentes solamente a la región paralela.

La primera región serie se corresponde mayormente con la función `makea()`. Esta función es la encargada de crear la matriz dispersa compactada y sus estructuras asociadas `rowstr` y `colidx`. Una vez disponemos de la matriz, comenzamos a pasar por distintas regiones paralelas donde entran en juego un elevado número de estructuras de datos. Las primeras de ellas comienzan siendo meras inicializaciones de las citadas estructuras y pequeños cálculos de valores que necesitaremos más adelante. El grueso

computacional de la aplicación se encuentra en las regiones C y F, donde se realizan las operaciones relacionadas con el sistema de ecuaciones, multiplicando la matriz dispersa por un vector unidimensional.

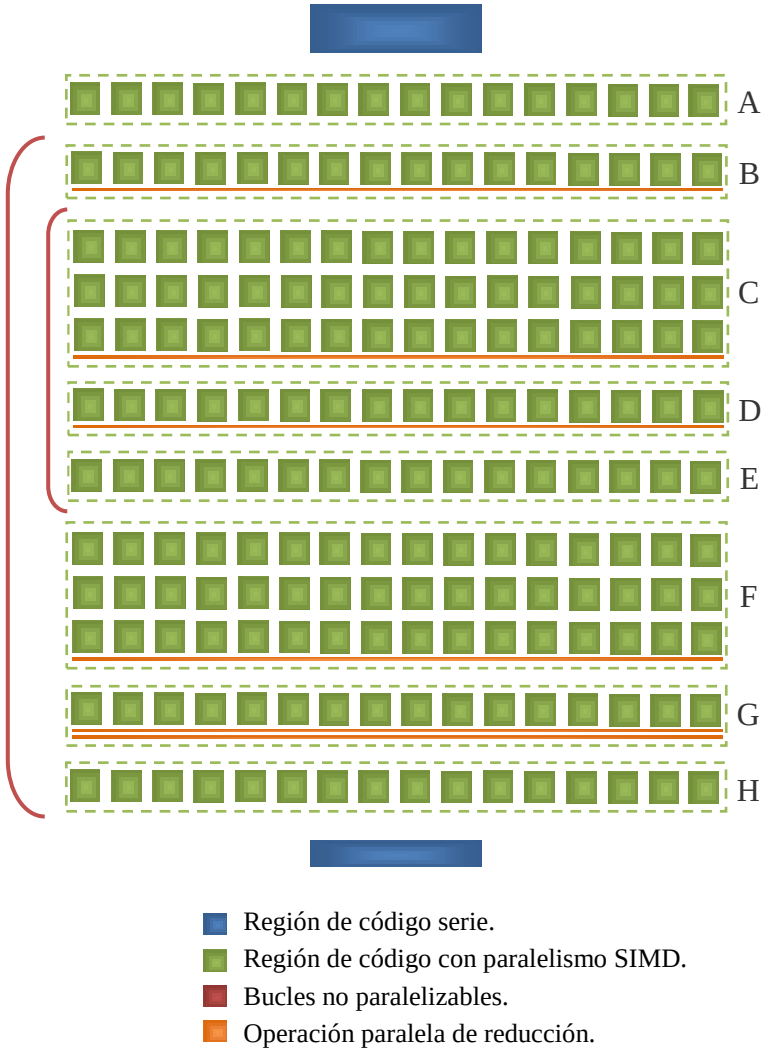


Figura 4.30. Estructura del benchmark CG

El bucle no paralelizable más externo se corresponde con el bucle principal del programa, donde cada iteración depende de los valores calculados en la iteración anterior para seguir avanzando con el método del gradiente. El bucle interior a este tampoco es paralelizable y su sus distintas iteraciones concentran en la región C un alto porcentaje del tiempo de ejecución.

Las operaciones de reducción se encuentran muy presentes en el algoritmo y han guiado la separación de las distintas regiones paralelas en la GPU por necesitar una última fase de cómputo en la CPU. Además, tras la mayoría de estas operaciones se suele encontrar una o varias instrucciones que calculan un valor a partir del resultado de la reducción, y que se suele usar por la siguiente región paralela.

El código fuente original de CG tuvo que modificarse previamente antes de comenzar con su migración a CUDA. Al tratarse de un *benchmark* procedente de

Fortran, en su traducción a C todos los bucles comenzaban por la iteración uno, en lugar de hacerlo en la cero, como se habitúa en este lenguaje. Ese pequeño detalle aparentemente sin importancia dejaba el elemento cero de todas las estructuras de datos sin utilizar. Debido a las estrictas restricciones de CUDA en cuanto a los accesos de memoria *coalesced*, se decidió adaptar el algoritmo para que cada bucle comenzara por la iteración cero. Dicho trabajo resultó especialmente arduo y costoso debido a la complejidad que encontramos en funciones como la que inicializa la matriz dispersa (`makea()`). El algoritmo quedo, de esta manera, más predispuesto a su migración a CUDA sin diferencias de rendimiento producidas por la modificación.

Si pasamos al análisis de las regiones paralelas, la Figura 4.31 y Figura 4.32 muestran un resumen de las mismas, donde la mayoría se encuentran albergadas en esta última, que hace referencia a la función `conj_grad()`, la cual es invocada en cada iteración del bucle principal. Podemos ver la baja cantidad de operaciones, y como consecuencia instrucciones, que tendrán lugar en la mayor parte de las regiones, sobre todo en aquellas en las que apenas se inicializan o intercambian valores entre estructuras, con la ligera excepción de C y F.

```
...
makea();
...
#pragma omp parallel for nowait                                   A
for (i = 0; i < NA+1; i++) {
  x[i] = 1.0;
}

zeta  = 0.0;

#pragma omp parallel private(it,i,j,k)
{
  for (it = 1; it <= NITER; it++) {

    conj_grad(colidx, rowstr, x, z, a, p, q, r, w, &rnorm);

    #pragma omp single{ norm_temp11 = 0.0; norm_temp12 = 0.0; }
    #pragma omp for reduction(+:norm_temp11,norm_temp12)          G
    for (j = 0; j < lastcol-firstcol; j++) {
      norm_temp11 = norm_temp11 + x[j]*z[j];
      norm_temp12 = norm_temp12 + z[j]*z[j];
    }

    #pragma omp single {
      norm_temp12 = 1.0 / sqrt( norm_temp12 );
      zeta = SHIFT + 1.0 / norm_temp11;
    }
    ...
    #pragma omp for                                               H
    for (j = 0; j < lastcol-firstcol; j++) {
      x[j] = norm_temp12*z[j];
    }
  }
}
```

Figura 4.31. CG: Resumen de código base para la CPU. Función *main*

```
...
#pragma omp for nowait                                                    B
for (j = 0; j < naa+1; j++) {
  q[j] = 0.0; z[j] = 0.0; r[j] = x[j]; p[j] = r[j]; w[j] = 0.0;
}
#pragma omp for reduction(+:rho)
for (j = 0; j < lastcol-firstcol; j++) { rho = rho + x[j]*x[j]; }

for (cgit = 1; cgit <= cgitmax; cgit++) {
  #pragma omp single nowait{ rho0 = rho; d = 0.0; rho = 0.0; }
  #pragma omp for private(sum,k)                                          C
  for (j = 0; j < lastrow-firstrow; j++) {
    sum = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
      sum = sum + a[k]*p[colidx[k]];
    }
    w[j] = sum;
  }
  #pragma omp for
  for (j = 0; j < lastcol-firstcol; j++) { q[j] = w[j]; }
  #pragma omp for nowait
  for (j = 0; j < lastcol-firstcol; j++) { w[j] = 0.0; }
  #pragma omp for reduction(+:d)
  for (j = 0; j < lastcol-firstcol; j++) { d = d + p[j]*q[j]; }
  #pragma omp single {alpha = rho0 / d;}
  #pragma omp for                                                         D
  for (j = 0; j < lastcol-firstcol; j++) {
    z[j] = z[j] + alpha*p[j];
    r[j] = r[j] - alpha*q[j];
  }
  #pragma omp for reduction(+:rho)
  for (j = 0; j < lastcol-firstcol; j++) {
    rho = rho + r[j]*r[j];
  }
  #pragma omp single { beta = rho / rho0; }
  #pragma omp for                                                         E
  for (j = 0; j < lastcol-firstcol; j++) {
    p[j]=r[j] + beta*p[j];
  }
}
#pragma omp single nowait { sum = 0.0; }
#pragma omp for private(d, k)                                             F
for (j = 0; j < lastrow-firstrow; j++) {
  d = 0.0;
  for (k = rowstr[j]; k <= rowstr[j+1]-1; k++) {
    d = d + a[k]*z[colidx[k]];
  }
  w[j] = d;
}
#pragma omp for
for (j = 0; j < lastcol-firstcol; j++) { r[j] = w[j]; }
#pragma omp for reduction(+:sum) private(d)
for (j = 0; j < lastcol-firstcol; j++) {
  d = x[j] - r[j];
  sum = sum + d*d;
}
#pragma omp single { (*rnorm) = sqrt(sum); }
```

Figura 4.32. CG: Resumen de código base para la CPU. Función *conj_grad*

La ejecución de CG sobre las CPUs *multicore* ha dado resultados dispares en función del tamaño del problema, como vemos en la Tabla 4.11. Este suceso no hace más que confirmar la importancia que la jerarquía de memoria juega en este sentido, al trabajar con un algoritmo que realiza accesos que no explotan la localidad espacial. La enorme diferencia en este sentido que existe entre ambos procesadores permite a Xeon alcanzar cotas de rendimiento muy altas en comparación con la CPU Opteron, mientras que la única excepción se da para el tamaño B, donde podemos ver como estas cotas se ven reducidas, probablemente porque el tamaño del problema da lugar a conflictos de caché que no aparecen en el resto de tamaños.

	S	W	A	B	C
Opteron Serie	1,00 X	1,00 X	1,00 X	1,00 X	1,00 X
Opteron 4 Cores	3,65 X	1,65 X	1,90 X	2,31 X	2,40 X
Xeon 8 Cores	5,18 X	9,97 X	6,64 X	3,91 X	9,88 X

Tabla 4.11. CG *Speedups*: *Multicores*

4.3.1 Versión Inicial en CUDA

La migración a CUDA da como resultado un código con multitud de *kernels* sencillos, en la mayoría, que serán invocados hasta miles de veces en la misma ejecución.

```
#define BLOCK_SIZE  128
#define GRID_SIZE   (NA%BLOCK_SIZE ? ((NA/BLOCK_SIZE)+1) :
                    (NA/BLOCK_SIZE))
dim3 grid(GRID_SIZE);
dim3 block(BLOCK_SIZE);

main(){
  ...
  cudaMalloc((void **)&colidx_d, (NZ)   * sizeof(int));
  cudaMalloc((void **)&rowstr_d, (NA+1) * sizeof(int));
  cudaMalloc((void **)&a_d, (NZ)   * sizeof(double));
  ...
  cudaMemcpy(colidx_d,..., cudaMemcpyHostToDevice);
  cudaMemcpy(rowstr_d,..., cudaMemcpyHostToDevice);
  cudaMemcpy(a_d,     ..., cudaMemcpyHostToDevice);

  init_x_kernel<<<grid, block>>>();                    //A

  for (it = 1; it <= NITER; it++) {
    cong_grad(...);
    norm_temp_kernel<<<grid, block>>> (...);           //G
    //CPU reduction phase
    x_norm_z_kernel<<<grid, block>>> (...);            //H
  }
  ...
}
```

Figura 4.33. CG *host*: CUDA V1. Función *main*

La Figura 4.33 y Figura 4.34 muestran parte del código *host*, correspondiente a la función `main()` y `conj_grad()` respectivamente. Podemos ver como la configuración de *threads* por bloque elegida es de 128, llegando a necesitar más de 1000 bloques para el tamaño C del problema, según podemos comprobar en el correspondiente *profiling* (Tabla II.9). La configuración es idéntica para todos los *kernels* del algoritmo, en función del tamaño del problema.

Las transferencias de memoria que se llevan a cabo entre el *host* y la GPU son relativamente pequeñas, ya que se ha procurado que la mayoría de las estructuras de datos del problema se inicialicen en el propio dispositivo para evitar su transferencia. Así, inicialmente sólo necesitaremos copiar a la GPU los valores de la matriz dispersa y sus estructuras asociadas (`a`, `rowstr` y `colidx`). En la dirección inversa encontraremos pequeñas transferencias relacionadas con la última fase de la reducción que se lleva a cabo en la CPU.

A pesar de que en los dispositivos utilizados no es posible ejecutar varios *kernels* concurrentemente, se han introducido puntos de sincronización para la GPU (`cudaThreadSynchronize()`), que serían necesarios en caso de que el *benchmark* fuera ejecutado sobre tarjetas de nueva generación que sí soportan esta característica. Así, evitaríamos posibles condiciones de carrera que pudieran producirse

```
conj_grad(){
  ...
  cudaThreadSynchronize();

  conj_grad_init_and_rho_kernel<<<grid, block>>> (); //B
  //CPU reduction phase

  for (cgit = 1; cgit <= cgitmax; cgit++) {

    conj_grad_sum_d_kernel<<<grid, block>>> ();       //C
    //CPU reduction phase

    conj_grad_z_r_rho_kernel<<<grid, block>>> ();     //D
    //CPU reduction phase
    conj_grad_p_kernel<<<grid, block>>> ();           //E

    cudaThreadSynchronize();
  }

  conj_grad_d_sum_kernel();                           //F
  //CPU reduction phase
}
```

Figura 4.34. CG *host*: CUDA V1. Función *conj_grad*

En lo relacionado con la implementación de los *kernels*, vamos a distinguir dos esquemas diferentes. El primero de ellos, Figura 4.35, englobará a la gran mayoría (A, B, D, E, G y H), que poseen un bucle simple que itera sobre alguna estructura unidimensional. El mapeo de *threads* que tendrá lugar será el de un *thread* por iteración del bucle. Además, es necesario incluir el código a ejecutar por cada *thread* dentro de un salto condicional, que garantizará que solamente aquellos *threads* que queden dentro

del rango de los datos ejecuten dicho código. Esto resulta imprescindible, pues existirán *threads* por encima de este rango en el último de los bloques cuando el tamaño de los datos no sea múltiplo del número de *thread* por bloque (128). Tras esta primera fase, los *kernels* que requieran de una operación de reducción la llevarán a cabo a continuación.

```
template <unsigned int gridSize,
          unsigned int blockSize,
          unsigned int nThreads>
__global__ void kernel(double * x_d){

    if (gid < N){
        code without loops
    }

    [reduction phase]
}
```

Figura 4.35. CG: Esquema de *kernels* A, B, D, E, G, H: CUDA V1

El segundo esquema es parecido al primero en cuanto a la condición para descartar la ejecución de ciertos *threads*. Este es seguido por los *kernels* C y F, que trabajan con la matriz dispersa. Puesto que en este caso nos encontramos con un bucle anidado, desplegaremos una estrategia de *threads* donde cada iteración del bucle exterior quede cubierta por uno de ellos, es decir, estaremos asignando un *thread* por fila de la matriz para realizar la multiplicación entre esta y el vector (Figura 4.36). Algunas operaciones sin mucha relevancia se ejecutarán a continuación de la citada multiplicación, junto con una operación de reducción similar a las ya descritas.

```
template <unsigned int gridSize, unsigned int     kSize, unsigned
int nThreads>
__global__ void conj_grad_sum_d_kernel(...){

  double sum = 0.0;
  __shared__ double d[blockSize];

  d[lid] = 0.0;

  if (gid < NA){
    for (k = rowstr_d[gid]; k < rowstr_d[gid+1]; k++){
      sum = sum + a_d[k]*p_d[colidx_d[k]];
    }
    w_d[gid] = sum;
    q_d[gid] = w_d[gid];
    w_d[gid] = 0.0;
    d[lid] = p_d[gid]*q_d[gid];
  }
...  //d reduction
}
```

Figura 4.36. CG *kernel* C (y F): CUDA V1

Es interesante destacar como en ambos esquemas nos encontramos con una instrucción fuera del la condición selectora de *threads* (`d[lid] = 0.0`, no se muestra

en el primero). Resulta imprescindible inicializar el array de datos de memoria compartida con el valor neutro para la operación de reducción antes de entrar a la condición, ya que los *threads* que no ejecuten dicho código posteriormente sí participaran en la reducción y podrían introducir valores no esperados.

Un aspecto interesante con el que nos hemos encontrado a nivel de compilación es que el *benchmark* deja de funcionar correctamente si es compilado con el nivel de optimización O3. Frente a todo pronostico, no se debe a nada relacionado directamente con las GPUs, sino que en alguna fase de optimización de código *host* las invocaciones a los primeros *kernels* son ejecutadas posteriormente a la utilización de los datos que estos producen. Según pudimos comprobar parece deberse a una incompatibilidad del compilador de CUDA con las versiones más recientes de GCC. El problema desaparece al bajar a dos el nivel de optimización, por lo que así hemos procedido.

En cuanto a los resultados sobre la GPU, podemos ver en la Tabla 4.12 como vamos obteniendo un mayor rendimiento conforme aumentamos el tamaño del problema, similar a lo ocurrido en EP. Sin embargo, debemos destacar como estos no llegan a alcanzar a los resultados obtenidos por una de las CPUs, por lo que debemos estudiar cuál es la causa de ello.

	S	W	A	B	C
Opteron 4 Cores	3,65 X	1,65 X	1,90 X	2,31 X	2,40 X
Xeon 8 Cores	5,18 X	9,97 X	6,64 X	3,91 X	9,88 X
CUDA V1	1,07 X	2,80 X	3,16 X	3,76 X	8,56 X

Tabla 4.12. CG *Speedups*: CUDA V1

4.3.2 Mapear Warps sobre filas

Si observamos el *profiling* de la versión anterior (Tabla II.9) encontramos que el tiempo de ejecución de los *kernels* que trabajan con las matrices dispersas (C y F) supera el 99% del total. Centrándonos en los datos de estos *kernels* descubrimos un cuello de botella inmediato: el throughput de acceso a memoria es lamentablemente bajo, sobre todo en el *kernel* que es invocado un mayor número de veces. Este fenómeno es producido por el tipo de mapeo de *threads* que hemos utilizado en la versión anterior. El hecho de que cada *thread* se encargue de manera independiente de una sola fila tiene como consecuencia que los accesos que realicen los *threads* dentro del mismo *half-warp* quedaren distantes, y desencadenen múltiples accesos a memoria en lugar de sólo uno (no *coalesced*). Por tanto debemos cambiar dicha estrategia de mapeo por una en la que se asignen *warps* sobre filas de la matriz dispersa, tal y como ya hemos descrito en el apartado 4.1.3. Además, por la naturaleza de las operaciones, necesitaremos aplicar un pequeño algoritmo de reducción sobre los *threads* del mismo *warp* para aunar los valores dentro de la misma fila. El código resultante de la aplicación de esta estrategia

para el *kernel* C se muestra en la Figura 4.37. Una modificación similar será necesaria para el *kernel* F.

```
template <unsigned int gridSize, unsigned int blockSize, unsigned
int nThreads, unsigned int nThreadsWarp>
__global__ void conj_grad_sum1_kernel(...){

  int w_id  = gid/nThreadsWarp;     //Global warp index
  int lw_id = gid&(nThreadsWarp-1); //Thread index within the warp

  __shared__ double sum[blockSize];

  if (w_id < NA){                   //One warp per row
    sum[lid] = 0.0;
    for (k=rowstr_d[w_id]+lw_id; k<rowstr_d[w_id + 1]; k+=32){
      sum[lid] = sum[lid] + a_d[k] * p_d[colidx_d[k]];
    }

    //sum warp reduction with warp size = 32
    if (lw_id < 16) sum[lid] = sum[lid] + sum[lid + 16];
    if (lw_id <  8) sum[lid] = sum[lid] + sum[lid +  8];
    if (lw_id <  4) sum[lid] = sum[lid] + sum[lid +  4];
    if (lw_id <  2) sum[lid] = sum[lid] + sum[lid +  2];
    if (lw_id <  1) sum[lid] = sum[lid] + sum[lid +  1];

    if (lw_id == 0) w_d[w_id] = sum[lid];
  }
}

template <unsigned int gridSize, unsigned int blockSize, unsigned
int nThreads>
__global__ void conj_grad_d1_kernel(...){

  __shared__ double d[blockSize];
  d[lid] = 0.0;

  if (gid < NA){
    q_d[gid] = w_d[gid];
    w_d[gid] = 0.0;
    d[lid] = p_d[gid]*q_d[gid];
  }
...  //d reduction
}
```

Figura 4.37. CG *kernels* C (y F): CUDA V2-Warp

Puesto que en este caso tendremos que adaptar el número de *threads* y bloques para que existan tantos *warps* como filas de la matriz, necesitaremos una configuración especial distinta de la que utilizan el resto de *kernels*. Esta situación originará también, que la parte de C y F que no tiene relación directa con la matriz dispersa tenga que ser separada a un *kernel* independiente para mantener su configuración original. En la Figura 4.38 se muestran las distintas configuraciones de *threads* y bloques que se necesitan en la nueva versión del programa, así como el esquema de invocación de los nuevos *kernels* tras la modificación. Todos los que no han sido modificados, más los nuevos que han resultado tras la separación, serán lanzados con la configuración uno, mientras que los *kernels* C1 y F1 utilizarán la configuración dos.

```
/* Grid Configuration 1 */

#define BLOCK_SIZE_1  128
#define GRID_SIZE_1   (NA%BLOCK_SIZE_1 ? ((NA/BLOCK_SIZE_1)+1) :
                      (NA/BLOCK_SIZE_1))

dim3 grid1(GRID_SIZE_1);
dim3 block1(BLOCK_SIZE_1);

/* Grid Configuration 2 */
#define WARP_SIZE     32
#define BLOCK_SIZE_2  128
#define GRID_SIZE_2   (NA%(BLOCK_SIZE_2/WARP_SIZE) ?
                      ((NA/(BLOCK_SIZE_2/WARP_SIZE))+1) :
                      (NA/(BLOCK_SIZE_2/WARP_SIZE)))
dim3 grid2(GRID_SIZE_2);
dim3 block2(BLOCK_SIZE_2);
...

conj_grad(){
  ...

  for (cgit = 1; cgit <= cgitmax; cgit++) {

    conj_grad_sum1_kernel<<<grid2, block2>>>(...);    //C1
    cudaThreadSynchronize();        //Waiting for warp reduction
    conj_grad_d1_kernel<<<grid1, block1>>>(...);      //C2
    //CPU reduction phase
    ...
  }

  conj_grad_d2_kernel<<<grid2, block2>>>(...);        //F1
  cudaThreadSynchronize();        //Waiting for warp reduction
  conj_grad_sum2_kernel<<<grid1, block1>>>(...);      //F2
  ...
}
```

Figura 4.38. CG *host*: CUDA V2-Warp

Como consecuencia de esta modificación el rendimiento ha aumentado muy considerablemente (Tabla 4.13), sobre todo para tamaños grandes del problema, como viene acostumbrando. El throughput de accesos a memoria alcanza en esta versión entre 45 y 50 GB/s más que en la versión anterior (Tabla II.10), valores que no resultan nada despreciables. Sin duda, nos encontramos ante una optimización imprescindible para este tipo de problemas.

