Universidad Politécnica de Valencia
Máster en Computación Paralela y Distribuida

Trabajo Fin de Máster

Modelo de estimación de rendimiento para arquitecturas paralelas heterogéneas

Alumno:
Cristina Yenyxe González García

Director:
Antonio M. Vidal Maciá

Febrero de 2012

Departamento de Sistemas Informáticos
y Computación

Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

Abstract

A performance model predicts the cost of an algorithm, based on a set of parameters. In the field of parallel CPU computing there are many models which allow to perform a theoretical estimation, but just a few of them exist for algorithms executed in GPU. Moreover, the latter present two important limitations:

1. They evaluate only GPU code. They can't estimate costs in a heterogeneous system, which uses both CPU and GPU simultaneously to solve a problem, because they don't consider decisive factors like the time consumed in transfers between them.

2. Some of them demand the algorithms to have been already implemented, because they conduct a low-level study.

This work aims to fulfill this void, providing a high-level model which allows to estimate cost easily and supports evaluating algorithms executed in heterogeneous architectures. In order achieve this goal, previous GPU models are compared using some simple algorithms, and the model which exhibits the best performance is the one described by Mukherjee *et al.* [1]. It is taken as a basis and new considerations are included in order to improve its estimations:

- Data transfers between CPU and GPU
- Execution divergencies due to conditional expressions
- Basic cache memory design
- Atomic operations

Finally, some algorithms are implemented in order to validate the new model estimations, compared both with experimental results and the ones got with the base model. These algorithms are reduction, matrix-vector product and Cholesky's decomposition, the latter being also a dense structured approach to solve the problem introduced by Ródenas *et al.* in [2].

Keywords: Heterogeneous computing, parallel computing, performance model, GPU (Graphics Processing Unit), CUDA (Compute Unified Device Architecture), numerical computing.

Resumen

Un modelo de estimación de rendimiento predice el coste de un algoritmo a partir de una serie de parámetros. En el campo de la computación paralela en CPU se dispone de múltiples modelos para realizar esta estimación de manera teórica, pero si el algoritmo se ejecuta sobre una GPU solo existen unos pocos escasamente extendidos. Además, estos últimos presentan dos importantes carencias:

1. Solo evalúan el código ejecutado en GPU. No permiten estimar el coste completo en un sistema heterogéneo, que emplea CPU y GPU simultáneamente para la resolución de un problema, al no tener en cuenta factores determinantes como el tiempo de copia de datos entre ellas.

2. En algunos casos exigen que los algoritmos ya hayan sido implementados, pues realizan estudios a bajo nivel.

El objetivo de este trabajo es cubrir de alguna manera este vacío, proporcionando un modelo de alto nivel que permita estimar costes de una manera sencilla y que dé soporte para la evaluación de algoritmos ejecutados sobre arquitecturas heterogéneas. Para ello, se comparan varios modelos de GPU ya existentes, y el que exhibe un mejor rendimiento es el modelo descrito por Mukherjee *et al.* [1]. Por ello, se toma como base y se le añaden nuevas consideraciones para mejorar las estimaciones:

- Copia de datos entre CPU y GPU
- Divergencias de ejecución debidas a expresiones condicionales
- Diseño básico del funcionamiento de la memoria caché
- Operaciones atómicas

Finalmente, se implementan varios algoritmos para validar las estimaciones del nuevo modelo, comparándolo con los resultados experimentales y con los obtenidos con el modelo de referencia. Dichos algoritmos son la reducción, el producto matriz-vector y la factorización de Cholesky. Este último ofrece, además, un enfoque denso estructurado orientado a resolver el problema planteado por Ródenas *et al.* en [2].

Palabras clave: Computación heterogénea, computación paralela, modelo de rendimiento, GPU (Unidad de Procesamiento Gráfico), CUDA Compute Unified Device Architecture), computación numérica.

Agradecimientos

Cuando empecé me parecía imposible llegar hasta aquí. Incluso hoy, con la mayor parte del trabajo hecho, me lo sigue pareciendo un poquito.

Gracias a todos los que confiaron más en mí que yo misma.

Índice general

Índice de figuras

1

Introducción

Tradicionalmente, el desarrollo de software paralelo se llevaba a cabo sobre arquitecturas basadas únicamente en CPU y, para medir el coste de un algoritmo ejecutado en ellas, se disponía de múltiples modelos como PRAM [3] para el caso de memoria compartida y DRAM [4] para sistemas con memoria distribuida. Sin embargo, en los últimos años ha proliferado el uso de tarjetas gráficas o GPU (*Graphics Processing Unit*) en la computación de propósito general, dando lugar al modelo de programación conocido como GPGPU.

Por las diferencias existentes entre la arquitectura de una CPU y una GPU, los modelos arriba citados no permiten estimar el coste de los algoritmos implementados sobre estas tarjetas, por lo que distintos autores han intentando desarrollar modelos adaptados a ellas [5] [6] [1] [7].

Pese a que en un primer *boom* en el uso de las GPU se dejó un poco de lado el empleo de la CPU, actualmente los desarrollos se están orientando al máximo aprovechamiento de los equipos, dando lugar a la denominada "computación híbrida". El objetivo principal de este trabajo consistirá en definir un nuevo modelo que englobe la parte de programación en GPU, pero que también incluya consideraciones como el coste de transferencia de datos entre CPU y GPU.

Los modelos orientados únicamente a CPU o a GPU no se pueden ajustar directamente a las arquitecturas híbridas, pero sí se estudiará su idoneidad para tomarlos como base para el nuevo modelo. En caso de que alguno de ellos resulte aceptable, las ampliaciones y mejoras se llevarán a cabo sobre él.

Una vez definido el modelo, para validarlo se emplearán varios algoritmos de diferente complejidad, y diseñados siguiendo un enfoque heterogéneo cuando sea posible: la reducción, el producto matriz-vector y la factorización de Cholesky.

Para cubrir todos estos aspectos, el trabajo se estructurará como sigue:

Capítulo 2. Describe las características de las arquitecturas heterogéneas y de la arquitectura CUDA [8], desarrollada por NVIDIA para facilitar el uso de sus GPU en computación de propósito general.

Capítulo 3. Muestra algunos de los modelos teóricos de evaluación de costes en arquitecturas orientadas a CPU y GPU.

Capítulo 4. Una vez conocido el estado del arte, establece los objetivos del proyecto y las características que debe cumplir el modelo obtenido como resultado.

Capítulo 5. En una primera parte evalúa de manera experimental los modelos descritos en el capítulo 3 para decidir si alguno de ellos es adecuado para tomarlo como base del modelo a desarrollar. En la segunda parte describe los puntos del modelo desarrollados en este trabajo y realiza lo define de manera comprensible para su posterior aplicación.

Capítulo 6. Describe el algoritmo de reducción, primer caso de uso para evaluar el rendimiento del modelo desarrollado. Para ello, se toman como base dos implementaciones con un uso diferente de las diferentes memorias que la arquitectura CUDA pone a disposición del desarrollador.

Capítulo 7. Describe el algoritmo del producto matriz-vector, segundo caso de uso a emplear en la evaluación del modelo. Se emplean tres enfoques, cada uno de los cuales permite evaluar una característica diferente de las GPU.

Capítulo 8. Describe el algoritmo de la factorización de Cholesky siguiendo varias aproximaciones, partiendo de un esquema denso secuencial hasta llegar a una implementación dispersa paralela en GPU.

Capítulo 9. Para cada uno de los algoritmos descritos en los 3 capítulos anteriores, realiza una estimación del coste empleando el modelo desarrollado y la compara con los resultados obtenidos de manera experimental.

Capítulo 10. Incluye las conclusiones al trabajo y posibles ampliaciones futuras.

Apéndices. El primer apéndice describe las bibliotecas basadas en la plataforma CUDA empleadas durante este trabajo. El segundo apéndice incluye la relación completa de tablas con mediciones realizadas experimentalmente y las estimaciones realizadas mediante el modelo teórico.

2

Modelo de computador heterogéneo

Durante años, los procesadores de propósito general han ido evolucionando en su arquitectura para permitir incluir un mayor número de transistores en menor espacio, aumentando su frecuencia de reloj o el número de núcleos que los componen. En la actualidad, cualquier usuario doméstico puede disponer de equipos basados en tecnología de 32 nm (y posiblemente 22 nm en un futuro próximo), con frecuencias de reloj superiores a los 3 GHz y 4 núcleos. En el mercado de servidores el número de núcleos se puede disparar hasta 12, sin contar las posibilidades ofrecidas por tecnologías como el *hyper-threading*.

Sin embargo, para obtener un alto rendimiento en computación paralela usando estos procesadores es necesario disponer de un gran número de equipos interconectados mediante una red de alta velocidad, lo que supone un coste considerable tanto de adquisición como de consumo. Por este motivo se ha intentado buscar alternativas que proporcionen una mejor relación capacidad de computación/precio, como el uso de tarjetas gráficas o GPU.

2.1. Arquitecturas heterogéneas y programación híbrida

Los sistemas heterogéneos son aquellos cuyas unidades computacionales poseen diferentes características: diferente juego de instrucciones, frecuencia, capacidades aritméticas, etc. [9] Un ejemplo son aquellos que combinan CPU y GPU, y que serán en los que se centre este trabajo.

Cuando los procesadores de un sistema son idénticos se intenta asignar la misma carga de trabajo a todos ellos, pero en máquinas heterogéneas los procesadores más rápidos terminarán antes el trabajo y deberán esperar por los más lentos. Es decir, el rendimiento será el mismo que si se utilizase una máquina homogénea compuesta únicamente por procesadores del modelo más lento.

Por tanto, una buena paralelización en este tipo de sistemas debe distribuir la carga de manera desigual, asignando la mayor cantidad del trabajo a los procesadores más rápidos. Esta estimación del reparto de la carga será la que decida en gran medida el rendimiento alcanzado.

En un sistema que combine CPU y GPU, lo habitual será proporcionar a la GPU la mayor carga de computación y hacer que la CPU se encargue del reparto de tareas y de una pequeña parte de la computación que no se ajuste al tamaño óptimo de trabajo de la GPU. Este enfoque ya está siendo empleado en bibliotecas como MAGMA [10], un *port* de LAPACK para este tipo de arquitecturas.

Otro punto destacable es la diferencia en la aritmética de coma flotante, que afecta especialmente al *software* de computación numérica. Dos procesadores distintos no garantizan que se utilice la misma representación para almacenar un número, además de que si se debe transmitir entre ellos, serán necesarias dos conversiones: una de la representación original a otra independiente de la plataforma, y de esta a la representación en el procesador de destino.

En el caso concreto de las GPU, se puede destacar que aquellas con una capacidad computacional inferior a la 1.3 (ver apéndice A de [11]) no podían realizar cálculos en coma flotante de doble precisión. En el caso de la capacidad 1.3, el *hardware* daba soporte a la multiplicación-suma fusionada (FMA) en doble precisión, y en aquellas GPU con capacidad 2.0 o superior se incluyó esta misma funcionalidad también para simple precisión [12].

Además, presentan varias diferencias respecto a la arquitectura x86:

- Los modos del redondeo (al más cercano, al alza, a la baja...) se codifican en cada instrucción en lugar de usar una palabra de control.
- No soportan manejadores de *traps* para excepciones en coma flotante.
- No disponen de banderas para indicar si ha habido un *overflow* o *underflow* en los cálculos.
- Al igual que en SSE, la precisión de las operaciones se codifica en la instrucción, mientras que en x87 se utiliza la palabra de control.

2.2. La arquitectura CUDA

Aunque en estos últimos años diferentes fabricantes han impulsado su propia arquitectura, el presente trabajo se centrará en CUDA, diseñada por NVIDIA para facilitar la explotación del paralelismo de sus GPU. Hasta el nacimiento de esta tecnología en 2006 con la arquitectura G80, el aprovechamiento de las tarjetas en aplicaciones no orientadas a gráficos era complicado [13]:

- La programación en la GPU solo se podía realizar a través de la API gráfica, con lo que la curva de aprendizaje y la sobrecarga por utilizar una API mal adaptada eran altas.

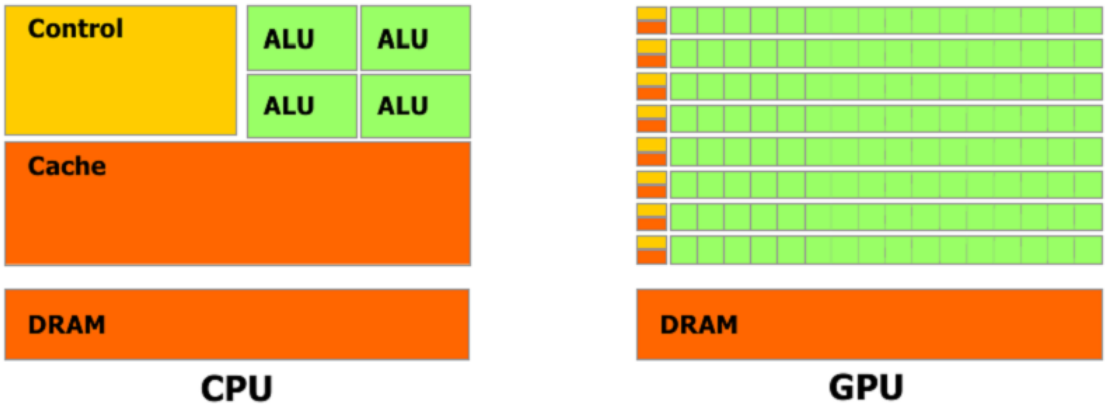

Figura 2.1: Estructura de una CPU y una GPU

- Los programas veían limitada su flexibilidad al poder leer pero no escribir sobre cualquier parte de la DRAM de la GPU.
- Algunas aplicaciones tenían el cuello de botella en el ancho de banda de la memoria DRAM e infrautilizaban la potencia computacional de la GPU.

Por ello, diseñaron un *hardware* compuesto de procesadores que, en lugar de estar orientados a tareas de propósito general, se centran únicamente en el procesamiento numérico. Puesto que se ejecuta un programa para cada dato, no es necesario que el control de flujo sea muy sofisticado, y debido a la gran carga de datos y de computación, la latencia de acceso a memoria se puede ocultar con cálculos [11]. Por tanto, las GPU están especialmente indicadas para problemas que se pueden expresar según el modelo *Single Instruction-Multiple Data* (SIMD), es decir, aquel donde un mismo programa se ejecuta sobre una gran cantidad de datos diferentes, y que además el *ratio* de operaciones aritméticas sobre las de acceso a memoria sea alto.

La pila del *software* CUDA se compone de varias capas, como se puede observar en la figura 2.2: un controlador del *hardware*, una API y su entorno de ejecución, y bibliotecas de alto nivel.

2.2.1. Modelo de programación

El entorno de desarrollo de CUDA permite utilizar C, FORTRAN y OpenCL [14] (estándar creado originalmente por Apple y orientado a la multiplataforma) como lenguajes de programación de alto nivel, así como la API DirectCompute [15].

El lenguaje CUDA C consiste en una serie de extensiones a C, entre las cuales destacan los *kernels*, es decir, las funciones ejecutadas en la GPU. Para ejecutar un *kernel* se emplean agrupaciones de hilos en bloques, y estos a su vez en *grids*.

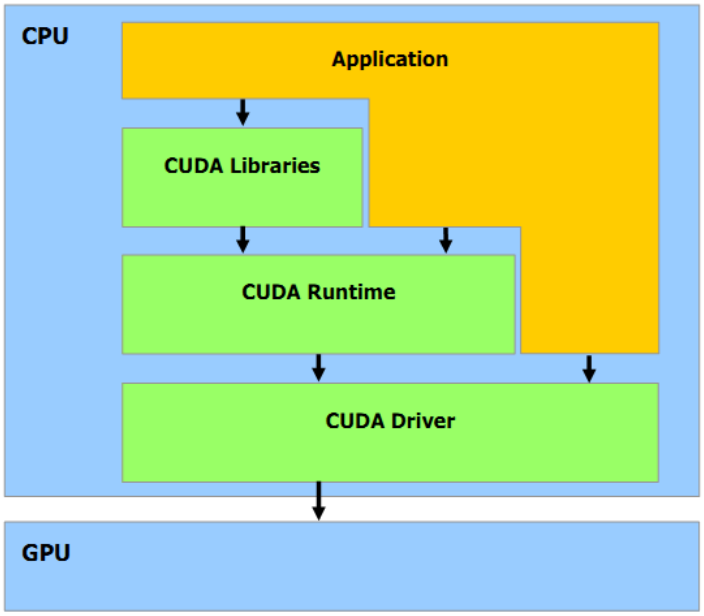

Figura 2.2: Pila del software que compone CUDA

Cada hilo que ejecuta el *kernel* recibe un identificador único `threadIdx`. Para que los hilos puedan compartir ciertos datos en memoria y sincronizar su ejecución se agrupan en bloques (*thread-blocks*) que también tienen asociado un identificador único `blockIdx`. A su vez, los bloques se organizan en un *grid* (véase la figura 2.3).

Tanto los *grids* como los bloques pueden tener hasta 3 dimensiones, para facilitar la asignación de hilos en problemas que trabajen sobre vectores, matrices o volúmenes. El número máximo de bloques del *grid* e hilos de un bloque viene impuesto por las propias especificaciones técnicas de la GPU.

Un *kernel* se especifica con el modificador `__global__`, y la asignación de bloques e hilos se realiza en su invocación con la sintaxis `<<<número de bloques, hilos por bloque >>>`.

2.2.2. Modelo de ejecución

Cuando desde la CPU se invoca a un *kernel*, los bloques que componen el *grid* se distribuyen entre los SM o *streaming multiprocessors*. Los hilos de cada bloque se ejecutan de manera concurrente, y varios bloques se pueden ejecutar también concurrentemente en un mismo multiprocesador. Cuando los bloques van terminando su trabajo, se lanzan otros nuevos en los multiprocesadores libres.

El multiprocesador crea, gestiona, planifica y ejecuta hilos en grupos de 32, denominados *warps*. Los hilos de un *warp* se inician a la vez en la misma dirección de un programa, pero tienen su propio contador de programa y estado de los registros, por lo que pueden ramificarse y ejecutarse de manera independiente.

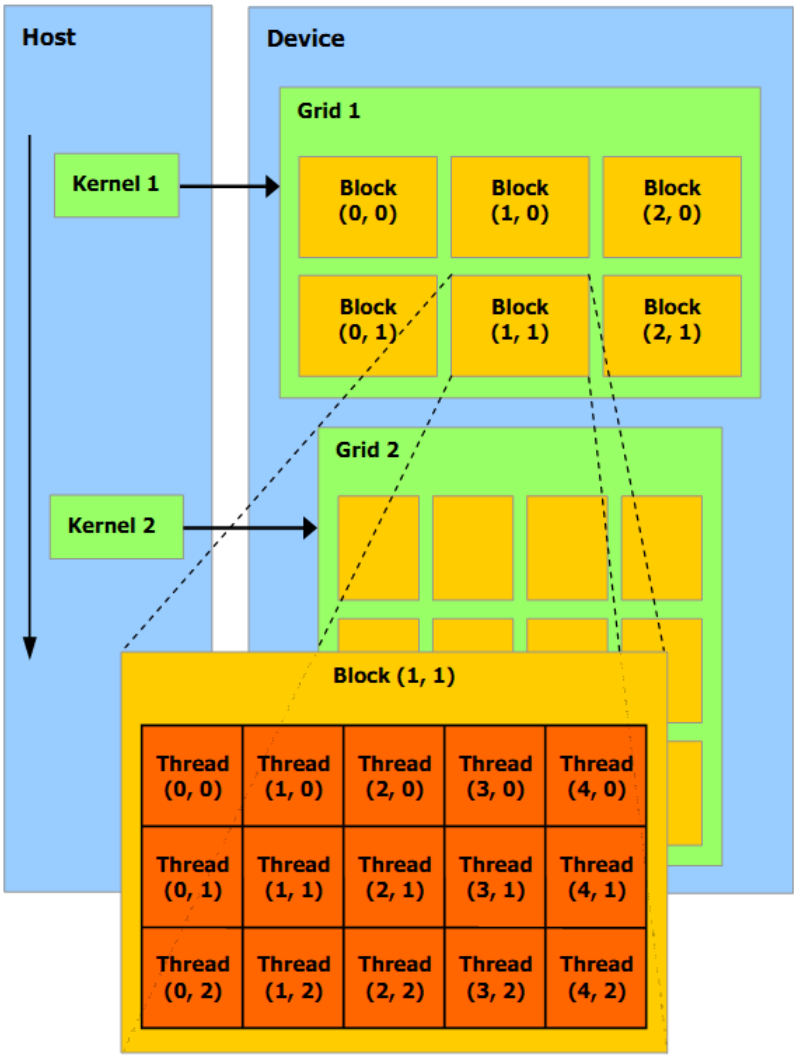

Figura 2.3: División de un *grid* en bloques y de un bloque en hilos

Todos los hilos de un *warp* ejecutan una misma instrucción a la vez pero, si se separan debido a alguna instrucción condicional, se serializará la ejecución de las distintas ramas, desactivando en cada momento los hilos que no pertenezcan a la que esté activa. Cuando todos los caminos se completen, los hilos convergerán de nuevo en el mismo flujo de ejecución. Por tanto, la máxima eficiencia se alcanza cuando los 32 hilos de un mismo *warp* siguen exactamente el mismo flujo de ejecución.

2.2.3. Modelo de memoria

Los hilos CUDA pueden acceder a varios tipos de memoria durante su ejecución:

- Cada hilo almacena las variables locales de dos maneras posibles. Los escalares

se almacenan en registros dentro del propio *chip* que permiten acceder rápidamente a los datos. Si son estructuras como *arrays* con un tamaño no-constante, se almacenan en un espacio en memoria global pero que se considera privado al hilo.

- Cada bloque dispone de una memoria compartida entre todos sus hilos. La vida de esta memoria es la misma que la del bloque, es decir, la ejecución de un *kernel*.

- Todos tienen acceso a una memoria global y a dos espacios de "solo lectura": la memoria constante y la de texturas. Estos tres espacios de memoria persisten entre diferentes ejecuciones de un *kernel* en la misma aplicación.

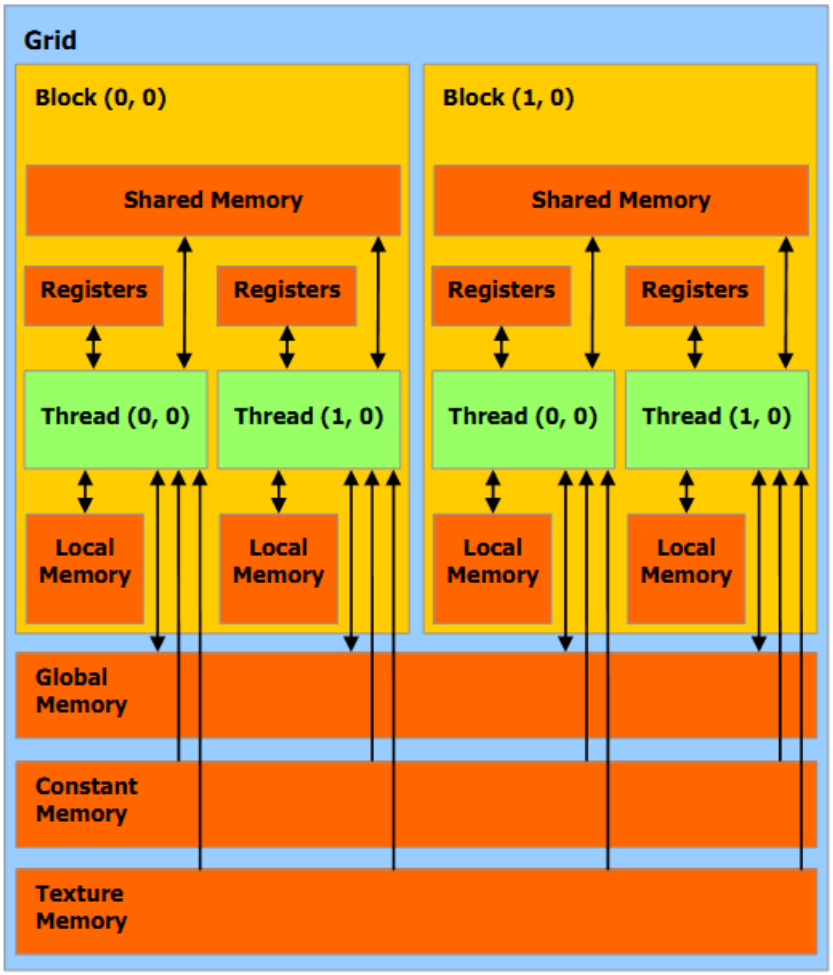

Figura 2.4: Modelo de memoria

Es importante diseñar los algoritmos para optimizar el uso de memoria teniendo en cuenta que, cuanto más rápido se puede acceder a un tipo de memoria, menor es el tamaño del que se dispone, como queda plasmado en las especificaciones técnicas en [11].

Por ejemplo, el acceso a registros se considera prácticamente gratuito, pero solo se dispone de 32 KB por multiprocesador. La siguiente memoria más rápida es la compartida entre los hilos de un bloque, con una latencia de 4 ciclos y una capacidad configurable de 16 ó 48 KB. La memoria global sin ningún tipo de cacheado está disponible en el orden de GB, pero el tiempo de latencia asociado a ella es de entre 400 y 800 ciclos.

3

Estado del arte

Aunque en la introducción se ha descrito el objetivo principal del proyecto, el campo de la computación híbrida permanece abierto, y para establecer los objetivos de manera más concreta es necesario estudiar previamente el estado del arte. Los modelos existentes se centran solo en la ejecución en CPU o GPU, sin plantear un enfoque híbrido. Se comprobará hasta dónde llegan las diferentes aproximaciones e intentará mejorarlas.

3.1. Modelos de rendimiento en CPU

Dado que las GPU no han adquirido verdadero protagonismo en la computación de propósito general hasta el surgimiento hace escasos años de tecnologías para facilitar su utilización, los primeros modelos de arquitecturas paralelas que se plantearon estaban orientados únicamente al uso de CPU.

Dichos modelos se pueden clasificar en dos grandes categorías: modelos para sistemas de memoria compartida y de memoria distribuida. Los primeros modelan una arquitectura en la que varios componentes (procesadores) se comunican a través de una misma memoria compartida y sobre la que se deben controlar los accesos concurrentes. En cambio, el segundo consiste en varias máquinas, cada una de las cuales posee una memoria local, y para comunicarse emplean el paso de mensajes.

Entre los primeros se encuentran PRAM [3] y derivados como QRQW PRAM [16] [17], y de los segundos se pueden destacar BSP [18] [19] o LogP [20], entre otros.

3.1.1. PRAM (*Parallel Random Access Machine*)

El modelo PRAM consiste en un conjunto de procesadores identificados por un índice o número único, cada uno de los cuales dispone de una memoria local. Además, pueden intercambiar información a través de una memoria compartida o global, pero nunca acceder a la memoria local de otro procesador.

El acceso concurrente a la memoria global puede producir conflictos entre los distintos procesadores, y según cómo se resuelvan nos encontraremos ante una de las siguientes variantes de PRAM:

- EREW (*Exclusive Read Exclusive Write*): No se permiten lecturas ni escrituras simultáneas sobre una misma posición de memoria.
- CREW (*Concurrent Read Exclusive Write*): Se permiten varias lecturas simultáneas sobre una misma posición de memoria, no así escrituras.
- ERCW (*Exclusive Read Concurrent Write*): Se permite la escritura concurrente sobre una posición de memoria, pero no lecturas.
- CRCW (*Concurrent Read Concurrent Write*): Se permite realizar lecturas y escrituras concurrentes sobre una misma posición de memoria. Para el caso de la escritura concurrente es necesario plantear, a su vez, varias posibilidades que definan exactamente el resultado que se obtendrá.
  - *Common*: La escritura tendrá éxito solo si todos los procesadores escriben el mismo valor.
  - *Arbitrary*: Un procesador decidido de manera arbitraria será el que escriba en la posición de memoria.
  - *Priority*: El identificador de los procesadores es el que indica cuál podrá escribir, por ejemplo, el que tenga el menor identificador.
  - *Sum*: Se escribe la suma de todos los datos.