	S	W	A	B	C
CUDA V1	1,07 X	2,80 X	3,16 X	3,76 X	8,56 X
CUDA V2-Warp	1,74 X	6,49 X	9,72 X	12,40 X	22,97 X

Tabla 4.13. CG *Speedups*: CUDA V2-Warp.

4.3.3 Mapear Half-Warps sobre filas

El hecho de haber destinado un *warp* completo por cada fila de la matriz dispersa hemos comprobado que supone un gran aumento del rendimiento. Sin embargo, cuando el número de elementos por fila no sea múltiplo del tamaño del *warp* (32 *threads*), algunos *threads* quedarán sin trabajo para el último conjunto de elementos de cada fila, provocando saltos divergentes. Una de las referencias bibliográficas consultada ([12]) sugiere utilizar la mitad de un *warp* por fila, en lugar de un *warp* completo. A nivel de accesos a memoria, la estrategia no supondría un problema, pues se trabaja con la unidad que se pretende instaurar. Sin embargo, a nivel de ejecución, la división de un *warp* en dos partes supone una inevitable serialización del mismo en un factor de dos. Al encontrarnos frente a un código con baja densidad computacional podría no suponer un problema, por lo que decidimos experimentar con esta alternativa.

Los cambios necesarios se reducen simplemente a cambiar la macro `WARP_SIZE` definida a 32, por `HALF_WARP_SIZE` definida a 16, y sus respectivas referencias a lo largo del código.

Como podemos ver en la Tabla II.10 y la Tabla II.11, la alternativa *half-warp* supone la reducción de cerca de 2.000.000 de saltos divergentes sólo para el *kernel* C, en contraposición a un aumento de 100.000.000 de *warp* serializados.

	S	W	A	B	C
CUDA V2-Warp	1,74 X	6,49 X	9,72 X	12,40 X	22,97 X
CUDA V3-Half-Warp	1,76 X	6,54 X	9,71 X	11,00 X	21,29 X

Tabla 4.14. CG *Speedups*: CUDA V3-Half-Warp

Los resultados de rendimiento parecen no ser tan optimistas como los conseguidos por los autores de la referencia. En la Tabla 4.14 podemos apreciar ligeras descendencias del rendimiento, con la excepción de un pequeño aumento para el tamaño W, por lo que parece que la cantidad de *warps* serializados neutraliza las posibles mejoras introducidas por la eliminación de saltos divergentes. En consecuencia, la optimización ha sido descartada.

4.3.4 Padding

Otra de las optimizaciones propuesta en la referencia anterior de IBM ([12]) es la de utilizar *padding* al final de cada fila de la matriz dispersa para evitar saltos divergentes. La idea consiste en introducir elementos con valores que no afecten a las operaciones que se realizarán con ellos, de cara a conseguir un número de estos que sea múltiplo del tamaño del *warp* (o *half-warp* en su caso).

La modificación necesaria, aunque simple en concepto, entraña cierta complejidad y gran cantidad de trabajo debido a la cripticidad por la que la función `makea()` se caracteriza, y el formato compactado de la matriz. Como vemos en la

Figura 4.39, bastó con añadir una pequeña fase al final de la citada función que introdujera los elementos nulos por fila que fueran necesarios para alcanzar el múltiplo más cercano al tamaño del *warp*.

```
makea(...){

  ... //a, rowstr and colidx initial configuration

  /* Padding to avoid divergence branches */

  rowstr_aux[0] = rowstr[0];

  for (j=0; j < nrows; j++){
    nzrow = rowstr[j+1] - rowstr[j];
    for (k=rowstr[j], i=rowstr_aux[j]; k < rowstr[j+1]; k++, i++){
      a_aux[i] = a[k];
      colidx_aux[i] = colidx[k];
    }

    if (nzrow % WARP_SIZE){
      padding = ((nzrow/WARP_SIZE)+1)*WARP_SIZE;
      while (nzrow < padding){
        a_aux[i] = 0.0;
        colidx_aux[i] = colidx_aux[i-1];
        nzrow++;
        i++;
      }
    }
    rowstr_aux[j+1] = rowstr_aux[j] + nzrow;
  }

  for (j=0; j < (NA+1); j++)
    rowstr[j] = rowstr_aux[j];

  for (j=0; j < NZ; j++){
    a[j] = a_aux[j];
    colidx[j] = colidx_aux[j];
  }
}
```

Figura 4.39. CG *host*: CUDA V4-Padding

Debemos destacar que esta parte no queda dentro del tiempo que es medido por NPB, aunque apenas supone modificación en el tiempo global del programa.

Como consecuencia del *padding*, los saltos divergentes pasan de 16 millones a 9 millones en el *kernel* C, según puede verse en la Tabla II.12. Esto se traduce en una mejora que se hace más sustancial con el aumento del tamaño del problema, llegando incluso hasta a más de 2 X (Tabla 4.15).

	S	W	A	B	C
CUDA V2-Warp	1,74 X	6,49 X	9,72 X	12,40 X	22,97 X
CUDA V4-Padding	1,75 X	6,68 X	10,19 X	14,28 X	25,41 X

Tabla 4.15. CG *Speedups*: CUDA V4-Padding

La alternativa también fue probada en la versión *half-warp*, donde también pudo observarse cierta mejora, aunque insuficiente para incluir dicha optimización.

4.3.5 Memoria Constante

El elevado número de invocaciones a *kernels* que los perfiles de cualquiera de las versiones de CG nos indican que se producen, y la gran cantidad de parámetros que se pasan en la mayoría de ellos, nos indujo a pensar que el uso de la memoria constante para este cometido podría suponer una alternativa que mejorara el rendimiento, ya que los parámetros solamente habría que copiarlos a este tipo de memoria una vez en toda la ejecución del programa.

Para ello seguimos lo descrito en el punto 4.1.7, dejando como parámetros directos de los *kernels* aquellos valores que se calculan en cada iteración, que resultan ser los menos. Sin embargo, no se muestran diferencias significativas en cuanto al rendimiento, sino que este se mantiene prácticamente inalterado (Tabla 4.16).

	S	**W**	**A**	**B**	**C**
CUDA V4-Padding	1,75 X	6,68 X	10,19 X	14,28 X	25,41 X
CUDA V5-Constant	1,76 X	6,63 X	10,18 X	14,28 X	25,41 X

Tabla 4.16. CG *Speedups*: CUDA V5-Constant

Al parecer nos encontrábamos equivocados con nuestra suposición, y la modificación no ha supuesto diferencia de rendimiento. Si observamos el *profiling* correspondiente a esta versión (Tabla II.13) podemos observar como la cantidad de memoria compartida utilizada por los diferentes *kernel* se ve reducida ligeramente, en función del número de parámetros que en la versión anterior se le pasaban a cada *kernel*. Puesto que esta solución es mucho más elegante, y no introduce overhead, nos quedamos con la modificación.

4.3.6 Memoria Pinned

La memoria *pinned* se ha usado en CG de manera similar al *benchmark* anterior. Sin embargo, en este caso hemos utilizado el parámetro *cudaHostAllocWriteCombined*, para caracterizar las estructuras que tienen que ver con la matriz dispersa, es decir, `rowstr`, `colidx` y `a`. De esta forma hemos podido eliminar el paso de dichos datos por los distintos niveles de caché de la CPU y acelerar aún más su transferencia. Como son estructuras de datos que no se utilizan desde la CPU, a excepción de en la función `makea()` que queda fuera de la región de tiempo, son las candidatas perfectas para aplicar la decisión mencionada.

	S	W	A	B	C
CUDA V5-Constant	1,76 X	6,63 X	10,18 X	14,28 X	25,41 X
CUDA V6-Pinned	1,79 X	6,76 X	10,35 X	14,35 X	25,49 X

Tabla 4.17. CG *Speedups*: CUDA V6-Pinned

A pesar de todo, como describíamos en la primera versión en CUDA, existen muy pocas transferencias. En su mayoría tienen lugar al principio de la ejecución, fuera de los bucles, y no son excesivamente grandes. Las transferencias relacionadas con las reducciones tampoco juegan un papel importante, por lo que Tabla 4.17 muestra como la mejora conseguida es muy poco significativa incluso para tamaños pequeños del problema. Por estos mismos motivos, la técnica de Zero Copy no ha sido tan siquiera probada.

4.3.7 Memoria de Texturas sobre P

Varias referencias de la bibliografía utilizan este tipo de memoria para cachear el vector de datos por el que se multiplica la matriz dispersa [11][12]. Comenzaremos intentando aplicar dicha técnica para el caso de nuestro algoritmo. En concreto nos centraremos en el vector P, que es el que se encuentra dentro del primer *kernel* que realiza este tipo de operación, y que es el que más tiempo de cómputo consume. Una vez nos hayamos cerciorado de los beneficios de esta técnica continuaremos con su aplicación sobre el segundo de los *kernels* que realiza la misma operación.

La teoría que se encuentra detrás de la técnica augura una mejora de rendimiento al cachear dicho vector, debido a que, al realizar la multiplicación matriz-vector, las posiciones del vector podrían ser reutilizadas, en mayor o menor medida, por cada fila de la matriz. Todo dependerá del grado de dispersión de esta, pues en el peor de los casos podría ser tan alto que no se reutilizara ninguno de los valores.

Para su aplicación, como estamos tratando con un array de elementos `double`, tendremos que descomponerlos en dos enteros para conseguir alcanzar nuestro fin. Se han seguido los pasos descritos en el punto 4.1.8. La Figura 4.40 muestra parte del código *kernel* que ha sido modificado, donde se puede ver el proceso de composición del elemento `double` a partir de los dos enteros extraídos de la textura referenciada por `pTextRef`.

Además, el uso de la textura se ha extendido al resto de *kernels* que utilizan a P en modo lectura. Hay que destacar que el vector es actualizado a través de memoria global en uno de los *kernels* posteriores a la multiplicación con la matriz, por lo que la textura deberá ser reasociada con los nuevos datos al inicio de cada iteración.

```
texture<int, 1, cudaReadModeElementType> pTextRef;

cudaChannelFormatDesc pChannel;
...

__global__ void conj_grad_sum1_kernel(){
  ...
  if (w_id < NA){             //One warp per row
    ...
    for(k=rowstr_c[w_id]+lw_id;
        k<rowstr_c[w_id+1];
        k+=nThreadsWarp){

      index = 2 * colidx_c[k];
      sum[lid] = sum[lid] + a_c[k] *
                  __hiloint2double( tex1Dfetch(pTextRef,index+1),
                                    tex1Dfetch(pTextRef, index));
    }
    ...
  }
  ...
}
```

Figura 4.40. CG *kernel* C: CUDA V7-TextP

Echándole un vistazo al rendimiento obtenido (Tabla 4.18), comprobamos que los resultados no mejoran los de la versión anterior, sino justo todo lo contrario. Para todos los tamaños, excepto el S, podemos apreciar una degradación del rendimiento, que se acentúa aún más conforme el problema aumenta. Por tanto, el uso de memoria de texturas no se ha extendido al segundo de los *kernels* candidatos y se ha descartado esta modificación.

	S	W	A	B	C
CUDA V6-Pinned	1,79 X	6,76 X	10,35 X	14,35 X	25,49 X
CUDA V7-TextP	1,87 X	6,58 X	9,68 X	10,61 X	21,60 X

Tabla 4.18. CG *Speedups*: CUDA V7-TextP

Si observamos el *profiling* de esta versión (Tabla II.15) podemos encontrar un número de fallos de caché de textura considerablemente elevado, superando incluso a los aciertos en la mayoría de los casos. Es posible que este mal rendimiento tengan que ver con la altísima dispersión de la matriz (escasos elementos por columna y muy pocos coinciden en posición entre filas) y del enorme tamaño del vector P, aunque, sin duda, si la estructura no fuese modificada en cada iteración, el número de aciertos de caché aumentaría al encontrar datos válidos en ella pertenecientes a la iteración anterior.

4.3.8 Memoria de Texturas sobre *rowstr*, *colidx* y *a*

Tras la iniciativa propia de crear diferentes versiones con distintas estructuras mapeadas sobre la memoria de texturas, y contra todo pronóstico realizado en los artículos

referenciados anteriormente (publicados por NVIDIA e IBM), hemos conseguido una configuración con la que obtenemos una mejora de rendimiento más que aceptable.

En concreto, las estructuras de datos que pasan a través de la memoria de texturas son `rowstr`, `colidx` y la matriz dispersa `a`. Para ello se utilizan tres texturas diferentes, de las cuales las dos primeras serán de tipo `int` y la última, también del mismo tipo, contendrá `doubles` de igual manera que el apartado anterior. Se utilizará un canal de datos para cada una de las texturas.

El *binding* en este caso se realiza antes de entrar en ninguno de los bucles principales del programa, porque las estructuras no se verán modificadas durante el cómputo del algoritmo. De esta forma la reutilización de datos en cada textura se llevará a cabo incluso entre iteraciones. Su uso se extiende tanto al *kernel* C1 como al F1. En la Figura 4.41 se muestra el código del primero, donde el uso de cada textura se ha coloreado de manera diferente. El segundo tendría una estructura similar.

```
__global__ void conj_grad_sum1_kernel(unsigned int my_NA){

  ...

  if (w_id < NA){                    //One warp per row
    ...
    for (k = tex1Dfetch(rowstrTextRef, w_id) + lw_id;
         k < tex1Dfetch(rowstrTextRef, w_id + 1);
         k += nThreadsWarp){

      sum[lid] = sum[lid] + __hiloint2double(
                                    tex1Dfetch(aTextRef, 2*k+1),
                                    tex1Dfetch(aTextRef, 2*k))
                                * p_c[tex1Dfetch(colidxTextRef,k)];
    }
    ...  //sum warp reduction with warp size = 32
  }
}
```

Figura 4.41. CG *kernel* C1 (y F1): CUDA V8-TextRCA

Como ya preveíamos, los resultados han sido satisfactorios, llegando a conseguir hasta casi 4 X más de *speedup* para el tamaño del problema C (Tabla 4.19). Aún para tamaños más pequeños también se ha producido cierta mejora, excluyendo al tamaño S, al que podríamos considerar que esta técnica no le ha afectado demasiado debido a sus reducidas dimensiones.

	S	W	A	B	C
CUDA V6-Pinned	1,79 X	6,76 X	10,35 X	14,35 X	25,49 X
CUDA V8-TextRCA	1,76 X	7,16 X	11,38 X	16,92 X	29,38 X

Tabla 4.19. CG *Speedups*: CUDA V8-TextRCA

4.3.9 Enfoque Multi-GPU

Como hemos podido ver en los *profilings* de las distintas versiones de CG, más del 99% del tiempo se emplea en los dos *kernels* que trabajan sobre el producto matriz-vector.

```
/* Grid Configuration 2 */

#define WARP_SIZE     32
#define BLOCK_SIZE_2  128
#define AUX           ((NA/NUM_GPUS)%(BLOCK_SIZE_2/WARP_SIZE) ?
                      (((NA/NUM_GPUS)/(BLOCK_SIZE_2/WARP_SIZE))+1)
                      :((NA/NUM_GPUS)/(BLOCK_SIZE_2/WARP_SIZE)))
#define GRID_SIZE_2   ((NA%NUM_GPUS) ? AUX+1 : AUX)
#define NTHREADS_2    (GRID_SIZE_2*BLOCK_SIZE_2)

dim3 grid2(GRID_SIZE_2);
dim3 block2(BLOCK_SIZE_2);

conj_grad(){
  ...
  for (cgit = 1; cgit <= cgitmax; cgit++) {
    ...
    #pragma omp master
    {
      cudaMemcpy(aux_h, p_d,(NA+2) * sizeof(double),
                 cudaMemcpyDeviceToHost);
    }                                              Scatter Operation

    #pragma omp barrier

    if (my_device != 0)
      cudaMemcpy(p_d, aux_h, (NA+2) * sizeof(double),
                 cudaMemcpyHostToDevice);

    conj_grad_sum1_kernel<<<grid2, block2>>>(my_NA, my_device);

    if (my_device != 0){
      cudaMemcpy(aux_h + (NA/NUM_GPUS)*my_device, w_d,
                 my_NA * sizeof(double), cudaMemcpyDeviceToHost));
    }
                                                   Gather Operation
    #pragma omp barrier

    #pragma omp master
    {
      cudaMemcpy(w_d + my_NA, aux_h + my_NA, (NA-my_NA) *
                 sizeof(double), cudaMemcpyHostToDevice));

      conj_grad_d1_kernel<<<grid1, block1>>>();
      ...
    }
    ...
  }
  ...
}
```

Figura 4.42. CG *host*: V9-Enfoque Multi-GPU

Puesto que existen dependencias entre los diferentes *kernels*, cuando el flujo de ejecución se vaya a extender sobre varios dispositivos, en primer lugar necesitaremos distribuir ciertos datos a estos (operación *Scatter*), y tras la ejecución, será necesario mover los resultados de cada GPU al dispositivo principal que continuará en solitario (operación *Gather*). Para desarrollar este esquema nos ayudaremos del descrito en la Figura 4.14, pero realizando ciertas modificaciones.

La Figura 4.42 muestra el código resultante para el *kernel* C1, donde debemos destacar la obligatoria intervención del *host* como punto intermedio de las comunicaciones entre los diferentes dispositivos, así como las sincronizaciones a nivel OpenMP que resultan imprescindibles para que los intercambios de información se lleven a cabo con éxito. También, puesto que sólo una GPU computará la mayoría de los *kernels*, no todas las estructuras necesitarán ser reservadas en todas las GPUs, llevando así una reserva de memoria selectiva en función del dispositivo que se trate.

```
#define NUM_GPUS    2

__global__ void conj_grad_sum1_kernel(unsigned int my_NA){

  ...

  if (w_id < my_NA){
    ...
    for (k = tex1Dfetch(rowstrTextRef, w_id) + lw_id;
         k < tex1Dfetch(rowstrTextRef, w_id + 1);
         k += nThreadsWarp){

      sum[lid] = sum[lid] + __hiloint2double(
                                  tex1Dfetch(aTextRef, 2*k + 1),
                                  tex1Dfetch(aTextRef, 2*k))
                            * p_c[tex1Dfetch(colidxTextRef,k)];
    }
    ...  //sum warp reduction with warp size = 32
  }
}
```

Figura 4.43. CG *kernel* C1 (y F1): V9. Enfoque Multi-GPU

Uno de los problemas iniciales con el que nos encontramos fue el de cómo extender el uso de texturas a varias GPUs, ya que la declaración de las mismas se hace globalmente y no parecía poder especificarse cuál iría a qué dispositivo. Apenas encontramos información al respecto, con lo que acabamos concluyendo que la declaración global de una sola textura se asocia independientemente a todos los dispositivos que terminen usándola. Lo mismo sucede con la memoria constante. También descubrimos que actualmente CUDA no soporta el uso de arrays de texturas de ningún tipo.

Si nos centramos en el código del dispositivo (Figura 4.43), hemos tomado una decisión importante para simplificar las modificaciones a realizar. La división de las estructuras de datos relacionadas con la matriz podría requerir de cambios considerables, llegando incluso a introducir cierto overhead. Es el caso de la estructura

`colidx`, ya que esta se referencia globalmente por los valores almacenados en `rowstr`, y tendríamos que tener en cuenta desplazamientos irregulares en función de los elementos que le tocaran a cada dispositivo. Lo mismo sucede con los valores de la matriz `a`. Por todos estos motivos decidimos no llevar a cabo la división de ambas estructuras, sino que procedimos a copiarlas por completo a cada una de las GPUs involucradas en el cálculo, realizando solamente la división de la estructura `rowstr`. Puesto que estas copias solamente son necesarias una vez durante toda la ejecución podemos asegurar que no supondrá una diferencia significativa de rendimiento, al menos para tamaños grandes del problema.

En el *speedup* recogido en la Tabla 4.20 podemos ver el escalado del programa al utilizar dos GPUs. Este queda lejos de un escalado tan óptimo como el conseguido por EP, pero resulta aceptable para tamaños grandes, sobre todo teniendo en cuenta el nivel de comunicación que existe entre los diferentes dispositivos. Es muy posible que un número más elevado de estos termine colapsando la zona de comunicaciones y el rendimiento comience a decaer.

	S	W	A	B	C
CUDA V8-TextRCA	1,76 X	7,16 X	11,38 X	16,92 X	29,38 X
CUDA V9-Multi-GPU	1,18 X	3,99 X	6,59 X	19,01 X	41,54 X

Tabla 4.20. CG *Speedups*: CUDA V9-Multi-GPU

Los tamaños más pequeños del problema no disponen de suficiente trabajo para repartir entre varios dispositivos, lo que, sumado al overhead de las comunicaciones, terminan por ocasionar peores resultados que la versión Mono-GPU.

4.3.10 Conclusiones

La Tabla 4.21 contiene el resumen de los *speedups* de CG de todas las versiones anteriores. Podemos apreciar como la primera de las versiones implementada en CUDA obtiene unos resultados que dejan bastante que desear si la comparamos con la más potente de las CPUs. Esta sensación no cambiará con el resto de versiones para tamaños del problema pequeños, como S y W, los cuales no son capaces de generar suficiente trabajo como para que compense el uso de una GPU.

En cuanto a las optimizaciones, es sorprendente el aumento de rendimiento que puede conseguirse simplemente cambiando la estrategia de mapeo de *threads* (V2-Warp). La modificación requiere de cierto estudio y grandes cambios, pero sin duda merece la pena la adaptación para solventar problemas de acceso a memoria global cuando no sea posible utilizar otro tipo de memoria con menos restricciones.

La introducción de *padding* sobre cada fila de la matriz para evitar saltos divergentes (V4-Padding) también ha supuesto cierto aumento de rendimiento, aunque entrañaba una complejidad no despreciable. Podemos concluir con que este factor también es uno de los que pueden influir enormemente en el tiempo de ejecución, por lo

que debemos percatarnos de en qué ocasiones el número de saltos es muy elevado para tratar de reducirlo. En muchas ocasiones no será posible, al menos de forma trivial.

	S	W	A	B	C
Opteron Serie	1,00 X	1,00 X	1,00X	1,00 X	1,00 X
Opteron 4 Cores	3,65 X	1,65 X	1,90 X	2,31 X	2,40 X
Xeon 8 Cores	5,18 X	9,97 X	6,64 X	3,91 X	9,88 X
CUDA V1	1,07 X	2,80 X	3,16 X	3,76 X	8,56 X
CUDA V2-Warp	**1,74 X**	**6,49 X**	**9,72 X**	**12,40 X**	**22,97 X**
CUDA V3-Half-Warp	1,76 X	6,54 X	9,71 X	11,00 X	21,29 X
CUDA V4-Padding	1,75 X	6,68 X	**10,19 X**	**14,28 X**	**25,41 X**
CUDA V5-Constant	1,76 X	6,63 X	10,18 X	14,28 X	25,41 X
CUDA V6-Pinned	1,79 X	6,76 X	10,35 X	14,35 X	25,49 X
CUDA V7-TextP	1,87 X	6,58 X	9,68 X	10,61 X	21,60 X
CUDA V8-TextRCA	1,76 X	7,16 X	**11,38 X**	**16,92 X**	**29,38 X**
CUDA V9-Multi-GPU	1,18 X	3,99 X	6,59 X	19,01 X	41,54 X

Tabla 4.21. CG: Resumen *Speedups*

El uso de texturas, aunque complejo, permite obtener grandes beneficios si la situación se presta a su uso y acertamos en la elección de sobre qué estructuras debemos usarlas. Hemos podido comprobar como una incorrecta aplicación de esta memoria puede empeorar el rendimiento de la aplicación. Un análisis en profundidad de los patrones de acceso que el algoritmo realiza sobre una estructura nos permitirá descartarla como candidata y ahorrar tiempo. Sin embargo, existirán casos en los que deberemos recurrir al ensayo y error.

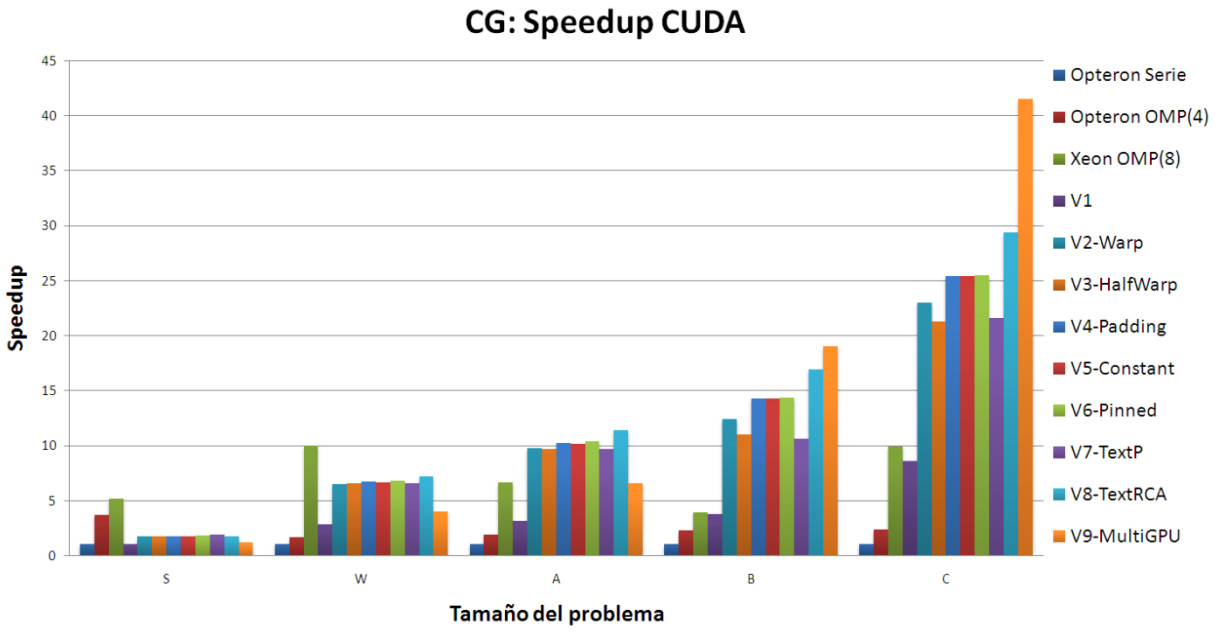


Figura 4.44. CG: Gráfica de *Speedups*

Como podemos ver también en la Figura 4.28, el uso de varias GPUs también resulta ser un éxito para los dos tamaños más grandes del problema, aunque auguramos cierto colapse del mismo si aumentamos demasiado el número de dispositivos. La culpa de ello la tiene la comunicación que es necesaria entre diferentes GPUs. La aplicación no escala de manera tan eficiente como EP, pero sin embargo, sí lo hace lo suficiente como para recomendar su extensión a varios dispositivos en lo referente a tamaños grandes del problema.

4.4 Fourier Transform Dock (FTDock)

Esta aplicación de dinámica de proteínas ha sido desarrollada por el *Biomolecular Modelling Laboratory* situado en Londres, Reino Unido, con la finalidad de estudiar y desarrollar posibles medicamentos contra el Cáncer. La creciente cantidad de proteínas individuales almacenadas en bases de datos, y el bajo número de relaciones estudiadas entre varias de ellas hacen de la predicción del acoplamiento de proteínas un método teórico muy relevante para el mundo científico. FTDock (Fourier Transform Dock) lleva a cabo un acoplamiento de los cuerpos rígidos de dos biomoléculas para predecir su correcta asociación geométrica. Para ello implementa una versión del algoritmo Katchalski-Katzir basado en transformadas de Fourier, que además incorpora funciones electrostáticas que han sido desarrolladas por el propio laboratorio.

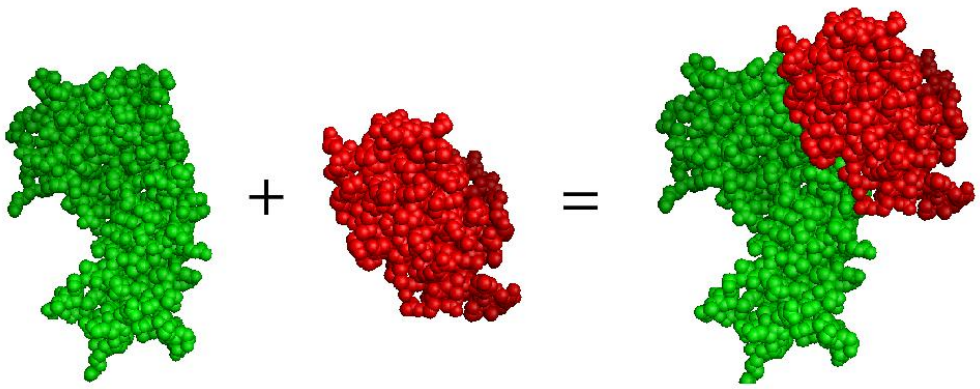


Figura 4.45. Representación gráfica del acoplamiento de dos proteínas

El algoritmo Katchalski-Katzir es un algoritmo puramente geométrico que mapea las moléculas sobre *grids* en 3D, donde cada punto del *grid* es marcado como "fuera de la molécula, "sobre la superficie de la molécula" o "dentro de la molécula". El algoritmo incrementa el contacto entre las distintas superficies de las moléculas minimizando el volumen de solapamiento de las mismas. La Figura 4.45 muestra gráficamente un ejemplo de acoplamiento de dos proteínas.

Para alcanzar dicho objetivo, una de las moléculas se mantiene estática (inmóvil) y sólo la otra es la que se va desplazando para conseguir el acoplamiento. A este acoplamiento o alineamiento se le asigna una puntuación que es fácilmente calculable a

partir de los parámetros anteriores, pero existen muchas posibilidades a la hora de acoplar dos moléculas.

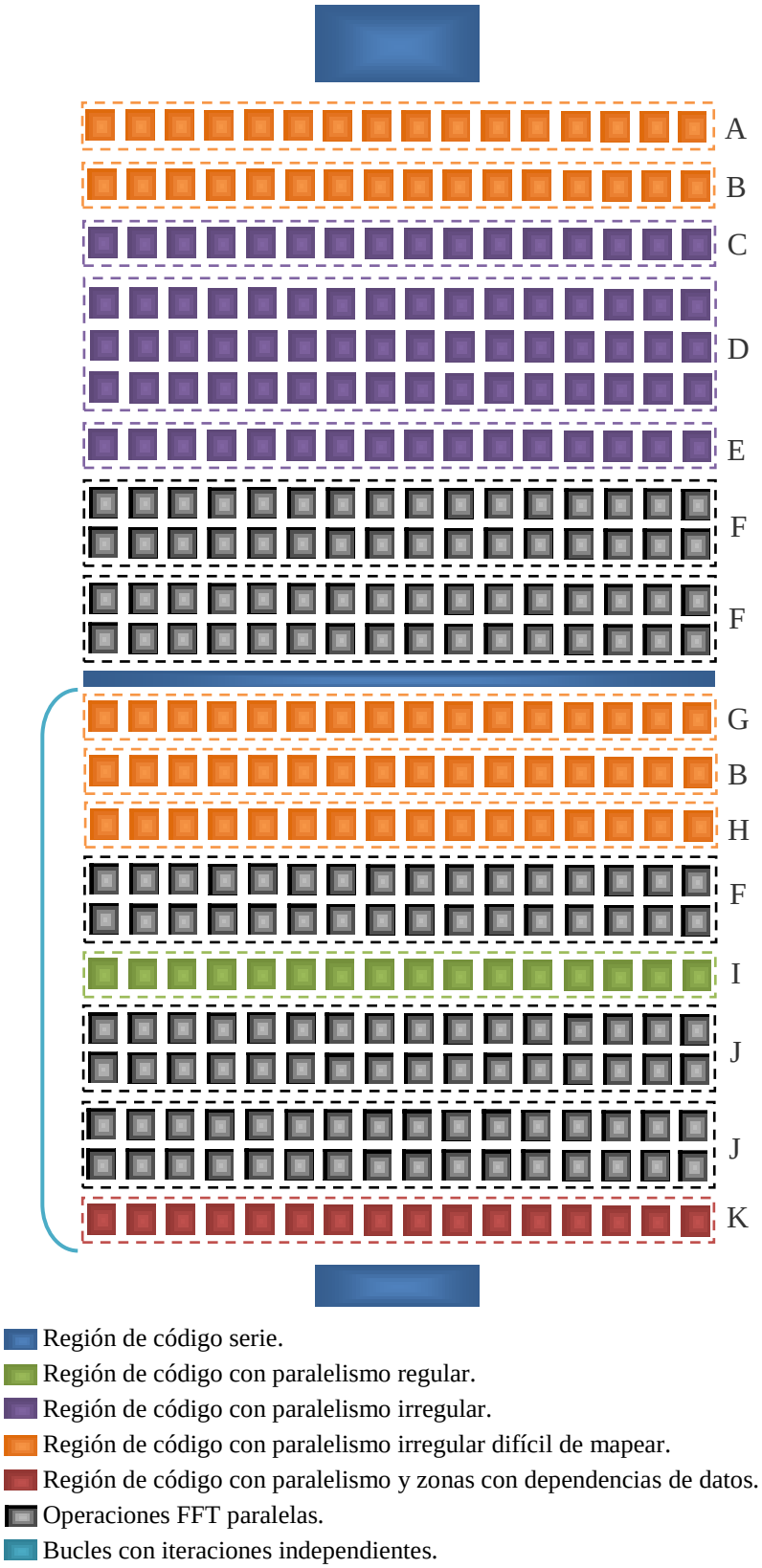


Figura 4.46. Estructura de la aplicación FTDock

Para calcular la puntuación de un elevado número de alineaciones eficientemente se aplica la transformada rápida de Fourier (FFT) sobre ambos *grids*. Teniendo los *grids* en formato FFT es posible calcular la puntuación de varias alineaciones muy rápidamente. De esta forma, FTDock permitirá calcular la puntuación de gran cantidad de alineaciones, dejando para la salida aquellas de mayor valor.

La Figura 4.46 describe la estructura de la aplicación, donde puede apreciarse a simple vista la complejidad introducida por sus dimensiones y las diferentes características del paralelismo de sus regiones. Pese a todo, el nivel de paralelismo es superior al 99%, donde el mayor tiempo de ejecución es consumido por las funciones que se encuentran dentro del bucle principal de la aplicación, y en concreto por las encargadas de realizar las operaciones FFT.

De cara a computar estas operaciones se hace uso de la conocida librería FFTW [50], que implementa una amplia variedad de funciones relacionadas con la transformada de Fourier.

En la Figura 4.48 mapeamos las distintas regiones de la estructura de la aplicación sobre parte del código fuente original de la misma. La mayoría de este queda desplegado en distintas funciones separadas en varios ficheros, debido a su extensión.

Las tres zonas serie más extensas que podemos encontrar a lo largo de toda la aplicación se corresponden con las partes del código encargadas de las declaraciones de datos, reserva de memoria, procesamiento de los parámetros de entrada, carga de los ficheros de entrada, escrituras en ficheros de salida, etc.

La región I, caracterizada como "región con paralelismo regular", hace referencia a una región con tres bucles anidados que llevan a cabo una operación de convolución sobre el *grid* de la proteína móvil (color verde). Cada una de las iteraciones de estos bucles es completamente independiente del resto, y además ejecutan el mismo número de operaciones sobre posiciones diferentes del *grid*. Su estructura puede observarse en la Figura 4.47.

```
for( fx = 0 ; fx < x_size; fx ++ ) {
  for( fy = 0 ; fy < y_size; fy ++ ) {
    for( fz = 0 ; fz < z_size; fz ++ ) {
      data[fx][fy][fz] ...
    }
  }
}
```

Figura 4.47. FTDock: Estructura de código. Paralelismo regular

Aquellas partes de código denominadas regiones "con paralelismo irregular" se caracterizan por ser bucles como los anteriores o similares, donde cada una de las iteraciones puede computar un número notablemente distinto de operaciones con respecto a otra iteración (color púrpura). Esto dependerá de factores pertenecientes a la propia molécula, por lo que el comportamiento es imposible de predecir.

```
main() {
  ... /* Declarations, Memory Allocation, Command Line Parse,
         Rescue Option, Load Input Files and Assign Charges */

  /* Store new structures centered on Origin */
  translate_structure_onto_origin( ... );                                  A
  ...

  /* Create FFTW plans */
  ...

  /* Discretise and surface the Static Structure */
  discretise_structure(Origin_Static_Structure,static_grid,... );          B
  surface_grid(static_grid, ...) ;                                         C

  /* Calculate electic field at all grid nodes */
  if( electrostatics == 1 ) {
    electric_field( Origin_Static_Structure,static_elec_grid,...);         D
    electric_field_zero_core(static_elec_grid,static_grid,...);            E
  }

  /* Fourier Transform the static grids */
  fftwf_execute_dft_r2c(p,static_grid,...);                                F
  if(electrostatics==1) fftwf_execute_dft_r2c(p,static_elec_grid,...);     F

  ... /* Store paramaters in case of rescue */

  /* Main program loop */
  for(rotation = 1; rotation <= Angles.n ; rotation ++ ) {
    /* Rotate Mobile Structure */
    Rotated_at_Origin_Mobile_Structure =                                   G
      rotate_structure( Origin_Mobile_Structure , Angles...) ;

    /* Discretise the rotated Mobile Structure */
    discretise_structure                                                   B
      (Rotated_at_Origin_Mobile_Structure ,mobile_grid,...);

    /* Electic point charge approximation onto grid calculations */
    if( electrostatics == 1 ) {
      electric_point_charge                                                H
        (Rotated_at_Origin_Mobile_Structure, mobile_elec_grid,...);
    }

    /* Forward Fourier Transforms */
    if( electrostatics == 1 )ftwf_execute_dft_r2c(p,mobile_elec_grid,..    F

    /* Convolution of the two sets of grids */
    ... //Pure SIMD Region (3 nested loops)                                I

    /* Reverse Fourier Transform */
    fftwf_execute_dft_c2r(pinv, multiple_fsg,(fftwf_real *)multiple_fsg)   J
    if(electrostatics==1)fftwf_execute_dft_c2r(pinv, multiple_elec_fsg,..  J

    ... /* Get best scores. (Loops with dependences) */                    K
  }
  ... /* Write Output Files */
}
```

Figura 4.48. FTDock: Resumen de código base para la CPU

Existen, además, regiones en las que la irregularidad alcanza a los propios bucles más externos (color naranja), haciendo especialmente difícil el mapeo entre estos y los *threads* de CUDA. La estructura de estas regiones es similar a la mostrada por la Figura 4.49, donde el número de iteraciones depende del número de residuos y átomos por residuo de la proteína de turno.

```
for(residue = 1; residue <= This_Structure.length; residue++){
  for(atom=1; atom<=This_Structure.Residue[residue].size; atom++){
    data[residue][atom]...
  }
}
```

Figura 4.49. FTDock: Estructura de código. Paralelismo irregular difícil de mapear

También nos encontramos con una región notablemente especial (color rojo), que es donde se van acumulando aquellos acoplamientos con puntuaciones más elevadas, y descartando aquellos de menor valor. Esta zona se caracteriza por estar compuesta de ciertas partes de las que podemos extraer paralelismo, y otras que deben ejecutarse completamente en serie, debido a dependencias de datos existentes.

Puesto que el código original viene escrito 100% en serie, carece de sentido la comparación entre los distintos *multicores*. De esta manera, inicialmente sólo se ha ejecutado el algoritmo sobre el procesador Opteron, utilizando dos entradas diferentes con configuraciones de *grid* de tamaño 128 y 256 para ambas, y simple precisión (*floats*) para los resultados. El tiempo de ejecución de cualquiera de ellas resulta especialmente prolongado sobre uno solo de los *cores* de una sola CPU, empleando 20 minutos la más rápida y 10,7 horas la más lenta.

4.4.1 Versión Inicial en CUDA

Tras un primer análisis de la aplicación serie mediante *gprof* (Tabla II.19), nos encontramos con que las funciones relacionadas con las operaciones FFT suponen casi la totalidad del tiempo de ejecución. Así pues, decidimos dedicar una primera versión a migrar solamente estas operaciones.

Para ello hicimos uso de la librería CUFFT [6], que viene a ser la implementación en CUDA de la librería FFTW, con ciertas salvedades. De esta manera no hicieron falta grandes cambios en el código fuente más allá de transferir datos entre el *host* y la GPU y cambiar el nombre de las funciones, pues la filosofía seguida es similar en ambos enfoques:

1. Se crea un plan de ejecución:
   a. FFTW

```
p = fftwf_plan_dft_r2c_3d( x_size, y_size, z_size,
                           input_grid, output_grid,
                           FFTW_MEASURE);
```

b. CUFFT

```
cufftPlan3d(&p_cufft, x_size, y_size, z_size, CUFFT_R2C);
```

2. Se ejecutan las operaciones:
   a. FFTW

```
fftwf_execute_dft_r2c(p, input_grid, output_grid);
```

   b. CUFFT

```
cudaMemcpy(input_grid, cudaMemcpyHostToDevice);
cufftExecR2C(p_cufft, input_grid, output_grid);
cudaMemcpy(output_grid, cudaMemcpyDeviceToHost);
```

3. Se destruye el plan:
   a. FFTW

```
fftwf_destroy_plan(p);
```

   b. CUFFT

```
cufftDestroy(p_cufft);
```

Cabría destacar que la librería CUFFT lanza numerosos *kernel* a la GPU por cada una de las operaciones que se le solicitan. La configuración de *grid* y bloque que se utiliza en cada caso queda fuera de nuestro control, aunque esperamos que sea la más adecuada.

Los resultados obtenidos con respecto al esfuerzo de realizar la modificación, una vez que se tiene el conocimiento de cómo funcionan las librerías, son bastante buenos, pues ganamos entre un 79% y un 157% de rendimiento con respecto a la versión serie, según vemos en la Tabla 4.22. Si nos damos cuenta, hemos conseguido la ganancia anterior aún sin tener que saber qué operaciones realiza cada una de las funciones que hemos visto que componen el algoritmo. Esto es algo muy positivo.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
Opteron Serie	1,00 X	1,00 X	1,00 X	1,00 X
CUDA V1-CUFFT	1,79 X	2,34 X	1,87 X	2,57 X

Tabla 4.22. FTDock *Speedups*: CUDA V1-CUFFT

4.4.2 Migración Completa a la GPU. Evitando Transferencias de Datos

Si observamos el *profiling* de la versión anterior (Tabla II.20), podemos ver como las transferencias de datos necesarias para usar la librería CUFFT suponen más tiempo de ejecución que alguna de las operaciones que la propia CUFFT realiza. Para evitar que estas transferencias sean necesarias de manera tan recurrente, deberíamos migrar el resto de las funciones del algoritmo a la GPU. Con ello conseguiríamos que todo se

ejecutase en el dispositivo, evitando las costosas transferencias de datos. Además estaríamos acelerando las citadas zonas del código.

Comenzando por la región A, la función `translate_structure_onto_origin()` lleva a cabo el desplazamiento de las estructuras para que sus coordenadas sean referentes a las de origen (0, 0, 0). Dicha función será sustituida por `translate_device_structure_onto_origin()` que, pese a su inapropiado nombre, además, recorre el número total de residuos y átomos de la estructura y va mapeando cada uno de ellos a un determinado *thread* GPU. Este mapeo consiste básicamente en asignar a cada identificador global del *thread* un par de índices que referencien al residuo y al átomo seleccionado, tal y como podemos ver en la Figura 4.50. Los datos se almacenan directamente en la memoria del dispositivo para su posterior uso. También se aprovecha para contabilidad cuál es el número total de átomos de la estructura. Como resultado, conseguiremos disponer de dos estructuras de datos (`gid2residue` y `gid2atom`) y una variable con el número de átomos para cada una de las dos estructuras del algoritmo: la proteína estática y la dinámica.

```
translate_device_structure_onto_origin
      (struct Structure This_Structure, int * total_atoms,
       int ** gid2residue_d, int ** gid2atom_d) {

  ... //Translation on CPU.

  /* Associate CUDA THREAD GID to Residue and Atom */
  gid2atom    = (int *) malloc (atoms * sizeof(int));
  gid2residue = (int *) malloc (atoms * sizeof(int));

  for(residue=1, k=0; residue<=New_Structure.length; residue++) {
    for(atom=1; atom<=New_Structure.Residue[residue].size;atom++, k++){
        gid2atom[k] = atom;
        gid2residue[k] = residue;
    }
  }

  cudaMalloc((void **) gid2residue_d, atoms * sizeof(int));
  cudaMalloc((void **) gid2atom_d,    atoms * sizeof(int));

  cudaMemcpy((*gid2residue_d), gid2residue,
              atoms * sizeof(int), cudaMemcpyHostToDevice);
  cudaMemcpy((*gid2atom_d),    gid2atom,
              atoms * sizeof(int), cudaMemcpyHostToDevice);

  free (gid2atom);  free (gid2residue);
  (*total_atoms) = atoms;
}
```

Figura 4.50. FTDock V2. Región A: *translate_device_structure_onto_origin*

Antes de comenzar a ejecutar código sobre la GPU necesitamos hallar la forma de transferir las estructuras `Structure`, que describen a cada proteína, al dispositivo. Esto no va a ser tarea fácil porque dentro de las citadas `Structure` nos encontramos un número de punteros variable, en función de la entrada, que hacen referencia a las

estructuras que describen a los residuos de cada proteína (`Amino_Acid`). A su vez, cada residuo contiene nuevamente un número indeterminado de punteros hacia las estructuras que representan a los átomos de cada residuo (`Atoms`).