3.1.2. QRQW PRAM (*Queue Read Queue Write PRAM*)

Este modelo surge como una evolución del PRAM que intenta reflejar las reglas de contención en accesos concurrentes de manera más realista y acorde a las arquitecturas comerciales que el estándar PRAM: las de acceso exclusivo son demasiado estrictas y las concurrentes no tienen en cuenta las penalizaciones al rendimiento derivadas de la contención. Se propone una regla de cola (*queue*), en la que todos los procesadores pueden leer y escribir sobre una posición de memoria, pero los accesos concurrentes se encolan y sirven de uno en uno.

Así mismo, este modelo expone algunas variantes. En el modelo QRQW PRAM original, cada procesador ejecuta una serie de lecturas de memoria compartida, varios pasos de computación, escrituras a la memoria compartida y finalmente se sincroniza con el resto de procesadores; por tanto, deberá esperar hasta que se vacíen las colas de todos ellos.

Sin embargo, en el modelo QRQW asíncrono cada procesador ejecuta las lecturas, computación y escrituras, pero puede continuar sin realizar las sincronizaciones en cuanto sus colas estén vacías. La mayoría de modelos de memoria compartida asíncronos asumen que un procesador solo puede tener una petición de acceso a memoria pendiente de manera simultánea, pero en el QRQW asíncrono se permite el *pipelining* de peticiones; un procesador puede ejecutar cualquier petición sobre memoria compartida y continuar sin esperar a que se completen, pero la primera operación que haga uso del valor leído en dicha petición provocará una espera.

3.1.3. BSP (*Bulk-Synchronous Parallel Model*)

El modelo BSP se define como una combinación de tres atributos:

1. Componentes que pueden llevar a cabo tareas de computación o acceso a memoria.

2. Un *router* que envía mensajes punto a punto entre componentes.

3. Facilidades para sincronizar los componentes a intervalos regular de L unidades de tiempo, donde L es el parámetro de periodicidad.

Un proceso de computación consiste en un conjunto de "superpasos". En cada superpaso, a cada componente se le asigna una tarea compuesta por cálculos locales y transmisión de mensajes. Tras cada período de L unidades de tiempo, se comprueba si todos los componentes han completado el superpaso. Si es así se procede con el siguiente y, si no, se asigna otro período de L unidades al mismo.

El mecanismo de sincronización se puede desactivar, de forma que un procesador pueda avanzar en procesos independientes de otros componentes sin esperar a que se completen el resto de procesos. Como método alternativo de sincronización se puede emplear el paso de mensajes.

En cuanto al acceso concurrente a memoria, no establece ningún modelo concreto, sino que se puede basar en otros ya existentes como los arriba citados.

3.1.4. LogP

LogP surge con la intención de introducir factores que no se tienen en cuenta en otros modelos, como el ancho de banda limitado o la latencia en las comunicaciones. Para ello emplea cuatro parámetros:

- Estimación superior de la latencia (L) en la comunicación de un mensaje.

- Sobrecarga (*overhead,* O), o tiempo de procesador consumido en el envío o recepción de cada mensaje.

- El intervalo (*gap,* g) mínimo de tiempo entre dos envíos o recepciones de mensajes en un procesador.

- El número de procesadores o módulos de memoria (P).

Los parámetros L, o y g se suelen medir en múltiplos del tiempo de ciclo en un procesador. Se asume que la red tiene una capacidad finita, por lo que el número máximo de mensajes en tránsito de un procesador a otro es $\lceil L/g \rceil$.

3.2. Modelos de rendimiento en GPU

Con la expansión de las GPU en la computación de propósito general han surgido también modelos orientados únicamente a esta arquitectura, entre los que se encuentran los propuestos por Hong y Kim [5], Baghsorkhi et al. [6], Mukherjee et al. [1] y Nugteren [7].

Algunos de ellos no giran en torno al concepto de hilo sino de *warp* que, como ya se ha comentado en 2.2, es la unidad mínima de agrupación de hilos, con 32 de ellos que, si no se da ninguna divergencia por la que la ejecución se divida en ramas, se ejecutan totalmente en paralelo. También se debe recordar que las latencias de los accesos a memoria se ocultan solapándolos con computación, y es un punto a tener presente para calcular las cotas mínimas y máximas.

3.2.1. Modelo de Hong y Kim

Estos autores proponen un modelo analítico que se puede utilizar de manera estática (sin ejecutar una aplicación) para evaluar la implementación de un *kernel* diseñado para GPU. Para ello emplea fórmulas con parámetros obtenidos de diversas formas:

- Con micro-benchmarks, para calcular valores como los ciclos de acceso coalescente o no a memoria, el retardo entre dos *warps* o el coste de las rutinas de sincronización.
- Mediante estudio del código, CUDA o incluso PTX.
- Derivados de las propias fórmulas.

Basándose en el hecho de que la latencia de las peticiones de memoria se intenta ocultar ejecutando cálculos u otras peticiones simultáneamente, el modelo define dos parámetros básicos:

- MWP (*Memory Warp Parallelism*): El número de *warps* de un SM que pueden acceder a memoria de manera simultánea desde que se inicia la ejecución de otro, denominado *memory warp*. Si el valor de MWP es mayor que 1, se podrá ocultar el coste de los accesos a memoria de $(MWP - 1)$ *warps*.
- CWP (*Computation Warp Parallelism*): El nivel de computación que otros *warps* pueden ejecutar mientras uno espera por algún valor de memoria.

Cuando $CWP > MWP$ significa que los accesos a memoria dominan el coste de ejecución, y si $MWP > CWP$, el coste será mayormente de computación.

Coste de computación

El coste de computación se calcula sumando el coste en ciclos de cada instrucción, en base a los detalles disponibles en [11]. El coste de computación se almacena en una variable $Comp\_cycles$.

Coste de acceso a memoria

Para hacer este cálculo se debe considerar por separado las latencias de los accesos coalescentes, iguales a la latencia de la memoria $Mem\_L\_Coal = Mem\_LD$, y las de los accesos no coalescentes, que suman el coste de cada uno de los accesos por separado, es decir, $Mem\_L\_Uncoal = Mem\_LD + (Num\_Uncoal\_per\_mw - 1) \times Departure\_del\_uncoal$.

El coste total se calcula como la media ponderada de las variables anteriores: $Mem\_L = Mem\_L\_Coal \times Weight\_coal + Mem\_L\_Uncoal \times Weight\_uncoal$.

Coste global

A partir de varias fórmulas más finalmente se podrán derivar los valores de CWP y MWP y, los ciclos totales de ejecución. Estos últimos dependen de la relación entre los anteriores:

- Si $MWP = CWP = N$ warps por SM:

$$\begin{aligned} Exec\_cycles\_app &= (Mem\_cycles + Comp\_cycles + \frac{Comp\_cycles}{Num\_Mem\_insts} \\ &\quad \times (MWP - 1)) \times Num\_Rep \end{aligned}$$

- Si $CWP >= MWP$ o $Comp\_cycles > Mem\_cycles$:

$$\begin{aligned} Exec\_cycles\_app &= (Mem\_cycles \times \frac{N}{MWP} + \frac{Comp\_cycles}{Num\_Mem\_insts} \\ &\quad \times (MWP - 1)) \times Num\_Rep \end{aligned}$$

- Si $MWP > CWP$:

$$Exec\_cycles\_app = (Mem\_L + Comp\_cycles \times N) \times Num\_Rep$$

Finalmente, si se quieren tener en cuenta las sincronizaciones dentro de un *kernel*, se puede calcular su coste mediante la fórmula siguiente y sumarlo a $Exec\_cycles\_app$:

$$\begin{aligned} Synch\_cost &= Departure\_delay \times (MWP - 1) \times Num\_synch\_insts \\ &\quad \times Num\_Active\_blocks\_per\_SM \times Num\_Rep \end{aligned}$$

Como limitaciones del modelo, los propios autores reconocen la carencia de un modelado de la memoria caché (que ha cobrado importancia en la generación de tarjetas Fermi) y de las divergencias entre ramas, que da lugar a la ejecución secuencial de los *kernel*.

3.2.2. Modelo de Baghsorkhi et al.

Este modelo introduce el concepto de "grafo de flujo de trabajo" (WFG, *work flow graph*) como una interpretación abstracta de un *kernel*. Los nodos de este tipo de grafos son operaciones de memoria o computación, y los arcos que los conectan representan el coste de transición de un nodo a otro, además de las relaciones entre una lectura de memoria y el uso del valor leído.

Parámetros básicos

Para construir este grafo se debe tener en cuenta el paralelismo a nivel de *warp*, datos e instrucciones.

El primero representa los *warps* que se pueden asignar a un SM simultáneamente sin exceder el límite de recursos disponibles. El máximo se denomina WLP_{max}, la cantidad de cierto fragmento de código es WLP_{local} y el efectivo es WLP_{effect}, que se calcula mediante la siguiente fórmula:

$$WLP_{effect} = \frac{WLP_{local} + (NUM_{blocks} - 1) \times WLP_{avg}}{NUM_{blocks}}$$

En un *kernel* sin control de divergencias, WLP_{effect}, WLP_{local} y WLP_{avg} son iguales a WLP_{max}, porque se dedicarán todos los recursos a un único camino del flujo de ejecución.

Para intentar reducir la latencia de acceso a memoria se aprovecha el paralelismo a nivel de datos o DLP, que consiste en operar simultáneamente sobre múltiples bancos de memoria a través de una instrucción vectorial, o emplear instrucciones SIMD sobre varios datos obtenidos en un solo acceso. Los conflictos y los accesos no-coalescentes disminuyen el DLP. Para estudiar los patrones de acceso e identificar los *warps* que siguen cada camino en la ejecución se emplea evaluación simbólica.

A nivel de los hilos, el paralelismo a nivel de instrucción contempla los bloqueos dentro de un bloque, derivados de las cargas desde memoria global y de las dependencias entre registros. ILP_{local} se aproxima como el número de instrucciones de dicha región entre la longitud de la cadena *def-use* más larga que se puede construir con ellas. El ILP_{effect} se puede calcular con la misma fórmula que el WLP_{effect}:

$$ILP_{effect} = \frac{ILP_{local} + (NUM_{blocks} - 1) \times ILP_{avg}}{NUM_{blocks}}$$

Para obtener los valores medios y locales se realiza un análisis del código fuente que genera un grafo de dependencias compuesto por nodos con las sentencias, predicados de control o los pesos de una rama cuando haya divergencias de flujo. Este

grafo se recorre de manera descendente realizando, mediante un motor de simplificación de expresiones, una evaluación simbólica de las condiciones y expresiones de acceso a memoria. Una vez se sabe qué hilos están activos en un momento dado se puede saber:

- El peso de cada rama
- La penalización media en el acceso a bancos de memoria
- El número medio de bytes transferido para las operaciones de memoria global

Recorriendo este grafo también se pueden computar las variables WLP_{local} e ILP_{local} de cada rama y calcular la media en base a los pesos y número de sentencias que las componen.

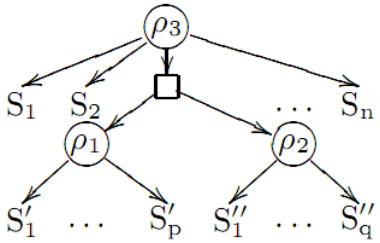

Figura 3.1: Grafo de dependencias para la obtención de parámetros en el modelo de Baghsorkhi

Por ejemplo, la figura 3.1 se correspondería con las fórmulas:

$$WLP_{avg} = \frac{p\rho_3\rho_1 WLP_{max} + q\rho_3\rho_2 WLP_{max} + n\rho_3 WLP_{max}}{p+q+n}$$

$$ILP_{avg} = \frac{p\rho_1 ILP_p + q\rho_2 ILPq + nILPn}{p\rho_1 + q\rho_2 + n}$$

Para combinar estos aspectos con el estudio de las latencias de memoria y del *pipeline* se emplean los grafos de control de flujo o WFG.

Construcción del WFG

Un WFG es una extensión a los grafos de control de flujo. Sus nodos pueden ser operaciones sobre memoria global o local, barreras de sincronización, un bloque de instrucciones de computación o entradas/salidas sintéticas. Además de los arcos de transición entre nodos, el WFG puede contener otros que conecten los puntos donde se realiza una lectura de memoria con aquellos donde se utilice el valor leído. En caso de divergencia entre ramas, se dividirá el grafo hasta representar de manera

completa e independiente los diferentes caminos, asignándole a cada uno el peso correspondiente a la fracción de *warps* que lo ejecuten.

En el WFG inicial no se asigna ningún peso a los arcos de transición, sino que estos simplemente indican que el nodo de destino se ejecuta después que el nodo origen. Para estimar el número de ciclos necesarios en una GPU determinada, se asignan a los arcos del WFG valores calculados en función de los parámetros del hardware, de las latencias de computación y de acceso a memoria. El primer tipo de latencia se calcula como:

$$Latency_{comp} = max(1, \frac{Latency_{pipeline}}{\frac{Warp_{size}}{SM_{num\_SP}} \times ILP_{effect} \times WLP_{effect}})$$

En la granuralidad de un *warp*, la latencia de ejecución de una instrucción de computación es $Latency_{comp} \times \frac{Warp_{size}}{SM_{num\_SP}}$, y el peso de un arco que sale desde un nodo de computación con n instrucciones:

$$C(n) = n \times Latency_{comp} \times \frac{Warp_{size}}{SM_{num\_SP}}$$

Una vez realizados los cálculos, se puede obtener el número de ciclos de computación asociados a un *warp* reduciendo el grafo hasta convertirlo en un único arco de transición entre un nodo de entrada y otro de salida. Este valor $CYC_{compute}$ no tiene en cuenta los arcos de dependencias de datos del WFG.

Los bloqueos de memoria se clasifican en aquellos por limitación del ancho de banda y los debidos únicamente a la latencia. Durante la reducción para obtener $CYC_{compute}$ se pueden obtener también el número global de operaciones de memoria NUM_{mem}, la cantidad de datos a transferir desde o hacia memoria global NUM_{bytes} y el número de barreras de sincronización NUM_{sync} en un *warp*. Así, la latencia de las operaciones de memoria se calcula como la diferencia entre CYC_{mem} y $CYC_{compute}$. Así, la latencia por falta de ancho de banda sería:

$$Latency_{BW} = max(0, \frac{CYC_{mem} - CYC_{compute}}{NUM_{mem}}) + \frac{Warp_{size}}{SM_{num\_SP}}$$

Este valor se establece como peso de las transiciones de arcos salientes desde operaciones de memoria global.

Para el segundo tipo de bloques se debe obtener el número medio de ciclos no-bloqueantes de un *warp* y a partir de él, la latencia que no se oculta al solapar la ejecución de varios *warps*:

$$NBC_{avg} = \frac{CYC_{compute}}{NUM_{mem} + NUM_{sync} + 1}$$

$$Latency_{exposed} = Latency_{mem} - (WLP_{effect} - 1) \times NBC_{avg}$$

Al reducir el WFG teniendo en cuenta las latencias de memoria, el peso del arco de transición del nodo de entrada al de salida ya refleja la latencia total del *warp* medio.

3.2.3. Modelo de Mukherjee et al.

Tomando como base otros modelos arriba citados (BSP, PRAM y QRQW PRAM) y relacionando la ejecución completa de un *kernel* con el concepto de superpaso de BSP, este modelo pretende representar la ejecución de un programa completo en lugar de modelar un solo *kernel*.

La estimación de coste para un *kernel* se corresponde con el tiempo máximo consumido por uno de sus hilos. El coste de un programa completo es la suma de los costes de todos los *kernel*.

Coste de computación

El coste de computación se calcula sumando el coste en ciclos de cada instrucción, en base a los detalles disponibles en [11]. El coste de computación se almacena en una variable N_{comp}.

Coste de acceso a memoria

Para cuantificar el coste de los accesos a memoria global, aunque en la fecha de publicación de la investigación en [11] se indicaba que varían entre 400 y 600 ciclos (actualmente entre 400 y 800), por simplicidad se tratan con una media de 500 ciclos. Así, si una transacción es coalescente para t hilos, el coste medio de acceso será de $\frac{500+t}{t}$.

En cuanto a la memoria compartida, cada acceso tiene un coste de 4 ciclos, pero si más de un hilo intenta acceder al mismo banco simultáneamente se produce una contención. Las contenciones se modelan de manera similar al PRAM QRQW asíncrono, por lo que si t hilos intentan acceder a una posición, el coste total es de $4t$ ciclos.

La suma de costes de acceso a memoria global y compartida se almacena en una variable N_{memory}.

Efectos de la planificación

El objetivo de la planificación es ocultar los tiempos de acceso a memoria con computación. Según el nivel en el que cumpla este objetivo, se pueden definir dos cotas para el número de ciclos necesarios para ejecutar un hilo $C(T)$:

- Máxima (modelo SUM): $C(T) = N_{comp} + N_{memory}$.
- Mínima (modelo MAX): $C(T) = max(N_{comp}, N_{memory})$.

Coste global

El objetivo es calcular el coste de un *kernel*, sabiendo que se asignan N_B bloques a los SM, cada uno con N_w *warps*, y estos a su vez con N_t hilos. El número de ciclos necesarios para ejecutar un *kernel* es el máximo requerido por alguno de sus hilos.

También se debe considerar que cada uno de los N_C *cores* o SP de un SM de la GPU posee un *pipeline* de longitud D que permite ejecutar D hilos en paralelo.

Por tanto, para obtener el número de ciclos de ejecución de un *kernel* se debe emplear la siguiente fórmula:

$$C(K) = N_B(K) \cdot N_w(K) \cdot N_t(K) \cdot C_T(K) \cdot \frac{1}{N_C \cdot D}$$

Si se desea obtener el tiempo en segundos, se divide el resultado anterior por la frecuencia del reloj de la GPU:

$$T(K) = \frac{C(K)}{R} \text{ s}$$

Como limitaciones reconocidas por los propios autores se encuentran el hecho de no tener en cuenta las sincronizaciones dentro de un mismo bloque mediante `__syncthreads()`, las operaciones atómicas ni los accesos a memoria de texturas y *shaders*.

3.2.4. Modelo de Nugteren

Este modelo se estructura en varias capas o niveles. El *core model* (cM) es el modelo básico de la GPU, y con el *roofline model* (rM) se le añaden ciertas restricciones. El *extensions model* (eM) presenta extensiones para casos especiales, y el *memory model* (mM) tiene en cuenta los diferentes tipos de acceso a memoria, la coalescencia, etc. Finalmente, el *null model* (nM) cubre todas las variables utilizadas y que no se toman directamente del código fuente, ni de las especificaciones de la GPU ni de valores constantes.

Core model

El primer paso es calcular el número de ciclos de GPU para ejecutar una instrucción, sea de computación o de memoria. Los ciclos de computación se obtienen multiplicando el número de instrucciones por los ciclos necesarios para una instrucción. Para obtener los ciclos de memoria se multiplica el número de accesos por la latencia total de un *warp*, es decir, la latencia de memoria más la de retardo para ejecutar dos peticiones consecutivas.

$$comp\_cycles = src.computation\_instr \times \frac{gpu.warp\_size}{gpu.processors}$$

$$mem\_cycles = memory\_instr \times (gpu.mem\_latency + mem\_rate)$$

Para calcular el coste, primero se calculan los ciclos consumidos por un SM para todos los *warps* asignados a él. Los factores por los que se multiplican los ciclos de memoria y computación se definen en el *roofline model*.

$$cycles\_per\_sm = mem\_factor \times mem\_cycles + comp\_factor \times comp\_cycles$$

También es necesario calcular cuántas ejecuciones necesita el SM:

$$sm\_executions = \left\lceil \frac{src.blocks}{gpu.multiprocessors \times blocks\_per\_sm} \right\rceil$$

Multiplicando el número de ejecuciones de un SM por los ciclos que necesita se obtiene el total de ciclos, que a su vez se puede dividir por la frecuencia del reloj para obtener el tiempo:

$$cycles = cycles\_per\_sm \times sm\_executions$$

$$time = \frac{cycles}{gpu.clock}$$

Roofline model

Puesto que este modelo impone restricciones al anterior, el primer paso es considerar las limitaciones en el ancho de banda. Para ello, primero se calculan los bytes leídos por un *warp* y a continuación el ancho de banda consumido por un SM:

$$bytes\_per\_warp = src.data\_size \times gpu.warp\_size \times min(1, src.memory\_instr)$$

$$bw\_per\_sm = \frac{bytes\_per\_warp \times gpu.clock}{gpu.mem\_latency + mem\_rate}$$

El límite de *warps* que se pueden utilizar hasta saturar el ancho de banda es el conciente entre el ancho de banda disponible en la GPU y el consumido en cada SM:

$$bw\_limit = \frac{gpu.bandwidth}{bw\_per\_sm \times gpu.multiprocessors}$$

Otro límite es la cantidad de peticiones de memoria que se pueden planificar simultáneamente, que viene dado en función del retardo introducido desde que se realiza una petición de memoria hasta que se puede realizar otra y de la latencia del acceso a memoria:

$$fire\_rate\_limit = \frac{gpu.mem\_latency + mem\_rate}{mem\_rate}$$

Uniendo estos dos límites al número máximo de *warps* por SM ya se dispone de las tres restricciones sobre las que aplicar el rM. La más restrictiva de ellas será la que indique el máximo de accesos simultáneos a memoria que se pueden llevar a cabo y se denominará sim_mem_limit.

Para calcular los factores mem_factor y $comp\_factor$ definidos en el cM se pueden dar dos situaciones:

- El programa está limitado por memoria:

$$mem\_factor = 1$$

$$comp\_factor = warps\_per\_sm$$

- El programa está limitado por computación:

$$mem\_factor = \frac{warps\_per\_SM}{sim\_mem\_limit}$$

$$comp\_factor = warps\_per\_sm - sim\_mem\_limit$$

Extensions model

Al coste antes calculado se debe añadir la sobrecarga debida a la invocación del *kernel*. La del lanzamiento en sí mismo es constante, pero se debe añadir un coste extra por cada bloque de hilos y si recibe más de 5 argumentos.

$$\begin{aligned} overhead\_per\_kernel \;=\; & gpu.overhead\_base + gpu.overhead\_block \times src.blocks \\ & + gpu.overhead\_arg \times max(0, src.arguments - 5) \end{aligned}$$

$$cycles = cycles + overhead\_per\_kernel$$

Otra extensión del modelo se debe a la latencia del *pipeline*, que impedirá obtener la máxima tasa de transferencia si no se trabaja con un número suficiente de hilos.

$$comp\_cycles = src.computation\_instr \times max(\frac{gpu.warp\_size}{gpu.processors}, \frac{src.pipeline\_latency}{warps\_per\_sm})$$

Memory model

Las instrucciones de lectura y escritura se consideran iguales, por lo que la el número de instrucciones de memoria es la suma de ambas.

En cuanto al retardo entre peticiones de memoria, depende de si estas son coalescentes o no y el valor de la variable *mem_rate* se establecerá en función de este criterio.

Null model

Por último, para saber el número de *warps* por SM se deben tener en cuenta varios factores limitantes:

- Bloques designados para ejecutar el *kernel*
- El máximo de bloques permitidos por el hardware
- El total de hilos permitidos por el hardware
- El uso de memoria compartida
- El uso de registros

Una vez tomado el más restrictivo de ellos, se pueden calcular los *warps* por bloque y por SM:

$$warps\_per\_block = \frac{src.threads\_per\_block}{gpu.warp\_size}$$

$$warps\_per\_sm = blocks\_per\_sm \times warps\_per\_block$$

4

Alcance del proyecto

Como se ha podido observar en el capítulo 3, los modelos planteados hasta ahora se centran únicamente en la CPU o en la GPU. Además, algunos de ellos se aplican a tan bajo nivel que antes de utilizarlos es necesario haber implementado el algoritmo. En contraposición, el modelo desarrollado en este trabajo debe intentar ajustarse a los siguientes requisitos:

Modelo teórico: Los modelos actuales evalúan algoritmos ya implementados sobre la GPU en lugar de trabajar a alto nivel. Dada la complejidad del entorno de programación es interesante disponer de un enfoque teórico que permita evaluar el coste de los algoritmos sin preocuparse por detalles específicos del lenguaje en que se realiza la implementación.

Modelo heterogéneo: La variación de rendimiento de un algoritmo para GPU por detalles como el acceso (no-)coalescente a memoria impide aplicar de manera única los modelos tradicionales en CPU, que se podrían considerar más genéricos. Los modelos de GPU tampoco podrían aplicarse directamente sobre una CPU por las diferencias a nivel de arquitectura, por detalles como pueden ser la organización en *grids*-bloques-*warps*-hilos o los diferentes tipos de memoria.

Por tanto, es necesario encontrar algún tipo de métrica que permita relacionar el rendimiento de una y otra arquitectura y evaluarlas conjuntamente. De esta forma también se podrían realizar comparativas entre algoritmos implementados siguiendo distintos enfoques, por ejemplo, uno híbrido contra otro que solo haga uso de GPU.

Mejoras en modelos de GPU: Además de extenderlos a aplicaciones completas, los modelos orientados exclusivamente a GPU son mejorables, como reconocen sus propios autores. El modelado de la sincronización, la memoria caché o la divergencia entre ramas son puntos que aún no están perfectamente trabajados y que se pueden tratar de manera más completa si se toman ideas de cada uno de los modelos estudiados.

Validación del modelo: Es interesante plantear algoritmos de diferente complejidad que permitan validar el modelo, realizando un estudio previo de los costes y

comprobando que se corresponden con medidas tomadas de manera experimental. Según el algoritmo, se puede tomar como base una implementación ya existente buscando posibles optimizaciones, adaptar una solución para matrices densas a dispersas o realizar una implementación completa.

Por tanto, el objetivo principal del proyecto es **crear un modelo teórico para arquitecturas híbridas CPU+GPU**, es decir, que permita evaluar los algoritmos sin desarrollar ninguna implementación.

En cuanto a los objetivos específicos necesarios para llevar a cabo esta tarea, se pueden resumir en los siguientes puntos:

- **Evaluar los modelos descritos en el estado del arte** (capítulo 3) para decidir si tomar alguno de ellos como base o partir de cero.
- **Extender la base del modelo** para que permita evaluar el coste de algoritmos que combinen el uso de CPU y GPU.
- **Desarrollar una serie de algoritmos** (solo en GPU o mediante un esquema híbrido) de distinta complejidad: reducción, producto matriz-vector y descomposición de Cholesky.
- **Optimizar el algoritmo de descomposición de Cholesky** para matrices densas estructuradas, empleando bibliotecas algebraicas y técnicas como la descomposición por bloques. Este algoritmo es **aplicable al problema de mallas de elementos finitos** descrito en [2].
- **Estimar el coste** de estos algoritmos mediante el modelo desarrollado **y comparar los resultados experimentales** con dichas estimaciones.

5

Modelo de rendimiento

Tal como se indicaba en el capítulo dedicado al alcance del proyecto (4), el objetivo principal es crear un modelo de rendimiento para arquitecturas heterogéneas CPU+GPU. En el estudio del estado del arte (capítulo 3) se ha observado la existencia de varios modelos ya existentes orientados al estudio de las aplicaciones para GPU.

En el primer punto de este capítulo se realizará un estudio teórico y práctico de los modelos, comparando sus resultados con otros experimentales. Tras concluir si resulta interesante tomar como base alguno de ellos o crear uno desde cero, en un segundo punto se planteará el nuevo modelo para arquitecturas heterogéneas.

5.1. Estudio de los modelos existentes

En esta sección se realizará un estudio teórico y práctico de los modelos estudiados en el capítulo 3. En la parte práctica se hará uso de dos núcleos computacionales básicos: la suma de vectores y el producto matriz-vector. Dichos núcleos, al tener un paralelismo claro y diferente complejidad computacional, permiten descartar aquellos modelos que no sean capaces de ofrecer un buen resultado en casos sencillos.

5.1.1. Consideraciones teóricas

A nivel teórico se pueden tener en cuenta varios puntos relativos a la exactitud y a la complejidad de cada uno de los modelos. Algunos son muy simples pero no obtienen resultados precisos, mientras que otros se ajustan bastante a los resultados experimentales pero resulta difícil aplicarlos a algoritmos complejos, así como justificar el porqué de los resultados.

Conteo de instrucciones y formulación

Como se ha visto en el capítulo 3.1, la base para calcular el coste computacional de un algoritmo es analizar el de todas sus instrucciones. De entre todos los modelos, el que en principio plantea mayores complicaciones es el de Hong y Kim, porque en su artículo plantean la necesidad de realizar esta cuenta sobre las instrucciones PTX,

es decir, a nivel de *bytecode*. Sin embargo, puesto que el modelo a desarrollar en este trabajo tendrá una base teórica, el conteo de instrucciones se realizará sobre código CUDA, de más alto nivel y con posibilidad de traducción a pseudocódigo.