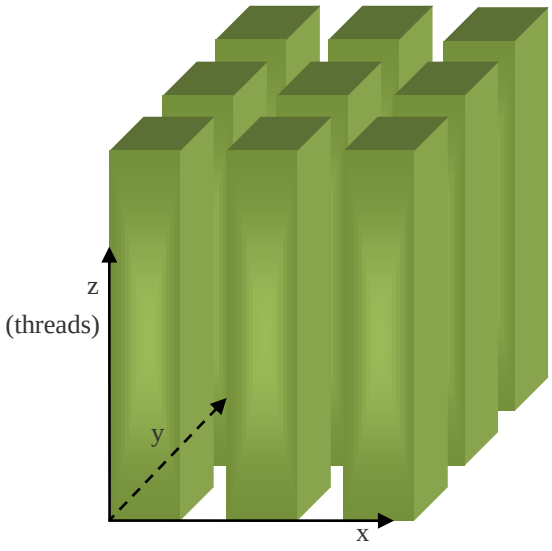
Si decidiéramos mover sin más la primera de ellas, tendríamos como resultado una estructura en memoria del dispositivo con punteros dirigidos al espacio de direcciones del *host*. Por tanto, para llevar a cabo una copia completa y correcta de los datos, necesitaremos agrupar todos los de cada tipo de estructura y moverlos conjuntamente al dispositivo, modificando previamente los punteros correspondientes desde el *host*, para que ya sean copiados apuntando a la región del dispositivo donde se encontrarán los datos originales. La Figura 4.51 muestra parte del código que implementa lo descrito anteriormente (función `copy_structure_to_device()`). Las partes de igual color se corresponden a la modificación de los punteros previa transferencia y a la transferencia propiamente dicha, respectivamente.

```
struct Amino_Acid * amino_acids_h;        //Residues
struct Atom ** atoms_h;struct Amino_Acid * amino_acids_d ;
struct Atom * atoms_d;
struct Structure * structure_d;

/* Structure and Amino Acids (Residues) Pool Allocation*/
atoms_h = (struct Atom **) malloc(...);
cudaMalloc((void **)&structure_d, sizeof(Structure)));
cudaMalloc((void **)&amino_acids_d,...);

/* Structure Transference */
amino_acids_h = structure_h.Residue;
structure_h.Residue = amino_acids_d;
cudaMemcpy(structure_d, &structure_h, ..., cudaMemcpyHostToDevice);
structure_h.Residue = amino_acids_h;

/* Atoms Pool Allocation */
for( num_atoms = 0, residue = 1 ; residue <= structure_h.length ; residue ++ )
  num_atoms += structure_h.Residue[residue].size + 1;

cudaMalloc((void **) &atoms_d, num_atoms * sizeof(Atom));

/* Atoms Pool Transference */
for ( offset = 0, residue = 1; residue <= structure_h.length ; residue ++ ){
  atoms_h[residue] = structure_h.Residue[residue].Atom;
  structure_h.Residue[residue].Atom = atoms_d + offset;
  cudaMemcpy(atoms_d + offset, atoms_h[residue], ..., cudaMemcpyHostToDevice);
  offset += (structure_h.Residue[residue].size + 1);
}

/* Amino Acids Pool Transference */
cudaMemcpy(amino_acids_d, structure_h.Residue, ..., cudaMemcpyHostToDevice);
 ...
return structure_d ;
```

Figura 4.51. FTDock V2: *copy_structure_to_device*

Esta función será utilizada en tres ocasiones para mover al dispositivo las estructuras `Origin_Static_Structure`, `Origin_Mobile_Structure` y `Rotated_at_Origin_Mobile_Structure`.

Con los datos ya sobre la GPU, procedemos a la migración de las diferentes funciones y regiones del algoritmo. Para aquellas que poseen una estructura externa similar, caracterizada por tres bucles (regiones con paralelismo regular e irregular, Figura 4.47), necesitaremos hacer uso de *grids* de *kernel* multidimensionales. En concreto, crearemos *grids* con bloques distribuidos en las dos primeras dimensiones (x e y), donde sus *threads* seguirán manteniendo una sola dimensión. Con esta configuración asociaremos las dos dimensiones de los bloques con los dos bucles más externos, dejando para el más interno la única dimensión que poseen los *threads*. Si imaginamos un prisma rectangular, los bloques, de tamaño (1, 1, |z|) se repetirán |x| e |y| veces respectivamente en cada eje hasta cubrir su totalidad. Internamente, cada bloque estará compuesto de |z| *threads* que cubrirán por completo la susodicha dimensión. La Figura 4.52 muestra todo lo descrito al respecto con un ejemplo gráfico y el código fuente genérico asociado.




```
cuda_block.x = x_size;
cuda_grid.x  = y_size;
cuda_grid.y  = z_size;

kernel<<<cuda_grid, cuda_block>>>(...);
```

```
__global__ void kernel(...)
{
  int fz = threadIdx.x;
  int fy = blockIdx.y;
  int fx = blockIdx.x;

  data[fx][fy][fz] ...
}
```

Figura 4.52. FTDock V2: Estructura y configuración de *kernels*.
Paral. reg. e irreg

Centrándonos ahora en las zonas irregulares difíciles de mapear a *threads* CUDA, gracias a las estructuras `gid2residue` y `gid2atom` y a la variable que creamos en la función `translate_device_structure_onto_origin()` (Figura

4.50), podremos asignar un *thread* por cada átomo de la proteína. Así, el *kernel* de dichas regiones tendrá un aspecto similar al de la Figura 4.53. La opción de utilizar un mapeo similar al empleado en la matriz dispersa de CG no es viable en este caso, ya que el número de átomos por residuo resulta bastante inferior al número de *threads* por *warp*.

Las configuraciones de *grid* y bloque, en este caso, continuarán manteniendo una sola dimensión. Sin embargo, debido al no excesivo nivel de paralelismo de las regiones mencionadas, dispondremos de bloques de solamente 32 *threads*, intentando conseguir un número lo más alto posible con el objetivo de cubrir cada uno de los multiprocesadores.

```
__global__ void kernel( int * gid2residue,
                        int * gid2atom,
                        int totalAtoms,...)
{

  if (gid < totalAtoms){

    residue = gid2residue[gid];
    atom    = gid2atom[gid];

    data[residue][atom] ...
}
```

Figura 4.53. FTDock V2: Estructura de *kernels*.
Paralelismo irreg. difícil de mapear

La región más peliaguda ha sido la relacionada con las dependencias de datos. La no migración de ésta habría conllevado el tener que seguir realizando grandes transferencias entre el *host* y el dispositivo por cada iteración del bucle principal del programa.

Como vemos en la Figura 4.54, aunque existe una pequeña región paralela regular que requiere de configuración de *kernel* multidimensional, nos encontramos con dos zonas en las que todo el trabajo debe realizarlo un solo *thread* de cada uno de los bloques. Precisamente estas zonas no se caracterizan por ser computacionalmente livianas, sino que en ellas podemos encontrar varios bucles y saltos condicionales. Como resultado, conseguimos que la reducción necesaria tras la ejecución de dicho código sea de tan sólo unos pocos elementos (en función del número de bloques).

Tras las modificaciones descritas, tenemos una versión de FTDock completamente migrada a la GPU. Todas y cada una de ellas han sido probadas individualmente, cerciorándonos de que su introducción suponía mejorar el rendimiento con respecto a la alternativa de mantener las transferencias de un gran volumen de datos. Obviamente no todas las mejoras han sido de la misma magnitud, pues la última modificación que hemos descrito ha supuesto una aceleración casi despreciable, que ha sido multiplicada tras la eliminación de las transferencias de gran tamaño.

```
__global__ void get_best_scores_kernel (...){

  int lid = threadIdx.x;
  __shared__ extern float my_scores[];

  ... //Parallel Zone

  /* Only thread 0 per block will compute */
  if (lid == 0){
    for (i=0; i < keep_per_rotation; i++){
      ...
    }
    for (i=0; i < blockDim.x; i++){
      if( my_scores[i] > value[keep_per_rotation-1] ) {
        ...
        while( (my_scores[i] > value[j] ) && ( j >= 0 ) ) {
          ...
        }
        ...
      }
    }
  }
  ...
  for (i=0; i < keep_per_rotation; i++){
    /* Only one thread per block per iteration will compute */
    if (pos[i] == lid){
      ...
    }
  }
}
```

Figura 4.54. FTDock V2: Estructura de *kernels*.
Paralelismo con dep. de datos

No se han creado versiones separadas con cada una de las modificaciones debido a la complejidad que suponía la función de devolver las estructuras a memoria global, cuando finalmente esta no iba a ser necesitada.

	E1-G128	E1-G256	E2-G128	E2-G256
CUDA V1-CUFFT	1,79 X	2,34 X	1,87 X	2,57 X
CUDA V2-Full	5,97 X	10,94 X	5,68 X	10,88 X

Tabla 4.23. FTDock *Speedups*: CUDA V2-Full

Esta versión ha llevado consigo un gran esfuerzo y contempla la fracción mayoritaria del tiempo de trabajo empleado sobre esta aplicación. El intento de migración al dispositivo de partes del código que son difíciles de mapear a *threads* CUDA, junto con la necesidad de hacer uso de configuraciones de *kernel* multidimensionales, y la acción de transferir estructuras de datos con varios niveles de indirección, han conllevado un gran esfuerzo de trabajo.

El rendimiento obtenido puede consultarse en la Tabla 4.23. Pensamos que el beneficio quizás es algo limitado para el esfuerzo de programación requerido, pero debemos matizar que la mayoría de las zonas del código poseen un paralelismo irregular

que provoca gran cantidad de saltos divergentes, además de mantener a gran parte del dispositivo ocioso mientras otra parte sigue computando.

4.4.3 Buffering de Escrituras en Disco

En esta versión se ha llevado a cabo una optimización que nada tiene que ver con CUDA, pero que se hace imprescindible para proceder con la utilización de varias GPUs (siguiente versión). En concreto, hemos modificado la forma en la que se escriben en disco los resultados. En la versión original, las escrituras se envían a disco en todas y cada una de las iteraciones. Sin embargo, para utilizar varias GPUs y que las escrituras se realicen de manera ordenada, resulta imprescindible modificar este patrón. Para ello dispondremos de un buffer que almacenará la información de cada una de las iteraciones y las volcará a disco al finalizar el bucle principal. Esto permitirá que con varias GPUs, cada una almacene su salida en un buffer privado y luego se realicen las escrituras de manera ordenada. Lo veremos más en detalle en el apartado siguiente.

En cuanto al rendimiento, Tabla 4.24, el empeoramiento de la entrada E2-G128 tienen que ver con el tamaño del buffer y los fallos de caché, teniendo en cuenta que la información que se escribe es de varios megabytes.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
CUDA V2-Full	5,97 X	10,94 X	5,68 X	10,88 X
CUDA V3-Buffering	5,95 X	11,15 X	5,33 X	10,99 X

Tabla 4.24. FTDock *Speedups*: CUDA V3-Buffering

4.4.4 Enfoque Multi-GPU

Todas las dificultades que hemos encontrado a la hora de portar FTDock a una sola GPU desaparecen cuando pretendemos escalar el programa a múltiples dispositivos. Gracias a que las iteraciones del bucle principal son elevadas y totalmente independientes unas de otras, podremos realizar un reparto equitativo de las mismas entre todas las tarjetas para cómputo de propósito general que queramos utilizar. De esta manera seguiremos un esquema como el descrito en la Figura 4.13, donde simplemente se lleva a cabo un reparto de trabajo a nivel de las iteraciones del bucle, sin ser necesario modificar en absoluto el código de ninguno de los *kernels*.

La complejidad radica entonces en determinar qué estructuras y variables serán privadas o compartidas por cada *thread* de *host*, y sobre todo, en llevar a cabo la escritura a disco en el orden en el que corresponda. Para este último aspecto, a partir de la versión anterior, será necesario disponer de un buffer privado por cada uno de los hilos, que debe ser volcado a disco en función del orden del conjunto de iteraciones que le haya sido asociado. Para evitar que este reparto sea no determinista, se utilizará un

schedule estático donde a cada *thread* le será repartida su parte, compuesta por iteraciones sucesivas entre sí.

```
#pragma omp parallel ...
{
  ...
  char aux_buffer[100];
  char * output_buffer = (char *) malloc (...);

  int chunk = ((Angles.n-1)%omp_get_num_threads()) ?
              (Angles.n-1)/omp_get_num_threads()+1 :
              (Angles.n-1)/omp_get_num_threads();
  ...

  /* Main program loop */
  #pragma omp for schedule(static, chunk) nowait
  for( rotation = 1; rotation <= Angles.n ; rotation++ ) {

    ... //GPU Computing similar to monoGPU version.

    /* Saving iteration scores in my private buffer */
    for( i = 0 ; i < keep_per_rotation ; i ++ ) {
      ...
      sprintf(aux_buffer, scores...);
      strcat(output_buffer, aux_buffer);
    }
  }  /* Finished main loop */


  /* Writing scratch_scores.dat */

  /* Master opens with "w" */
  #pragma omp master
  {
    ftdock_file = fopen( "scratch_scores.dat" , "w");
    fputs(output_buffer, ftdock_file);
    fclose( ftdock_file ) ;
  }

  #pragma omp barrier

  /* The other threads opens with "a" and write in order */
  #pragma omp for ordered schedule(static,1)
  for (i=0; i < omp_get_num_threads(); i++){
    #pragma omp ordered
    {
      if (omp_get_thread_num() != 0){
        ftdock_file = fopen( "scratch_scores.dat" , "a");
        fputs(output_buffer, ftdock_file);
        fclose( ftdock_file ) ;
      }
    }
  }
}
```

Figura 4.55. FTDock V4: Esquema Multi-GPU

Tras el cómputo de las rotaciones y el almacenamiento de los resultados en el correspondiente buffer privado, la escritura tendrá lugar en orden, donde el *thread*

master será el primero en escribir el fichero, y el resto continuará añadiendo la información en el mismo orden en el que le han sido repartidas las iteraciones. Este propósito se consigue haciendo uso de la directiva `ordered`, como podemos ver en la Figura 4.55.

Como era de esperar, la aplicación escala de manera óptima al duplicar el número de GPUs (Tabla 4.25), ya que no es necesaria ningún tipo de comunicación entre ellas y el algoritmo es suficientemente denso como para dar trabajo a más de un dispositivo. Tras los resultados medianamente aceptables de la versión con una sola GPU, podemos ver como una forma de acelerar rápidamente la ejecución es desplegar la aplicación sobre un mayor número de dispositivos.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
CUDA V3-Buffering	5,95 X	11,15 X	5,33 X	10,99 X
CUDA V4-Multi-GPU	11,60 X	22,11 X	10,84 X	21,75 X

Tabla 4.25. FTDock *Speedups*: CUDA V4-Multi-GPU

4.4.5 Enfoque Multi-GPU Multi-Nodo

La versión Multi-Nodo despliega la aplicación sobre tres nodos dotados de dos GPUs cada uno, utilizando MPI para la comunicación entre ellos a partir de la versión OpenMP anterior. Como ya hemos comentado, el bucle principal del programa se caracteriza por poseer iteraciones independientes entre sí. Este hecho nos permitirá que cada nodo pueda procesar de manera autónoma sus iteraciones para, posteriormente, llevar a cabo el envío de todos los datos procesados al nodo principal, que será el encargado de escribirlos en disco.

El esquema de cómputo utilizado es exactamente el descrito en la Figura 4.15, donde se lleva a cabo una primera repartición del trabajo homogénea a nivel de nodos, una segunda distribución de trabajo a nivel de GPUs dentro del mismo nodo, un almacenamiento privado de los resultados por *thread* OpenMP asociado a cada dispositivo, un movimiento y ordenación de los resultados a la memoria global de cada uno de los nodos y una envío de todos los resultados al proceso cero para que los agrupe y proceda con su escritura en disco.

La Figura 4.56 muestra el código que hace referencia a la aplicación del esquema y a todo lo descrito. La escritura de datos a disco, en este caso, varía ligeramente con respecto a la versión anterior, ya que aquí es el *thread master* del nodo 0 el que realiza el volcado de datos de una sola vez, previa ordenación de los mismos.

Puesto que estamos lanzando el ejecutable sobre arquitecturas que son diferentes incluso a nivel de GPU, será necesario utilizar un binario distinto compilado de manera específica para cada *Compute Capability* de cada dispositivo. En concreto necesitaremos un binario compilado para la *Compute Capability* 1.3, que será lanzado a Obelix, y otro para la *Compute Capability* 1.0, que será ejecutado tanto en Praliné como

en Gelatine. Mediante *mpirun* nos encargaremos de el binario adecuado sea invocado en el servidor que le corresponde.

```
char * global_output_buffer = (char *) malloc(...);

  my_process_chunk = Angles.n%num_processes ?
                     (Angles.n/num_processes) + 1 :
                     Angles.n/num_processes;
  my_process_from  = 1 + my_process_chunk * my_process_id;
  my_process_to    = min ((Angles.n+1), my_process_from +
                     my_process_chunk);

#pragma omp parallel ...
{
  int my_device_chunk = my_process_chunk%omp_get_num_threads() ?
                        my_process_chunk/omp_get_num_threads()+1 :
                        my_process_chunk/omp_get_num_threads();
  char aux_buffer[100];
  char * local_output_buffer = (char *) malloc (...);
  ...

  /* Main program loop */
  #pragma omp for schedule(static, my_device_chunk)
  for( rotation = 1; rotation <= Angles.n ; rotation++ ) {
    ...
  }  /* Finished main loop */

  /* Ordering iterations results at process memory */
  #pragma omp for ordered schedule(static,1)
  for (i=0; i < omp_get_num_threads(); i++){
    #pragma omp ordered
    strcat(global_output_buffer, local_output_buffer);
  }

  #pragma omp master
  {
    /* Sending iterations results to process 0 */
    MPI_Gather(global_output_buffer,
               strlen(global_output_buffer), MPI_CHAR,
               global_output_buffer, global_output_buffer_size,
               MPI_CHAR, 0, MPI_COMM_WORLD);

    /* Writing scratch_scores.dat */
    if (my_process_id == 0){
      ftdock_file = fopen( "scratch_scores.dat" , "w");{
      for (i=0; i < num_processes; i++){
        fputs(global_output_buffer, ftdock_file);
        global_output_buffer +=
            BYTES_PER_LINE * keep_per_rotation * my_process_chunk;
      }
      global_output_buffer -= BYTES_PER_LINE * keep_per_rotation
                              * my_process_chunk * num_processes;
      fclose( ftdock_file ) ;
    }
  }
}
```

Figura 4.56. FTDock V5: Esquema Multi-Nodo

En cuanto a los resultados, el escalado no es tan bueno como cabría esperar (Tabla 4.26). El hecho de que los nodos no sean similares en cuanto a procesador ni en cuanto a GPUs tiene mucho que ver en esto, ya que estamos repartiendo iteraciones de manera homogénea entre nodos que no lo son. Además, la infraestructura de red MPI, compartida por otros muchos servidores y no especialmente diseñada para tal uso, también ha hecho mella en el rendimiento, pues aunque no se transfieren datos continuamente sí que se envían mensajes hacia el proceso 0 con información para mostrar en pantalla, ya que por cada rotación procesada se imprime un carácter mediante `printf`. Este flujo de comunicación no puede ser abolido ni redirigiendo la salida a */dev/null*, ya que ese directorio hace referencia al nodo principal y los datos viajan igualmente por la red. También se estudiaron las distintas opciones que `mpirun` ofrece al respecto, sin ningún éxito.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
CUDA V4-MultiGPU	11,60 X	22,11 X	10,84 X	21,75 X
CUDA V5-MultiNodo	17,26 X	22,22 X	18,90 X	25,45 X

Tabla 4.26. FTDock *Speedups*: CUDA V5-Multi-GPU Multi-Nodo

4.4.6 Enfoque Multi-GPU Multi-Nodo. Distribución de Carga Heterogénea

La introducción de un factor de heterogeneidad que realice un reparto desbalanceado de las iteraciones permite asignar una mayor carga de trabajo a aquel nodo que es capaz de computarlas a una mayor velocidad (Obelix). De esta forma podemos ajustar la carga que se distribuye para que todos los servidores terminen su ejecución prácticamente al mismo tiempo.

Puesto que este factor de heterogeneidad variará dependiendo de la entrada, se ha decidido proceder con su ajuste utilizando E1, y resultará en valores diferentes dependiendo del tamaño del *grid* utilizado.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
CUDA V4-MultiGPU	11,60 X	22,11 X	10,84 X	21,75 X
CUDA V5-MultiNodo	17,26 X	22,22 X	18,90 X	25,45 X
CUDA V6-MultiNodo-Het	22,33 X	35,26 X	21,72 X	35,37 X

Tabla 4.27. FTDock *Speedups*: CUDA V5-Multi-GPU Multi-Nodo Heterogénea

En la Tabla 4.27 podemos ver como los resultados mejoran notablemente gracias a esta repartición de trabajo desbalanceada, dándose un mayor aumento de rendimiento sobre la primera de las entradas, que ha sido la utilizada para fijar los valores de heterogeneidad. Estos han sido de +55% y +88% para *grids* de 128 y 256

respectivamente. Consideramos que el rendimiento obtenido se corresponde con un buen escalado de la aplicación si tenemos en cuenta la diferencia en cuanto a capacidad que existe entre los dispositivos de cada uno de los nodos.