El coste de los accesos a memoria se calcula de manera similar, teniendo en cuenta cuándo son coalescentes o no, aplicando los tiempos de latencia en cada nueva petición, etc. El modelo de Baghsorkhi es el más sofisticado a este respecto, que parte de un árbol con el que expresar los patrones de acceso para comprobar si estos son coalescentes o no. Sin embargo, la herramienta no está accesible y no se pueden realizar estudios prácticos del modelo.

Una vez obtenidos los tiempos de computación y acceso a memoria, cada modelo emplea una serie de fórmulas para obtener el tiempo final, teniendo en cuenta el nivel de paralelismo.

El modelo de Mukherjee es el que aplica la fórmula más sencilla, tomando simplemente el tiempo de un hilo y multiplicándolo por el número de hilos, *warps* y bloques, y dividiéndolo por la cantidad de ellos que se pueden ejecutar en paralelo.

El modelo de Nugteren añade consideraciones más complejas para calcular el máximo de *warps* que se pueden ejecutar simultáneamente en un multiprocesador.

El trabajo de Hong y Kim desarrolla especialmente este último aspecto teniendo muy en cuenta el ancho de banda disponible, aunque ello implica una mayor complejidad en la formulación.

Limitaciones

Los modelos presentan varios puntos mejorables y que los propios autores reconocen. Parte de las limitaciones ya quedan plasmadas en el modelado únicamente de la parte ejecutada en GPU, mientras que otras incluirían también la comunicación con la parte de CPU en un sistema heterogéneo.

Exceptuando el modelo de Baghsorkhi, ninguno de ellos modela de manera explícita la divergencia entre ramas, que supone una gran penalización en el rendimiento y un factor a tener en cuenta. Se puede generar un pseudo-fragmento de código al dividir los cálculos entre cada una de las ramas, y después fusionar los resultados, pero no está contemplado directamente sino que supondría un añadido.

Las operaciones atómicas, que al igual que las divergencias implican una serialización, no están soportadas en ninguno de ellos. Se debería generar un nuevo fragmento de código y seguir el mismo procedimiento que en la divergencia entre ramas.

Otra característica de las últimas versiones de CUDA es el uso de *streams*, que permiten utilizar ejecuciones fuera-de-orden cuando hay varios flujos de trabajo independientes unos de otros. Esta característica tampoco está contemplada en ninguno de los modelos, en algunos casos por la fecha en que fueron publicados, y en general por su complejidad a la hora de modelarlos.

Dado que el modelo de Mukherjee está basado en superpasos y estos se establecen al finalizar cada *kernel*, no contempla la sincronización entre hilos mediante `__syncthreads()`.

Como última limitación en el modelado únicamente de la GPU, ninguno incluye el modelado la caché entre sus posibilidades, debido a la complejidad de su funcionamiento.

A la hora de modelar la comunicación con la CPU de cara a aplicarlo a un programa completo, Nugteren incluye fórmulas asociadas a la sobrecarga de invocación de un *kernel*. Esta es la única consideración que se hace al respecto, puesto que ninguno de los modelos incluye referencias a la copia de datos mediante funciones como `cudaMemcpy`.

5.1.2. Suma de vectores

Este es uno de los algoritmos más básicos que se pueden implementar en GPU y en el que se puede observar de manera muy clara el paralelismo, pues cada hilo realiza la suma de un elemento de cada vector. Su complejidad es lineal y no hace uso de memoria compartida, únicamente de la global.

Listing 5.1: Suma de vectores en CUDA

```
__global__ void addKernel(double *a, double *b,
                          double *c, int n)
{
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  if (i < n) {
    c[i] = a[i] + b[i];
  }
}
```

En este algoritmo, cada hilo ejecuta 3 operaciones de acceso a memoria global y una de computación. A pesar de la condición $i < n$, no hay divergencias apreciables en cuestión de rendimiento porque los hilos que no se encuentran dentro del rango no ejecutan ninguna instrucción.

En la figura 5.1 se pueden apreciar los resultados obtenidos con cada uno de los modelos asignando 128 hilos a cada bloque. Para consultar los resultados numéricos de manera más precisa, véase el apéndice B.1. Debe tenerse en cuenta que, al no disponer de las herramientas necesarias para calcular los parámetros WLP e ILP, no ha sido posible obtener resultados para el modelo de Baghsorkhi.

En la gráfica se observa cómo el modelo de Mukherjee (en el submodelo SUM, puesto que los cálculos no pueden solapar los accesos a memoria) es el que más rápido se aleja de los resultados experimentales. Realmente ninguno de los modelos

ajusta demasiado bien el problema, aunque es digna de mención la gran mejoría que supone, como se muestra en la figura 5.2, incluir un modelo simplificado de la caché en el modelo de Mukherjee. Al hacerlo consigue mejorar los resultados de Nugteren, que era el más cercano a los resultados experimentales.

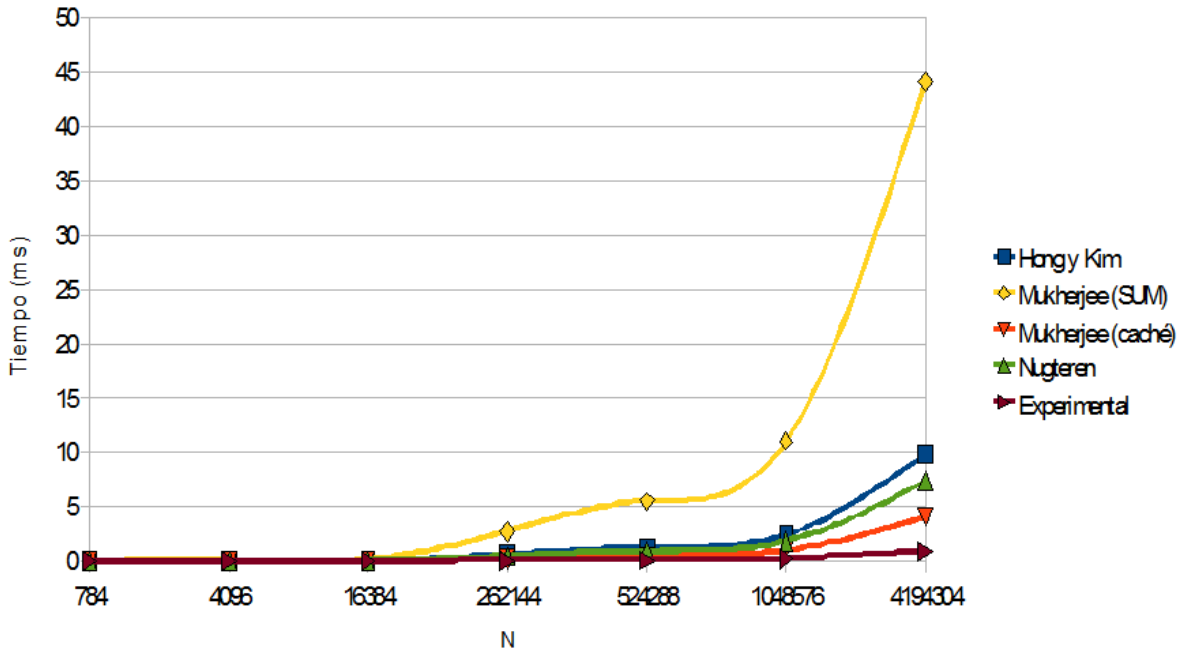

Figura 5.1: Estudio de modelos: suma de vectores (1)

5.1.3. Producto matriz-vector

Este algoritmo conforma uno de los núcleos computacionales básicos. Dada una matriz $A \in \mathbb{R}^{n \times m}$, un vector $x \in \mathbb{R}^m$, el vector resultante $y \in \mathbb{R}^n$ es el producto escalar de las filas de A por los elementos de x.

Puesto que el objetivo principal de este capítulo es comparar la calidad de los modelos existentes, para más información sobre el algoritmo véase el capítulo 7.

La paralelización de este algoritmo se puede afrontar desde varios enfoques, por lo que ofrece bastantes posibilidades para evaluar la corrección de un modelo. Las variantes que se han considerado para este estudio son las siguientes:

- Asignación de un hilo a cada fila de la matriz
- Asignación de un hilo a cada columna de la matriz
- Asignación de un hilo a un bloque de filas de la matriz

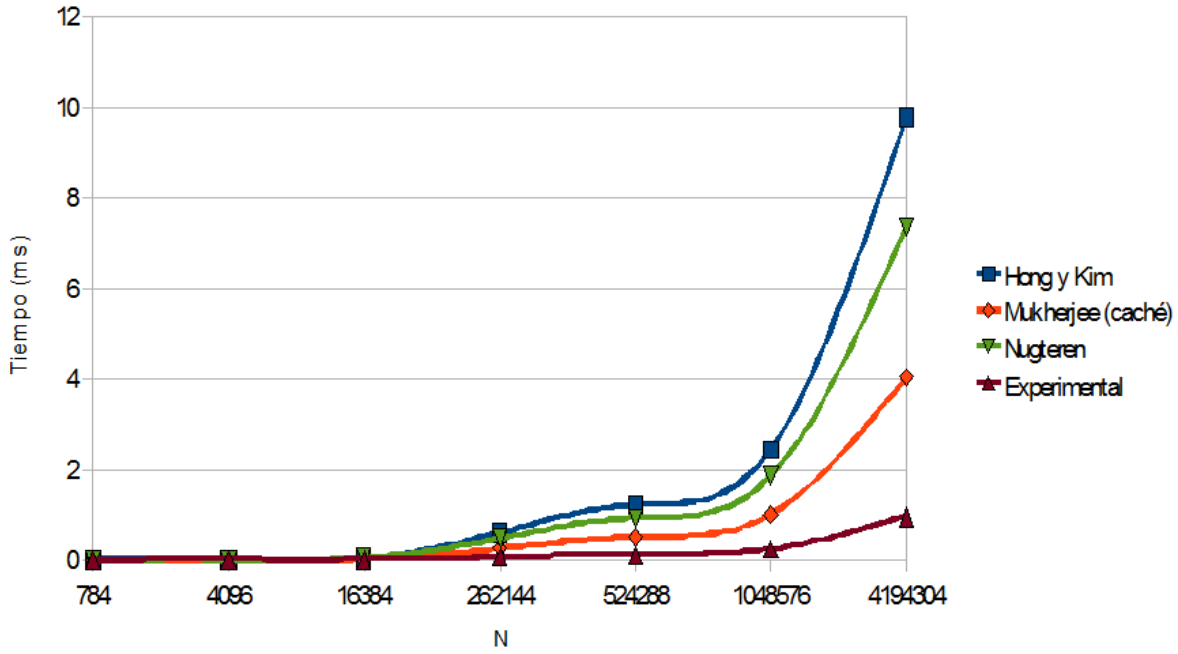

Figura 5.2: Estudio de modelos: suma de vectores (2)

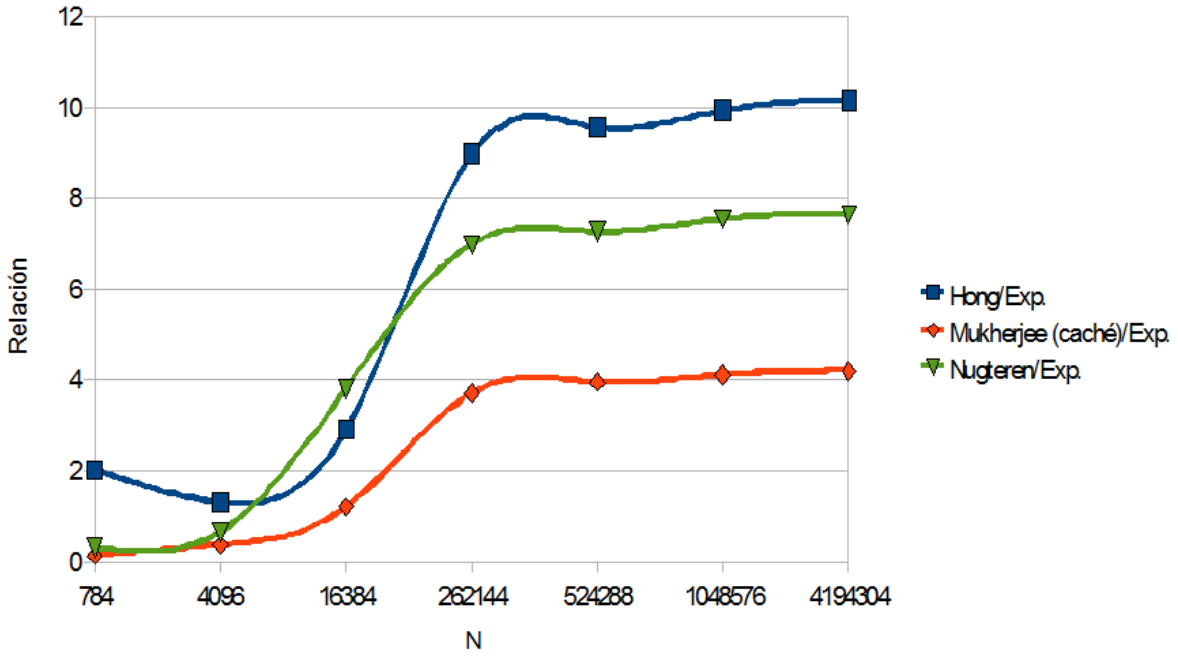

Figura 5.3: Estudio de modelos: suma de vectores, relación modelo/experimental

Al tratarse de un problema más complejo se han empleado dos aproximaciones: además de modificar el tamaño del problema, se ha hecho lo mismo con el tamaño de bloque. Para este segundo enfoque, el tamaño de la matriz es de 4000×2000 elementos, mientras que el vector contiene 2000 elementos.

Un hilo por fila

En la figura 5.4 se puede observar cómo en el modelo de Mukherjee los tiempos se disparan muy por encima del valor real, mientras que el de Nugteren y el de Hong y Kim se acercan más a la curva experimental.

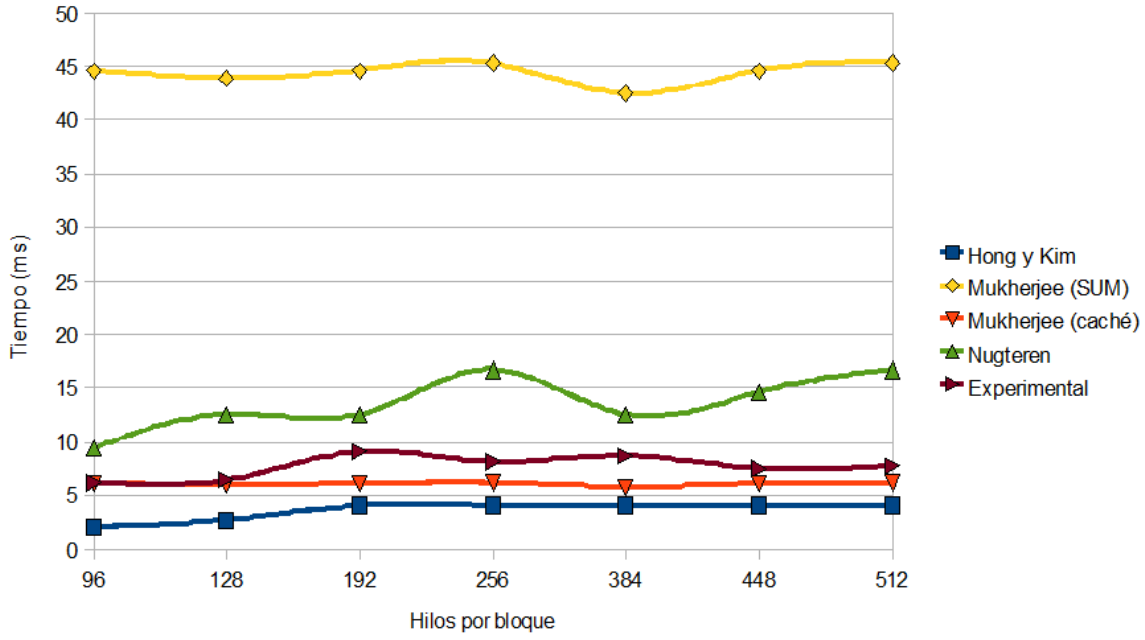

Figura 5.4: Estudio de modelos: matriz-vector por filas - cambio hilos (1)

Al igual que ocurría en el caso de la suma de vectores, en la figura 5.5 se puede observar una gran mejora al incluir la consideración de la caché en el modelo de Mukherjee, convirtiéndolo en el modelo que mejor realiza el ajuste.

Veamos ahora qué ocurre cuando lo que se modifica es el tamaño del problema (figuras 5.7 y 5.8). El modelo de Nugteren y el de Mukherjee modificado siguen de manera bastante fiel la curva del rendimiento. Al comprobar la relación entre la estimación y el resultado experimental, el de Nugteren sigue una evolución un poco más errática, mientras que la proporción del modelo de Mukherjee modificado es más estable.

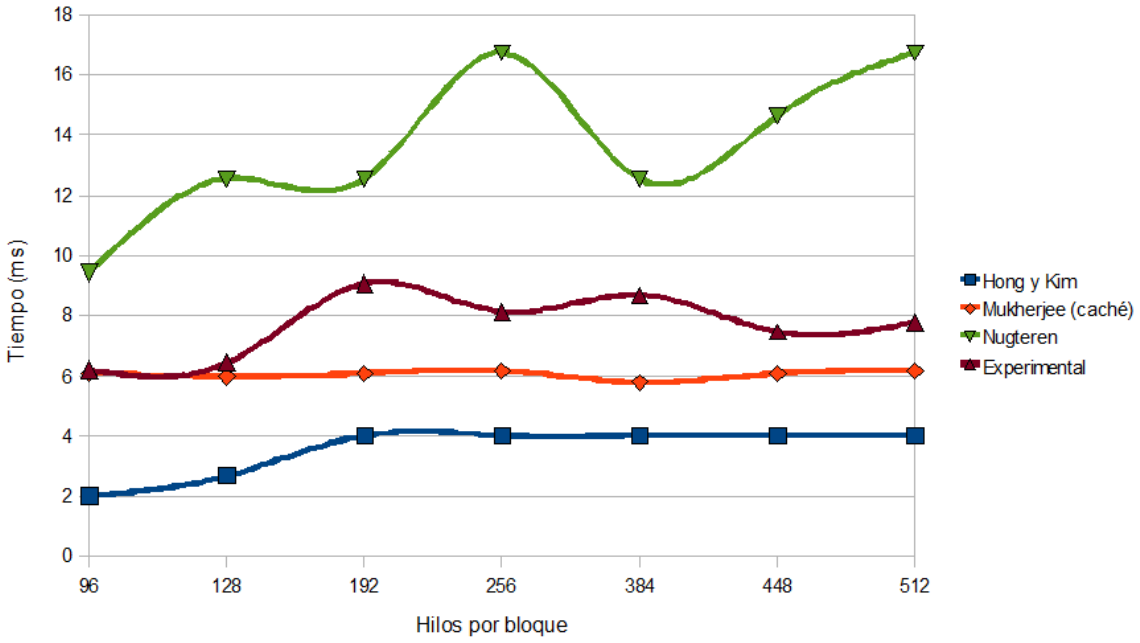

Figura 5.5: Estudio de modelos: matriz-vector por filas - cambio hilos (2)

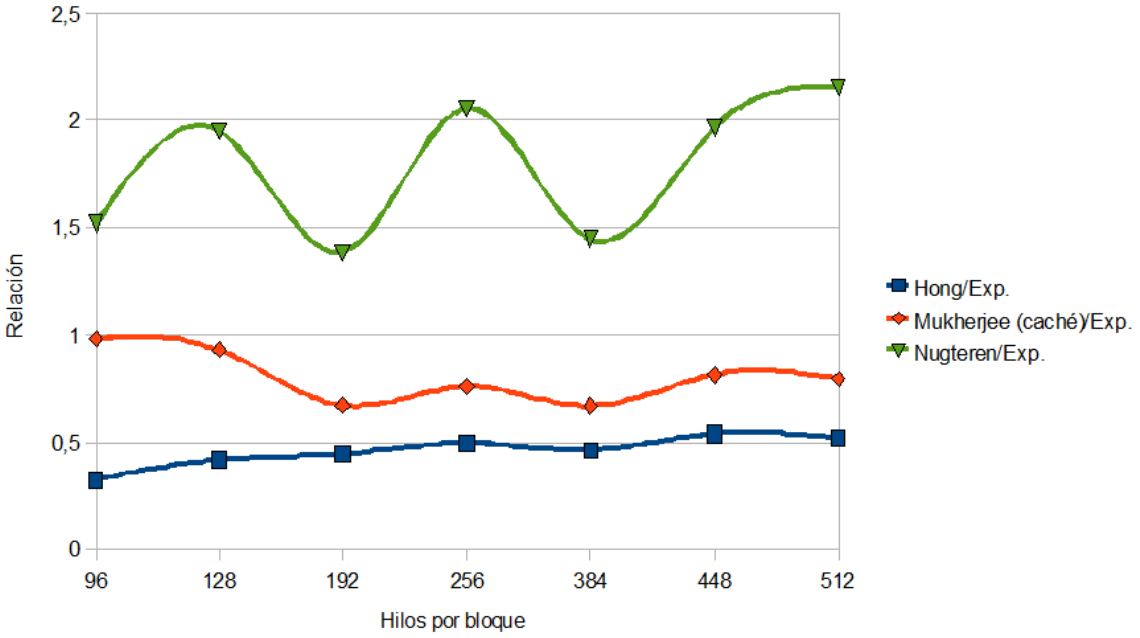

Figura 5.6: Estudio de modelos: matriz-vector por filas - cambio hilos, relación modelo/experimental

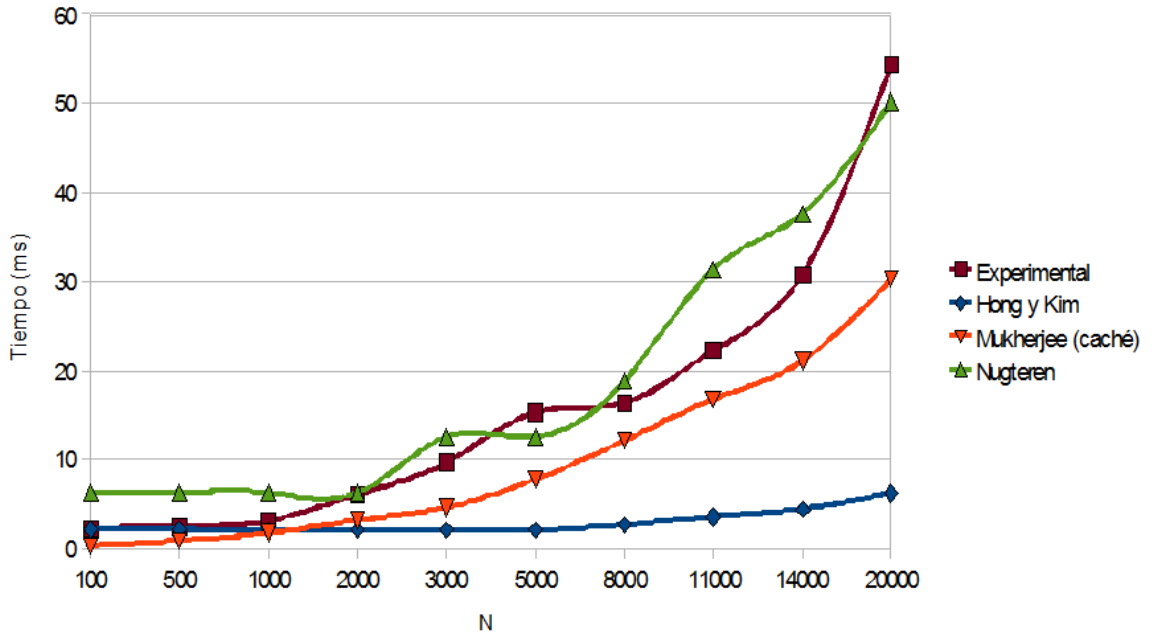

Figura 5.7: Estudio de modelos: matriz-vector por filas - cambio tamaño

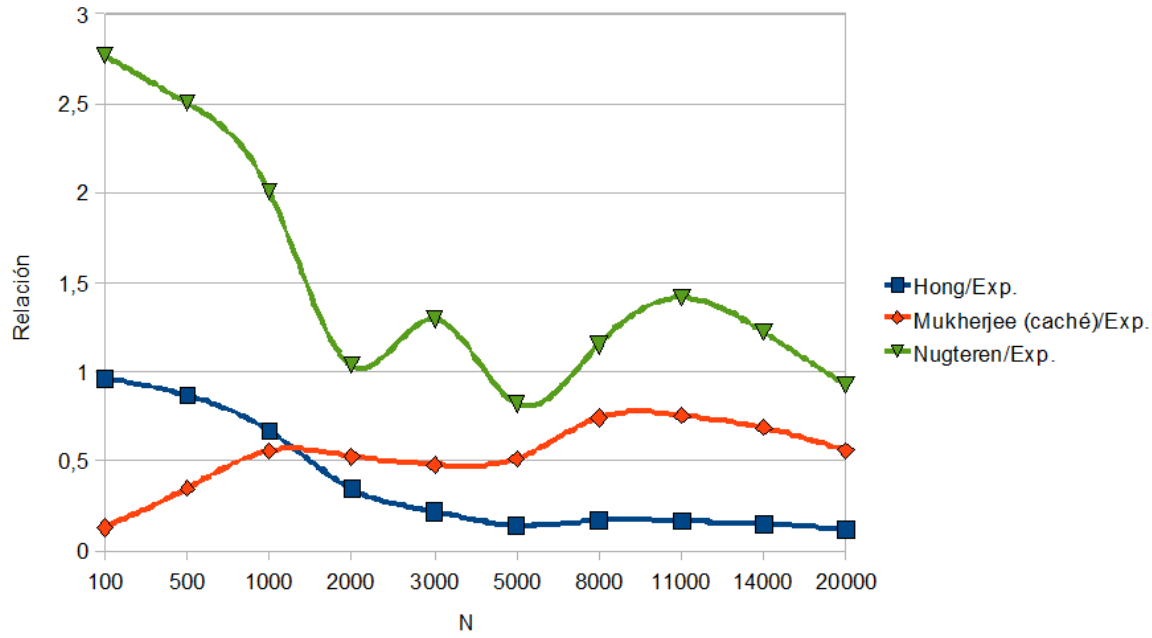

Figura 5.8: Estudio de modelos: matriz-vector por filas, cambio tamaño - relación modelo/experimental

Un hilo por columna

En este enfoque, todos los hilos modifican todas las posiciones del vector resultado. Aunque se podría emplear un algoritmo de reducción, de cara a nuestro estudio resulta más interesante considerar el uso de las denominadas "operaciones atómicas". Para que no se realicen dos actualizaciones de manera simultánea ni se produzca otro tipo de conflicto es necesario serializarlas, pero el único modelo que podría dar buen soporte a esta situación sería el de Baghsorkhi, sobre el que no se puede realizar un estudio práctico por carencia de herramientas.

El enfoque del modelo de Baghsorkhi será tomado como base de cara a incluir esta característica en el modelo diseñado en este trabajo.

Un hilo por bloque de filas

En este último caso, el modelo de Mukherjee en su diseño original no consigue acercarse a los valores reales, ni tampoco lo hace el de Nugteren. El de Hong y Kim está algo más próximo en los tamaños de bloque inferiores a 256, pero tampoco se puede considerar excepcional.