4.4.7 Conclusiones

Del trabajo sobre esta aplicación hemos aprendido varias lecciones. La primera de ellas es que gracias a librerías que ya poseen implementación en CUDA, y realizando una muy ligera modificación del código fuente, sin necesidad apenas de estudiarlo, podemos aumentar nuestro *speedup* con respecto a la CPU, aunque la mejora puede verse mermada por las transferencias de memoria que tengan que realizarse. En segundo lugar, la migración de ciertas regiones de código puede resultar especialmente compleja, sobre todo si tratamos con bucles irregulares. La heterogeneidad en cuanto al trabajo que realizan los *threads* de la GPU también es un factor negativo que puede traducir nuestros esfuerzos de programación en reducidas mejoras de rendimiento. Sin embargo, una vez obtenida la versión completa para una sola GPU, el moderado esfuerzo de extender la ejecución a un entorno múltiple puede resultar gratamente recompensado si el algoritmo se presta a ello.

	E1-G128	**E1-G256**	**E2-G128**	**E2-G256**
Opteron Serie	1,00 X	1,00 X	1,00X	1,00 X
CUDA V1-CUFFT	1,79 X	2,34 X	1,87 X	2,57 X
CUDA V2-Full	5,97 X	10,94 X	5,68 X	10,88 X
CUDA V3-Buffering	5,95 X	11,15 X	5,33 X	10,99 X
CUDA V4-MultiGPU	11,60 X	22,11 X	10,84 X	21,75 X
CUDA V5-MultiNodo	17,26 X	22,22 X	18,90 X	25,45 X
CUDA V6-MultiNodo-Het	22,33 X	35,26 X	21,72 X	35,37 X

Tabla 4.28. FTDock: Resumen *Speedups*

Un aspecto interesante que hemos podido verificar es que puede darse el caso en el que compense ejecutar cierto código en serie sobre la GPU antes que realizar transferencias de datos para computar dicha parte sobre la CPU, a pesar de ir en contra de toda la filosofía *pro-many-threading* de CUDA. En la Tabla 4.28 y Figura 4.57 quedan recogidos todos los resultados de las distintas versiones.

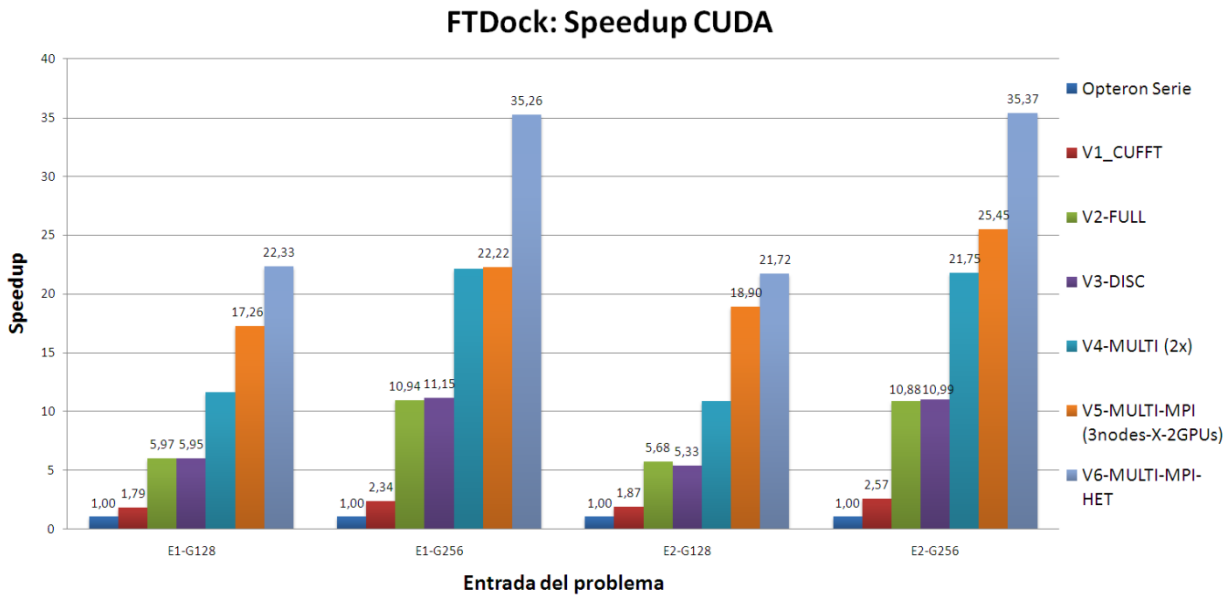


Figura 4.57. FTDock: Gráfica de *Speedups*

4.5 Comparación con otras Arquitecturas

Este apartado pretende enmarcar los resultados obtenidos en un contexto en el que se vean involucradas otras arquitecturas de alto rendimiento conocidas. Para ello, nos valdremos de 32 *cores* del supercomputador SGI Altix 4700 [34], donde ejecutaremos versiones OpenMP de EP y CG, y de un Blade QS20 [51], en el que lanzaremos la versión Cell de FTDock implementada por el Departamento de Arquitectura de Computadores de la UPC.

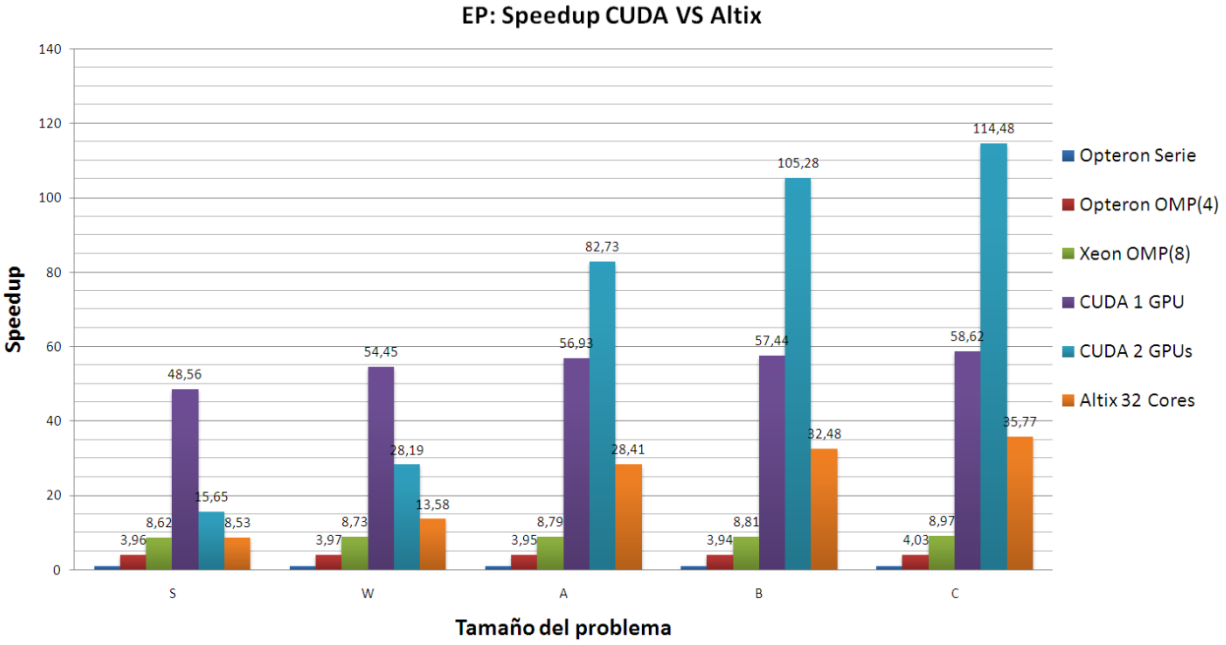


Figura 4.58. EP: Gráfica de *Speedups*. CUDA vs Altix

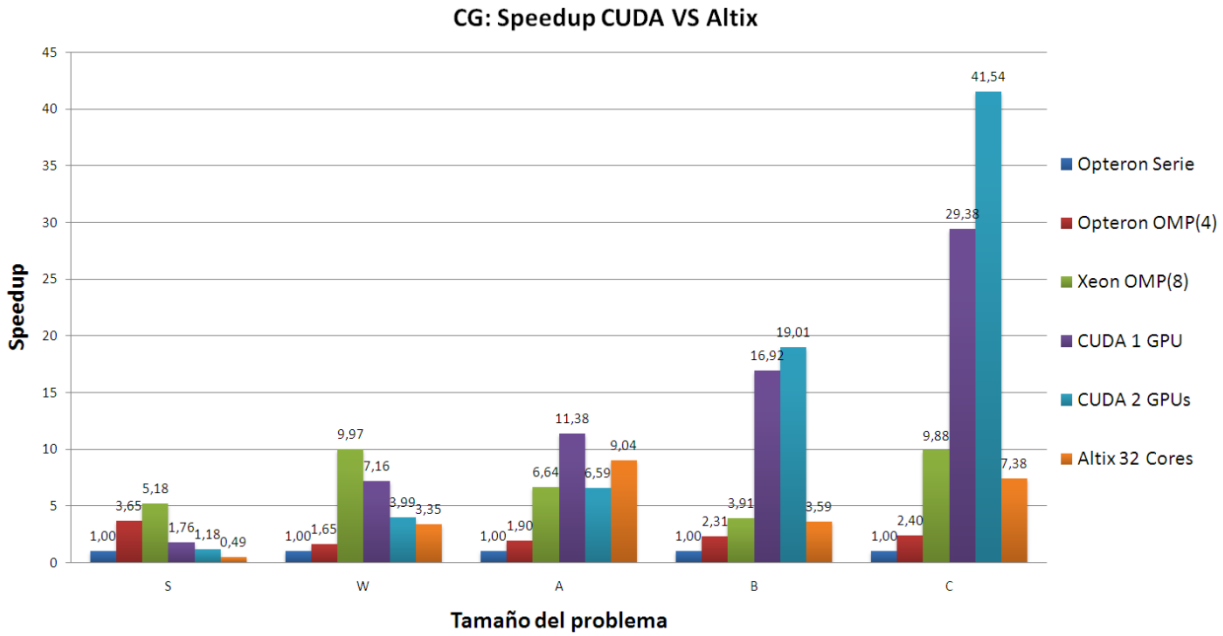


Figura 4.59. CG: Gráfica de *Speedups*. CUDA vs Altix

Refiriéndonos a la Figura 4.58 y Figura 4.59, podemos observar una gran diferencia de rendimiento entre a la versión CUDA, con una sola GPU, y los 32 *cores* Itanium de Altix. Mientras que en EP la distancia entre ambas arquitecturas se mantiene una vez alcanzados los tamaños grandes del problema, es destacable como esta diferencia aumenta para el problema CG conforme nos acercamos al tamaño C. La explicación de dicho comportamiento se encuentra en las comunicaciones que son necesarias en el segundo programa, donde Altix, por su diseño, arquitectura y número de núcleos, se ve claramente penalizada.

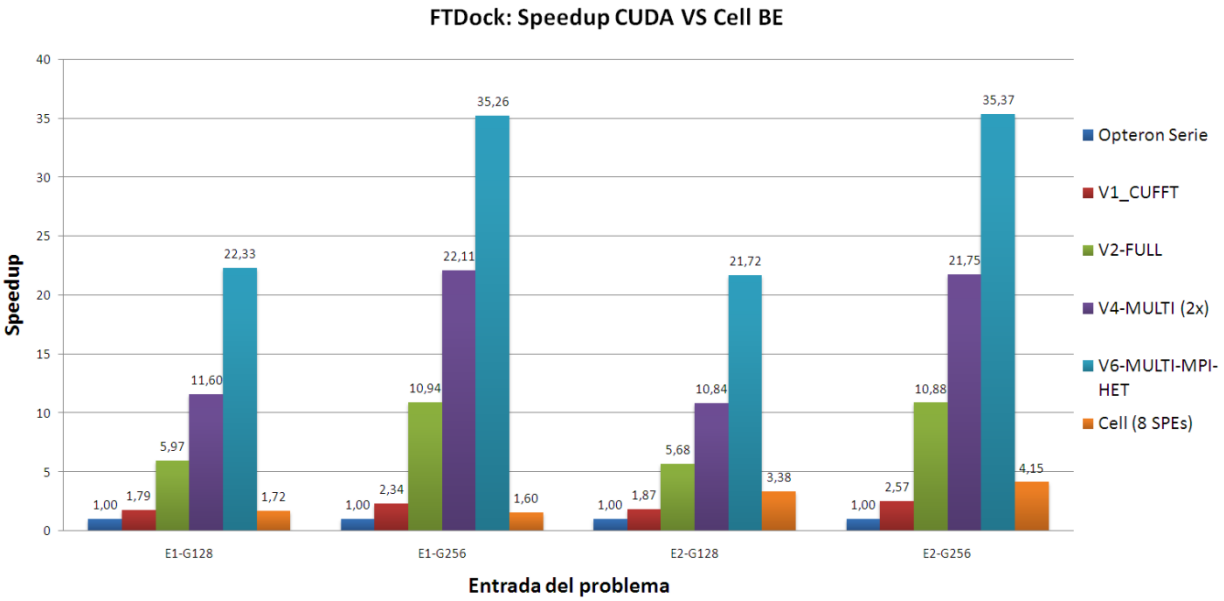


Figura 4.60. FTDock: Gráfica de *Speedups*. CUDA vs Cell BE

Aún superior ha sido la diferencia entre la versión Mono-GPU de FTDock y su correspondiente versión en Cell utilizando 8 SPEs – *Synergistic Processing Elements* –. Sin embargo, en este caso debemos matizar que la función `discretize_structure()` no se encuentra paralelizada ni tan siquiera nivel de PPE – *Power Processing Element* – lo que podría suponer un aumento de rendimiento de aproximadamente 1,2 X.

Muy lejos de poder eliminar casi por completo las transferencias de datos, como hemos conseguido en la versión CUDA, las grandes limitaciones que imponen las capacidades de los *Local Stores* de los SPEs origina un gran número de comunicaciones entre los mismos y el PPE, hasta el punto de que, dependiendo del tamaño de los *grids*, cada una de las operaciones que se realiza en cada iteración ha de abordarse por fases.

En conclusión, podríamos decir que las GPUs parecen ser la solución más potente en este caso, y sobre todo del orden de unas 10 veces más económica que la opción Cell, su competidor más directo. Mientras que un Blade QS20 podríamos encontrarlo alrededor de los 6000 euros, dos tarjetas gráficas NVIDIA como las utilizadas se encontrarían sobre los 600 euros.

Capítulo 5

Planificación Final

5.1 Planificación Final

La planificación final del proyecto se corresponde con la mostrada en la Figura 5.1.

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	**Búsqueda, Análisis y Estudio Bibliográfico**	01/09/2009	01/06/2010	49,5h
2	**EP**	01/11/2009	30/01/2010	109h
3	**CG**	15/02/2010	21/04/2010	97,5h
4	**FTDock**	02/05/2010	28/05/2010	90,5h
5	**Memoria del Proyecto**	01/01/2010	21/06/2010	238h
6	**Total**	**01/09/2009**	**21/06/2010**	**584,5h**

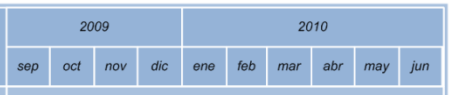


Figura 5.1. Planificación Final del Proyecto

5.2 Coste del proyecto

A partir de las suposiciones llevadas a cabo en el apartado 1.5 y del número de horas contabilizadas en el punto anterior, el coste de mano de obra total del proyecto ha sido de:

4383,75 €

Capítulo 6

Conclusiones y Vías Futuras

6.1 Conclusiones

Las conclusiones más importantes que podemos extraer de todo lo aprendido a lo largo del proyecto se centran, mayormente, en las optimizaciones que han sido aplicadas y ciertas peculiaridades que hemos podido extraer del comportamiento de la arquitectura.

En primer lugar, respecto a la aritmética de operaciones en punto flotante de doble precisión, aunque los resultados han sido relativamente buenos, consideramos que esta debe ser mejorada en arquitecturas futuras. El hecho de que exista una relación de unidades funcionales de ocho a una con respecto a las operaciones de simple precisión, junto con la incompatibilidad y comportamiento negativo de la memoria de texturas y compartida, respectivamente, podrían deteriorar la imagen de estas opciones en el ámbito científico y de centros de supercomputación.

Otras limitaciones que encontramos en el modelo de programación hacen referencia al uso de múltiples dispositivos, ya que se hecha en falta cierta automatización y mayor versatilidad, que consigan relajan la complejidad que conlleva el tener que introducir múltiples modelos de programación.

En cuanto al proceso de migración de código desde la CPU a la GPU, resulta muy interesante poder contar con versiones de librerías bastante utilizadas en el ámbito científico, que hacen uso del dispositivo de manera automática. Gracias a ellas, puede ser especialmente sencillo el obtener una primera aceleración de la aplicación sin demasiado esfuerzo, lo que ayudaría con la expansión de esta tecnología.

Si nos centramos en las optimizaciones, podríamos destacar tres factores primordiales que debemos cuidar al enfrentarnos a este tipo de arquitecturas. El primero de ellos se trata de los accesos a la memoria global. Unos accesos no *coalesced* pueden suponer la pérdida de cualquier mejora que una GPU pudiera ofrecer, por lo que deberán estudiarse a conciencia los patrones de acceso de la aplicación, y buscar la estrategia de mapeo de *threads* más adecuada para favorecer este factor. Por este motivo, el uso de memoria compartida resulta imprescindible, y la memoria de texturas y constante también pueden ayudar, pero siempre dependeremos de las características del problema al que nos estemos enfrentando.

El segundo de los factores se refiere a las causas de serialización que podemos encontrar por distintos frentes. El más importante es el de saltos divergentes, que puede llegar a eliminar el paralelismo más extremo, y con ello hacer desaparecer cualquier mejora que un dispositivo pudiera introducir. Podremos luchar contra algunos de ellos, mediante técnicas como la de *padding*, pero si el código es masivamente divergente es posible que no sea adecuado para ejecutarse en una GPU.

El último de los factores apunta a las transferencias de memoria entre el *host* y el dispositivo. Como hemos podido comprobar, estas pueden suponer la mayor parte del tiempo de ejecución para según qué tipos de problema. Su eliminación resulta necesaria para mejorar el rendimiento, llegando incluso a compensar el ejecutar código serie en el dispositivo, como hemos podido comprobar. Cuando no es posible deshacerse de ellas, siempre tendremos en nuestra mano técnicas para optimizar las mismas, mediante el uso de memoria no paginable.

En definitiva, nos encontramos ante una arquitectura con grandes capacidades de cómputo, incluso con posibilidades de superar a su competidor más directo, el Cell BE, tanto a nivel de rendimiento como a nivel económico. Resulta ser el complemento perfecto para acelerar la potencia de cálculo de las CPUs de hoy día. No podemos hablar de ella como una arquitectura "todoterreno" autosuficiente, pues hemos podido comprobar las grandes limitaciones que posee, pero si su versatilidad y potencia continúan en aumento, auguramos su firme consolidación en entornos de computación de altas prestaciones.

6.2 Vías futuras

Se plantea el estudio de otras aplicaciones en busca de nuevas situaciones con las que no nos hayamos enfrentado. Podrían estar relacionadas con la bioinformática, el tratamiento de imágenes, análisis de señales, etc. Además, algo indispensable sería el estudio de la nueva gama de dispositivos NVIDIA que acaba de salir al mercado, dando paso a la *Compute Capability* 2.0.

Posteriormente, vista la complejidad que puede llegar a entrañar el desarrollo de aplicaciones relacionadas con GPUs, se podría buscar la construcción de una capa de abstracción que permitiera ocultar ciertos detalles de bajo nivel, y automatizar operaciones como las transferencias de datos, para facilitar la labor del programador.

Bibliografía y Referencias

[1] *CUDA Programming Guide*. Versión 2.3 y 3.0. NVIDIA. 2009-2010.

[2] *CUDA Reference Manual*. Versión 2.3 y 3.0. NVIDIA. 2009-2010.

[3] *CUDA Best Practices Guide*. Versión 2.3 y 3.0. NVIDIA. 2009-2010.

[4] *CUDA Getting Started*. Versión 2.3 y 3.0. NVIDIA. 2009-2010.

[5] *CUDA GDB.* Versión 3.0. NVIDIA. 2009-2010.

[6] *CUFFT Library*. Versión 3.0. NVIDIA. 2010.

[7] *CUDA GPU Computing SDK 3.0*. NVIDIA. 2010.

[8] *NVIDIA's GT200: Inside a Parallel Processor*. D. Kanter, 2008.

[9] *Seminarios CUDA Illinois*. D. Kirk (NVIDIA) and W. Hwu. 2006-2009.

[10] *OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization*. S. Lee et ál. Pardue University.

[11] *Efficient Sparse Matrix-Vector Multiplication on CUDA.* N. Bell et ál. NVIDIA. 2008.

[12] *Optimizing Sparse Matrix-Vector Multiplication on GPUs.* M. Baskaran and R. Bordawekar. IBM. 2009.

[13] *SC07: High Performance Computing with CUDA. Optimizing CUDA.* M. Harris. NVIDIA.

[14] *Benchmarking GPUs to Tune Dense Linear Algebra.* V. Volkov, and J. Demmel. University of California at Berkeley.

[15] *Parallel Algorithm for Solving Kepler's Equation on Graphics Processing Units.* E. Ford. University of Florida. 2008.

[16] *CUDA Libraries and Tools.* J. Cohen. NVIDIA. SC 2009.

[17] *Seismic Imaging on NVIDIA GPUs.* S. Morton. Hess Corporation. SC 2009.

[18] *Case Study: Molecular Modeling Applications.* J.E. Stone. Beckman Institute for Advanced Science and Technology. SC 2009.

[19] *Fluid Simulation with CUDA.* J. Cohen. NVIDIA. SC 2009.

[20] *Irregular Algorithms & Data Structures.* J. Owens. University of California. SC 2009.

[21] *CUDA Optimization.* P. Micikevicius. NVIDIA. SC 2009.

[22] *High Performance Computing with CUDA.* G. Luebke. NVIDIA. SC 2009.

[23] *GPU Mixed-Precision Linear Equation Solver For Lattice QCD.* M. Clark. Harvard University. SC 2009.

[24] *The NAS Parallel Benchmarks.* D. Bailey et ál. NAS System Division. 1994.

[25] *The OpenMP Implementation of NAS. Parallel Benchmarks and Its Performance.* H. Jin et ál. NAS System Division. 1999.

[26] *Modelling Protein Docking using Shape Complementarity, Electrostatics and Biochemical Information.* H. Gabb et ál. Biomolecular Modelling Laboratory. Imperial Cancer Research Fund. 1997.

[27] *Drug Design Issues on the Cell BE.* D. Jimenez et al. UPC-BSC. PACT 2007.

[28] *CUDA Webinars: An Introduction to GPU Computing and CUDA Architecture; Performance Considerations for CUDA Programming; Further CUDA Optimization Techniques; An Introduction to GPU Computing and OpenCL; Best Practices for OpenCL programming.* NVIDIA. 2009-2010.

[29] *Intel's Teraflops Research Chip. Advancing Multi-Core Technology into the Tera-Scale Era.* Intel. 2009

[30] *Belgian researchers from University of Antwerp develop new mini supercomputer: Fastra2.* Press Release. IBBT.

[31] *OpenCL. Parallel Computing for Heterogeneous Devices*. Khronos Group. 2009.

[32] *OpenCL. The Open Standard for Parallel Programming of Heterogeneous Systems.* J. Y. Xu. University at Albany.

[33] *Nvidia Tesla: A Unified Graphics and Computing Architecture.* E. Lindholm, J. Kicholls, S. Oberman, J. Montrym. NVIDIA 2008.

[34] *SGI Altix 4700 User's Guide*. Barcelona Supercomputing Center. 2008.

Referencias Web

[35] *NAS Parallel Benchmarks.*
http://www.nas.nasa.gov/Resources/Software/npb.html

[36] *FTDock.*
http://bmm.cancerresearchuk.org/docking

[37] *Algoritmo Katchalski-Katzir, Graphics processing unit, pixel shader, vertex shader, geometry shader*.
http://en.wikipedia.org.

[38] *CUDA Webinars*.
http://www.nvidia.com/object/webinar.html

[39] *The Top 500 Project* .
http://www.top500.org

[40] *The Green 500 Project* .
http://www.green500.org

[41] *Fastra II*.
http://fastra2.ua.ac.be
[42] *OpenCL*.
http://www.khronos.org/opencl
[43] *Direct X*.
http://www.microsoft.com/games/en-US/aboutGFW/pages/directx.aspx
[44] *CG Language*.
http://developer.nvidia.com/page/cg_main.html
[45] *HLSL Language*.
http://msdn.microsoft.com/en-us/library/bb509561(v=VS.85).aspx
[46] *GPGPU*.
http://www.gpgpu.org
[47] *Pthreads*.
https://computing.llnl.gov/tutorials/pthreads
[48] *OpenMP*.
http://openmp.org
[49] *MPI*.
http://www.mcs.anl.gov/research/projects/mpi/index.htm
[50] *FFTW*.
http://www.fftw.org/
[51] *IBM Blade Center QS20*.
http://www-01.ibm.com/common/ssi/rep_ca/7/897/ENUS106-677/index.html
[52] *Barcelona Supercomputing Center*.
http://www.bsc.es

Anexo I

Utilidades Auxiliares Desarrolladas

I.1. Función *cudaInit*

La primera vez que se ejecuta una función que tiene relación con una GPU, CUDA lleva a cabo una serie de procedimientos que inicializan al dispositivo involucrado. Esto origina cierta sobrecarga sobre la primera llamada a CUDA que tenga lugar, por lo que se ha diseñado la función `cudaInit()` para controlar esta inicialización.

El código de la misma puede consultarse en la Figura I.1, donde el parámetro necesario para su invocación se corresponde con el identificador del componente compatible con la tecnología (0, 1, 2, 3, etc.…). Básicamente, esta función obtiene información sobre el sistema de los dispositivos CUDA y comprueba que el identificador del dispositivo que se ha pasado como parámetro sea un dispositivo existente y compatible.

```
void cudaInit(unsigned int deviceNumber){
  int deviceCount;
  cudaDeviceProp deviceProp;

  cudaGetDeviceCount(&deviceCount);
  if (deviceCount == 0) {
    fprintf(stderr, "Error: no devices supporting CUDA.\n");
    exit(-1);
  }
  if (deviceNumber > deviceCount){
    fprintf(stderr, "Error: device %d does not exist or it does
                     not support CUDA.\n", deviceNumber);
    exit(-1);
  }
  cudaGetDeviceProperties(&deviceProp, deviceNumber);
  if (deviceProp.major < 1) {
    fprintf(stderr, "Error: device %d does not
                     support CUDA.\n", deviceNumber);
    exit(-1);
  }
  cudaSetDevice(deviceNumber);
}
```

Figura I.1. Función *cudaInit*

I.2. Macros *CHECK_CUDA* y *CHECK_CUFFT*

Se han implementado dos macros encargadas de comprobar si las funciones de CUDA y la librería CUFFT se ejecutan correctamente. En caso de que se produzca algún error durante la ejecución de alguna de ellas, las macros mostrarán el mensaje de error que la función nos devuelve e interrumpirán la ejecución del programa. Este tipo de medida es necesaria debido a que si se producen reiterados errores en el dispositivo y no se detiene la ejecución, este puede quedar en un estado corrupto y no funcionar correctamente, además de poder colgar por completo el sistema.

```
#ifdef PERFORMANCE
  #define CHECK_CUDA(function) (function)
  #define CHECK_CUFFT(function) (function)

#else
  #define   CHECK_CUDA(function){                    \
                                                     \
    if ((function) != cudaSuccess){                  \
      printf("CHECK_CUDA ERROR: %s\n",
            cudaGetErrorString(cudaGetLastError())); \
      exit (1);                                      \
                                                     \
    }                                                \
  }

  #define CHECK_CUFFT(function) {                    \
    if ((function) != CUFFT_SUCCESS){                \
      printf("CHECK_CUFFT ERROR: %s\n",
            cudaGetErrorString(cudaGetLastError())); \
            exit (1);                                \
    }                                                \
  }
#endif
```

Figura I.2. Macros *CHECK_CUDA* y *CHECK_CUFFT*

`CHECK_CUDA` y `CHECK_CUFFT` han sido diseñadas para poder ser activadas y desactivadas mediante la definición en tiempo de compilación de la macro `PERFORMANCE`, lo que permitirá eliminar este tipo de comprobaciones cuando deseemos obtener medidas de rendimiento (`-DPERFORMANCE`). La Figura I.2 muestra el código de la implementación.

I.3. Shell Scrips *timing_NPB.sh* y *timing_ftdock.sh*

El script *timming_NPB.sh* ha sido desarrollado para ejecutar de una sola vez todos los tamaños de problema de uno de los *benchmarks* un número de veces establecido y obtener el tiempo medido de dichas ejecuciones. Para usarlo debemos pasarle como parámetro el nombre del *benchmark* y el número de iteraciones que queremos que ejecute cada tamaño de problema, como por ejemplo `./timing_NPB.sh ep 100`. El código del mismo se muestra en la Figura I.3. En este caso, puesto que los *benchmarks* no hacen uso de entrada salida no será necesario que la ejecución se realice sobre ninguna unidad local en especial

```
#!/bin/bash

iters=$2

echo $iters Iteraciones >> tiempos

for j in S W A B C
do
  make $1 CLASS=$j
  i=0
  media=0
  while [ $i -lt $iters ]
  do
    time=$(./bin/$1.$j | grep "Time in seconds" | tr -s "\t" " "
         | cut -d " " -f 6)
    media=$(echo $media + $time | bc -l)
    echo $i: $time $media

    dmesg | grep Xid
    i=`expr $i + 1`
  done

  media=$(echo $media "/" $iters | bc -l)
  echo $1.$j $media >> tiempos
done
```

Figura I.3. Script *timing_NPB.sh*

En cuanto a *timing_ftdock.sh* disponemos de dos versiones y la finalidad es la misma que la del script anterior: obtener los tiempos de ejecución medios del programa con las cuatro configuraciones que hemos elegido previamente Sin embargo, FTDock hace un uso intensivo de la entrada salida, por lo que necesitaremos que la ejecución se lleve a cabo sobre un disco duro local. Así, el script moverá todo lo necesario a la dirección contenida en la variable `$SCRATCH`.

La primera de las versiones (Figura I.4) es utilizada en aquellos casos en los que se trabaja con un solo nodo de ejecución. La segunda versión (Figura I.5) muestras los cambios, con respecto a la primera, necesarios para ejecutar las versiones que requieren de múltiples nodos.

En este caso, para lanzar cualquiera de las dos versiones del script solamente necesitamos pasar como parámetro el número de iteraciones que queremos que se repita cada ejecución con la misma configuración, por ejemplo `./timing_ftdock.sh 5`.

```
#!/bin/bash

iters=$1

echo $iters Iteraciones >> tiempos

cp -f ./ftdock $HOME/scratch-local
cp -f ../test/A/test3/* $HOME/scratch-local
cp -f ../test/B/test4/* $HOME/scratch-local

cd $HOME/scratch-local

for j in 128 256
do

  echo ENTRADA 1 GRID $j
  echo "ENTRADA 1: -grid $j -static 2PKA.parsed -mobile
        1BPI.parsed" >> tiempos

  i=0
  media=0
  while [ $i -lt $iters ]
  do
    time=$((/usr/bin/time -f %e ./ftdock -grid $j -static
         2PKA.parsed -mobile 1BPI.parsed > /dev/null) 2>&1)

    media=$(echo $media + $time | bc -l)
    echo $i: $time $(echo $media "/" "(" $i + 1 ")" | bc -l)

    i=`expr $i + 1`
  done

  media=$(echo $media "/" $iters | bc -l)
  echo GRID $j $media >> tiempos

  echo ENTRADA 2 GRID $j
  ...
    time=$((/usr/bin/time -f %e ./ftdock -angle_step 10 -grid $j
         -static 2PTN.parsed -mobile 4PTI.parsed > /dev/null) 2>&1)
  ...
done

mv tiempos $HOME/PFC/3D_Dock/progs_cuda

cd $HOME/PFC/3D_Dock/progs_cuda
```

Figura I.4. Script *timing_ftdock.sh.* Versión para un solo nodo

```
#!/bin/bash

SCRATCH=$HOME/scratch-local
HOSTFILE=/home/users/dcaballero/PFC/3D_Dock/progs_cuda/hostfile
NODES=3
...
scp ./ftdock_sm13 dcaballero@obelix:/$SCRATCH
scp ./ftdock_sm10 dcaballero@obelix:/$SCRATCH
scp ../test/A/test3/* dcaballero@obelix:/$SCRATCH
scp ../test/B/test4/* dcaballero@obelix:/$SCRATCH

scp ./ftdock_sm10 dcaballero@gelatine:$SCRATCH
scp ./ftdock_sm13 dcaballero@gelatine:/$SCRATCH
scp ../test/A/test3/* dcaballero@gelatine:$SCRATCH
scp ../test/B/test4/* dcaballero@gelatine:$SCRATCH

scp ./ftdock_sm10 dcaballero@praline:$SCRATCH
scp ./ftdock_sm13 dcaballero@praline:/$SCRATCH
scp ../test/A/test3/* dcaballero@praline:$SCRATCH
scp ../test/B/test4/* dcaballero@praline:$SCRATCH


for j in 128 256
do
  ...
    time=$((/usr/bin/time -f %e mpirun --hostfile $HOSTFILE
      --prefix $PREFIX -np 1 $SCRATCH/ftdock_sm13
      -static $SCRATCH/2PKA.parsed -mobile $SCRATCH/1BPI.parsed
      -grid $j : -np 2 $SCRATCH/ftdock_sm10
      -static $SCRATCH/2PKA.parsed -mobile $SCRATCH/1BPI.parsed
      -grid $j > /dev/null) 2>&1)
  ...
    time=$((/usr/bin/time -f %e mpirun --hostfile $HOSTFILE
      --prefix $PREFIX -np 1 $SCRATCH/ftdock_sm13 -angle_step 10
      -static $SCRATCH/2PTN.parsed -mobile $SCRATCH/4PTI.parsed
      -grid $j : -np 2 $SCRATCH/ftdock_sm10 -angle_step 10
      -static $SCRATCH/2PTN.parsed -mobile $SCRATCH/4PTI.parsed
      -grid $j > /dev/null) 2>&1)
  ...
done
```

Figura I.5. Cambios en script *timing_ftdock.sh* para versión Multi-Nodo

Anexo II

Profilings

Los *profilings* han sido realizados, en la mayoría, con la herramienta *CUDA Visual Profiler* (3.5.1). Además, para FTDock se incluye un profile de la versión serie efectuado con *gprof* (3.5.5).

En los correspondientes a EP es posible mostrar el tamaño total de los datos que se envían durante las distintas transferencias de memoria. Sin embargo, debido a las miles de entradas de los perfiles de CG y FTDock, resulta imposible realizar este cálculo, por lo que no se lleva a cabo.

También debemos destacar que *CUDA Visual Profiler* no muestra las transferencias de memoria de todos los dispositivos cuando se están utilizando varios, aunque podemos asegurar que se llevan a cabo. Posiblemente se deba a limitaciones de la herramienta o la arquitectura.

En los perfiles referentes a la técnica *Zero Copy* no son capturadas las transferencias de memoria porque la propia técnica las elimina, al menos de manera explicita.

II.1. EP CUDA Profilings

Method	ep_kernel_double	memcpyHtoD	memcpyDtoH
#Calls	1	1	3
GPU usec	21.400.000,00	4,03	239.670,00
CPU usec	21.400.000,00	3,00	246.553,00
%GPU time	98,89	0,00	1,10
gridSizeX	32.768		
gridSizeY	1		
blockSizeX	256		
blockSizeY	1		
blockSizeZ	1		
registerPerThread	26		
dynSmemPerBlock	0		
staSmemPerBlock	56		
occupancy	0,50		
divergent branch	4.520.000,00		
warp serialize	415.000.000,00		
cta launched	3.277		
local load	127.000.000,00		
local store	724.000.000,00		
gld 32b	6		
gld 64b	516		
gld 128b	53.700.000,00		
gst 32b	6		
gst 64b	4.290.000.000,00		
gst 128b	53.800.000,00		
tex cache hit	0		
tex cache miss	0		
glob mem read throughput (GB/s)	3		
glob mem write throughput (GB/s)	131,40		
glob mem overall throughput (GB/s)	134,61		
instruction throughput	0,20		
Total memTransfersSize (Bytes)		134.217.768,00	40

Tabla II.1. Profiling EP V1. Tamaño C

Method	ep_kernel_double	memcpyDtoH
#Calls	1	3
GPU usec	12.804.200,00	596,64
CPU usec	12.804.600,00	3.454,00
%GPU time	99,99	0,00
gridSizeX	32.768	
gridSizeY	1	
blockSizeX	256	
blockSizeY	1	
blockSizeZ	1	
registerPerThread	40	
dynSmemPerBlock (Bytes)	0	
staSmemPerBlock (Bytes)	14.392	
%occupancy	**0.25**	
divergent branch	4.519.760	
warp serialize	73.896.000	
cta launched	3.277	
local load	53.690.400	
local store	429.523.000	
gld 32b	0	
gld 64b	0	
gld 128b	0	
gst 32b	39.324	
gst 64b	4.294.970.000	
gst 128b	0	
tex cache hit	0	
tex cache miss	0	
glob mem read throughput (GB/s)	0	
glob mem write throughput (GB/s)	214,67	
glob mem overall throughput (GB/s)	214,67	
instruction throughput	0,35	
Total memTransfersSize (Bytes)		1.835.008,00

Tabla II.2. Profiling EP V2-Shared Memory. Tamaño C

Method	ep_kernel_double	memcpyDtoH
#Calls	1	3
GPU usec	12.700.000,00	601,86
CPU usec	12.700.000,00	3.852,00
%GPU time	99,99	0,00
gridSizeX	32.768	
gridSizeY	1	
blockSizeX	256	
blockSizeY	1	
blockSizeZ	1	
registerPerThread	20	
dynSmemPerBlock	0	
staSmemPerBlock	4.160	
occupancy	**0,75**	
divergent branch	4.530.000	
warp serialize	166.000.000	
cta launched	3.277	
local load	134.000.000	
local store	967.000.000	
gld 32b	17.800.000	
gld 64b	55.900.000	
gld 128b	0	
gst 32b	17.900.000	
gst 64b	4.290.000.000	
gst 128b	0	
tex cache hit	0	
tex cache miss	0	
glob mem read throughput (GB/s)	3	
glob mem write throughput (GB/s)	217,41	
glob mem overall throughput (GB/s)	220,69	
instruction throughput	0,36	
Total memTransfersSize (Bytes)		1.835.008,00

Tabla II.3. Profiling EP V3-Occupancy 0,75. Tamaño C

Method	ep_kernel_double	memcpyDtoH
#Calls	1	3
GPU usec	12.700.000,00	1.329,09
CPU usec	12.700.000,00	1.367,00
%GPU time	99,98	0,01
gridSizeX	32.768	
gridSizeY	1	
blockSizeX	256	
blockSizeY	1	
blockSizeZ	1	
registerPerThread	20	
dynSmemPerBlock	0	
staSmemPerBlock	4.160	
occupancy	0,75	
divergent branch	4.530.000	
warp serialize	166.000.000	
cta launched	3.277	
local load	134.000.000	
local store	967.000.000	
gld 32b	17.800.000	
gld 64b	55.900.000	
gld 128b	0	
gst 32b	17.900.000	
gst 64b	4.290.000.000	
gst 128b	0	
tex cache hit	0	
tex cache miss	0	
glob mem read throughput (GB/s)	3	
glob mem write throughput (GB/s)	217,43	
glob mem overall throughput (GB/s)	220,70	
instruction throughput	0,36	
Total memTransfersSize (Bytes)		1.835.008,00

Tabla II.4. Profiling EP V4-Pinned. Tamaño C

Method	ep_kernel_double
#Calls	1
GPU usec	12.700.000,00
CPU usec	12.700.000,00
%GPU time	100,00
gridSizeX	32.768
gridSizeY	1
blockSizeX	256
blockSizeY	1
blockSizeZ	1
registerPerThread	20
dynSmemPerBlock	0
staSmemPerBlock	4.160
occupancy	0,75
divergent branch	4.530.000
warp serialize	167.000.000
cta launched	3.277
local load	134.000.000
local store	967.000.000
gld 32b	17.800.000
gld 64b	55.900.000
gld 128b	0
gst 32b	17.900.000
gst 64b	4.290.000.000
gst 128b	0
tex cache hit	0
tex cache miss	0
glob mem read throughput (GB/s)	3
glob mem write throughput (GB/s)	217,45
glob mem overall throughput (GB/s)	220,73
instruction throughput	0,36

Tabla II.5. Profiling EP V5-Zero Copy. Tamaño C

Method	ep_kernel_double	memcpyDtoH
#Calls	1	3
GPU usec	12.600.000,00	605,86
CPU usec	12.600.000,00	641,00
%GPU time	99,99	0,00
gridSizeX	32.768	
gridSizeY	1	
blockSizeX	256	
blockSizeY	1	
blockSizeZ	1	
registerPerThread	20	
dynSmemPerBlock	0	
staSmemPerBlock	4.152	
occupancy	0,75	
divergent branch	4.530.000	
warp serialize	157.000.000	
cta launched	3.277	
local load	262.000.000	
local store	1.690.000.000	
gld 32b	0	
gld 64b	0	
gld 128b	0	
gst 32b	39.324	
gst 64b	4.290.000.000	
gst 128b	0	
tex cache hit	0	
tex cache miss	0	
glob mem read throughput (GB/s)	0	
glob mem write throughput (GB/s)	218,37	
glob mem overall throughput (GB/s)	218,37	
instruction throughput	0,36	
Total memTransfersSize (Bytes)		1.835.008,00

Tabla II.6. Profiling EP V6-QQs Local. Tamaño C

Method	ep_kernel_double	memcpyDtoH
#Calls	1	3
GPU usec	12.600.000,00	598,98
CPU usec	12.600.000,00	632,00
%GPU time	99,99	0,00
gridSizeX	32.768	
gridSizeY	1	
blockSizeX	256	
blockSizeY	1	
blockSizeZ	1	
registerPerThread	20	
dynSmemPerBlock	0	
staSmemPerBlock	4.152	
occupancy	0.75	
divergent branch	4.530.000	
warp serialize	158.000.000	
cta launched	3.277	
local load	241.000.000	
local store	1.610.000.000	
gld 32b	0	
gld 64b	0	
gld 128b	0	
gst 32b	39.324	
gst 64b	4.290.000.000	
gst 128b	0	
tex cache hit	0	
tex cache miss	0	
glob mem read throughput (GB/s)	0	
glob mem write throughput (GB/s)	218,75	
glob mem overall throughput (GB/s)	218,75	
instruction throughput	0,36	
Total memTransfersSize (Bytes)		1.835.008,00

Tabla II.7. Profiling EP V7-Unroll. Tamaño C

Method	ep_kernel_double GPU0	memcpyDtoH GPUs	ep_kernel_double GPU1
#Calls	1	3	1
GPU usec	6.290.000,00	303,71	6.290.000,00
CPU usec	6.290.000,00	348,00	6.290.000,00
%GPU time	100,00	0,00	100,00
gridSizeX	16.384		16.384
gridSizeY	1		1
blockSizeX	256		256
blockSizeY	1		1
blockSizeZ	1		1
registerPerThread	20		20
dynSmemPerBlock	0		0
staSmemPerBlock	4.160		4.160
occupancy	0,75		0,75
divergent branch	2.260.000		2.260.000
warp serialize	79.000.000		79.000.000
cta launched	1.639		1.639
local load	121.000.000		121.000.000
local store	806.000.000		806.000.000
gld 32b	0		0
gld 64b	0		0
gld 128b	0		0
gst 32b	19.668		19.668
gst 64b	4.290.000.000		4.290.000.000
gst 128b	0		0
tex cache hit	0		0
tex cache miss	0		0
glob mem read throughput (GB/s)	0		0
glob mem write throughput (GB/s)	436,78		436,78
glob mem overall throughput (GB/s)	436,78		436,78
instruction throughput	0,36		0,36
Total memTransfersSize (Bytes)		917.504,00	1