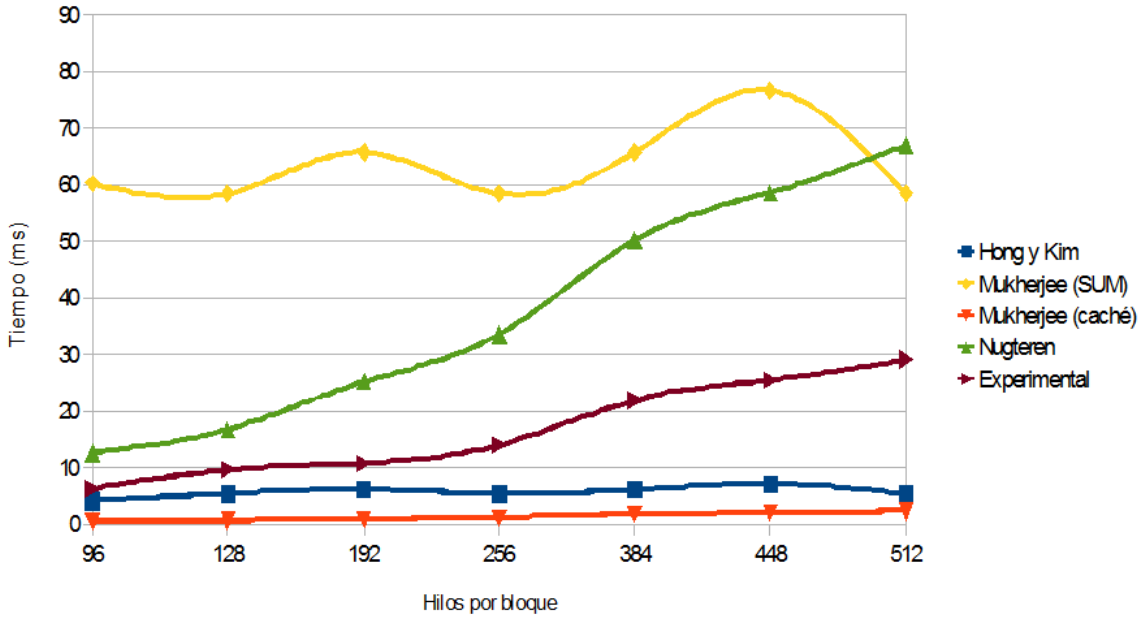

Figura 5.9: Estudio de modelos: matriz-vector por bloques de filas - cambio hilos (1)

Al comprobar la evolución por cambio del tamaño del problema, sin embargo, es el modelo modificado de Mukherjee el que de nuevo sigue más fielmente la curva de evolución. La figura 5.14 muestra que para los tamaños más pequeños el modelo de Hong y Kim realiza la mejor aproximación, pero no hace más que distanciarse

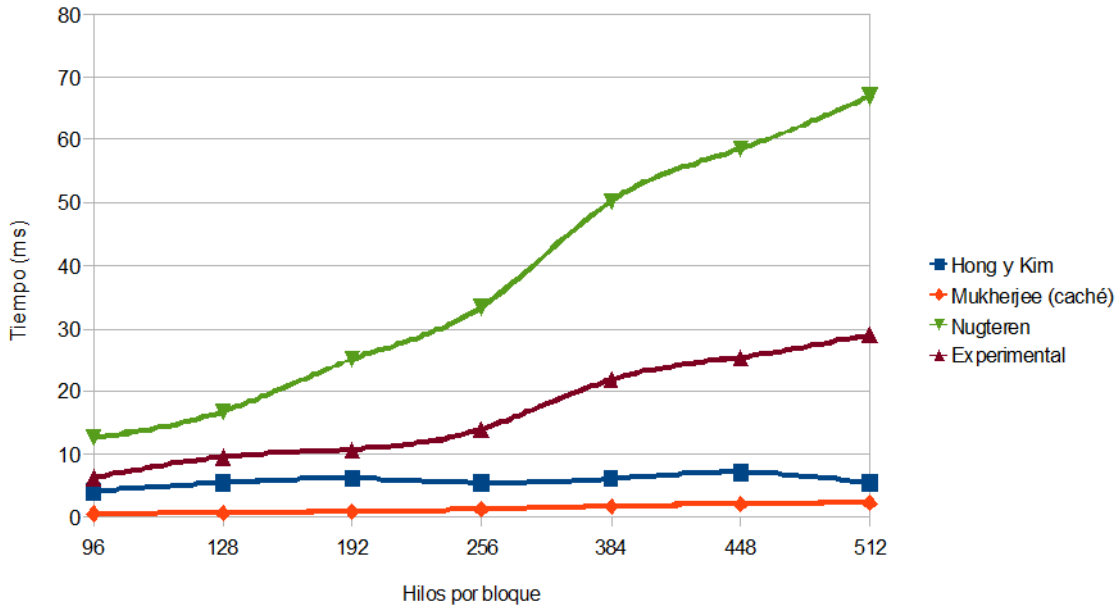

Figura 5.10: Estudio de modelos: matriz-vector por bloques de filas - cambio hilos (2)

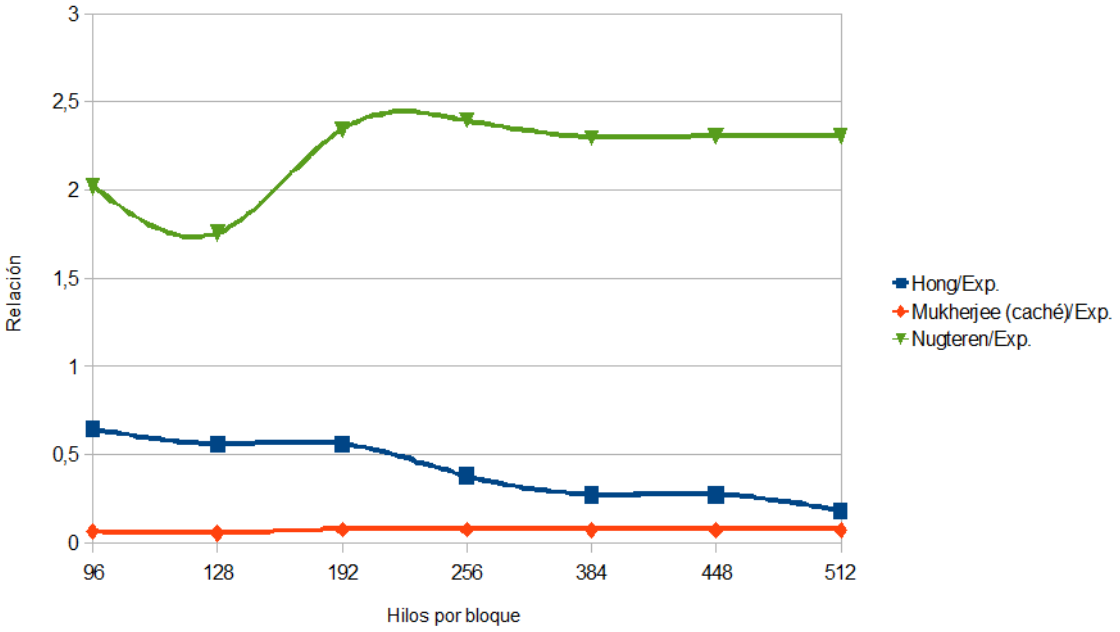

Figura 5.11: Estudio de modelos: matriz-vector por bloques de filas - cambio hilos, relación modelo/experimental

cuando el problema crece por encima de las 10000 filas, mientras que el modelo modificado de Mukherjee se acerca progresivamente al valor experimental.

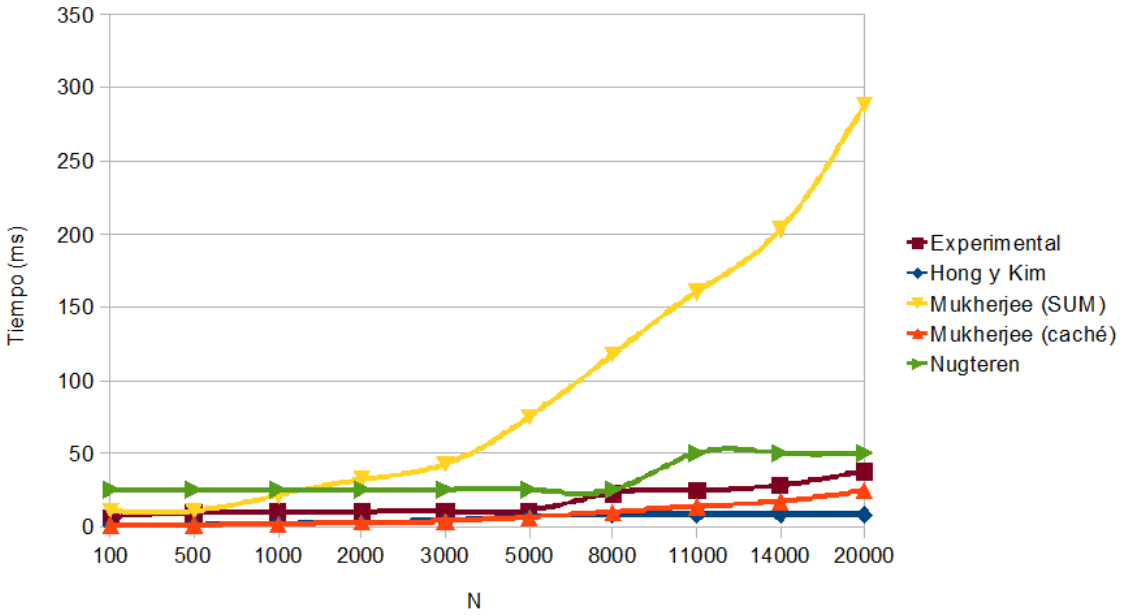

Figura 5.12: Estudio de modelos: matriz-vector por bloques de filas - cambio tamaño (1)

5.1.4. Conclusión

De media, el modelo de Mukherjee modificado es el que más se ajusta a los resultados experimentales. Aunque en algún estudio puntual los otros modelos hayan podido situarse al mismo nivel o mejorarlo ligeramente, a nivel general se ha mostrado superior en todas las pruebas.

En la suma de vectores es el que menos se ha distanciado conforme aumentaba el tamaño del problema. En el producto matriz-vector por filas, aunque en ningún momento se ha aproximado tanto al resultado experimental como el modelo de Nugteren, se comporta de manera menos errática que este. En cuanto al producto matriz-vector por bloques de filas, de nuevo los otros modelos se aproximan más en algún punto, pero el de Mukherjee modificado muestra un comportamiento más estable y que permite apreciar de manera más precisa la evolución.

Por tanto, de cara al diseño de un nuevo modelo se tomará como base el de Mukherjee, y se integrarán en él de manera natural funcionalidades asociadas a la secuenciación de la ejecución, es decir, la divergencia entre ramas y las operaciones atómicas. Puesto que para su correcta aplicación es imprescindible, se realizará un

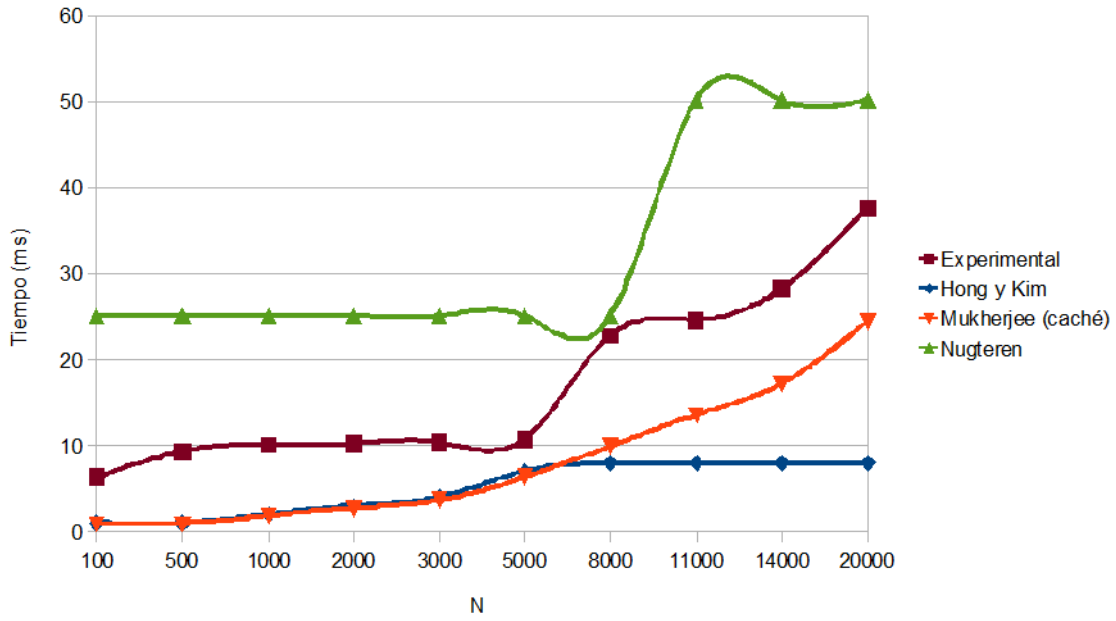

Figura 5.13: Estudio de modelos: matriz-vector por bloques de filas - cambio tamaño (2)

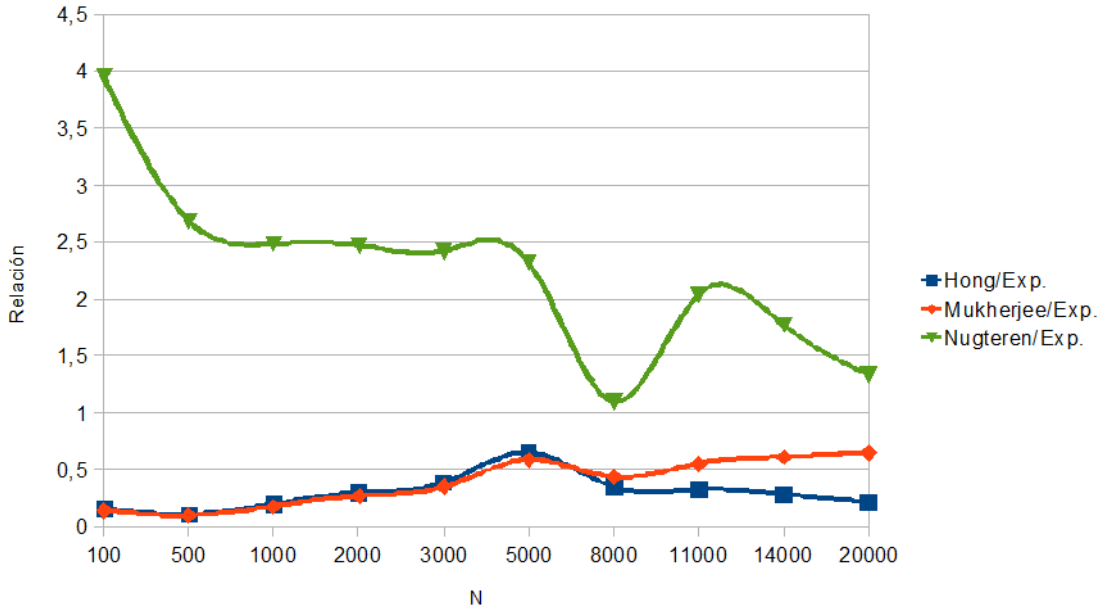

Figura 5.14: Estudio de modelos: matriz-vector por bloques de filas - cambio tamaño, relación modelo/experimental

modelado al menos sencillo de la memoria caché. Por último, se intentará plantear ideas relativas al uso de *streams*.

En cuanto al modelado de aplicaciones completas, se incluirá la información relativa a la copia de datos de CPU a GPU y la sobrecarga de invocación de los *kernel*.

5.2. Diseño del modelo

Como ya se ha indicado en el apartado anterior, la base para el modelo desarrollado en este trabajo será el de Mukherjee. Para más información sobre él, véanse [1] y la sección 3.2 de esta documentación.

Las consideraciones añadidas sobre esta base son:

- Copia de datos CPU $\longleftrightarrow$ GPU
- Sobrecarga de invocación de un *kernel*
- Divergencia entre ramas
- Memoria caché
- Operaciones atómicas

5.2.1. Copia de datos entre CPU y GPU

La copia de datos entre el *host* y el *device* se realiza mediante la función `cudaMemcpy`. Suponiendo una dedicación completa del bus PCI Express en teoría se dispondría, para cada sentido, de 4 GB/s en PCIe 1.1 y de 8 GB/s en PCIe 2.0.

Para verificaro esta suposición, se empleó el ejecutable *bandwithTest* incluido con el CUDA SDK, que comprueba el ancho de banda en MB/s para las copias CPU $\longrightarrow$ GPU, GPU $\longrightarrow$ CPU y GPU $\longrightarrow$ GPU, tanto para memoria paginable como *pinned*, esta última reservada en la memoria del *host* para optimizar los tiempos de transferencia para copias de gran tamaño.

Las pruebas se han realizado sobre tarjetas Fermi (C2070, C2075 y GeForce GT520M) y Tesla (C1060) sobre un bus PCIe 2.0, y los resultados obtenidos sobre algunas de ellas se muestran en las tablas 5.1 y 5.2 en MB/s. Los tiempos obtenidos en los distintos modelos de Fermi son bastante similares a los mostrados en dichas tablas.

De estas medidas se puede deducir que para las transferencias en memoria paginable, el aprovechamiento del bus en el sentido CPU $\longrightarrow$ GPU es de 4 GB/s, y de unos 3 GB/s en el sentido GPU $\longrightarrow$ CPU, independientemente de la generación a la que pertenezca la GPU.

Cuando la memoria se reserva como *pinned*, tal como se explica en [21], el controlador lleva un registro de los rangos reservados y puede acelerar automáticamente

Sentido de copia	Tesla C2070	Tesla C1060
Host a *device*	4355,4	4196,4
Device a *host*	3683,3	2801,4
Device a *device*	81386,2	73670,8

Cuadro 5.1: Ancho de banda en transferencias de memoria paginable (MB/s)

Sentido de copia	Tesla C2070	Tesla C1060
Host a *device*	5761,0	5740,9
Device a *host*	6222,3	3118,7
Device a *device*	81403,4	73570,8

Cuadro 5.2: Ancho de banda en transferencias de memoria *pinned* (MB/s)

las llamadas a funciones como `cudaMemcpy`. La memoria pasa a estar directamente accesible al dispositivo, por lo que el ancho de banda puede aumentar respecto a la memoria paginable.

Las transferencias CPU $\longrightarrow$ GPU están al mismo nivel de rendimiento en ambas generaciones, no así en la copia GPU $\longrightarrow$ CPU, por lo que se ha aproximado el uso del bus PCIe 2.0 para cada una de ellas. Estos porcentajes y los valores en GB/s asociado se muestran en las tablas 5.3 y 5.4.

Sentido de copia	Fermi/Tesla serie 20		Tesla serie 10	
	Paginada	*Pinned*	Paginada	*Pinned*
Host a *device*	50 %	70 %	50 %	70 %
Device a *host*	45 %	75 %	35 %	40 %

Cuadro 5.3: Ocupación del bus PCIe 2.0 según generación de GPU (porcentaje)

Sentido de copia	Fermi/Tesla serie 20		Tesla serie 10	
	Paginada	*Pinned*	Paginada	*Pinned*
Host a *device*	4 GB/s	5'6 GB/s	4 GB/s	5'6 GB/s
Device a *host*	3'6 GB/s	6 GB/s	2'8 GB/s	3'2 GB/s

Cuadro 5.4: Ocupación del bus PCIe 2.0 según generación de GPU

Para obtener el tiempo consumido basta con dividir la cantidad de información transferida por la tasa de transferencia.

5.2.2. Sobrecarga de invocación de un *kernel*

El coste que supone invocar a un *kernel* dentro de una aplicación está modelado en [7], donde se indica que la sobrecarga base de la invocación es constante y de unos 6000 ciclos, y que se debe añadir un coste extra por cada bloque de hilos y si recibe más de 5 argumentos. Desde los foros de NVIDIA, sin embargo, afirman que se encuentra en un rango de 2'5 μs, que en una C2070 sería el equivalente a 2875 ciclos. Para comprobar la opción que se ajusta más a la realidad, se realizaron mediciones experimentales del coste de invocación con 1, 16 ó 64 bloques a:

- Un *kernel* vacío y sin argumentos
- Un *kernel* vacío y con un argumento
- Un *kernel* vacío y con 3 argumentos
- Un *kernel* vacío y con más de 5 argumentos

Argumentos	Tesla C2070			Tesla C1060		
	1 bloque	16 bloques	64 bloques	1 bloque	16 bloques	64 bloques
0	0,002870	0,002888	0,002877	0,003827	0,004019	0,004048
1	0,003008	0,002958	0,002997	0,004093	0,004174	0,004262
3<5	0,002944	0,002933	0,002954	0,003859	0,004045	0,004064
6>5	0,002886	0,002893	0,002893	0,003891	0,004022	0,004042

Cuadro 5.5: Coste de la invocación de un *kernel* (ms)

En los resultados mostrados en la tabla 5.5, el modelo C2070 (arquitectura Fermi) no muestra ninguna sobrecarga por el número de bloques ni de argumentos, mientras que el C1060 (arquitectura Tesla), existente cuando se llevó a cabo el estudio en febrero de 2010, sí sufre una pequeña penalización. Es de suponer, por tanto, que esta penalización apenas perceptible (unos 250 ciclos) se haya eliminado con mejoras en la arquitectura.

Se puede observar que los valores oscilan entre unos 0'003 ms = 3 μs = 3450 ciclos para la C2070 y los 0'004 ms = 4 μs = 5200 ciclos para la C1060. Por ello, se utilizarán estos tiempos como cotas fijas para cada generación, y los ciclos se calcularán en función del reloj del procesador.

5.2.3. Divergencia entre ramas

Cuando dentro de un *kernel* hay algún tipo de expresión condicional (`if`, `for`, `while`...), si los hilos dentro de un *warp* siguen caminos diferentes, su ejecución se serializa: primero se ejecutan todos los hilos de una rama, y a continuación todos

los hilos de la otra. Por tanto, un enfoque intuitivo sería que el coste equivale al de evaluar la condición y ejecutar todas las instrucciones de ambas ramas en paralelo en todos los hilos en paralelo y sin esperas.

Para comprobar la corrección de este enfoque se han implementado dos *kernel*. En 5.2, los hilos que pertenecen a la mitad inferior de cada *warp* suman 1 a cada posición del *array*, mientras que los de la mitad superior suman la raíz cuadrada de su identificador de hilo.

Listing 5.2: *Kernel* con divergencia entre ramas

```
__global__ void divergenceKernel(float *v, int n) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  if (i < n) {
    if (threadIdx.x % 32 < 16) {
      v[i] = v[i] + 1;
    } else {
      v[i] = v[i] + sqrtf(threadIdx.x);
    }
  }
}
```

En 5.3 se fusionan las dos ramas y, para tener en cuenta el coste de todos los cálculos, se añade también la evaluación de la condición en una sentencia independiente.

Listing 5.3: *Kernel* que fusiona ramas divergentes

```
__global__ void mergeBranchesKernel(float *v, int n) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j;

  if (i < n) {
    j = threadIdx.x % 32 < 16;
    v[i] = v[i] + 1;
    v[i] = v[i] + sqrtf(threadIdx.x);
  }
}
```

Así pues, se ha intentado que ambas implementaciones sean lo más similares posibles en cuanto a coste. Sin embargo, al ejecutar estos algoritmos, existe una diferencia bastante estable entre ellos, independentemente de la arquitectura (el modelo C2070 es Fermi y el C1060 es Tesla) y del número de bloques y/o hilos escogido. Como ejemplo ilustrativo, la siguiente tabla muestra las mediciones (en

milisegundos) tomadas al emplear 64 bloques de 128 hilos, así como la diferencia y relación entre ellas.

Kernel	Tesla C2070	Tesla C1060
divergent	0,007763	0,008448
mergeBranches	0,004698	0,005690
Diferencia	0,003065	0,002758
Relación	1,652405	1,484710

La diferencia de tiempos oscila en torno a los 3×10^{-3} ms, suponiendo para este caso una sobrecarga de aproximadamente un 50 %. Para averiguar qué produce esta sobrecarga tan notoria con un nivel de exactitud que permita extraer conclusiones para el modelo es necesario trabajar con el código PTX.

Comparativa del PTX divergente y fusionado

El primer paso para averiguar qué parte del código produce la sobrecarga es interpretar el código PTX [22]. El cuerpo (sin cabeceras) de los *kernel* a comparar se muestra en los *listings* 5.4 y 5.5.

Las líneas 2 a 6 representan el cálculo de la variable `i = threadIdx.x + blockIdx.x * blockDim.x`. Su comparación con `n` se lleva a cabo entre las líneas 7 y 9, donde se produce la primera ramificación. En las líneas 11 a 15 se lee el valor de `v[i]`, y en la 16 a 20 se ejecuta la condición que da lugar a la segunda ramificación. En el fragmento con las ramas unificadas este último cálculo se omite.

En el primer fragmento, la rama afirmativa se ejecuta dentro de un bloque iniciado por la directiva `bra` y terminado por `bra.uni`, mientras que en el segundo dichas directivas no aparecen. Lo mismo ocurre con la rama negativa, iniciada por la etiqueta a la que se produjo el salto desde la condición.

Por tanto, las diferencias se encuentran en dos puntos:

- La evaluación de la condición más interna, que en el segundo fragmento se omite por no ser necesaria en ninguna otra parte del código.
- Las directivas de ramificación `bra` y `bra.uni`.

Listing 5.4: PTX del *kernel* con divergencia entre ramas

```
$LDWbegin__Z16divergenceKernel:
  mov.u16    %rh1, %ctaid.x;
  mov.u16    %rh2, %ntid.x;
  mul.wide.u16   %r1, %rh1, %rh2;
  cvt.u32.u16    %r2, %tid.x;
  add.u32    %r3, %r1, %r2;
  ld.param.s32   %r4, [↩
      __cudaparm__Z16divergenceKernel_n];
  setp.le.s32    %p1, %r4, %r3;
  @%p1 bra   $Lt_0_1794;

  ld.param.u64   %rd1, [↩
      __cudaparm__Z16divergenceKernel_v];
  cvt.s64.s32    %rd2, %r3;
  mul.wide.s32   %rd3, %r3, 4;
  add.u64    %rd4, %rd1, %rd3;
  ld.global.f32    %f1, [%rd4+0];
  and.b32    %r5, %r2, 31;
  mov.u32    %r6, 15;
  setp.gt.u32    %p2, %r5, %r6;
  @%p2 bra   $Lt_0_2562;
  .loc   16   12   0
  mov.f32    %f2, 0f3f800000;       // 1
  add.f32    %f3, %f1, %f2;
  st.global.f32    [%rd4+0], %f3;
  bra.uni    $Lt_0_2306;
$Lt_0_2562:
  .loc   16   14   0
  cvt.rn.f32.u32   %f4, %r2;
  sqrt.approx.f32    %f5, %f4;
  add.f32    %f6, %f1, %f5;
  st.global.f32    [%rd4+0], %f6;
$Lt_0_2306:
$Lt_0_1794:
  .loc   16   17   0
  exit;
$LDWend__Z16divergenceKernel:
  } // _Z16divergenceKernel
```

Listing 5.5: PTX del *kernel* que fusiona ramas divergentes

```
$LDWbegin__Z19mergeBranchesKernel:
  mov.u16    %rh1, %ctaid.x;
  mov.u16    %rh2, %ntid.x;
  mul.wide.u16   %r1, %rh1, %rh2;
  cvt.u32.u16    %r2, %tid.x;
  add.u32    %r3, %r1, %r2;
  ld.param.s32   %r4, [↩
      __cudaparm__Z19mergeBranchesKernel_n];
  setp.le.s32    %p1, %r4, %r3;
  @%p1 bra   $Lt_1_1026;
  .loc   16   24   0
  ld.param.u64   %rd1, [↩
      __cudaparm__Z19mergeBranchesKernel_v];
  cvt.s64.s32    %rd2, %r3;
  mul.wide.s32   %rd3, %r3, 4;
  add.u64    %rd4, %rd1, %rd3;
  ld.global.f32    %f1, [%rd4+0];
                                  // 1
  mov.f32    %f2, 0f3f800000;
  add.f32    %f3, %f1, %f2;
  st.global.f32    [%rd4+0], %f3;

  .loc   16   25   0
  cvt.rn.f32.u32   %f4, %r2;
  sqrt.approx.f32    %f5, %f4;
  add.f32    %f6, %f3, %f5;
  st.global.f32    [%rd4+0], %f6;

$Lt_1_1026:
  .loc   16   27   0
  exit;
$LDWend__Z19mergeBranchesKernel:
  } // _Z19mergeBranchesKernel
```

Coste según número de divergencias y *warps* en ejecución

Una vez conocida la parte del código que provoca la diferencia de tiempo, se tomaron como referencia los *micro-benchmarks* llevados a cabo en [23] para el estudio de la divergencia. Según afirman en dicho artículo, cuando se producen N divergencias en un *warp*, cada una de las ramas se serializa, pero se puede seguir produciendo solapamiento entre distintos *warps*. Esta situación se ilustra en la figura 5.15, extraída de [23].