Tabla II.8. Profiling EP V8-Multi-GPU. Tamaño C. 2 GPUs

II.2. CG CUDA Profilings

Method	conj_grad_ sum_d_kernel	conj_grad_ d_sum_kernel	conj_grad_z_ r_rho_kernel	conj_grad _p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	76	76	76	2	6	4.104
GPU usec	155.000.000	6.210.000	119.726	62.634	6.809	4.635	1.744	41	577.534	25.298
CPU usec	155.000.000	6.210.000	143.093	85.929	7.749	5.563	2.682	85	580.659	92.447
%GPU time	95,68	3,82	0,07	0,03	0,00	0,00	0,00	0,00	0,35	0,01
gridSizeX	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1		
registerPerThread	14	13	10	7	9	9	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.096	1.104	1.088	40	1.096	2.096	40	24		
occupancy	1,00	1,00	1,00	1,00	1,00	0,75	1,00	1,00		
divergent branch	4.860.000	194.508	74.230	0	2.969	2.969	0	0		
warp serialize	22.600.000	910.523	14.200.000	0	570.006	962.538	0	0		
cta launched	222.680	8.907	222.680	222.680	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0		
gld 32b	3.390.000.000	823.000.000	0	0	0	0	0	0		
gld 64b	5.610.000	224.328	0	0	0	0	0	0		
gld 128b	11.700.000	396.115	7.130.000	3.560.000	142.512	142.512	71.256	0		
gst 32b	222.680	8.907	222.680	0	8.907	17.814	0	0		
gst 64b	1.990.000.000	938.000.000	58.500.000	26.200.000	3.840.000	3.280.000	859.703	18.898		
gst 128b	5.340.000	142.512	3.560.000	1.780.000	356.280	0	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	7	42	76	72	26,61	39,09	51,95	0		
glob mem write throughput (GB/s)	8,19	96,16	349,27	302,27	425,62	450,80	365,37	351,61		
glob mem overall throughput (GB/s)	15,25	138,40	424,94	374,59	452,23	489,89	417,33	351,61		
instruction throughput	0,02	0,02	0,43	0,17	0,34	0,51	0,22	0,19		

Tabla II.9. Profiling CG V1. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	6	4.104
GPU usec	57.300.000,00	2.290.000,00	119.804,00	113.832,00	62.755,20	6.833,28	4.652,45	3.435,30	1.735,46	41,06	577.271,00	25.250,20
CPU usec	57.300.000,00	2.290.000,00	142.970,00	137.239,00	85.948,00	7.772,00	5.583,00	4.362,00	2.652,00	88,00	580.334,00	87.229,00
%GPU time	94,69	3,78	0,19	0,18	0,10	0,01	0,00	0,00	0,00	0,00	0,95	0,04
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1		
registerPerThread	13	13	10	7	7	9	9	7	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.080	1.080	1.088	1.072	40	1.096	2.096	1.072	40	24		
occupancy	1	1	1	1	1	1	0,75	1	1	1		
divergent branch	18.300.000	730.307	74.230	74.225	0	2.969	2.969	2.969	0	0		
warp serialize	1.130.000.000	45.200.000	14.200.000	13.800.000	0	570.115	969.127	495.272	0	0		
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0		
gld 32b	3.560.000.000	314.000.000	0	0	0	0	0	0	0	0		
gld 64b	435.000.000	17.400.000	0	0	0	0	0	0	0	0		
gld 128b	605.000.000	24.200.000	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0		
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0		
gst 64b	1.990.000.000	423.000.000	58.700.000	65.800.000	26.500.000	3.840.000	3.290.000	2.230.000	851.506	19.537		
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	37,99	61,81	75,62	59,69	72,18	26,51	38,94	52,74	52,20	0,00		
glob mem write throughput (GB/s)	22,28	117,33	349,65	407,67	304,02	424,28	450,99	439,14	364,09	360,95		
glob mem overall throughput (GB/s)	60,27	179,15	425,27	467,36	376,20	450,79	489,93	491,88	416,29	360,95		
instruction throughput	0,08	0,08	0,43	0,43	0,17	0,34	0,51	0,54	0,22	0,19		

Tabla II.10. Profiling CG V2-Warp. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	6	4.104
GPU usec	61.800.000,00	2.470.000,00	119.819,00	113.967,00	62.682,10	6.835,87	4.658,69	3.437,76	1.737,86	40,70	577.040,00	25.251,30
CPU usec	61.900.000,00	2.470.000,00	142.991,00	137.124,00	85.891,00	7.757,03	5.583,00	4.368,00	2.665,00	86,00	580.117,00	90.554,00
%GPU time	94,80	3,79	0,18	0,17	0,09	0,01	0,00	0,00	0,00	0,00	0,88	0,03
gridSizeX	18.750	18.750	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1		
registerPerThread	13	13	10	7	7	9	9	7	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.080	1.080	1.088	1.072	40	1.096	2.096	1.072	40	24		
occupancy	1	1	1	1	1	1	0,75	1	1	1		
divergent branch	16.700.000	669.016	74.200	74.200	0	2.969	2.969	2.969	0	0		
warp serialize	1.230.000.000	43.700.000	14.200.000	13.800.000	0	570.000	968.009	493.626	0	0		
cta launched	3.560.000	142.500	223.000	223.000	222.680	8.907	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0		
gld 32b	3.510.000.000	312.000.000	0	0	0	0	0	0	0	0		
gld 64b	436.000.000	17.400.000	0	0	0	0	0	0	0	0		
gld 128b	605.000.000	24.200.000	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0		
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0		
gst 64b	1.760.000.000	414.000.000	58.600.000	65.600.000	26.400.000	3.850.000	3.290.000	2.230.000	856.537	20.415		
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	34,93	57,00	75,61	59,62	72,26	26,50	38,89	52,70	52,13	0,00		
glob mem write throughput (GB/s)	18,28	106,54	349,51	406,43	303,66	424,88	450,61	439,77	365,43	377,78		
glob mem overall throughput (GB/s)	53,21	163,54	425,12	466,04	375,92	451,39	489,50	492,47	417,55	377,78		
instruction throughput	0,06	0,06	0,43	0,43	0,17	0,34	0,51	0,54	0,22	0,19		

Tabla II.11. Profiling CG V3-Half-Warp. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	6	4.104
GPU usec	51.700.000,00	2.070.000,00	119.844,00	113.889,00	62.792,20	6.823,04	4.656,06	3.435,17	1.740,83	40,67	586.162,00	25.256,60
CPU usec	51.700.000,00	2.070.000,00	142.924,00	137.124,00	86.044,00	7.752,00	5.585,00	4.372,00	2.669,00	91,00	588.986,00	86.619,00
%GPU time	94,52	3,78	0,21	0,20	0,11	0,01	0,00	0,00	0,00	0,00	1,07	0,04
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1		
registerPerThread	13	13	10	7	7	9	9	7	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.080	1.080	1.088	1.072	40	1.096	2.096	1.072	40	24		
occupancy	1	1	1	1	1	1	0,75	1	1	1		
divergent branch	9.500.000	380.000	74.230	74.225	0	2.969	2.969	2.969	0	0		
warp serialize	1.200.000.000	48.000.000	14.200.000	13.800.000	0	569.851	967.868	493.872	0	0		
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0		
gld 32b	3.140.000.000	297.000.000	0	0	0	0	0	0	0	0		
gld 64b	458.000.000	18.300.000	0	0	0	0	0	0	0	0		
gld 128b	459.000.000	18.400.000	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0		
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0		
gst 64b	1.370.000.000	398.000.000	58.600.000	65.700.000	26.400.000	3.850.000	3.290.000	2.220.000	860.183	19.889		
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	36,21	62,62	75,59	59,66	72,14	26,55	38,91	52,74	52,04	0,00		
glob mem write throughput (GB/s)	17,03	122,38	349,12	406,91	303,11	425,16	450,34	438,74	366,13	369,86		
glob mem overall throughput (GB/s)	53,23	185,00	424,71	466,56	375,24	451,71	489,25	491,48	418,17	369,86		
instruction throughput	0,09	0,09	0,43	0,43	0,17	0,34	0,51	0,54	0,22	0,19		

Tabla II.12. Profiling CG V4-Padding V2. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	28	4.104
GPU usec	51.700.000,00	2.070.000,00	121.374,00	117.535,00	62.559,80	7.039,71	4.657,41	3.515,01	1.758,27	45,06	577.526,00	25.250,80
CPU usec	51.700.000,00	2.070.000,00	144.732,00	140.742,00	85.654,00	7.965,00	5.595,00	4.451,00	2.673,00	97,00	580.301,00	85.568,00
%GPU time	94,53	3,78	0,22	0,21	0,11	0,01	0,00	0,00	0,00	0,00	1,05	0,04
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1		
registerPerThread	13	13	10	7	7	9	9	7	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.040	1.040	1.048	1.040	24	1.040	2.064	1.040	24	0		
occupancy	1	1	1	1	1	1	0,75	1	1	1		
divergent branch	9.500.000	380.000	74.230	74.225	0	2.969	2.969	2.969	0	0		
warp serialize	1.200.000.000	48.000.000	14.200.000	13.900.000	0	564.521	965.123	499.300	0	0		
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0		
gld 32b	3.140.000.000	297.000.000	0	0	0	0	0	0	0	0		
gld 64b	458.000.000	18.300.000	0	0	0	0	0	0	0	0		
gld 128b	459.000.000	18.400.000	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0		
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0		
gst 64b	1.380.000.000	397.000.000	60.800.000	68.000.000	27.800.000	4.000.000	3.300.000	2.260.000	872.460	21.033		
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	36,21	62,62	74,64	57,81	72,40	25,74	38,90	51,54	51,52	0,00		
glob mem write throughput (GB/s)	17,09	122,15	356,22	406,80	318,89	426,25	451,18	435,99	366,94	350,01		
glob mem overall throughput (GB/s)	53,30	184,76	430,85	464,61	391,30	451,99	490,08	487,53	418,46	350,01		
instruction throughput	0,09	0,09	0,43	0,41	0,17	0,33	0,51	0,53	0,22	0,17		

Tabla II.13. Profiling CG V5-Constant. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	28	4.103
GPU usec	51.700.000,00	2.070.000,00	121.366,00	117.424,00	62.503,30	7.072,67	4.657,82	3.491,90	1.755,84	43,46	298.432,00	40.891,40
CPU usec	51.700.000,00	2.070.000,00	144.761,00	140.859,00	86.020,00	8.021,00	5.602,00	4.418,00	2.706,00	80,00	298.501,00	85.634,00
%GPU time	94,99	3,79	0,22	0,21	0,11	0,01	0,00	0,00	0,00	0,00	0,54	0,07
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172		
gridSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeX	128	128	128	128	128	128	128	128	128	128		
blockSizeY	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1		
registerPerThread	13	13	10	7	7	9	9	7	5	4		
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0		
staSmemPerBlock	1.040	1.040	1.048	1.040	24	1.040	2.064	1.040	24	0		
occupancy	1	1	1	1	1	1	0,75	1	1	1		
divergent branch	9.500.000	380.000	74.230	74.225	0	2.969	2.969	2.969	0	0		
warp serialize	1.200.000.000	48.000.000	14.200.000	13.900.000	0	561.516	962.999	496.629	0	0		
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236		
local load	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0		
gld 32b	3.140.000.000	297.000.000	0	0	0	0	0	0	0	0		
gld 64b	458.000.000	18.300.000	0	0	0	0	0	0	0	0		
gld 128b	459.000.000	18.400.000	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0		
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0		
gst 64b	1.380.000.000	399.000.000	60.900.000	68.100.000	27.700.000	4.020.000	3.290.000	2.260.000	875.748	20.707		
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888		
tex cache hit	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0		
glob mem read throughput (GB/s)	36,21	62,62	74,64	57,86	72,47	25,62	38,90	51,89	51,59	0,00		
glob mem write throughput (GB/s)	17,11	122,74	356,61	407,94	318,32	425,92	450,88	438,80	368,64	358,13		
glob mem overall throughput (GB/s)	53,32	185,36	431,26	465,80	390,79	451,53	489,78	490,68	420,23	358,13		
instruction throughput	0,09	0,09	0,43	0,41	0,17	0,32	0,51	0,53	0,22	0,18		

Tabla II.14. Profiling CG V6-Pinned Memory. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH	Memcpy DtoHasync
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	28	4.103	1
GPU usec	61.500.000,00	2.070.000,00	128.924,00	118.211,00	62.442,50	7.060,93	4.666,59	3.495,71	1.761,09	43,84	298.436,00	40.832,90	9,95
CPU usec	61.500.000,00	2.070.000,00	152.316,00	141.658,00	85.790,00	8.002,00	5.607,00	4.421,00	2.713,00	88,00	298.495,00	85.436,00	2
%GPU time	95,74	3,21	0,20	0,18	0,09	0,01	0,00	0,00	0,00	0,00	0,46	0,06	0
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172			
gridSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeX	128	128	128	128	128	128	128	128	128	128			
blockSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeZ	1	1	1	1	1	1	1	1	1	1			
registerPerThread	13	13	10	7	7	9	9	7	5	4			
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0			
staSmemPerBlock	1.040	1.040	1.048	1.040	24	1.040	2.064	1.040	24	0			
occupancy	1	1	1	1	1	1	0,75	1	1	1			
divergent branch	9.500.000	380.000	74.230	74.225	0	2.969	2.969	2.969	0	0			
warp serialize	1.150.000.000	48.000.000	14.300.000	13.600.000	0	557.723	966.312	500.099	0	0			
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236			
local load	0	0	0	0	0	0	0	0	0	0			
local store	0	0	0	0	0	0	0	0	0	0			
gld 32b	570.000.000	297.000.000	0	0	0	0	0	0	0	0			
gld 64b	456.000.000	18.300.000	0	0	0	0	0	0	0	0			
gld 128b	456.000.000	18.400.000	5.340.000	3.560.000	1.780.000	142.512	142.512	142.512	71.256	0			
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0			
gst 64b	2.380.000.000	399.000.000	73.700.000	65.800.000	39.400.000	3.990.000	3.290.000	2.260.000	868.854	21.181			
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888			
tex cache hit	2.950.000.000	0	3.170.000	3.410.000	3110000	0	0	0	0	0			
tex cache miss	2.820.000.000	0	3.960.000	3.710.000	4010000	0	0	0	0	0			
glob mem read throughput (GB/s)	17,09	62,62	52,70	38,32	36,27	25,66	38,82	51,83	51,44	0,00			
glob mem write throughput (GB/s)	24,73	122,83	399,18	392,53	437,11	423,47	449,45	438,21	365,05	361,87			
glob mem overall throughput (GB/s)	41,82	185,45	451,88	430,85	473,38	449,13	488,27	490,04	416,49	361,87			
instruction throughput	0,09	0,09	0,43	0,44	0,23	0,33	0,51	0,53	0,22	0,18			

Tabla II.15. Profiling CG V7-TextP. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH	Memcpy DtoHasync
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	28	4.103	1
GPU usec	44.800.000,00	1.790.000,00	121.402,00	117.228,00	62.494,10	7.033,41	4.664,77	3.493,44	1.757,28	44,03	298.433,00	40.783,00	9,92
CPU usec	44.800.000,00	1.790.000,00	144.826,00	140.618,00	85.932,00	7.964,00	5.615,00	4.426,00	2.706,00	82,00	298.499,00	85.327,00	3
%GPU time	94,81	3,79	0,25	0,24	0,13	0,01	0,00	0,00	0,00	0,00	0,63	0,08	0
gridSizeX	37.500	37.500	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172			
gridSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeX	128	128	128	128	128	128	128	128	128	128			
blockSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeZ	1	1	1	1	1	1	1	1	1	1			
registerPerThread	13	13	10	7	7	9	9	7	5	4			
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0			
staSmemPerBlock	1.040	1.040	1.048	1.040	24	1.040	2.064	1.040	24	0			
occupancy	1	1	1	1	1	1	0,75	1	1	1			
divergent branch	9.500.000	380.000	74.230	74.225	0	2.969	2.969	2.969	0	0			
warp serialize	1.160.000.000	46.500.000	14.200.000	13.900.000	0	563.572	964.198	499.452	0	0			
cta launched	7.130.000	285.000	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236			
local load	0	0	0	0	0	0	0	0	0	0			
local store	0	0	0	0	0	0	0	0	0	0			
gld 32b	2.570.000.000	275.000.000	0	0	0	0	0	0	0	0			
gld 64b	1.320.000	52.597	0	0	0	0	0	0	0	0			
gld 128b	2.730.000	109.068	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0			
gst 32b	28.500.000	1.140.000	222.680	222.680	0	8.907	17.814	8.907	0	0			
gst 64b	3.650.000.000	318.000.000	60.700.000	67.900.000	27.800.000	3.980.000	3.290.000	2.260.000	869.005	20.571			
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888			
tex cache hit	1.930.000.000	77.100.000	0	0	0	0	0	0	0	0			
tex cache miss	1.950.000.000	78.000.000	0	0	0	0	0	0	0	0			
glob mem read throughput (GB/s)	18,33	48,83	74,62	57,96	72,48	25,76	38,84	51,86	51,55	0,00			
glob mem write throughput (GB/s)	52,09	112,95	355,87	407,69	318,67	424,26	450,06	437,18	365,90	351,48			
glob mem overall throughput (GB/s)	70,42	161,79	430,49	465,65	391,15	450,02	488,90	489,04	417,45	351,48			
instruction throughput	0,13	0,13	0,43	0,41	0,17	0,33	0,51	0,53	0,22	0,18			

Tabla II.16. Profiling CG V8-TextRCA. Tamaño C

Method	conj_grad_ sum1_kernel	conj_grad_ d2_kernel	conj_grad_z_ r_rho_kernel	conj_grad_ d1_kernel	conj_grad_ p_kernel	conj_grad_init_ and_rho_kernel	norm_temp _kernel	conj_grad_ sum2_kernel	x_norm_ z_kernel	init_x_ kernel	Memcpy HtoD	Memcpy DtoH	Memcpy DtoHasync
#Calls	1.900	76	1.900	1.900	1.900	76	76	76	76	2	2.004	6.004	76
GPU usec	22.400.000,00	896.454,00	121.420,00	117.066,00	62.517,50	7.061,70	4.670,02	3.499,07	1.766,40	44,03	683.966,00	1.820.000,00	758,34
CPU usec	22.400.000,00	897.875,00	145.013,00	142.344,00	86.048,00	7.989,00	5.594,00	4.500,00	2.714,00	92,00	1.140.000,00	2.600.000,00	193
%GPU time	85,76	3,43	0,46	0,44	0,23	0,02	0,01	0,01	0,00	0,00	2,61	6,96	0
gridSizeX	18.750	18.750	1.172	1.172	1.172	1.172	1.172	1.172	1.172	1.172			
gridSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeX	128	128	128	128	128	128	128	128	128	128			
blockSizeY	1	1	1	1	1	1	1	1	1	1			
blockSizeZ	1	1	1	1	1	1	1	1	1	1			
registerPerThread	13	13	10	7	7	9	9	7	5	4			
dynSmemPerBlock	0	0	0	0	0	0	0	0	0	0			
staSmemPerBlock	1.040	1.040	1.048	1.040	24	1.040	2.064	1.040	24	0			
occupancy	1	1	1	1	1	1	0,75	1	1	1			
divergent branch	4.750.000	190.000	74.230	74.225	0	2.969	2.969	2.969	0	0			
warp serialize	581.000.000	23.200.000	14.200.000	13.900.000	0	563.288	963.888	498.316	0	0			
cta launched	3.560.000	142.500	222.680	222.680	222.680	8.907	8.907	8.907	8.907	236			
local load	0	0	0	0	0	0	0	0	0	0			
local store	0	0	0	0	0	0	0	0	0	0			
gld 32b	3.430.000.000	137.000.000	0	0	0	0	0	0	0	0			
gld 64b	665.380	26.608	0	0	0	0	0	0	0	0			
gld 128b	1.400.000	55.838	7.130.000	5.340.000	3.560.000	142.512	142.512	142.512	71.256	0			
gst 32b	14.300.000	570.000	222.680	222.680	0	8.907	17.814	8.907	0	0			
gst 64b	4.030.000.000	161.000.000	60.800.000	67.800.000	27.800.000	3.980.000	3.300.000	2.250.000	871.359	22.541			
gst 128b	0	0	3.560.000	3.560.000	1.780.000	356.280	0	71.256	71.256	1.888			
tex cache hit	960.000.000	38.400.000	0	0	0	0	0	0	0	0			
tex cache miss	977.000.000	39.100.000	0	0	0	0	0	0	0	0			
glob mem read throughput (GB/s)	48,73	48,73	74,61	58,04	72,45	25,66	38,80	51,78	51,28	0,00			
glob mem write throughput (GB/s)	114,64	114,50	356,27	407,69	318,72	423,05	449,89	436,06	364,85	379,92			
glob mem overall throughput (GB/s)	163,37	163,22	430,88	465,73	391,17	448,71	488,69	487,84	416,14	379,92			
instruction throughput	0,13	0,13	0,43	0,41	0,17	0,33	0,51	0,53	0,22	0,18			

Tabla II.17. Profiling CG V9-Multi-GPU. Tamaño C. 2 GPUs (GPU0)

Method	conj_grad_sum1_kernel	conj_grad_d2_kernel	MemcpyHtoD	MemcpyDtoH
#Calls	1.900	76	2.004	1.976
GPU usec	22.400.000,00	896.040,00	7.230.000,00	424.677,00
CPU usec	22.400.000,00	897.161,00	8.790.000,00	1.000.000,00
%GPU time	72,37	2,89	23,36	1,37
gridSizeX	18.750	18.750		
gridSizeY	1	1		
blockSizeX	128	128		
blockSizeY	1	1		
blockSizeZ	1	1		
registerPerThread	13	13		
dynSmemPerBlock	0	0		
staSmemPerBlock	1.040	1.040		
occupancy	1	1		
divergent branch	4.750.000	190.000		
warp serialize	577.000.000	23.100.000		
cta launched	3.560.000	142.500		
local load	0	0		
local store	0	0		
gld 32b	3.430.000.000	137.000.000		
gld 64b	649.800	25.992		
gld 128b	1.350.000	54.112		
gst 32b	14.300.000	570.000		
gst 64b	4.050.000.000	162.000.000		
gst 128b	0	0		
tex cache hit	961.000.000	38.400.000		
tex cache miss	975.000.000	39.000.000		
glob mem read throughput (GB/s)	49,07	49,06		
glob mem write throughput (GB/s)	115,95	115,76		
glob mem overall throughput (GB/s)	165,02	164,82		
instruction throughput	0,13	0,13		

Tabla II.18. Profiling CG V9-Multi-GPU. Tamaño C. 2 GPUs (GPU1)

II.3. FTDock Profilings

index	% time	self	children	called	name
[1]	100.0	589.53	8646.94		main
		0.00	4151.47	9314/9314	fftwf_execute_dft_r2c
		0.01	4150.57	9312/9312	fftwf_execute_dft_c2r
		86.35	50.58	01/01	electric_field
		67.34	28.32	4657/4657	discretise_structure
		41.36	0.37	4656/4656	electric_point_charge
		0.00	35.16	01/01	fftwf_plan_dft_r2c_3d
		0.00	35.16	01/01	fftwf_plan_dft_c2r_3d
		0.19	0.00	01/01	surface_grid
		0.03	0.02	4656/4656	rotate_structure
		0.01	0.00	01/01	electric_field_zero_core
		0.00	0.00	02/02	translate_structure_onto_origin
		0.00	0.00	02/02	fftwf_destroy_plan
		0.00	0.00	06/06	fftwf_malloc
		0.00	0.00	03/03	fftwf_free
		0.00	0.00	02/02	read_pdb_to_structure
		0.00	0.00	02/02	assign_charges
		0.00	0.00	01/01	generate_global_angles
		0.00	0.00	01/01	total_span_of_structures
		0.00	0.00	01/01	qsort_scores
		0.00	4151.47	9314/9314	fftwf_execute_dft_r2c
…	…	…	…	…	…

Tabla II.19. Profiling FTDock Versión Serie (*gprof*). E1-128

(*) La realización de este *profiling* tuvo lugar con la librería FFTW compilada con las opciones –g –pg, por lo que el tiempo de ejecución final no debe ser tenido en cuenta.