Suponiendo la ejecución de dos *warps* donde todos los hilos son divergentes, se observa cómo el tiempo consumido por cada *warp* se divide en 32 segmentos ejecutados en serie. Sin embargo, para un mismo instante de tiempo, los hilos de ambos *warps* que siguen el mismo camino se ejecutan de manera simultánea. Así, en los primeros 5000 ciclos de reloj se completaría el camino de los hilos 0 a 4 de ambos *warps*, antes de los 10000 ciclos se habrían ejecutado los hilos 5 a 10, y así sucesivamente.

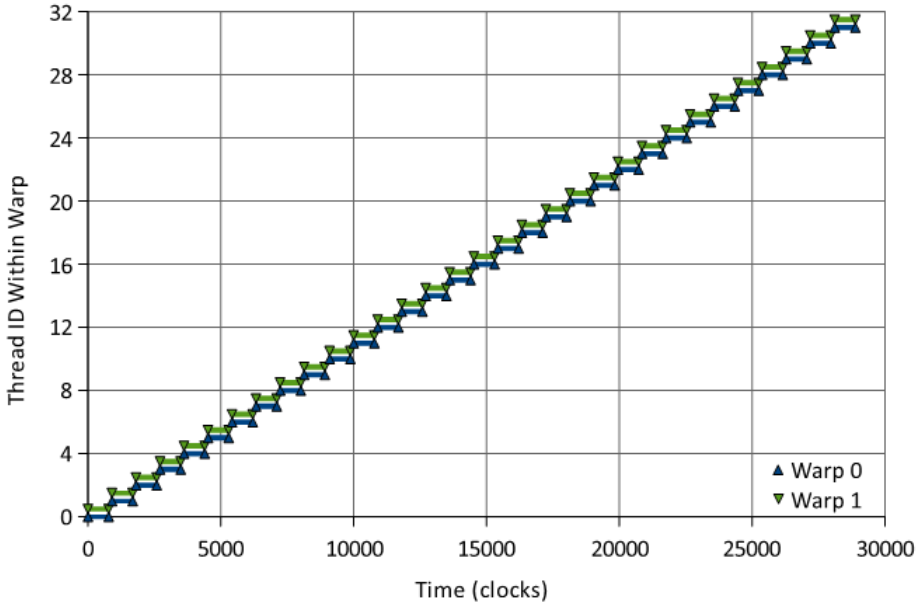

Figura 5.15: Paralelismo entre *warps* con divergencia entre ramas

Por desgracia, el estudio llevado a cabo en dicho artículo no va más allá del empleo de un par de *warps* simultáneamente, lo cual es claramente insuficiente para extraer una conclusión más general. Por tanto, tomando como base su código se realizaron pruebas para distintas combinaciones.

El coste experimental se calculó empleando la función `clock()` desde el interior del propio *kernel* para evaluar el coste de cada rama, equivalente al siguiente código:

Listing 5.6: Modelo de rama en el *kernel* de [23] para estudiar la divergencia

```
start_time = clock();
repeat16(t1+=t2;t2+=t1;)
stop_time = clock();
```

Siguiendo las indicaciones del artículo, se considera que una suma en coma flotante de 32 bits consume 24 ciclos. Mediante la línea `repeat16(t1+=t2;t2+=t1;)` se ejecutan dos sumas 16 veces, por lo que el total de ciclos consumidos será de $24 \times 2 \times 16 = 768$. Ya que la ejecución de las ramas se serializa, se debe sumar este coste tantas veces como divergencias haya. En la tabla siguiente se muestran la estimación para cierto número de *warps* y divergencias, los resultados obtenidos de manera experimental y el porcentaje de error en la estimación.

Warps	Divergencias	Estimación	Experimental	% error
1	16	12288	11400	8,88
1	2	1536	1100	4,36
2	32	24576	23100	14,76
2	2	1536	1100	4,36
8	8	6144	5800	3,44
8	4	3072	2750	3,22
8	2	1536	1240	2,96
16	8	6144	7100	9,56
16	2	1536	1700	1,64
32	2	1536	3370	18,34
32	32	24576	24700	1,24
Promedio				6,61

Exceptuando casos con una cantidad grande de divergencias, lo que posiblemente implicaría un mal diseño del algoritmo, o con el máximo posible de hilos por bloque (32 *warps* $\times$ 32 hilos = 1024 hilos/bloque), el margen de error se encuentra por debajo del 5 %, siendo el promedio inferior al 7 %.

5.2.4. Memoria caché

En la sección 5.1 se ha observado cómo, para que el modelo de Mukherjee realice un ajuste adecuado, es necesario tener en cuenta la presencia de memoria caché L1/L2 en las GPU de arquitectura Fermi. En las primeras tarjetas con soporte a CUDA ya fueron incluidas caché de texturas, constantes, etc., pero no una caché genérica para acceso a memoria global.

En la arquitectura Fermi, se incluye una caché L1 para cada multiprocesador, y una caché L2 compartida entre todos. El tamaño de la memoria caché L1 es configurable, pudiendo asignarle 16 ó 48 KB.

El tamaño de la línea de caché es de 128 bytes, equivalentes a 32 datos en coma flotante de simple precisión, o a 16 datos si se representan en doble precisión. Por tanto, una aproximación primitiva (empleada en el capítulo 5.1) podría consistir en dividir el coste de las operaciones de acceso a memoria entre el número N de datos que se van a cargar en la caché, es decir, que solo una operación de cada N se realiza sobre memoria global, y el resto sobre caché. La fórmula que se aplicaría en este caso sería $cyc_{mem} = cyc_{mem}/N$, siendo $N = 16$ si los datos son de tipo *double* y $N = 32$ si son de tipo *float*. Sin embargo, esto no ofrece el valor real sino la cota inferior.

Cuando los datos están presentes en caché L1 y L2 simultánemente se sirven en transacciones de memoria de 128 bytes, mientras que si solo están presentes en la caché L2 se servirán en transacciones de 32 bytes. Una aproximación más fiel que la arriba descrita debe tener en cuenta la tasa de acierto y fallo en el acceso a los diferentes niveles de caché, tal como se define en [24]. Esto implica realizar un análisis exhaustivo del comportamiento del algoritmo. Dado que se busca un modelo de alto nivel que resulte rápido de aplicar, otra alternativa es aproximarlo a la mitad de lecturas sobre caché L1 y la mitad sobre caché L2. En este caso se aplicarían las fórmulas siguientes:

$$cache\_factor = \frac{N_{max} + N_{min}}{2}$$

$$cyc_{mem} = \frac{gmem\_latency \times op_{mem}}{cache\_factor} + \frac{cache\_latency \times op_{mem} \times (cache\_factor - 1)}{cache\_factor}$$

Primera fórmula: La variable $cache\_factor$ es la media entre el número de datos que se pueden transferir en una sola transacción cuando estos están presentes en caché L1 y L2 simultánemente (transacciones de 128 bytes) o solo en L2 (32 bytes).

Segunda fórmula: El primer sumando representa que la primera de cada $cache\_factor$ operaciones de memoria se realiza sobre memoria global. El segundo, que las siguientes hasta consumir toda la información transferida en la primera se obtienen de caché y, por tanto, la latencia es mucho menor (aproximadamente 4 ciclos, la misma que en memoria *shared*).

5.2.5. Operaciones atómicas

Este tipo de operaciones garantizan que, cuando varios hilos escriban sobre la misma posición de memoria, se acceda a ella a través de una sección crítica, actualizando de manera secuencial su valor. De manera intuitiva, el coste debería ser equivalente a la suma del coste de todas las operaciones de lectura, computación y escritura; también se le debe añadir el coste de contención, lo que supone uno de los mayores problemas por depender de muchos factores.

Para verificar esta hipótesis, se implementaron dos *kernel* por cada operación atómica disponible. Uno de ellos ejecuta la operación N veces en un solo hilo y un solo bloque. El otro ejecuta la operación atómica, usando la función `atomicXXX`, por parte de N hilos en un solo bloque.

Puesto que el número de hilos máximo en un bloque depende de la *compute capability* del dispositivo, se realizaron pruebas en una tarjeta Tesla C2070 con CC2.0 y en una Tesla C1060 con CC1.3. La primera acepta 1024 hilos por bloque como máximo y la segunda 512. En la figura 5.16 se compara el coste de ejecutar la operación atómica en las diferentes tarjetas, así como su ejecución secuencialmente sobre un registro y sobre memoria global (en estos dos últimos casos el coste es similar en ambos dispositivos).

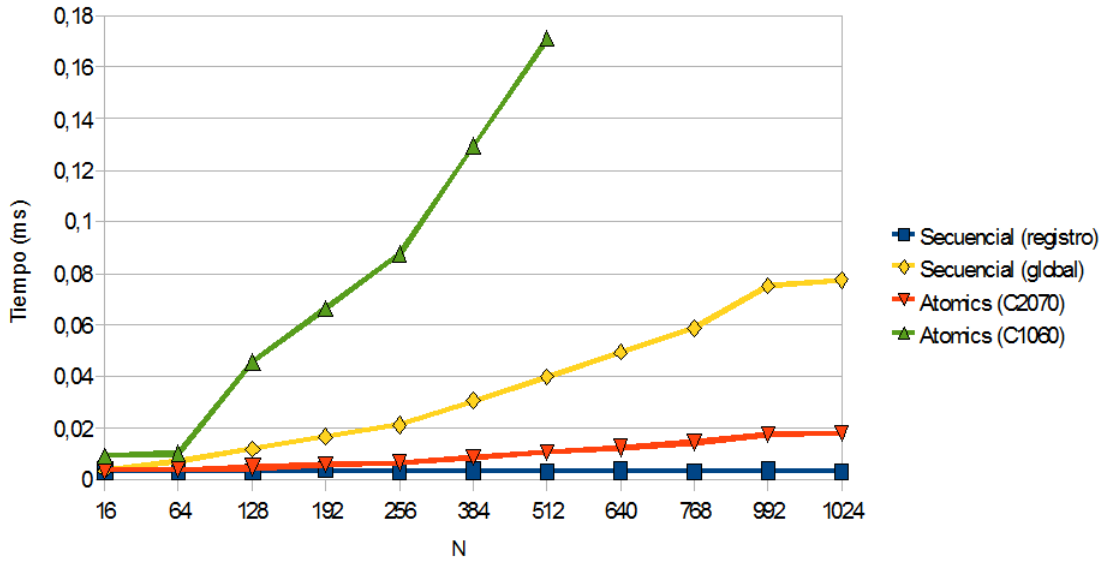

Figura 5.16: Comparativa del coste de una operacion secuencial y su equivalente atómico

Como se puede apreciar, la curva de evolución es distinta y difícil de conciliar en todas las implementaciones. Sin embargo, se ha comprobado experimentalmente que son varias las operaciones que siguen exactamente la misma evolución temporal hasta una precisión de microsegundos:

- `atomicAdd`
- `atomicMax`
- `atomicAnd`
- `atomicSub`
- `atomicInc`
- `atomicOr`
- `atomicMin`
- `atomicDec`
- `atomicXor`

Las operaciones que se comportan de manera diferente son `atomicExch` y `atomicCAS`. Por tanto, se puede intentar definir al menos un modelo para todas las operaciones que muestran el mismo comportamiento.

Para ello, a partir de los valores experimentales y empleando MATLAB se realizó una aproximación para cada una de las *compute capability* estudiadas. Es-

ta aproximación toma como variable el número de hilso que ejecutan la operación atómica. Las fórmulas obtenidas son:

- CC2.0: $T = 0,0000147 \times N + 0,003$ ms o, de manera aproximada, $C = 17 \times N + 3450$ ciclos
- CC1.3: $T = 0,00033 \times N + 0,003$ ms o, de manera aproximada, $C = 380 \times N + 3450$ ciclos

En las gráficas 5.17 y 5.18 se puede apreciar cómo la aproximación se ajusta con precisión a la realidad. Sin embargo, al tratarse de un resultado obtenido a través de una prueba experimental muy concreta, su validación en otras situaciones queda pospuesta hasta la sección 9.2, donde se evaluará el producto matriz-vector mediante un enfoque por columnas.

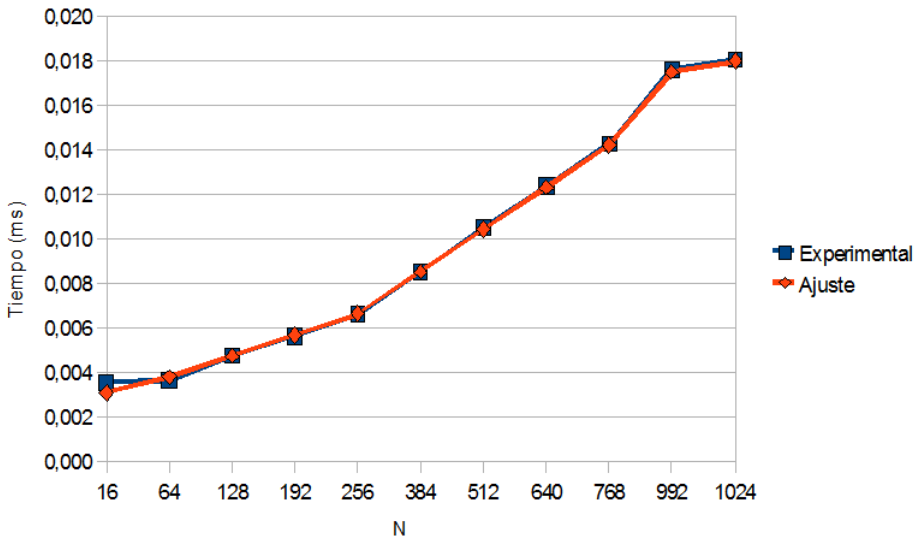

Figura 5.17: Aproximación del coste de una operación atómica en una tarjeta con *compute capability* 2.0

5.3. Definición completa del modelo

Tras decidir qué modelo tomar como base y qué modificaciones aplicar sobre él, en esta sección se define de manera completa el modelo resultante. Los parámetros en los que se basa son los siguientes:

- *comp_insts*: Número de instrucciones de computación.

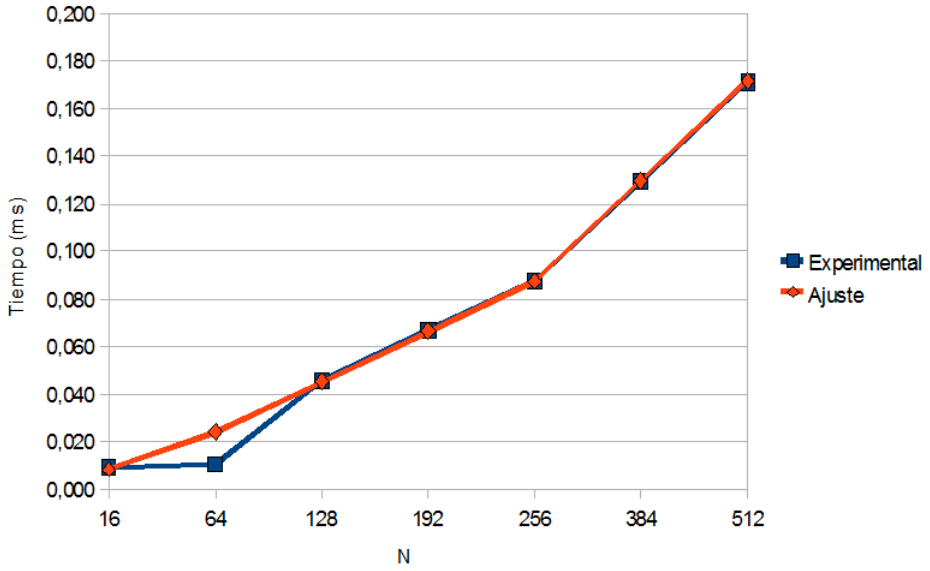

Figura 5.18: Aproximación del coste de una operación atómica en una tarjeta con *compute capability* 1.3

- $issue\_cycles$: Coste de una operación de computación (dependiente principalmente del tipo de dato a usar).
- C_{comp}: Ciclos consumidos en instrucciones de computación.
- mem_insts: Número de instrucciones de acceso a memoria.
- $data\_size$: Tamaño del tipo de datos con los que se está trabajando.
- N_{max}: Máximo de datos que se pueden transferir en una petición de caché (128 bytes).
- N_{min}: Mínimo de datos que se pueden transferir en una petición de caché (32 bytes).
- $cache\_factor$: Media entre el máximo y el mínimo número de datos que se pueden transferir en una petición de caché.
- $latency_{gmem}$: Latencia de acceso a memoria global en ciclos.
- $latency_{smem}$: Latencia de acceso a memoria *shared* en ciclos.
- $latency_{cache}$: Latencia de acceso a memoria caché en ciclos.
- C_{mem}: Ciclos consumidos en instrucciones de acceso a memoria.

- C_{max}: Máximo de los ciclos de computación o de acceso a memoria (si hay mucho solapamiento).
- C_{sum}: Suma de los ciclos de computación o de acceso a memoria (si apenas se solapan).
- $N_B(K)$: Número de bloques asignados a la ejecución de un *kernel* K.
- $N_t(K)$: Número de hilos que puede contener un *warp*.
- $N_w(K)$: Número de *warps* contenidos en cada bloque que ejecuta un *kernel* K.
- N_C: Número de *cores* o procesadores contenidos en un multiprocesador.
- D: Profundidad del *pipeline* de cada *core* de la GPU.
- R: Frecuencia del procesador de la GPU.
- $C(K)$: Coste de ejecución del *kernel* K en ciclos.
- $T(K)$: Coste de ejecución del *kernel* K en segundos.
- BW_{HD}: Ancho de banda de las transferencias de *host* a *device* en GB/s.
- BW_{DH}: Ancho de banda de las transferencias de *device* a *host* en GB/s.
- $T(HD)$: Coste de la transferencia de *host* a *device* en segundos.
- $T(DH)$: Coste de la transferencia de *device* a *host* en segundos.
- T: Coste total del programa en segundos.

A continuación se muestra cómo derivar unos de otros en el caso más sencillo, es decir, cuando la ejecución sigue un camino único. El primer paso es contar el número de instrucciones de computación. Supongamos que $comp\_insts = 10$ y que los datos tratados se representan en coma flotante de doble precisión, por lo que $issue\_cycles = 48$.

$$C_{comp} = comp\_insts \times issue\_cycles = 10 \times 48 = 480 \text{ ciclos}$$

Para el coste de acceso a memoria, supongamos que cada hilo necesita acceder a 20 datos. La primera petición se realizará sobre memoria global y las siguientes sobre caché. Por tanto, el coste será el siguiente:

$$N_{max} = 128/data\_size = 128/8 = 16 \text{ datos}$$

$$N_{min} = 32/data\_size = 32/8 = 4 \text{ datos}$$

$$cache\_factor = \frac{N_{max} + N_{min}}{2} = \frac{16+4}{2} = 10 \text{ datos}$$

$$C_{gmem} = \frac{mem\_insts \times latency_{gmem}}{cache\_factor} = \frac{20 \times 600}{10} = 1200 \text{ ciclos}$$

$$C_{cache} = \frac{mem\_insts \times latency_{cache} \times (cache\_factor - 1)}{cache\_factor} = \frac{20 \times 4 \times 9}{10} = 72 \text{ ciclos}$$

$$C_{mem} = C_{gmem} + C_{cache} = 1200 + 72 = 1272 \text{ ciclos}$$

Según las operaciones de computación y de acceso a memoria puedan solaparse o no, el coste de ejecución medio de un hilo será uno de los siguientes:

$$C_{max} = max(C_{comp}, C_{mem}) = max(480, 1272) = 1272 \text{ ciclos}$$

$$C_{sum} = C_{comp} + C_{mem} = 480 + 1272 = 1752 \text{ ciclos}$$

Si el *kernel* se invoca con 4 bloques de 128 hilos cada uno, el coste total se calcula siguiendo los pasos mostrados a continuación:

$$N_B(K) = 4 \text{ bloques}$$

$$N_t(K) = 32 \text{ hilos}$$

$$N_w(K) = 128/N_t(K) = 128/32 = 4 \textit{ warps}$$

$$N_C = 32 \textit{ cores} \text{ por SM (en una Tesla C2070)}$$

$$D = 4$$

$$\begin{aligned} C_{max}(K) &= N_B(K) \times N_w(K) \times N_t(K) \times C_{max} \times \frac{1}{N_C \times D} \\ &= 4 \times 4 \times 32 \times 1272 \times \frac{1}{32 \times 4} = 5088 \text{ ciclos} \end{aligned}$$

$$T_{max}(K) = \frac{C_{max}(K)}{R} = \frac{5088 \text{ ciclos}}{1,15 \times 10^9 \text{ GHz}} = 4424,34 \times 10^{-9} \text{ s}$$

$$\begin{aligned} C_{sum}(K) &= N_B(K) \times N_w(K) \times N_t(K) \times C_{sum} \times \frac{1}{N_C \times D} \\ &= 4 \times 4 \times 32 \times 1752 \times \frac{1}{32 \times 4} = 7008 \text{ ciclos} \end{aligned}$$

$$T_{sum}(K) = \frac{C_{sum}(K)}{R} = \frac{7008 \text{ ciclos}}{1,15 \times 10^9 \text{ GHz}} = 6093,91 \times 10^{-9} \text{ s}$$

Si el algoritmo emplea estructuras de control con límites claramente delimitados, se descompondrá en diferentes partes y se irá sumando el coste de cada una de ellas. Supongamos un algoritmo que se puede representar mediante el grafo 5.19.

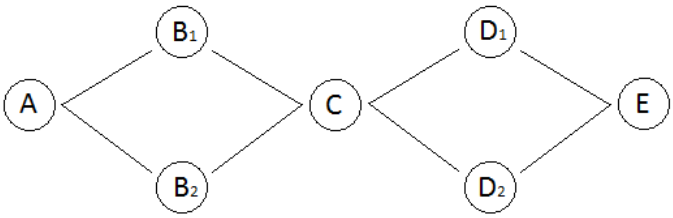

Figura 5.19: Grafo de la estructura de un programa de ejemplo

En este caso, el coste de la ejecución del algoritmo sería, de acuerdo a lo explicado en el apartado 5.2.3:

$$C_B = C_B1 + C_B2$$
$$C_D = C_D1 + C_D2$$
$$C_{max} = C_{max\_A} + C_{max\_B} + C_{max\_C} + C_{max\_D} + C_{max\_E}$$
$$C_{sum} = C_{sum\_A} + C_{sum\_B} + C_{sum\_C} + C_{sum\_D} + C_{sum\_E}$$

Para cada uno de los nodos basta con seguir el mismo proceso que para un algoritmo sin ninguna divergencia.

Finalmente, se puede añadir a la consideración el coste de transferencia desde y hacia CPU. Supongamos el mismo problema que al inicio del apartado, donde cada hilo trabaja con 20 datos en coma flotante de doble precisión. Si hay 4 bloques de 128 hilos y los datos se envían como entrada y se reciben como salida, el total de datos a transferir sería de 80 KB en cada sentido, por tanto:

$$BW_{HD} = 4 \text{ GB/s}$$
$$BW_{DH} = 3,6 \text{ GB/s}$$

$$\begin{aligned} T(HD) &= transfer\_size \times \text{ conversión a GB} \times BW_{HD} \\ &= 80 \text{ KB} \times \frac{1 \text{ GB}}{10^6 \text{ KB}} \times 4 \text{ GB/s} = 0,00032 \text{ s} \end{aligned}$$

$$\begin{aligned} T(DH) &= transfer\_size \times \text{ conversión a GB} \times BW_{HD} \\ &= 80 \text{ KB} \times \frac{1 \text{ GB}}{10^6 \text{ KB}} \times 3,6 \text{ GB/s} = 0,000288 \text{ s} \end{aligned}$$

$$\begin{aligned} T_{max} &= T(HD) + T_{max}(K) + T(DH) \\ &= 320 \times 10^{-6} + 4,424 \times 10^{-6} + 288 \times 10^{-6} \text{ s } = 612,424 \times 10^{-6} \text{ s} \end{aligned}$$

$$\begin{aligned} T_{sum} &= T(HD) + T_{sum}(K) + T(DH) \\ &= 320 \times 10^{-6} + 6,094 \times 10^{-6} + 288 \times 10^{-6} \text{ s } = 614,094 \times 10^{-6} \text{ s} \end{aligned}$$

En resumen, el primer paso consiste en calcular el coste de un algoritmo en un solo hilo. El coste computacional se expresa como el producto del número de instrucciones de cálculo por los ciclos que se tarda en ejecutar una de ellas. El coste de acceso a memoria es la suma de los accesos a memoria global (con una latencia grande) y a memoria caché (con una latencia mucho menor). Si existen accesos consecutivos cacheables, el modelo supone que el primero se realiza sobre memoria global y los siguientes, hasta cierta cantidad, sobre memoria caché. La suma del coste de computación y de acceso a memoria será el coste global de un hilo.

A continuación se debe calcular el coste de todos los hilos multiplicando el de uno por el número de ellos en ejecución, divididos por el número que se pueden lanzar simultáneamente.

Para obtener los costes de transferencia de datos entre CPU y GPU basta con dividir el tamaño de los datos a transferir por la velocidad del bus PCI Express.

Finalmente, el coste global del algoritmo será la suma del coste de ejecución y de copia de los datos que necesite, como parámetros o como resultado.

Como se puede observar, el modelo sigue un camino directo con una formulación de baja dificultad. En algoritmos complejos la mayor complicación consiste en extraer el coste de un solo hilo pero, una vez realizada esta tarea, derivar el coste global no supone más que dejarse guiar por las fórmulas planteadas.

6

Caso de uso 1: Reducción

En este capítulo, así como en los dos siguientes, se presentarán los algoritmos a emplear como casos de uso para validar el modelo desarrollado. Los algoritmos serán descritos a nivel teórico y mediante algunos códigos o pseudocódigos. La comparativa entre la estimación del modelo y las medidas experimentales se mostrará en el capítulo 9. Tras esta breve introducción, pasemos a la descripción del primer algoritmo.

Un algoritmo de reducción consiste en "reducir" una secuencia de entrada a un único valor mediante el uso de un operador binario. Por ejemplo, dado un *array* de valores numéricos, se podría obtener la suma de todos ellos siguiendo el esquema mostrado en la figura 6.1.

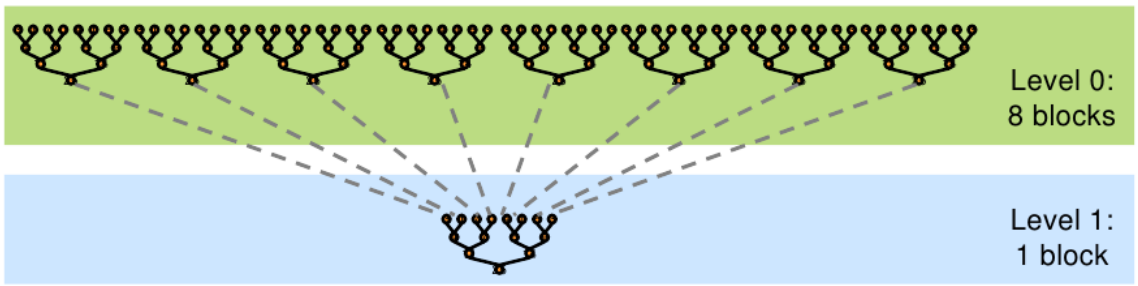

Figura 6.1: Árbol de reducción en dos pasos

La reducción es una de las directivas paralelas más tradicionales, estando presente en entornos como OpenMP y MPI. Sin embargo, para implementarla en GPU es importante considerar las diferencias entre los tipos de memoria de esta arquitectura, y así comprender las dificultades con las que nos podemos enfrentar:

- La memoria compartida entre todos los hilos de un bloque, a través de la que pueden compartir información de manera rápida. Un bloque no puede acceder a la memoria compartida de otro.
- La memoria global a todo el dispositivo, pero de acceso más lento.

La situación más sencilla es aquella donde todos los datos se pueden tratar en un solo bloque. En este caso, basta con realizar una reducción entre todos los hilos. Sin embargo, si se trabaja con grandes cantidades de datos será necesario emplear varios bloques. En este caso, cada uno procesará un pedazo del *array* de datos, pero se plantea la problemática de cómo comunicar de manera eficiente los diferentes bloques. Por ello, será necesario ejecutar el algoritmo siguiendo la estructura de árbol mostrada en la figura 6.1.

En la primera iteración, el pedazo de *array* correspondiente a cada bloque pasa a la memoria compartida de este, y el resultado se almacena en la memoria global. En la segunda iteración, la cantidad de datos se habrá reducido lo suficiente como para poder tratarla en un solo bloque. De nuevo, los datos pasan a la memoria compartida de dicho bloque y el resultado se almacena en memoria global.

Puesto que el algoritmo es el mismo en todas las iteraciones, basta con implementar un kernel e invocarlo en cada uno de los niveles, modificando el número de bloques, hilos y tamaño de memoria compartida con los que debe trabajar.

6.1. Estudio de implementaciones existentes

La implementación más conocida en CUDA es la de la biblioteca Thrust [25], basada en un estudio de optimización realizado por NVIDIA [26] e incluida en el SDK de CUDA 4.0. La interfaz que proporciona es muy sencilla, basta con indicar los límites de la colección, el valor inicial de la reducción y la operación a realizar. Las operaciones que se pueden emplear derivan de las `binary_function` de la STL [27], por lo que en cualquier momento podrían añadirse otras nuevas, siempre que sean conmutativas y asociativas.

Listing 6.1: Uso de la reducción en Thrust

```
1 int max = thrust::reduce(D.begin(), D.end(), -1,
                           thrust::maximum<int>());
3 int sum = thrust::reduce(D.begin(), D.end(), (int) 0,
                           thrust::plus<int>());
```

En esta sección se estudiarán las 7 implementaciones del estudio de optimización que han dado lugar a la versión de Thrust y la motivación de cada una de ellas:

1. Inicial
2. Supresión de divergencias dentro de un *warp*
3. Direccionamiento secuencial
4. Primera operación (suma) durante la carga

5. *Loop unrolling* basado en (3)
6. *Loop unrolling* basado en (4)
7. Múltiples sumas en cada hilo

6.1.1. Implementación inicial

La implementación básica realiza la combinación siguiendo una estructura de árbol y sin ninguna optimización aparte del uso de memoria compartida.

La principal sobrecarga en este caso viene dada del uso de la operación módulo, que es una de las operaciones más lentas. Además, la divergencia provocada por la condición dentro del bucle limita el paralelismo dentro de los *warps* (los distintos caminos se ejecutan de manera secuencial).

Listing 6.2: Reducción: Versión inicial

```
__global__ void reduce0_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;

  sdata[tid] = (i < n) ? g_idata[i] : 0;
  __syncthreads();

  // do reduction in shared mem
  for(unsigned int s=1; s < blockDim.x; s *= 2) {
      // modulo arithmetic is slow!
      if ((tid %(2*s)) == 0) {
          sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
  }

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

En la imagen 6.2 se ilustra el funcionamiento del algoritmo. Los hilos con índice par reciben los datos de las posiciones impares, duplicando el tamaño del intervalo entre ellos en cada paso.

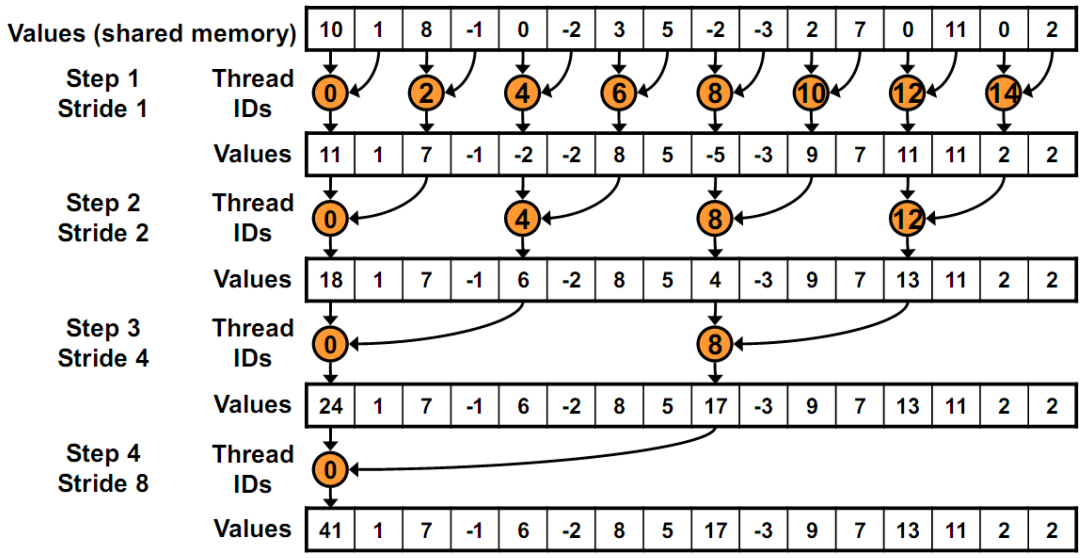

Figura 6.2: Reducción: Versión inicial

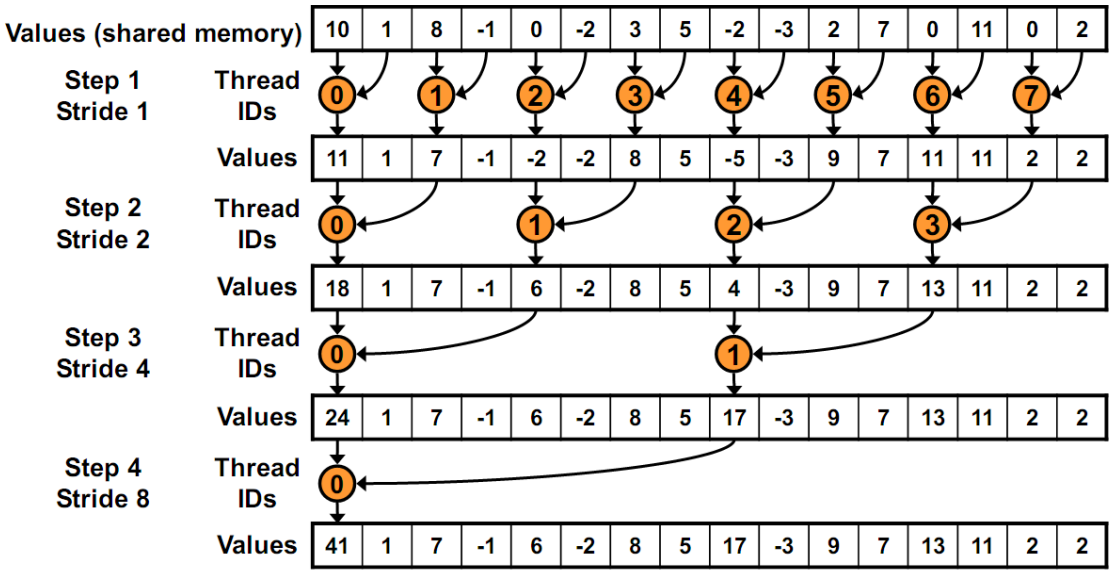

Figura 6.3: Reducción: Supresión de divergencias en un *warp*

6.1.2. Supresión de divergencias dentro de un *warp*

El primer paso para acelerar la ejecución es eliminar la costosa operación módulo, y especialmente las divergencias entre los hilos de un mismo *warp*. Con el cambio de condición, las divergencias no se producen de manera alterna en cada hilo, sino que los primeros hilos del bloque siguen un camino y los últimos hilos otro, reduciéndose a únicamente 2 ramas. Sin embargo, con este enfoque surge un nuevo problema: los

conflictos en los bancos de memoria compartida.

Listing 6.3: Reducción: Supresión de divergencias en un *warp*

```
__global__ void reduce1_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];
  // ...

  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    index = 2 * s * tid;
    if (index < blockDim.x) {
      sdata[index] += sdata[index + s];
    }
    __syncthreads();
  }

  // ...
}
```

En la imagen 6.3 se observa cómo el esquema de direccionamiento es el mismo, pero han cambiado los índices de los hilos que acceden a los datos.

6.1.3. Direccionamiento secuencial

Este caso tiene como objetivos eliminar los conflictos en bancos de memoria y las divergencias en el flujo de ejecución dentro de un mismo warp. Para ello, en lugar de trabajar con un índice calculado, en la indexación se utilizan directamente los identificadores de los hilos.

Listing 6.4: Reducción: Direccionamiento secuencial

```
__global__ void reduce2_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];
  // ...

  for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // ...
}
```

En la imagen 6.4 se observa cómo el esquema de direccionamiento ha cambiado y se accede siempre a una serie de posiciones consecutivas.

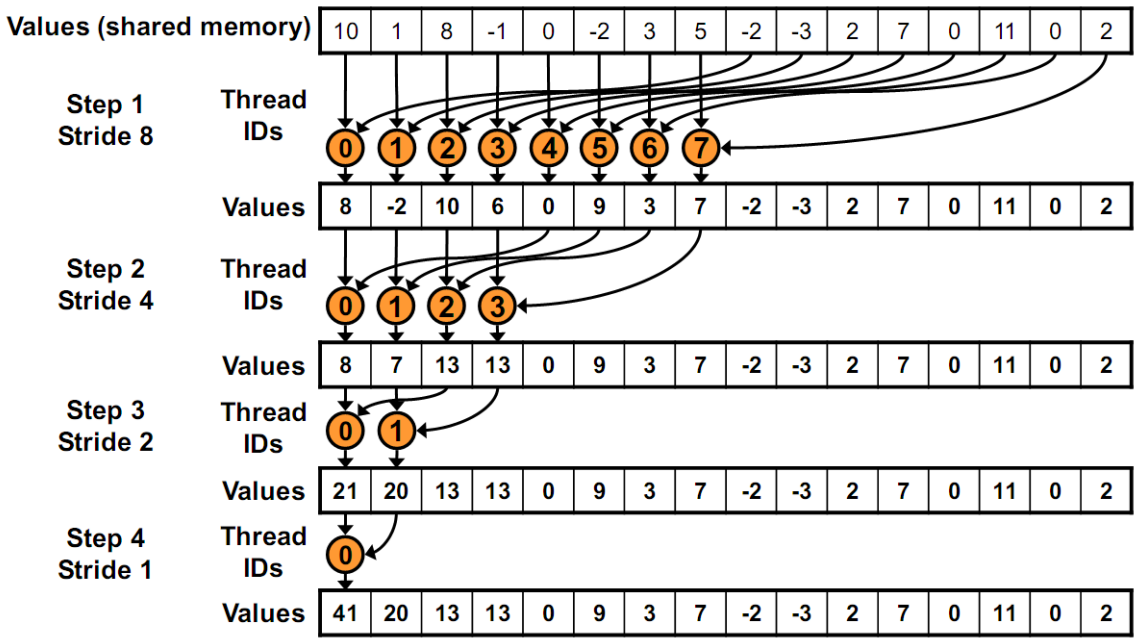

Figura 6.4: Reducción: Direccionamiento secuencial

6.1.4. Primera operación durante la carga

El problema de la solución anterior es que la mitad de hilos están ociosos (véase que se inicia el bucle en `blockDim.x/2`), por lo que en este caso la optimización no se realiza sobre el bucle sino sobre la primera reducción.

Se divide entre 2 el número de bloques con los que se está trabajando, con lo que virtualmente se trabajaría con bloques del doble de tamaño. Los hilos que perteneciesen a la primera mitad de este "bloque virtual" realizarían directamente la suma con los de la segunda mitad, y después ya se trabajaría con la primera mitad del "bloque virtual", que se correspondería con todo el bloque original.

6.1.5. *Loop unrolling* basado en 6.1.3

Para disminuir la carga asociada a la gestión del bucle, se puede realizar un loop unrolling. Cuando solo queden 32 hilos significará que solo se está trabajando con un warp, por lo que se pueden ahorra la condición y la sincronización entre hilos que se llevan a cabo en cada iteración del bucle.

Por ello, se desenrollan las 6 últimas iteraciones del bucle. Para que esta solución funcione, es necesario declarar la variable de memoria compartida como `volatile` para que el compilador no realice optimizaciones que puedan generar resultados incorrectos.

Listing 6.5: Reducción: Primera operación durante la carga

```
__global__ void reduce3_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];

  // perform first level of reduction,
  // reading from global memory, writing to shared memory

  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;

  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();

  // ...

}
```

Listing 6.6: Reducción: *Loop unrolling* basado en 6.1.3

```
__global__ void reduce4_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];
  // ...

  for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid < 32) warpReduce(sdata, tid);

  // ...

}
```

Listing 6.7: Reducción: *Loop unrolling*

```
__device__ void warpReduce(volatile double *sdata, int tid) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
```

6.1.6. *Loop unrolling* basado en 6.1.4

Siguiendo con el enfoque anterior, se ha realizado la misma comprobación del rendimiento del loop unrolling sobre el algoritmo de reducción que realizaba la primera operación durante la carga.

Listing 6.8: Reducción: *Loop unrolling* basado en 6.1.4

```
__global__ void reduce5_kernel(double *g_idata, double *g_odata
    , unsigned int n) {
  extern __shared__ double sdata[];
  // ...

  for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }

  if (tid < 32) warpReduce(sdata, tid);

  // ...
}
```

6.1.7. Múltiples sumas en cada hilo

Puesto que una sola operación por hilo supone muy poca carga computacional, asignarles más trabajo puede incrementar el rendimiento. Para ello, es posible definir un tamaño del bloque utilizando *templates* de C++. El único problema que tiene este caso es que para aprovecharlo se debe conocer en tiempo de compilación el número de hilos a utilizar.

Listing 6.9: Reducción: Múltiples sumas en cada hilo

```
template <unsigned int blockSize>
__global__ void reduce6_kernel(double *g_idata, double *g_odata, unsigned int n) {
  extern __shared__ double sdata[];
  // ...
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + threadIdx.x;
  unsigned int gridSize = blockSize*2*gridDim.x;

  sdata[tid] = 0;
  while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();

  // ...
}

double reduce6(double *in_data, double *out_data, unsigned int n, unsigned int numblocks, unsigned int numthreads, double cpu_result) {
  // ...
  // case example with 512 threads per block
    reduce6_kernel<512><<<numblocks, numthreads, numthreads*sizeof(double)>>>(gpu_in_data, gpu_out_data, n); break;
  }
  // ...
}
```

6.2. Implementación final

De la última aproximación derivó el algoritmo incluido en la biblioteca Thrust. En este caso de uso se evaluarán las dos aproximaciones incluidas en ella, cuyo código es idéntico exceptuando que en uno de los casos se trabaja siempre sobre memoria global y en otro sobre la compartida.

En el *listing* 6.10 se muestra el pseudocódigo asociado a ellas. La única diferencia una vez llegado el momento de la implementación es la ubicación de la variable `datos`.

Listing 6.10: Algoritmo de reducción en Thrust

```
1 for i = tam_bloque + hilo to tam_datos
    datos[hilo] := datos[hilo] + datos[i]
3   i := i + tam_bloque

5 activos := tam_bloque / 2
  while activos >= 1
7   if hilo < activos
      datos[hilo] := datos[hilo] + datos[hilo + activos]
```

Como se citó en el apartado anterior, la reducción es una operación con poca carga de computación y no sería eficiente que cada hilo tratase un solo dato. Por tanto, en la primera parte del algoritmo cada hilo realiza la suma de aproximadamente *tam_datos*/*tam_bloque* elementos.

En la segunda parte se aplica el árbol de reducción usando el número de hilos definido en la variable `activos`. En cada iteración, dichos hilos suman el elemento que contienen además de otro en un hilo ya inactivo, y se va dividiendo el número de activos por la mitad. Esta acción provoca la mínima divergencia posible con únicamente dos ramas, y al aplicarse un desplazamiento igual para todos los hilos el acceso a memoria es coalescente.

7

Caso de uso 2: Producto matriz-vector

Una de las operaciones más básicas en el campo de la computación numérica es el producto matriz-vector, que se corresponde con la operación xGEMV del nivel 2 de BLAS. A partir de una matriz $A \in \mathbb{R}^{n \times m}$ y un vector $v \in \mathbb{R}^m$, se obtiene como resultado otro vector $y \in \mathbb{R}^n$. Esta operación será empleada como segundo caso de uso para la validación del modelo teórico desarrollado.

7.1. Una fila por hilo

Una posible implementación consiste en que cada hilo procese una fila de la matriz A, multiplicando sus elementos por todos los del vector v. En la figura 7.1 se muestra, mediante un código de colores, qué elementos de la matriz y los vectores están relacionados en este caso.

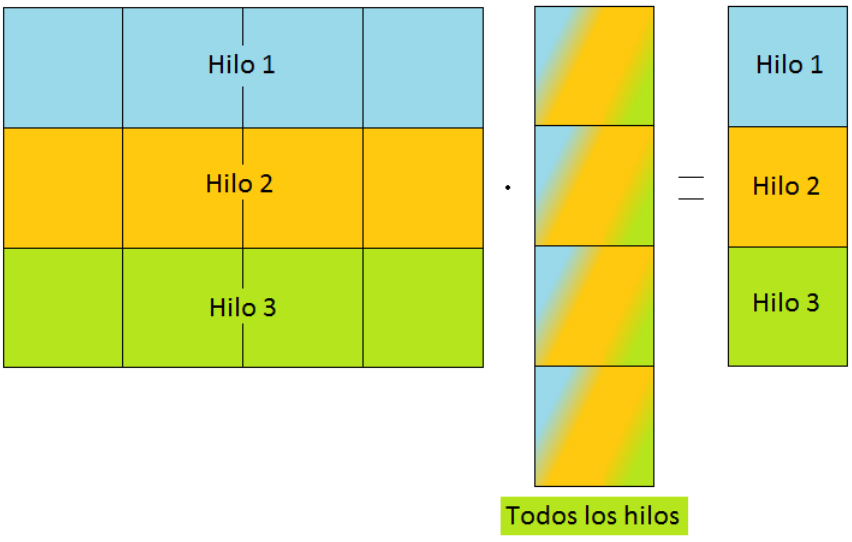

Figura 7.1: Producto matriz-vector con cada hilo procesando una fila

En cuanto al código de la implementación en CUDA, en el *listing* 7.1 se puede observar cómo, para reducir el número de escrituras sobre memoria global, la suma acumulada se almacena temporalmente en una variable local *result*. Cuando se termina de procesar toda la fila, el resultado se copia a memoria global. De esta forma, cada hilo realiza $2 \times m$ lecturas (sin contar los efectos de la caché) y una escritura sobre memoria global.

Listing 7.1: Producto matriz-vector en CUDA por filas

```
__global__ void matvecKernel(float *y, float *A, float *v,
                             int n, int m) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  float result = 0.0f;

  if (i < n) {
    for (int j = 0; j < m; j++) {
      result += A[i*m+j] * v[j];
    }
    y[i] = result;
  }
}
```

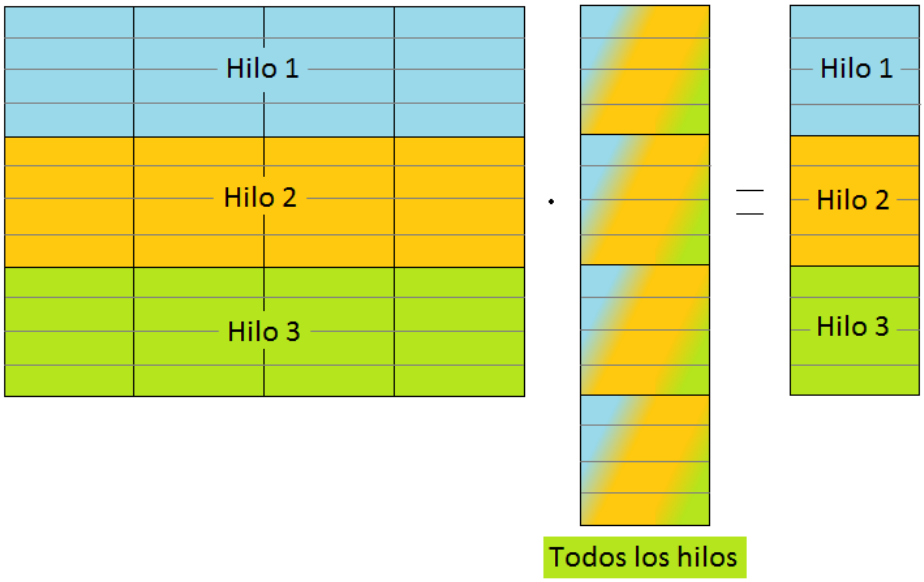

Figura 7.2: Producto matriz-vector con cada hilo procesando un bloque de filas

7.2. Un bloque de filas por hilo

Un segundo enfoque asigna una mayor carga de trabajo a cada uno de los hilos de forma que, en lugar de trabajar sobre una sola fila, operan sobre varias de ellas. La acumulación se realiza de la misma manera, pero a la hora de asignar el resultado sobre el vector es importante tener en cuenta el tamaño de los bloques y el desplazamiento dentro de ellos.

Listing 7.2: Producto matriz-vector en CUDA por bloques de filas

```
__global__ void matvecKernel(float *y, float *a, float *v,
                             int n, int m) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  float result;

  if (i <= ceil((float) n/BLOCK_SIZE))  {
    for (int k = 0; k < BLOCK_SIZE && i*BLOCK_SIZE+k < n; k++) {
      result = .0f;
      for (int j = 0; j < m; j++) {
        result += a[(i+k)*m+j] * v[j];
      }
      y[i*BLOCK_SIZE+k] = result;
    }
  }
}
```

Listing 7.3: Producto matriz-vector en CUDA por columnas

```
__global__ void matvecKernel(float *y, float *a, float *v,
                             int n, int m) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;

  if (i < m) {
    for (int j = 0; j < n; j++) {
      atomicAdd( &y[j], a[j*m+i] * v[i] );
    }
  }
}
```

7.3. Una columna por hilo

Otra posible implementación orientada a aumentar el número de accesos coalescentes es la que asigna una columna a cada hilo. De esta forma se disminuye el coste de los accesos a memoria.

Por contra, puesto que todos los hilos trabajan sobre un mismo dato del vector resultante y, es necesario que se acceda a él a través de una sección crítica, empleando operaciones atómicas para actualizar de manera secuencial su valor.

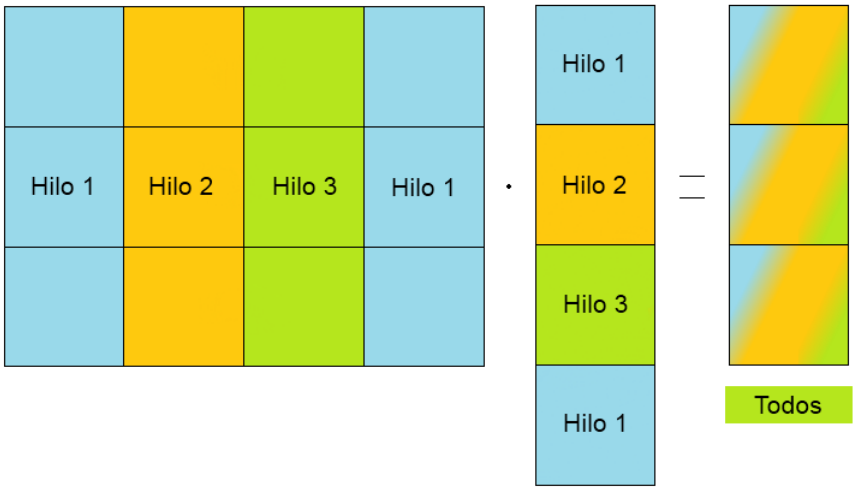

Figura 7.3: Producto matriz-vector con cada hilo procesando una columna

8

Caso de uso 3: Descomposición de Cholesky

Este capítulo describe el último caso de uso, que resulta ser también el más complejo de los tres, para la validación del modelo teórico. El motivo de su complejidad es el hecho de que engloba código verdaderamente heterogéneo, con transferencias de datos entre *host* y *device*, invocación de diferentes *kernels* CUDA y uso de *streams*.

Este método permite descomponer una matriz A simétrica y definida positiva en la forma $A = RR^T$. R será una matriz triangular inferior cuyas entradas de la diagonal principal sean todas positivas, y para ser definida positiva A deberá cumplir las siguientes propiedades [28]:

- Ser una matriz real $N \times N$
- Ser simétrica
- Cumplir $x^T Ax > 0$ para todo $x \in \mathbb{R}^n$ distinto de cero

En computación científica es habitual tener que resolver sistemas de la forma $Ax = b$ con n ecuaciones y n incógnitas, siendo x la solución. Si A es definida positiva y se conoce su descomposición R, se puede escribir el sistema como $R^T y = b$, siendo $y = Rx$.

Puesto que no se conoce x, tampoco se conoce y, pero se sabe que esta última satisface la igualdad $R^T y = b$. Al ser R^T triangular inferior, se puede obtener y por sustitución progresiva, y a continuación resolver $Rx = y$ por sustitución regresiva.

En este trabajo se intentará implementar esta descomposición para el caso denso por bloques estructurado pero, puesto que no existe apenas trabajo previo al respecto en GPU, se tomarán como referencia las implementaciones del caso denso básico y se irán realizando iteraciones hasta llegar a esa última versión del algoritmo.

El primer paso consistirá en partir del caso denso implementado en la biblioteca MAGMA [10] y que divide el problema en *tiles* (descritas más adelante). A partir de él se derivará un algoritmo orientado a bloques que después permitirá realizar optimizaciones para el caso denso estructurado, eliminando las operaciones que

impliquen bloques compuestos totalmente por ceros. Finalmente, se realizará la implementación para el caso disperso y se intentará optimizar en caso de que la matriz tenga una estructura determinada.

8.1. Caso denso secuencial

En los *listings* 8.1 y 8.2 se puede observar el pseudocódigo para realizar esta descomposición de manera secuencial básica y por bloques, respectivamente.

Listing 8.1: Algoritmo de la descomposición de Cholesky

```
R := A
for k = 1 to n do
  r_kk := sqrt(r_kk)
  for i = k + 1 to n do
    r_ik := r_ik / r_kk
  for j = k + 1 to n do
    for i = j to n do
      r_ij := r_ij - r_ik r_jk
```

Listing 8.2: Algoritmo de la descomposición de Cholesky por bloques