Method	SP_c2c_transpose_kernel	SP_c2c_radix2_sp_kernel	SP_c2r_radix2_sp_kernel	SP_r2c_radix2_sp_kernel	memcpyHtoD	memcpyDtoH
#Calls	9.540.000	4.770.000	9.312	9.314	18.626	18.626
GPU usec	80.700.000,00	53.700.000,00	9.790.000,00	9.300.000,00	71.400.000,00	122.000.000,00
CPU usec	189.000.000,00	114.000.000,00	9.900.000,00	9.440.000,00	79.700.000,00	131.000.000,00
%GPU time	23,27	15,51	2,82	2,68	20,59	35,10
gridSizeX	3	65	16.384	16.384		
gridSizeY	4	1	1	1		
blockSizeX	32	64	64	64		
blockSizeY	8	1	1	1		
blockSizeZ	1	1	1	1		
registerPerThread	12	13	13	13		
dynSmemPerBlock	8576	1024	1024	1024		
staSmemPerBlock	64	80	80	80		
occupancy	0,25	0,5	0,5	0,5		
divergent branch	10.200.000	0	5.080.000	5.090.000		
warp serialize	0	1.530.000.000	609.000.000	605.000.000		
cta launched	11.400.000	31.000.000	15.300.000	15.300.000		
local load	0	0	0	0		
local store	0	0	0	0		
gld 32b	183.000.000	0	61.000.000	61.000.000		
gld 64b	122.000.000	0	30.500.000	61.000.000		
gld 128b	477.000.000	248.000.000	53.400.000	45.800.000		
gst 32b	183.000.000	0	61.000.000	45.800.000		
gst 64b	2.640.000.000	3.650.000.000	2.850.000.000	2.500.000.000		
gst 128b	477.000.000	248.000.000	45.800.000	53.400.000		
tex cache hit	0	0	0	0		
tex cache miss	0	0	0	0		
glob mem read throughput (GB/s)	9,26	5,90	10,97	12,60		
glob mem write throughput (GB/s)	29,26	49,34	194,15	181,10		
glob mem overall throughput (GB/s)	38,52	55,24	205,12	193,70		
instruction throughput	0,00	0,13	0,94	0,95		

Tabla II.20. Profiling FTDock V1-CUFFT. E1-128

Method	SP_c2c_ trans	SP_c2c_ radix2_sp	get_best_ scores	SP_c2r_ radix2_sp	SP_r2c_ radix2_sp	Convolution _kernel	discretise_ structure	electric_ field_calc	rotate_mobile _structure	electric_point_ charge_kernel	Surface grid	Memcpy HtoD	Memcpy DtoH
#Calls	9.540.000	4.770.000	4.656	9.312	9.314	4.656	4.657	1	4.656	4.656	1	590	4656
GPU usec	80.700.000,00	53.800.000,00	26.900.000	9.790.000,00	9.300.000,00	3.020.000,00	2.490.000,00	313.360,00	93.185,10	87.721,40	4.957,44	2.135,36	7740000
CPU usec	190.000.000,00	114.000.000,00	27.000.000,00	9.900.000,00	9.410.000,00	3.080.000,00	2.550.000,00	313.376,00	153.214,00	145.465,00	4.970,00	1.644,00	10100000
%GPU time	41,54	27,68	13,85	5,03	4,78	1,55	1,28	0,16	0,04	0,04	0,00	0,00	3,98
gridSizeX	3	65	128	16.384	16.384	128	15	128	15	15	128		
gridSizeY	4	1	128	1	1	128	1	128	1	1	128		
blockSizeX	32	64	128	64	64	65	32	128	32	32	128		
blockSizeY	8	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1	1		
registerPerThread	12	13	13	13	13	3	23	16	20	25	23		
dynSmemPerBlock	8576	1024	532	1024	1024	0	0	0	0	0	0		
staSmemPerBlock	64	80	64	80	80	80	80	64	64	80	48		
occupancy	0,25	0,5	1	0,5	0,5	0,75	0,25	1	0,25	0,25	0,625		
divergent branch	10.200.000	0	19.500.000	5.090.000	5.090.000	0	118.969	9.812	155	2.484	15921		
warp serialize	0	1.530.000.000	0	609.000.000	605.000.000	0	0	0	0	0	0		
cta launched	11.400.000	31.000.000	7.630.000	15.300.000	15.300.000	7.630.000	6.994	1.639	6.985	6.985	1638		
local load	0	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0	0		
gld 32b	183.000.000	0	83.200.000	61.000.000	61.000.000	91.500.000	15.900.000	102.000.000	142.962	1.230.000	60447		
gld 64b	122.000.000	0	54.200.000	30.500.000	61.000.000	61.000.000	11.900.000	6.560	103.379	110.099	36667		
gld 128b	477.000.000	248.000.000	45.500.000	56.300.000	53.400.000	114.000.000	69.100.000	5.736	358.072	190.592	27477		
gst 32b	183.000.000	0	114.000.000	61.000.000	45.800.000	45.800.000	4.870.000	6.948	26.544	505.156	383		
gst 64b	2.640.000.000	3.660.000.000	2.670.000.000	2.840.000.000	2.500.000.000	1.180.000.000	1.820.000.000	229.000.000	59.600.000	55.700.000	3650000		
gst 128b	477.000.000	248.000.000	0	53.400.000	57.200.000	57.200.000	39.199	5.817	194.169	1.548	75		
tex cache hit	0	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0	0		
g mem rd (GB/s)	9,51	6,07	4,57	11,66	14,03	72,96	41,77	106,87	6,29	8,27	16,16		
g mem wr (GB/s)	30,01	50,81	66,57	199,91	186,22	285,91	482,14	480,06	422,99	419,61	484,69		
g mem ov (GB/s)	39,52	56,88	71,13	211,57	200,24	358,87	523,91	586,93	429,28	427,88	500,85		
instruction thrput	0,00	0,13	0,36	0,94	0,95	0,29	0,03	1,33	0,03	0,04	0,44		

Tabla II.21. Profiling FTDock V2-Full. E1-128

Method	SP_c2c_ trans	SP_c2c_ radix2_sp	get_best_ scores	SP_c2r_ radix2_sp	SP_r2c_ radix2_sp	Convolution _kernel	discretise_ structure	electric_ field_calc	rotate_mobile _structure	electric_point_ charge_kernel	Surface grid	Memcpy HtoD	Memcpy DtoH
#Calls	9.540.000	4.770.000	4.656	9.312	9.314	4.656	4.657	1	4.656	4.656	1	590	4656
GPU usec	80.700.000,00	53.800.000,00	26.900.000,00	9.790.000,00	9.300.000,00	3.020.000,00	2.490.000,00	313.360,00	93.185,10	87.721,40	4.957,44	2.135,36	7740000
CPU usec	190.000.000,00	114.000.000,00	27.000.000,00	9.900.000,00	9.410.000,00	3.080.000,00	2.550.000,00	313.376,00	153.214,00	145.465,00	4.970,00	1.644,00	10100000
%GPU time	41,54	27,68	13,85	5,03	4,78	1,55	1,28	0,16	0,04	0,04	0,00	0,00	3,98
gridSizeX	3	65	128	16.384	16.384	128	15	128	15	15	128		
gridSizeY	4	1	128	1	1	128	1	128	1	1	128		
blockSizeX	32	64	128	64	64	65	32	128	32	32	128		
blockSizeY	8	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1	1		
registerPerThread	12	13	13	13	13	3	23	16	20	25	23		
dynSmemPerBlock	8576	1024	532	1024	1024	0	0	0	0	0	0		
staSmemPerBlock	64	80	64	80	80	80	80	64	64	80	48		
occupancy	0,25	0,5	1	0,5	0,5	0,75	0,25	1	0,25	0,25	0,625		
divergent branch	10.200.000	0	19.500.000	5.090.000	5.090.000	0	118.969	9.812	155	2.484	15921		
warp serialize	0	1.530.000.000	0	609.000.000	605.000.000	0	0	0	0	0	0		
cta launched	11.400.000	31.000.000	7.630.000	15.300.000	15.300.000	7.630.000	6.994	1.639	6.985	6.985	1638		
local load	0	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0	0		
gld 32b	183.000.000	0	83.200.000	61.000.000	61.000.000	91.500.000	15.900.000	102.000.000	142.962	1.230.000	60447		
gld 64b	122.000.000	0	54.200.000	30.500.000	61.000.000	61.000.000	11.900.000	6.560	103.379	110.099	36667		
gld 128b	477.000.000	248.000.000	45.500.000	56.300.000	53.400.000	114.000.000	69.100.000	5.736	358.072	190.592	27477		
gst 32b	183.000.000	0	114.000.000	61.000.000	45.800.000	45.800.000	4.870.000	6.948	26.544	505.156	383		
gst 64b	2.640.000.000	3.660.000.000	2.670.000.000	2.840.000.000	2.500.000.000	1.180.000.000	1.820.000.000	229.000.000	59.600.000	55.700.000	3650000		
gst 128b	477.000.000	248.000.000	0	53.400.000	57.200.000	57.200.000	39.199	5.817	194.169	1.548	75		
tex cache hit	0	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0	0		
g mem rd (GB/s)	9,51	6,07	4,57	11,66	14,03	72,96	41,77	106,87	6,29	8,27	16,16		
g mem wr (GB/s)	30,01	50,81	66,57	199,91	186,22	285,91	482,14	480,06	422,99	419,61	484,69		
g mem ov (GB/s)	39,52	56,88	71,13	211,57	200,24	358,87	523,91	586,93	429,28	427,88	500,85		
instruction thrput	0,00	0,13	0,36	0,94	0,95	0,29	0,03	1,33	0,03	0,04	0,44		

Tabla II.22. Profiling FTDock V4-Multi-GPU. E1-128. (GPU 0)

Method	SP_c2c_ trans	SP_c2c_ radix2_sp	get_best_ scores	SP_c2r_ radix2_sp	SP_r2c_ radix2_sp	Convolution _kernel	discretise_ structure	electric_ field_calc	rotate_mobile _structure	electric_point_ charge_kernel	Surface grid	Memcpy HtoD	Memcpy DtoH
#Calls	4.770.000	2.380.000	2.327	4.655	4.658	2.328	2.329	1	2.328	2.328	1	590	2327
GPU usec	42.400.000,00	26.300.000,00	12.300.000,00	4.890.000,00	4.650.000,00	1.510.000,00	1.250.000,00	310.178,00	46.664,10	43.929,30	4.954,37	2.120,10	2550000
CPU usec	108.000.000,00	57.200.000,00	12.300.000,00	4.960.000,00	4.720.000,00	1.550.000,00	1.280.000,00	310.196,00	88.411,00	74.869,00	4.967,00	1.800,00	3840000
%GPU time	44,07	27,29	12,75	5,08	4,83	1,57	1,29	0,32	0,04	0,04	0,00	0,00	2,65
gridSizeX	3	65	128	16.384	16.384	128	15	128	15	15	128		
gridSizeY	4	1	128	1	1	128	1	128	1	1	128		
blockSizeX	32	64	128	64	64	65	32	128	32	32	128		
blockSizeY	8	1	1	1	1	1	1	1	1	1	1		
blockSizeZ	1	1	1	1	1	1	1	1	1	1	1		
registerPerThread	12	13	13	13	13	3	23	16	20	25	23		
dynSmemPerBlock	8576	1024	532	1024	1024	0	0	0	0	0	0		
staSmemPerBlock	64	80	64	80	80	80	80	64	64	80	48		
occupancy	0,25	0,5	1	0,5	0,5	0,75	0,25	1	0,25	0,25	0,625		
divergent branch	5.090.000	0	11.000.000	2.540.000	2.540.000	0	59.765	9.812	78	1.243	15921		
warp serialize	0	763.000.000	2.540.000	304.000.000	303.000.000	0	0	0	0	0	0		
cta launched	5.720.000	15.500.000	3.810.000	7.630.000	7.630.000	3.810.000	3.503	1.639	3.492	3.492	1638		
local load	0	0	0	0	0	0	0	0	0	0	0		
local store	0	0	0	0	0	0	0	0	0	0	0		
gld 32b	91.600.000	0	41.600.000	30.500.000	30.500.000	45.800.000	7.950.000	102.000.000	71.474	612.762	60447		
gld 64b	61.000.000	0	27.100.000	15.300.000	30.500.000	30.500.000	5.960.000	6.560	51.689	54.978	36667		
gld 128b	238.000.000	124.000.000	22.700.000	28.100.000	26.700.000	57.200.000	34.600.000	5.736	179.003	95.237	27477		
gst 32b	91.600.000	0	67.600.000	30.500.000	22.900.000	22.900.000	2.440.000	6.948	13.270	252.578	383		
gst 64b	1.320.000.000	1.830.000.000	455.000.000	3.570.000.000	3.400.000.000	590.000.000	912.000.000	227.000.000	29.800.000	27.900.000	3660000		
gst 128b	238.000.000	124.000.000	0	26.700.000	28.600.000	28.600.000	19.157	5.817	97.066	748	75		
tex cache hit	0	0	0	0	0	0	0	0	0	0	0		
tex cache miss	0	0	0	0	0	0	0	0	0	0	0		
g mem rd (GB/s)	9,05	6,21	5,01	11,66	14,03	72,94	41,81	107,96	6,28	8,26	16,17		
g mem wr (GB/s)	28,60	52,04	26,21	488,44	489,89	286,27	482,07	481,76	422,68	419,05	485,11		
g mem ov (GB/s)	37,66	58,25	31,22	500,10	503,91	359,21	523,88	589,72	428,96	427,31	501,28		
instruction thrput	0,14	0,35	0,78	0,94	0,95	0,29	0,03	1,35	0,03	0,04	0,44		

Tabla II.23. Profiling FTDock V4-Multi-GPU. E1-128. (GPU 1)

Anexo III

Tiempos de Ejecución

En este anexo se incluyen los tiempos de ejecución de las distintas versiones, **en segundos**, con el objetivo de que puedan compararse los resultados con cualquier otro tipo de arquitectura.

III.1. EP

	S	W	A	B	C
Opteron Serie	2,8245	5,6670	45,1905	181,0447	737,0196
Opteron 4 Cores	0,7132	1,4289	11,4360	45,9587	182,9605
Xeon 8 Cores	0,3278	0,6489	5,1431	20,5443	82,1905
CUDA V1	0,1957	0,2967	1,5157	5,6002	21,9072
CUDA V2-SM	0,1340	0,1808	0,8830	3,2848	12,8882
CUDA V3-Occup	0,1344	0,1806	0,8759	3,2536	12,7520
CUDA V4-Pinned	0,0588	0,1051	0,8003	3,1778	12,6755
CUDA V5-Zero	0,0586	0,1049	0,8002	3,1783	12,6780
CUDA V6-QQsLM	0,0583	0,1043	0,7951	3,1574	12,5951
CUDA V7-Unroll	0,0582	0,1041	0,7937	3,1519	12,5733
CUDA V8-MultiGPU (2x)	0,1804	0,2010	0,5462	1,7197	6,4377
Altix (32 Cores)	0,3312	0,4173	1,5905	5,5734	20,6053

Tabla III.1. EP: Tiempos de Ejecución

III.2. CG

	S	W	A	B	C
Opteron Serie	0,0869	0,8412	3,5445	202,0841	1367,6627
Opteron 4 Cores	0,0238	0,5108	1,8697	87,5053	570,4877
Xeon 8 Cores	0,0168	0,0844	0,5342	51,7104	138,4181
CUDA V1	0,0813	0,3001	1,1220	53,7293	159,8296
CUDA V2-Warp	0,0501	0,1297	0,3645	16,3016	59,5318
CUDA V3-Half-Warp	0,0495	0,1287	0,3650	18,3666	64,2326
CUDA V4-Padding	0,0496	0,1259	0,3478	14,1489	53,8212
CUDA V5-Constant	0,0495	0,1268	0,3483	14,1494	53,8227
CUDA V6-Pinned	0,0486	0,1244	0,3425	14,0792	53,6503
CUDA V7-TextP	0,0464	0,1279	0,3661	19,0550	63,3112
CUDA V8-TextRCA	0,0494	0,1176	0,3116	11,9416	46,5476
CUDA V9-MultiGPU	0,0739	0,2110	0,5380	10,6330	32,9276
Altix (32 Cores)	0,1758	0,2510	0,3922	56,2229	185,3109

Tabla III.2. CG: Tiempos de Ejecución

III.3. FTDock

	E1-G128	E1-G256	E2-G128	E2-G256
Opteron Serie	1.216,2500	11.384,4800	3.912,1600	38.673,5450
CUDA V1-CUFFT	679,9400	4.869,3433	2.090,6967	15.053,4967
CUDA V2-Full	203,7400	1.041,1040	689,0100	3.553,2760
CUDA V3-Buffering	204,3275	1.021,1375	734,0900	3.517,6750
CUDA V4-MultiGPU	104,8380	514,9500	360,9360	1.778,4000
CUDA V5-MultiNodo	70,4620	512,2820	206,9620	1.519,4471
CUDA V6-MultiNodo-Het	54,4560	322,8820	180,1000	1.093,3700
Cell (8 SPEs)	707,4500	7.137,3320	1.158,1000	9.313,4700

Tabla III.3. FTDock: Tiempos de Ejecución

Anexo IV

Guía de compilación y ejecución del código fuente

IV.1. NAS Parallel Benchmarks

La versiones de NPB que van adjuntas a esta documentación vienen listas para ser ejecutadas sin demasiado esfuerzo. Antes de compilar, para el caso de CUDA, debemos elegir las opciones de compilación de la versión correspondiente. Para ello editaremos el fichero `./config/make.def` desde el directorio principal de NPB y quitaremos los comentarios de la opción de compilación de la versión con la que vayamos a trabajar. En la Figura IV.1 podemos ver las distintas opciones que existen, donde en este caso la compilación está configurada para la versión nueve de CG.

```
#---------------------------------------------------------------------
# Global *compile time* flags for CUDA programs
#---------------------------------------------------------------------

### EP (V1,V2)
#CUDAFLAGS = -O3 -arch sm_13 -DPERFORMANCE

### EP (V3-V8)
#CUDAFLAGS = -O3 -arch sm_13 -DPERFORMANCE -maxrregcount=20

### EP (V9)
#CUDAFLAGS=-O3 -arch sm_13 -DPERFORMANCE -maxrregcount=20 –Xcomp...

### CG (V1-V8)
#CUDAFLAGS = -O2 -arch sm_13 -DPERFORMANCE

### CG (V9)
CUDAFLAGS = -O2 -arch sm_13 -DPERFORMANCE -Xcompiler -fopenmp
```

Figura IV.1. Fichero *./config/make.def*

La interfaz de compilación que traen por defecto los NAS necesita del nombre del *benchmark* que se quiere compilar, así como del tamaño o clase del problema. El comando correspondiente deben ser ejecutados desde el directorio raíz del propio *benchmark* y dará como resultado un fichero ejecutable en formato `nombreBenchmark.tamaño` que será alojado en la carpeta `./bin`. La Figura IV.2 muestra diferentes ejemplos de compilación y ejecución bastante auto explicativos.

`make EP CLASS=S` → Compila el benchmark EP con el tamaño S.
`make EP CLASS=B` → Compila el benchmark EP con el tamaño B.
`make CG CLASS=W` → Compila el benchmark CG con el tamaño W.
`make CG CLASS=C` → Compila el benchmark CG con el tamaño C.
`./bin/ep.S` → Ejecuta EP con el tamaño S.
`./bin/ep.B` → Ejecuta EP con el tamaño B.
`./bin/cg.W` → Ejecuta CG con el tamaño W.
`./bin/cg.C` → Ejecuta CG con el tamaño C.

Figura IV.2. Comandos para compilar y ejecutar NPB

Para alternar entre las distintas versiones CUDA implementadas será necesario copiar el contenido de la versión correspondiente al directorio principal del *benchmark* al que hacen referencia y proceder con los pasos de compilación y ejecución antes descritos. Cada una de las versiones se encuentra alojada en un directorio independiente, dentro de `./EP` o `./CG` respectivamente.

Por ejemplo, para hacer uso de la versión nueve de CG tendríamos que anteponer el comando `cp ./CG/V9-TextRowstr/* ./CG` a la fase de compilación, y sustituir todos los ficheros que se encuentran en `./CG`.

La versión utilizada del kit de desarrollo de CUDA ha sido la 3.0, por lo que no garantizamos el funcionamiento con versiones anteriores.

IV.2. FTDock

Para compilar la versión serie de FTDock necesitaremos tener instalada la librería FFTW. La utilizada en el proyecto para su ejecución ha sido la versión 3.3.2, por lo que podemos garantizar el funcionamiento sólo con la misma. Las versiones CUDA no necesitan de librerías especiales más allá de las que ya vienen instaladas con kit de desarrollo 3.0. Con versiones anteriores, FTDock podría no ser compatible.

El código de ambas versiones se encuentra en el directorio `./progs` y `./progs_cuda` respectivamente, dentro de la carpeta raíz destinada al contenido de la aplicación.

La compilación se reduce a hacer un *make* dentro del directorio donde se encuentra el código fuente, y la interfaz de ejecución en ambas es idéntica:

```
./ftdock –static fichero_proteina_estática
         –mobile fichero_proteina_móvil
         [–grid tamaño_grid]
```

Para alternar entre las distintas versiones CUDA, al igual que con NPB, necesitaremos copiar todos los ficheros relacionados con la versión al directorio principal del código CUDA.

Por ejemplo, para hacer uso de la versión dos de FTDock tendríamos que anteponer el comando `cp ./progs_cuda/V2-Full/* ./CG` a la fase de compilación, y sustituir todos los ficheros que se encuentran en `./progs_cuda`. Resulta imprescindible que si existen ficheros que no son sustituidos en la copia sean borrados, pues sino obtendremos errores en la compilación. Así, podríamos utilizar el comando *rm* `./progs_cuda/*` como previo al comando `cp`.

La versión utilizada del kit de desarrollo de CUDA ha sido la 3.0, por lo que no garantizamos el funcionamiento con versiones anteriores.