```
R := A
for k = 1 to n do
  R_kk := cholesky(R_kk)
  for i = k + 1 to n do
    R_ik := R_ik R_kk^-1
  for j = k + 1 to n do
    for i = j to n do
      R_ij := R_ij - R_ik R_jk
```

Se ha realizado una implementación en MATLAB para ambos casos, que por su similitud con el pseudocódigo no se referenciarán en esta documentación. Por otra parte, la versión por bloques se ha implementado utilizando la biblioteca LAPACK (ver figura 8.1).

Las operaciones utilizadas son:

- xSYRK: Actualiza un bloque de la diagonal tomando la factorización de los que se encuentran a su izquierda. La expresión equivalente es $B = B - A * A'$.
- xPOTRF: Realiza la factorización de Cholesky de un bloque de la diagonal.

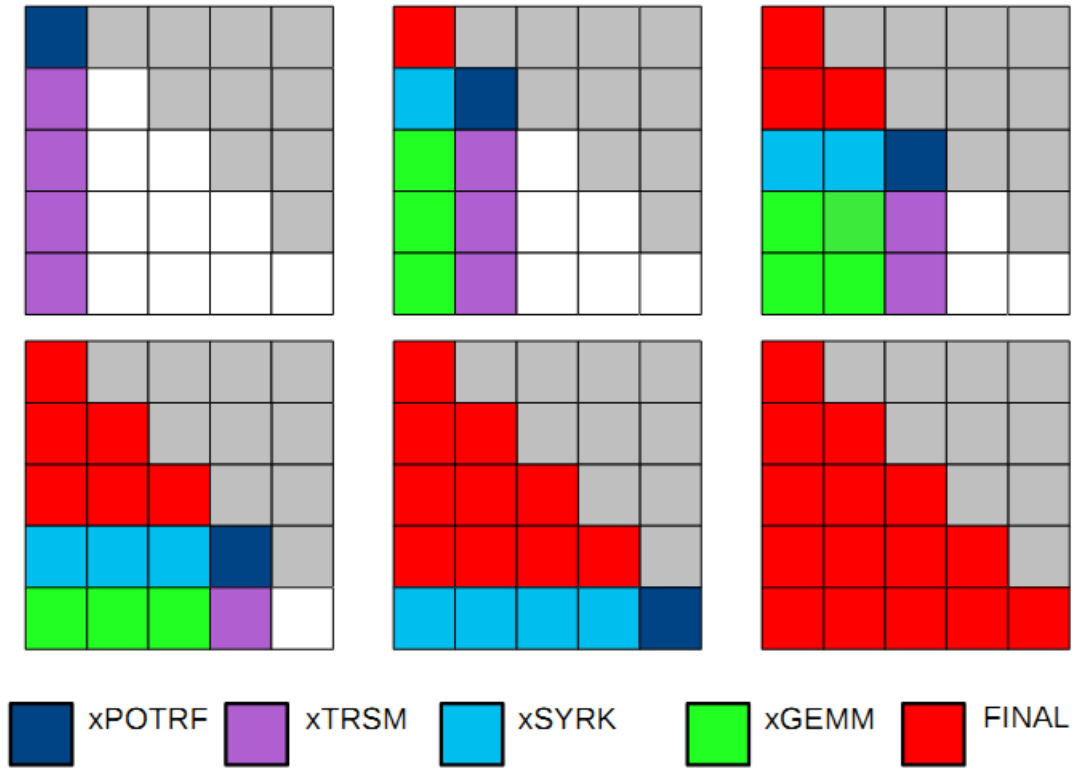

Figura 8.1: Factorización de Cholesky empleando bloques

- xGEMM: Actualiza los bloques debajo de una diagonal tomando las factorizaciones a su izquierda, mediante una multiplicación de matrices. Equivale a la expresión $D = D - C * A'$.

- xTRSM: Actualiza los bloques debajo de una diagonal tomando la factorización de esta, mediante la resolución de un sistema triangular. La expresión a la que equivale es $D = D \backslash B$.

8.2. Caso denso paralelo

Una de las implementaciones para este caso se incluye dentro de la biblioteca MAGMA [10], desarrollada por equipos de las universidades de Berkeley y Tennesse, entre otras, como un *port* de LAPACK a arquitecturas híbridas. El modelo de resolución se basa en *tiles*, agrupaciones rectangulares o cuadradas de elementos. Esta diferenciación no se debe a que el tamaño de la matriz no sea múltiplo del tamaño del bloque, sino a sobre qué elementos se puede aplicar una operación común.

El modelo de resolución por bloques es muy similar al de *tiles* pero empleando

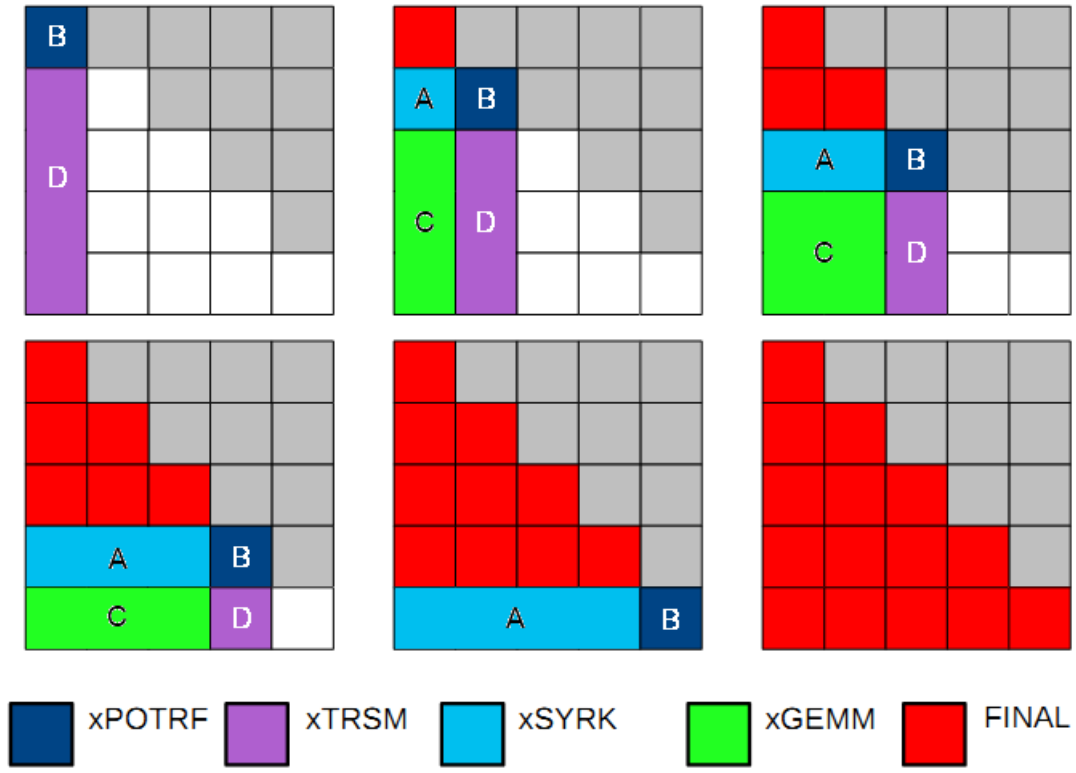

Figura 8.2: Factorización de Cholesky empleando *tiles*

divisiones de menor tamaño. Para un caso denso sin ceros esto no supone una ventaja clara pues se limita a tratar subproblemas más pequeños y esto puede disminuir el nivel de paralelismo en la GPU, sin embargo, en casos dispersos permite eliminar ciertas operaciones sobre los bloques que solo contienen ceros.

8.2.1. Implementación de MAGMA empleando *tiles*

Puesto que es una biblioteca orientada a arquitecturas heterogéneas, el modelado de los algoritmos en MAGMA busca que la división y la planificación permita la ejecución asíncrona y el balanceo de carga entre los componentes. Para ocultar los efectos de las pequeñas tareas no paralelizables y planificadas en CPU, se solapa su ejecución con las de otras más paralelizables, que son las operaciones de nivel 3 de BLAS [29] planificadas en GPU. Para estas últimas se hace uso de la biblioteca CUBLAS, y son iguales a las empleadas en la versión secuencial.

Puesto que el código de MAGMA se ha tomado como base para posteriores iteraciones, en este apartado se describe en profundidad su funcionamiento, de forma que en los siguientes solo sea necesario mostrar las diferencias para ofrecer una total comprensión de todas las versiones.

El algoritmo está enfocado hacia la factorización en la triangular superior o inferior y varía ligeramente entre ellas en el direccionamiento y en la transferencia de un dato en un punto concreto. En los *listings* 8.3 y 8.4 se pueden observar el pseudocódigo y el código de la factorización sobre la triangular inferior. Como se puede observar, no aparecen bucles como en la versión por bloques, porque todos los bloques sobre los que se aplica una misma operación son agrupados.

Listing 8.3: Algoritmo de la descomposición de Cholesky en MAGMA

```
for k = 1 to n do
  copy_matrix(R_kk , A_kk , hostToDevice)
  R_kk := R_kk − R'_k0 R_k0
  copy_async(A_k0 , R_k0 , deviceToHost , stream_1)
  i := k + 1
  R_ik = R'_k0 R_i0
  synchronize(stream_1)
  A_kk := cholesky(A_kk)
  copy_async(R_kk , A_kk , hostToDevice)
  R_ik := R_ik R_kk^-1
```

En la línea 2 del código fuente se calcula el ancho y alto máximos de la *tile* con la que trabajar, puesto que si el lado de la matriz no es potencia de dos el último bloque será más pequeño. Una vez la *tile* del tamaño necesario se ha enviado a la GPU, en la línea 7 se ejecuta la operación xSYRK (en este caso para dobles). De acuerdo a la figura 8.2, `dA(j, j)` se corresponde con B y `dA(0, j)` con A, es decir, la expresión resultante sería `dA(j, j) = dA(j, j) - dA(0, j)' * dA(0, j)`, donde j es la columna inicial de la *tile* que se encuentra en la diagonal principal.

Por simplicidad, la interfaz de CUBLAS es igual que la de BLAS añadiendo únicamente un manejador especial como primer parámetro de las invocaciones. Debe tenerse en cuenta que el direccionamiento es en base a las columnas, por lo que las coordenadas aparecen invertidas (más información en el apéndice A.1 sobre CUBLAS).

Al realizar la copia de datos de GPU a CPU de manera asíncrona, se puede solapar con la operación xGEMM, que modifica los datos de la *tile* D a partir del producto $A' * C$.

La factorización en la diagonal principal se realiza en CPU. En la línea 26 se puede observar cómo se está invocando una función de LAPACK. De nuevo, este bloque se copia de manera asíncrona pero esta vez hacia la CPU. Mientras tanto, se ejecuta la instrucción xTRSM para actualizar la *tile* D.

Listing 8.4: Factorización de Cholesky en MAGMA

```
for (j=0; j<n; j += nb) {
  jb = min(nb, (n-j));
  cublasSetMatrix(jb, (n-j), sizeof(double),
                  A(j, j), lda,
                  dA(j, j), ldda);

  stat = cublasDsyrk(handle, FillUpper, Transposed, jb, j,
                     &mdone, dA(0, j), ldda,
                     &done,  dA(j, j), ldda);

  cudaMemcpy2DAsync( A(0, j), lda  * sizeof(double),
                    dA(0, j), ldda * sizeof(double),
                    sizeof(double) * (j+jb), jb,
                    cudaMemcpyDeviceToHost, stream[1]);

  if ( (j+jb) < n) {
    stat = cublasDgemm(handle, Transposed, NoTransposed,
                       jb, (n-j-jb), j,
                       &mzone, dA(0, j   ), ldda,
                               dA(0, j+jb), ldda,
                       &zone,  dA(j, j+jb), ldda);
  }

  cudaStreamSynchronize(stream[1]);

  dpotrf_(FillUpperStr, &jb, A(j, j), &lda, &info);
  if (info != 0) {
    info = info + j;
    break;
  }

  cudaMemcpy2DAsync(dA(j, j), ldda * sizeof(double),
                     A(j, j), lda  * sizeof(double),
                    sizeof(double)*jb, jb,
                    cudaMemcpyHostToDevice,stream[0]);

  if ( (j+jb) < n )
    stat = cublasDtrsm(handle, SideLeft, FillUpper,
                       Transposed, DiagNonUnit,
                       jb, (n-j-jb),
                       &zone, dA(j, j   ), ldda,
                              dA(j, j+jb), ldda);
}
```

La factorización con el almacenamiento en la parte inferior de la matriz sigue el mismo proceso, pero invirtiendo las coordenadas y los tamaños de las *tiles*.

8.2.2. Implementación propia empleando bloques

Este caso se han sustituido las operaciones sobre una *tile* por una iteración por varios bloques hasta cubrir los mismos datos que ella. Como se ha comentado en la introducción de la sección, para un caso denso sin ceros no supone una ventaja, pero sirve como base para los casos denso estructurado y disperso, pues permitiría eliminar ciertas operaciones sobre los bloques que solo contienen ceros.

Listing 8.5: Algoritmo de la descomposición de Cholesky en CUDA por bloques

```
1  for k = 1 to n do
     copy_matrix(R_kk, A_kk, hostToDevice)
3    for i = 1 to j do
       R_kk := R_kk − R'_ki R_ki
5    copy_async(A_k0, R_k0, deviceToHost, stream_1)

7    i := k + 1
     for i = 1 to n−k do
9      for j = 1 to k do
         R_{k+i,k} = R'_kj R_{k+i,j}
11
     synchronize(stream_1)
13   A_kk := cholesky(A_kk)
     copy_async(R_kk, A_kk, hostToDevice)
15
     for i = k + 1 to n do
17     R_ik := R_ik R_kk^{-1}
```

El pseudocódigo que resume el funcionamiento de esta implementación se muestra en el *listing* 8.5. En cuanto al código fuente, la operación xSYRK se traduce en que, en lugar de posicionarse en `dA(0, j)` y realizar una sola operación, se itera por toda la fila, desde `dA(0, j)` hasta el contiguo a `dA(j, j)`.

Listing 8.6: Uso de xSYRK por bloques

```
1  for (i = 0; i < j; i += nb) {
       cublasDsyrk(handle, FillUpper, Transposed,
3                  nb, nb,
                   &mdone, dA(i, j), ldda,
5                  &done,  dA(j, j), ldda);
   }
```

Para la operación xGEMM, por cada bloque de D se itera de izquierda a derecha por todos los de C que se encuentran en la misma fila.

Finalmente, en xTRSM se itera desde `dA(j, j+jb)`, primer bloque de D, y se dividen todos ellos por `dA(j, j)`.

Listing 8.7: Uso de xGEMM por bloques

```
for (i = jb; i < n-j; i += nb) {
  for (k = 0; k < j-1; k += nb) {
    cublasDgemm(handle, Transposed, NoTransposed,
                nb, jb, nb,
                &mzone, dA(k, j  ), ldda,
                        dA(k, j+i), ldda,
                &zone,  dA(j, j+i), ldda);
  }
}
```

Listing 8.8: Uso de xTRSM por bloques

```
for (i = j+jb; i < n; i += nb) {
  ib = min(jb, (n-i));
  cublasDtrsm(handle, SideLeft, FillUpper,
              Transposed, DiagNonUnit,
              jb, ib,
              &zone, dA(j, j), ldda,
                     dA(j, i), ldda);
}
```

8.3. Caso denso estructurado paralelo

Una posible optimización para reducir el número de operaciones realizadas es aprovechar que la descomposición de Cholesky de una matriz conserva su perfil. Para ello, antes de la factorización se realiza un preprocesado que comprueba cuál es el primer bloque de una fila o columna, según la factorización sea superior o inferior, con algún elemento distinto de cero.

Supongamos el caso de una matriz 5x5 donde las tres primeras filas se procesan desde el primer bloque, la fila 4 a partir del tercero y en la fila 5 solo el último. En este caso, se emplearía un vector descriptor de la estructura como el siguiente:

$$blocks = \begin{pmatrix} 0 & 0 & 0 & 2 & 4 \end{pmatrix}$$

La manera de acotar las posiciones a partir de las que se realiza alguna operación se puede observar en el pseudocódigo 8.9.

Listing 8.9: Algoritmo de la descomposición de Cholesky en el caso denso estructurado

```
for k = 1 to n do
  copy_matrix(R_kk , A_kk , hostToDevice)
  for i = 1 to j do
    if i >= blocks_k
      R_kk := R_kk − R'_ki R_ki
  copy_async(A_k0 , R_k0 , deviceToHost , stream_1)

  i := k + 1
  for i = 1 to n−k do
    for j = 1 to k do
      if j >= blocks_k and j >= blocks_k+i
        R_k+i,k = R'_kj R_k+i,j

  synchronize(stream_1)
  A_kk := cholesky(A_kk)
  copy_async(R_kk , A_kk , hostToDevice)

  for i = k + 1 to n do
    if k >= blocks_i
      R_ik := R_ik R_kk^-1
```

Ya en el código fuente y sabiendo que se está trabajando con los bloques que comienzan en la posición j, se obtiene el índice del primero con contenido, almacenado en `blocks[j/nb]`. Si el bloque con el que se está trabajando es posterior a aquel, entonces se ejecuta la operación. De esta forma, en la penúltima fila de la matriz se evitaría operar sobre dos bloques y en la última sobre tres.

Listing 8.10: xSYRK en el caso denso estructurado

```
block_index = j/nb;
for (i = 0; i < j; i += nb) {
  if ((i/nb) >= blocks[block_index]) {
    cublasDsyrk( ... );
  }
}
```

Para xGEMM, puesto que se trata de un producto de matrices se debe comprobar que **ambas** tengan contenido, pues en cuanto una esté compuesta totalmente de ceros se anulará dicho producto.

Listing 8.11: xGEMM en el caso denso estructurado

```
for (i = jb; i < n-j; i += nb) {
  for (k = 0; k < j-1; k += nb) {
    block_index = j/nb;
    block_index_2 = (j+i)/nb;
    if ((k/nb) >= blocks[block_index] &&
        (k/nb) >= blocks[block_index_2]) {
      cublasDgemm( ... );
    }
  }
}
```

En el caso de xTRSM vuelve a ser necesaria solo la comprobación sobre un bloque, de la misma forma que con xSYRK pero, puesto que D tiene una orientación distinta de A, se comprueba la posición sobre la otra coordenada.

Listing 8.12: xTRSM en el caso denso estructurado

```
for (i = j+jb; i < n; i += nb) {
  block_index = i/nb;
  ib = min(jb, (n-i));
  if ((j/nb) >= blocks[block_index]) {
    cublasDtrsm(handle, SideLeft, FillUpper,
                Transposed, DiagNonUnit,
                jb, ib,
                &zone, dA(j, j), ldda,
                       dA(j, i), ldda);
  }
}
```

8.4. Caso denso estructurado paralelo sin *kernels* de CUBLAS

El código de la versión 2.0 de CUBLAS no está disponible públicamente, lo que impide realizar una estimación detallada del coste del algoritmo. Por ello, se ha realizado una implementación propia de los *kernels* para el caso a evaluar (no orientada como biblioteca por la complejidad que entraña), y que será a partir de la cual se realice dicha estimación.

La operación xSYRK se expresa como $B = B - A \times A'$ y xGEMM como $D = D - C \times A'$. Por tanto, es necesario trasponer la matriz A, realizar un producto de matrices y a continuación una resta. El *kernel* que realiza la trasposición se ha extraído del SDK de CUDA y viene acompañado de un estudio de rendimiento [30].

Es un algoritmo orientado a bloques, de manera que se puedan mantener los datos en memoria *shared* para aumentar el rendimiento (véase el *listing* 8.13).

Listing 8.13: Trasposición de una matriz

```
__global__ void transpose( double *odata, double *idata, int ↩
    width, int height )
{
  __shared__ double tile[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }

  __syncthreads();

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i↩
        ];
  }
}
```

En cuanto al código del producto de matrices extraído del SDK, también es un algoritmo por bloques que se aprovecha del hecho de que, en un producto $A \times B$, las primeras posiciones de las filas de A siempre operen con las primeras posiciones de las columnas de B. En el *listing* 8.14 se muestra el código de este producto.

Finalmente, la resta de matrices se ha implementado con un *kernel* idéntico al de la suma de vectores ilustrada en la sección 5.1.2.

Todos estos componentes se han unido en los núcleos xSYRK y xGEMM. El primero se muestra en el *listing* 8.15, aunque el código de xGEMM se puede deducir fácilmente a partir de él. Para invocar ambas funciones es necesario copiar las submatrices a un nuevo espacio de memoria dentro de la GPU mediante una copia de tipo `cudaMemcpyDeviceToDevice`, y restaurar el resultado a su emplazamiento original una vez realizado el cálculo.

Listing 8.14: Producto de matrices por bloques

```
template <int BLOCK_SIZE> __global__ void
matrixMul( double* C, double* A, double* B, int wA, int wB )
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;

    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix computed by the thread
    double Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from device memory to shared memory
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];

        __syncthreads();

        // Multiply the two matrices together
    # pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);

        __syncthreads();
```

```
    }
44
    // Write the block sub-matrix to device memory
46  int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
48 }
```

Listing 8.15: Implementación propia de DSYRK y DGEMM

```
void dsyrk ( double* A, double* B, int wA, int hA, int wB, int ↩
    hB )
2 {
  // transpose A
4 dim3 blocks1((int) ceil((float) wA/TILE_DIM), (int) ceil((↩
     float) hA/TILE_DIM));
  dim3 threads1(TILE_DIM, BLOCK_ROWS);
6 double* A_tr; cudaMalloc( &A_tr, wA * hA * sizeof(double) );
  transpose<<< blocks1, threads1 >>>( A_tr, A, wA, hA );
8
  // aux = A * A'
10 double* aux; cudaMalloc( &aux, hA * hA * sizeof(double) );
  dim3 threads2(32, 32);
12 dim3 blocks2((int) ceil((float) hA/threads2.x), (int) ceil((↩
     float) hA/threads2.y));
  matrixMul<32><<< blocks2, threads2 >>>( aux, A, A_tr, wA, hA ↩
     );
14
  // B = B - aux
16 dim3 threads3(192);
  dim3 blocks3((int) ceil((float) (wB * hB) / threads3.x));
18 matrixSub<<< blocks3, threads3 >>>( B, aux, wB, hB );

20 cudaFree(A_tr); cudaFree(aux);
}
```

Para la implementación del núcleo xTRSM se ha tomado como referencia el código de BLAS [29]. Aunque dicha biblioteca ofrece mucha versatilidad en cuanto a opciones, por sencillez se ha implementado únicamente el caso aplicable a la estructura de matrices que será probada en capítulos posteriores. Este caso es $X \times A = \alpha \cdot B$, siendo A una matriz triangular inferior cuya diagonal no es unitaria.

El algoritmo de resolución se compone de varias iteraciones con dependencias entre ellas, por lo que cada una se corresponde con una invocación del *kernel* mostrado en el *listing* 8.16. En un primer paso de cada k-ésima iteración, se calculan todos los elementos de una columna determinada en base a un elemento ubicado en

la diagonal de A. Utilizando estos valores, cada hilo recalcula el valor del elemento correspondiente a sus coordenadas (x, y) en el bloque de hilos.

Listing 8.16: Implementación propia de DTRSM

```
__global__ void dtrsm_kernel ( double *A, double *B, int width,
    int height, int k )
{
        int x = blockIdx.x * blockDim.x + threadIdx.x;
        int y = blockIdx.y * blockDim.y + threadIdx.y;

        double temp = 1 / A[k * width + k];

        if (y < height) {
                // Loop 1.1
                B[y * width + k] = temp * B[y * width + k];

                // Loop 1.2
                if (x > k && x < width) {
                        temp = A[x * width + k];
                        B[y * width + x] -= temp * B[y * width
                            + k];
                }
        }
}
```

9

Evaluación teórica y pruebas experimentales

En este capítulo se evaluará el modelo teórico desarrollado. Para ello, se realizarán pruebas experimentales del coste de los algoritmos descritos en los capítulos 6, 7 y 8, así como una estimación empleando el modelo teórico. Finalmente, se compararán los resultados para concluir cuáles son los casos a los que mejor se adapta y en cuáles es posible una mejora sustancial.

Los computadores heterogéneos que se emplearán en las pruebas experimentales, ya utilizados para las tomas de tiempos en capítulos anteriores (aunque no se explicitasen los resultados obtenidos en todos ellos), son los siguientes:

eleanorrigby: Dispone de dos procesadores Intel Xeon X5680 a 3,33 GHz, cada uno de los cuales es un *hexa-core* que, haciendo uso de la tecnología *hyper-threading*, proporciona 24 procesadores virtuales. En cuanto a las GPU, dispone de dos Nvidia Tesla C2070 con 448 núcleos CUDA y 6 GB de memoria GDDR5.

elbulli: Dispone de un procesador Intel Xeon X5675 a 3,07 GHz, un *hexa-core* que, haciendo uso de la tecnología *hyper-threading*, proporciona 12 procesadores virtuales. En cuanto a las GPU, dispone de dos Nvidia Tesla C2075 con 448 núcleos CUDA y 6 GB de memoria GDDR5.

harajuku: Dispone de un procesador Intel Core 2 Duo E6600 con dos núcleos a 2,4 GHz, y de una GPU Nvidia GeForce GTX 560Ti con 384 núcleos CUDA y 1 GB de memoria GDDR5.

micromachin: Dispone de un procesador Intel Xeon E5530 con 4 núcleos funcionando a 2,4 GHz, y de 4 GPU Nvidia Tesla C1060 con 240 núcleos CUDA y 4 GB de memoria GDDR3 cada una.

Las máquinas `eleanorrigby` y `elbulli` se han empleado de manera indistinta en varias pruebas, pues sus especificaciones son iguales exceptuando el consumo energético. Son las principales representantes de la arquitectura Fermi. `harajuku` ha

servido como respaldo a la hora de comprobar que algunas mediciones se aplicaban igualmente a tarjetas de distinto perfil dentro de la misma arquitectura.

Por otro lado, `micromachin` se ha empleado en las mediciones realizadas durante la fase de diseño del modelo para la arquitectura Tesla. Puesto que pertenece a una arquitectura a extinguir, en la comparativa del modelo con las pruebas experimentales se ha dado preferencia a la arquitectura Fermi frente a esta.

9.1. Caso de uso 1: Reducción

A continuación se evaluará el coste de la implementación de este algoritmo descrita en la sección 6.2. La evaluación se realizará desde dos enfoques: el primero teniendo en cuenta solo el coste del *kernel*, y la segunda, añadiendo el tiempo necesario para copiar los elementos del *host* al *device*.

En cuanto a la configuración, cada bloque está compuesto de 512 hilos y los tamaños de problema varían entre 500.000 y 100.000.000 elementos.

Evaluación teórica

El algoritmo tal como se describe en la sección 6.2 se puede descomponer en dos partes principales: la primera de ellas realiza la acumulación inicial para asociar más carga computacional a los hilos, y la segunda ejecuta el proceso de reducción en sí mismo. Por tanto, se calculará el coste de cada una de las partes y a continuación se sumarán ambos.

En la acumulación inicial, cada hilo procesa $\frac{n}{512}$ elementos. En cada iteración se realiza una suma, para la que son necesarios 3 accesos a memoria (2 de lectura y 1 de escritura). Finalmente, se realiza una sincronización entre todos los hilos del bloque para garantizar que todos terminen esta parte del trabajo simultánamente.

Si tomamos como ejemplo concreto para el cálculo el caso con 100.000.000 elementos de doble precisión empleando en todo momento memoria global, obtenemos los siguientes valores para los parámetros del modelo:

- $comp\_insts = \left\lceil \frac{100000000}{512} \right\rceil + 1 = 195314$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 48 \times 195314 = 9375072$ ciclos
- $mem\_insts = 3 \times \left\lceil \frac{100000000}{512} \right\rceil = 585938$ instrucciones
- $data\_size = 8$ bytes
- $N_{max} = \frac{128}{8} = 16$ datos/petición
- $N_{min} = \frac{32}{8} = 4$ datos/petición

- $cache\_factor = \frac{16+4}{2} = 10$ datos/petición
- $latency_{gmem} = 600$ ciclos
- $latency_{smem} = 4$ ciclos
- $latency_{cache} = 4$ ciclos
-

$$\begin{aligned} C_{mem} &= latency_{gmem} \times \frac{mem\_insts}{cache\_factor} + latency_{cache} \times \frac{mem\_insts \times (cache\_factor - 1)}{cache\_factor} \\ &= 600 \times \frac{585938}{10} + 4 \times \frac{585938 \times 9}{10} = 37265657 \text{ ciclos} \end{aligned}$$

- $C_{max} = max(9375072, 37265657) = 37265657$ ciclos
- $C_{sum} = 9375072 + 37265657 = 46640729$ ciclos
- $N_B(K) = 1$ bloque
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{512}{32} = 16$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
-

$$\begin{aligned} C_{sum}(K) &= N_B(K) \times N_w(K) \times N_t(K) \times C_{sum} \times \frac{1}{N_C \times D} \\ &= 1 \times 16 \times 32 \times 46640729 \times \frac{1}{32 \times 4} = 186562916 \text{ ciclos} \end{aligned}$$

- $T(K) = \frac{186562916}{1,15 \times 10^9} = 0,162$ s

A continuación se calcula el coste de efectuar la reducción final.

- $comp\_insts = \log_2 512 = 9$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 48 \times 9 = 432$ ciclos

- $mem\_insts = 3 \times \log_2 512 = 27$ instrucciones
- $C_{mem} = 600 \times \frac{27}{10} + 4 \times \times \frac{27 \times 9}{10} = 1718$ ciclos
- $C_{max} = max(432, 1718) = 1718$ ciclos
- $C_{sum} = 432 + 1718 = 2150$ ciclos
- $N_B(K) = 1$ bloque
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{512}{32} = 16$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 1 \times 16 \times 32 \times 2150 \times \frac{1}{32 \times 4} = 8600$ ciclos
- $T(K) = \frac{8600}{1,15 \times 10^9} = 7,478 \times 10^{-6}$ s

El último paso es calcular el coste de transferencia de los datos y sumarlo al de las dos partes anteriores para obtener el coste total del programa. Dado que $BW_{HD} = 4$ GB/s y $BW_{DH} = 3{,}6$ GB/s:

$$T(HD) = 100000000 \text{ elementos} \times 8 \text{ bytes/elemento} \times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 4 \text{ GB/s} = 0,186 \text{ s}$$

$$T(DH) = 1 \text{ elemento} \times 8 \text{ bytes/elemento} \times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 3{,}6 \text{ GB/s} = 2 \times 10^{-9} \text{ s}$$

$$T = 0,162 + 7,478 \times 10^{-6} + 0,186 + 2 \times 10^{-9} = \mathbf{0{,}348 \text{ s}}$$

Comparativa con pruebas experimentales

Al ejecutar las dos versiones del algoritmo se aprecia cómo la versión que emplea memoria global ajusta de manera satisfactoria, mientras que al emplear memoria *shared* aparece más distanciada. En las gráficas 9.1 y 9.2 se comparan los resultados experimentales, la estimación obtenida con la primera versión del modelo y la estimación empleando el modelo desarrollado posteriormente.

Si se considera la copia de datos, las estimaciones siguen aproximadamente la misma evolución que en las gráficas anteriores, por lo que se puede considerar que en este punto alcanza un buen nivel de exactitud.

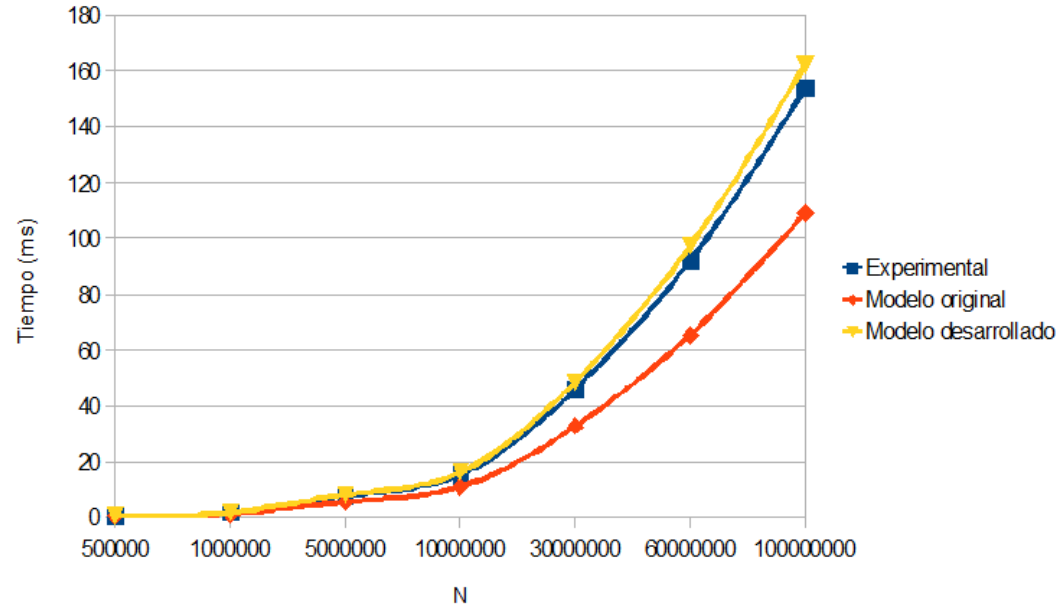

Figura 9.1: Comparativa de la estimación teórica y resultados experimentales del algoritmo de reducción (memoria global)

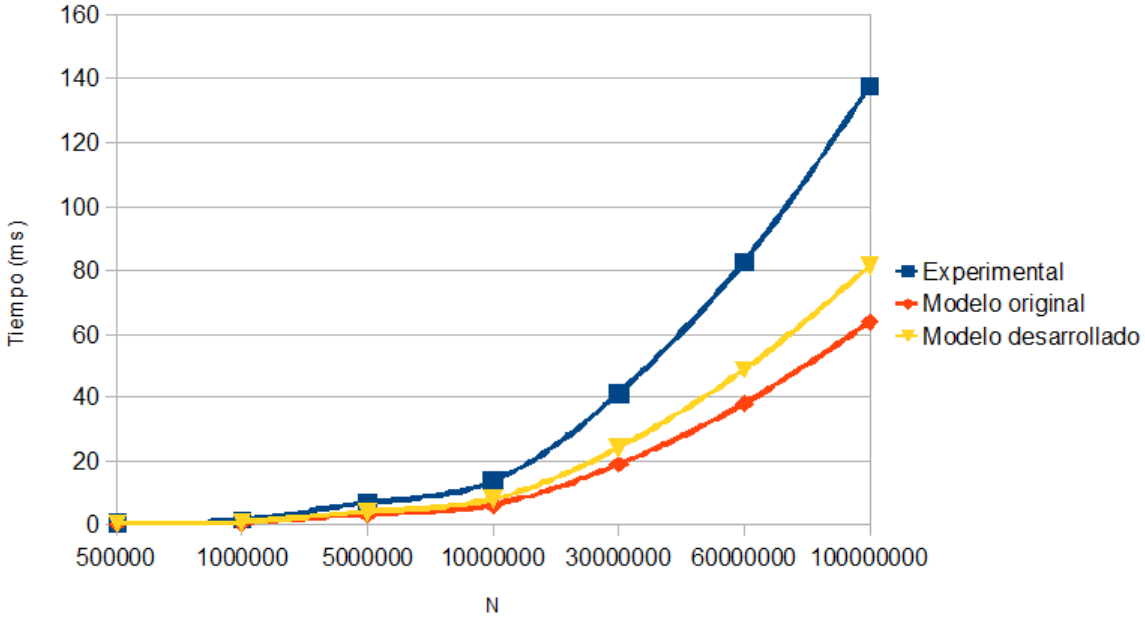

Figura 9.2: Comparativa de la estimación teórica y resultados experimentales del algoritmo de reducción (memoria *shared*)

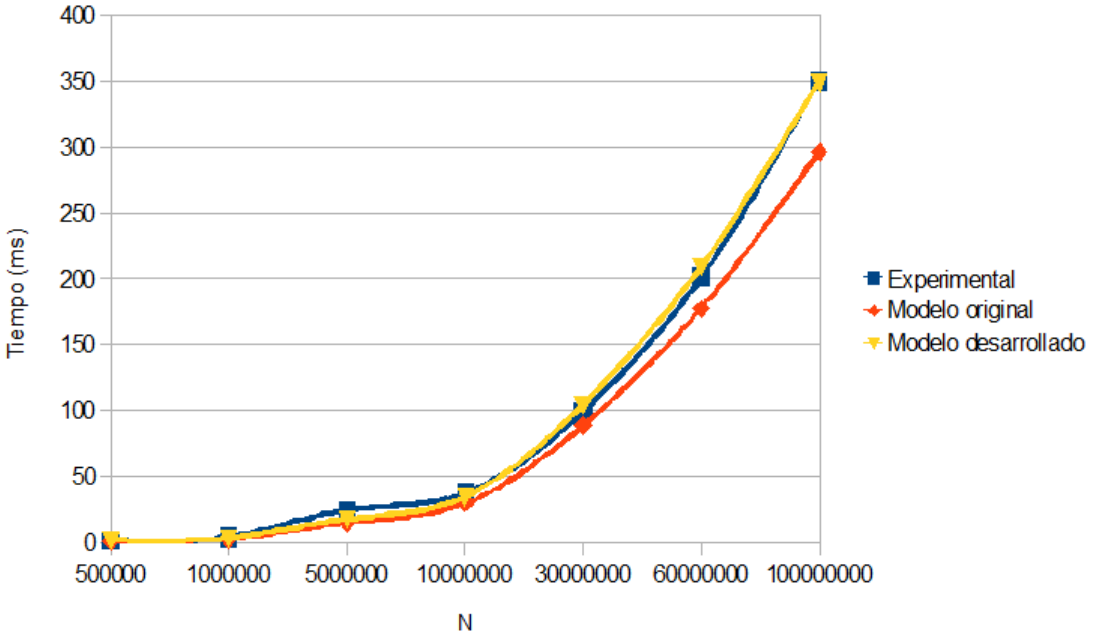

Figura 9.3: Comparativa de la estimación teórica y resultados experimentales del algoritmo de reducción (memoria global, con copia)

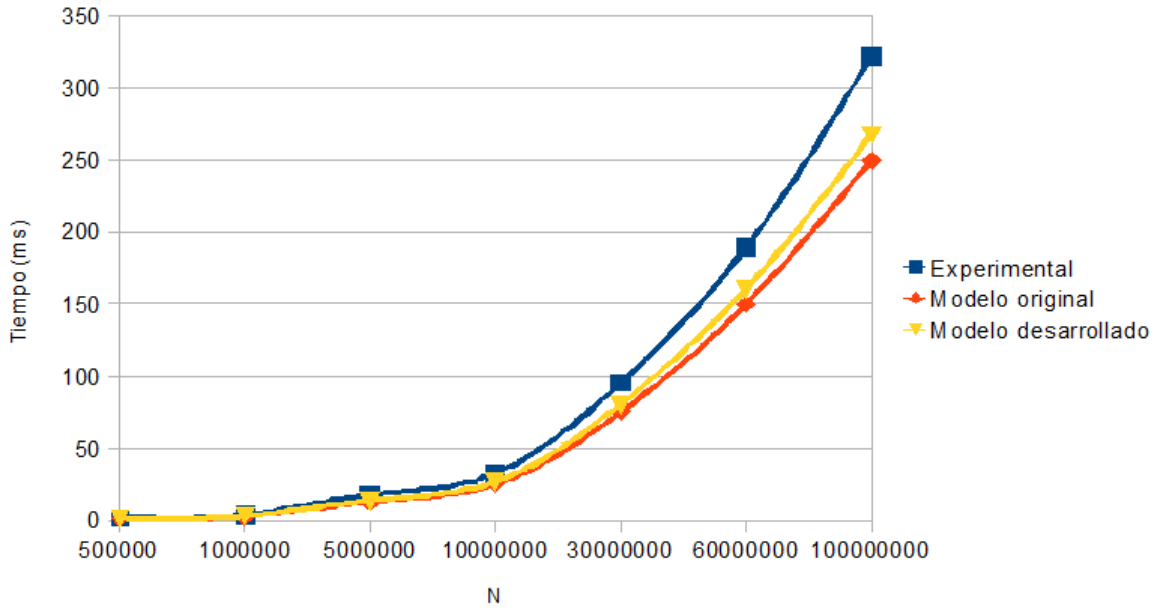

Figura 9.4: Comparativa de la estimación teórica y resultados experimentales del algoritmo de reducción (memoria *shared*, con copia)

Conclusión

De este caso de estudio se puede concluir que el modelo desarrollado consigue mejorar las estimaciones realizadas por el modelo tomado como base, incluso con la primera modificación llevada a cabo en el capítulo 5.1.

El trabajo actual es muy preciso en el caso de uso de la memoria global. Al emplear memoria *shared* se aleja un poco más del resultado experimental, pero si se tiene en cuenta el tiempo de transferencia resulta bastante aceptable en términos relativos.

9.2. Caso de uso 2: Producto matriz-vector

En esta sección se evaluará el coste de las diferentes versiones de este algoritmo, descritas en el capítulo 7. La evaluación se ha llevado a cabo para datos en coma flotante de distinta precisión, variando el número de filas de la matriz entre 500 y 20000, y manteniendo fijo en 2000 el número de columnas. Cada bloque de hilos está compuesto de 192 hilos.

Evaluación teórica

En la aproximación por filas, cada hilo debe realizar 2000 operaciones, una por cada columna. El resultado temporal se almacena en un registro, por lo que se realizan 2 operaciones de lectura de memoria, una sobre la matriz y otra sobre el vector, pero ninguna de escritura. Para el caso de una matriz con 1000 filas de datos en coma flotante de doble precisión, los parámetros adoptan los siguientes valores:

- $comp\_insts = 2000$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 48 \times 2000 = 96000$ ciclos
- $mem\_insts = 2 \times 2000 + 1 = 4000$ instrucciones + 1 no cacheable
- $data\_size = 8$ bytes
- $N_{max} = \frac{128}{8} = 16$ datos/petición
- $N_{min} = \frac{32}{8} = 4$ datos/petición
- $cache\_factor = \frac{16+4}{2} = 10$ datos/petición
- $latency_{gmem} = 600$ ciclos
- $latency_{smem} = 4$ ciclos

- $latency_{cache} = 4$ ciclos
- $C_{mem} = 600 \times \frac{4000}{10} + 4 \times \frac{4000 \times 9}{10} + 600 \times 1 = 255000$ ciclos
- $C_{max} = max(96000, 255000) = 255000$ ciclos
- $C_{sum} = 96000 + 255000 = 351000$ ciclos
- $N_B(K) = 6$ bloque
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{192}{32} = 6$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 6 \times 6 \times 32 \times 351000 \times \frac{1}{32 \times 4} = 3159000$ ciclos
- $T(K) = \frac{3159000}{1,15 \times 10^9} = 0,002747$ s

Para obtener el coste total del programa se debe añadir el tiempo consumido en transferir los datos entre CPU y GPU. El *kernel* recibe como parámetros la matriz y el vector a multiplicar, y se devuelve a la CPU el vector resultado.

$$T(HD) = (2000 \times 1000) + 2000 \text{ elementos} \times 8 \text{ bytes/elemento} \times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 4 \text{ GB/s} = 0,00373 \text{ s}$$

$$T(DH) = 1000 \text{ elementos} \times 8 \text{ bytes/elemento} \times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 3,6 \text{ GB/s} = 2,070 \times 10^{-6} \text{ s}$$

$$T = 0,002747 + 0,00373 + 2,070 \times 10^{-6} = \mathbf{0{,}006478 \text{ s}}$$

Para el enfoque orientado a bloques de filas, consideremos bloques de 4 filas, datos en coma flotante de simple precisión y una matriz de 5000 filas. El coste sería el siguiente:

- $comp\_insts = 4 \times 2000 = 8000$ instrucciones
- $issue\_cycles = 24$ ciclos
- $C_{comp} = 24 \times 8000 = 192000$ ciclos
- $mem\_insts = 4 \times (2 \times 2000 + 1) = 16000$ instrucciones + 4 no cacheables

- $data\_size = 4$ bytes
- $N_{max} = \frac{128}{4} = 32$ datos/petición
- $N_{min} = \frac{32}{4} = 8$ datos/petición
- $cache\_factor = \frac{32+8}{2} = 20$ datos/petición
- $latency_{gmem} = 600$ ciclos
- $latency_{smem} = 4$ ciclos
- $latency_{cache} = 4$ ciclos
- $C_{mem} = 600 \times \frac{16000}{20} + 4 \times \frac{16000 \times 19}{20} + 600 \times 1 = 543200$ ciclos
- $C_{max} = max(192000, 255000) = 543200$ ciclos
- $C_{sum} = 192000 + 543200 = 735200$ ciclos
- $N_B(K) = 7$ bloque
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{192}{32} = 6$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 7 \times 6 \times 32 \times 735200 \times \frac{1}{32 \times 4} = 7719600$ ciclos
- $T(K) = \frac{7719600}{1,15 \times 10^9} = 0,006713$ s
- $T(HD) = (2000 \times 5000) + 2000$ elementos $\times$ 4 bytes/elemento $\times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times$ 4 GB/s $= 0,009315$ s
- $T(DH) = 5000$ elementos $\times 4$ bytes/elemento $\times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 3,6$ GB/s $= 5 \times 10^{-6}$ s
- $T = 0,006713 + 0,009315 + 0,000005 =$ **0,016033 s**

Por último, el enfoque por columnas emplea operaciones atómicas, por lo que para el cálculo del coste se deben emplear las fórmulas descritas en la sección 5.2.5 y solo es necesario un subconjunto de los parámetros del modelo. Por ejemplo, para el caso de una matriz de enteros con 2000 filas, bloques de 192 hilos y ejecutando el algoritmo sobre una GPU con *compute capability* 2.0, el coste se calcularía de acuerdo a los siguientes valores:

- $C(K) = 2000 \times (17 \times 192 + 3450) = 13428000$ ciclos
- $R = 1,15$ GHz
- $T(K) = \frac{13428000}{1,15 \times 10^9} = 0,01168$ s
- $T(HD) = (2000 \times 2000) + 2000$ elementos $\times$ 4 bytes/elemento $\times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times$ 4 GB/s $= 0,003727$ s
- $T(DH) = 2000$ elementos $\times 4$ bytes/elemento $\times \frac{1 \text{ GB}}{10^9 \text{ bytes}} \times 3,6$ GB/s $= 2 \times 10^{-6}$ s
- $T = 0,006713 + 0,003727 + 0,000002 =$ **0,01541 s**

Comparativa con pruebas experimentales

En el caso de una fila por hilo, la figura 9.5 muestra que el modelo realiza un ajuste bastante preciso, siguiendo la media entre las fluctuaciones que se producen.

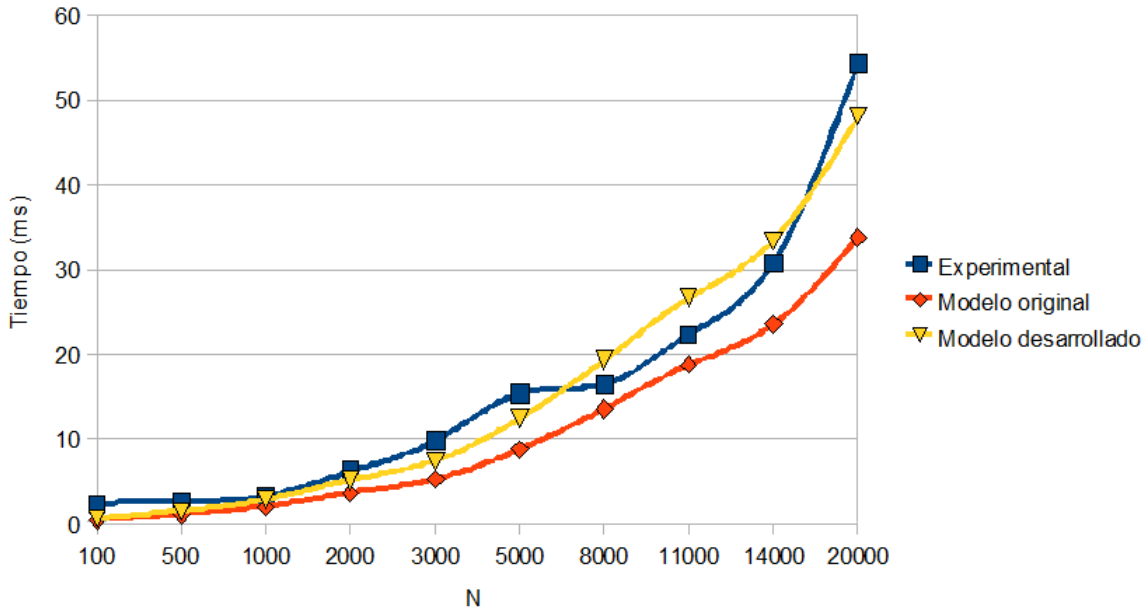

Figura 9.5: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por filas

Al pasar a considerar también el tiempo de copia se observa cómo la estimación de esta tarea se ha realizado de manera correcta, pues en la gráfica 9.6 las líneas experimental y teórica están prácticamente solapadas.

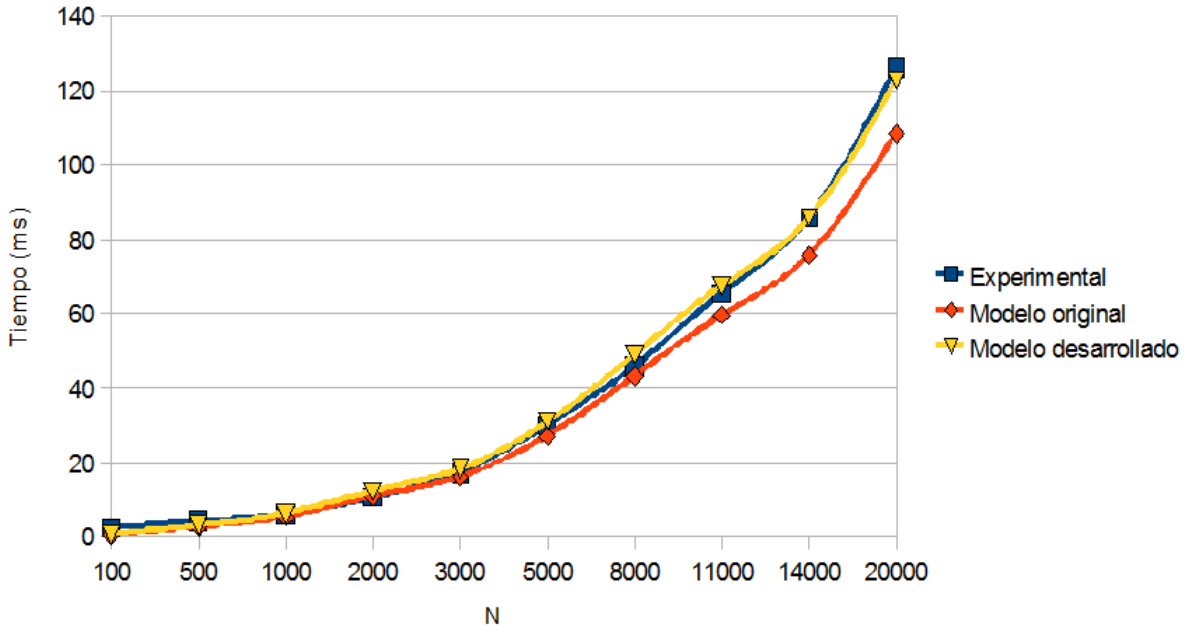

Figura 9.6: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por filas (con copia)

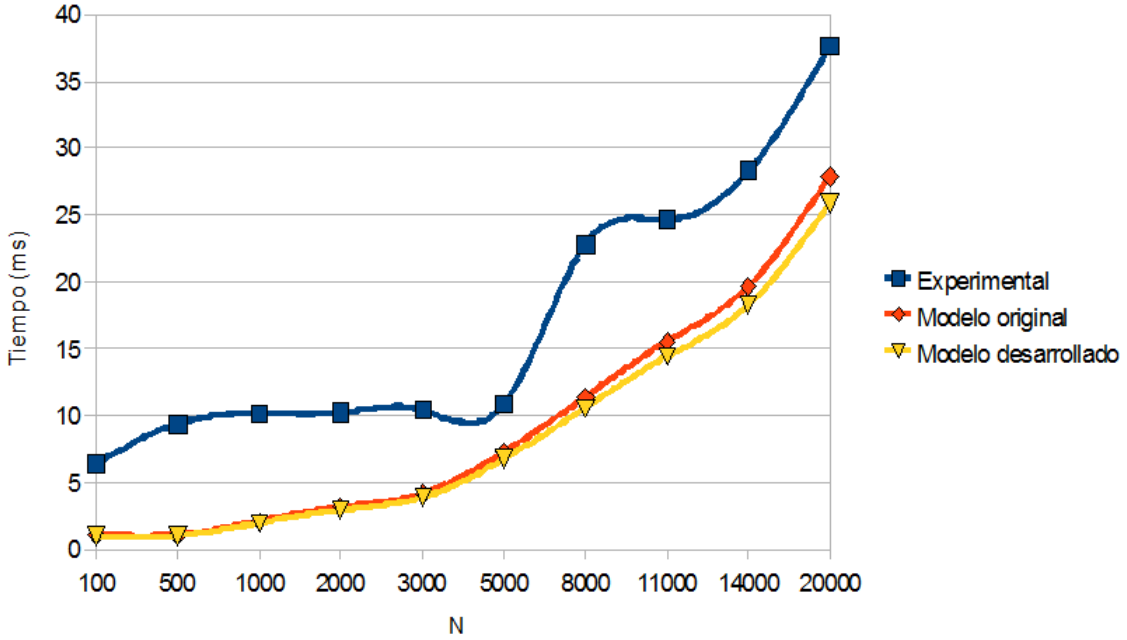

Figura 9.7: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por bloques de filas

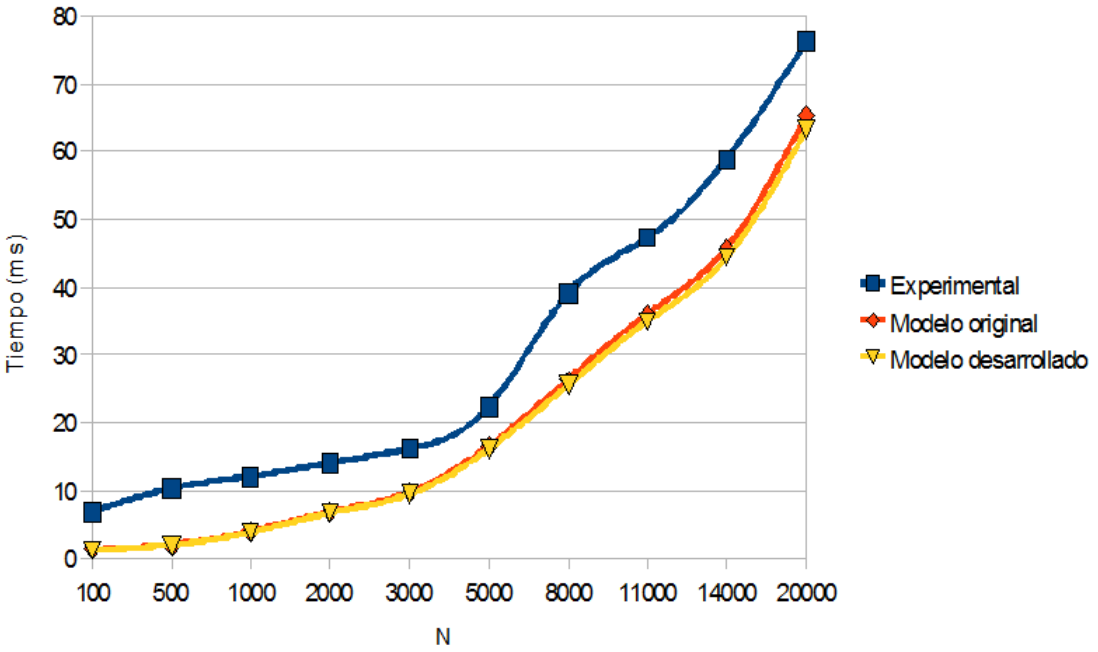

Figura 9.8: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por bloques de filas (con copia)

Cuando se estudia la asignación de un bloque de filas a un hilo, el ajuste no mejora respecto al modelo original, pues sigue la curva de evolución pero un 50 % por debajo de media (figuras 9.7 y 9.8).

Las pruebas de la asignación de una columna a cada hilo se han realizado desde dos perspectivas: modificando el número de hilos y el tamaño del problema. El problema que plantea el primer caso es que, en la práctica, el coste no aumenta linealmente con el número de hilos, sino que con los tamaños más bajos se mantiene relativamente constante. En el momento en el que evoluciona de manera lineal, en la imagen 9.10 se observa que la estimación ya se ajusta más al valor real.

Si se modifica el tamaño del problema (figuras 9.11 y 9.12), de nuevo la aproximación es mejor en el caso de la tarjeta con CC 1.3, aunque en ambos casos sigue la curva con la precisión suficiente como permitir realizar un estudio del comportamiento a nivel asintótico.

Si se tiene en cuenta también el tiempo de copia el error absoluto se mantiene pero, al tratarse de un intervalo temporal mayor, el error relativo disminuye, por lo que se puede emplear de manera un poco más fiable a la hora de realizar la estimación. Como muestra, véanse las gráficas 9.13 y 9.14.

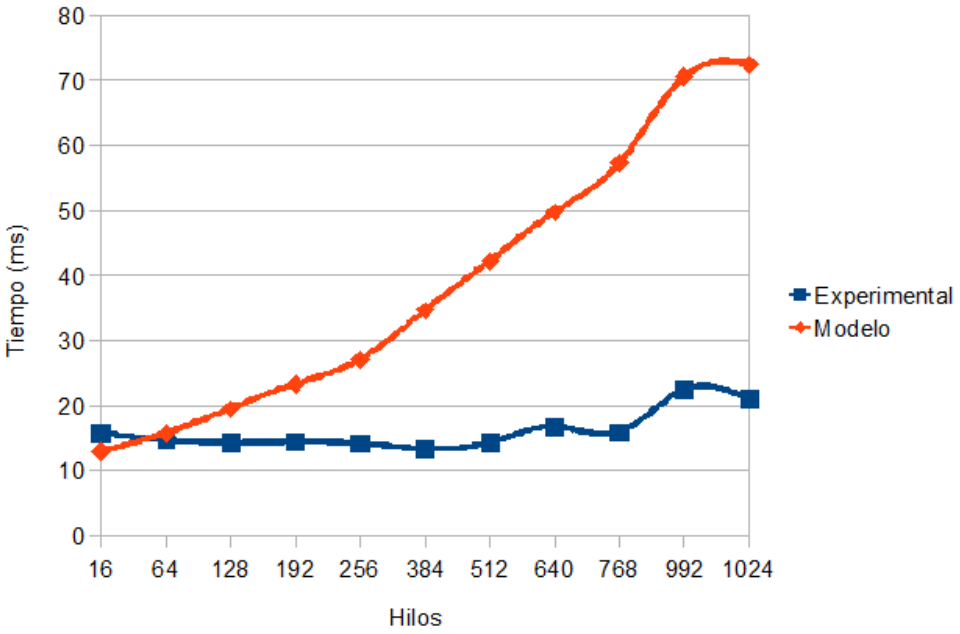

Figura 9.9: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 2.0 - cambio de hilos

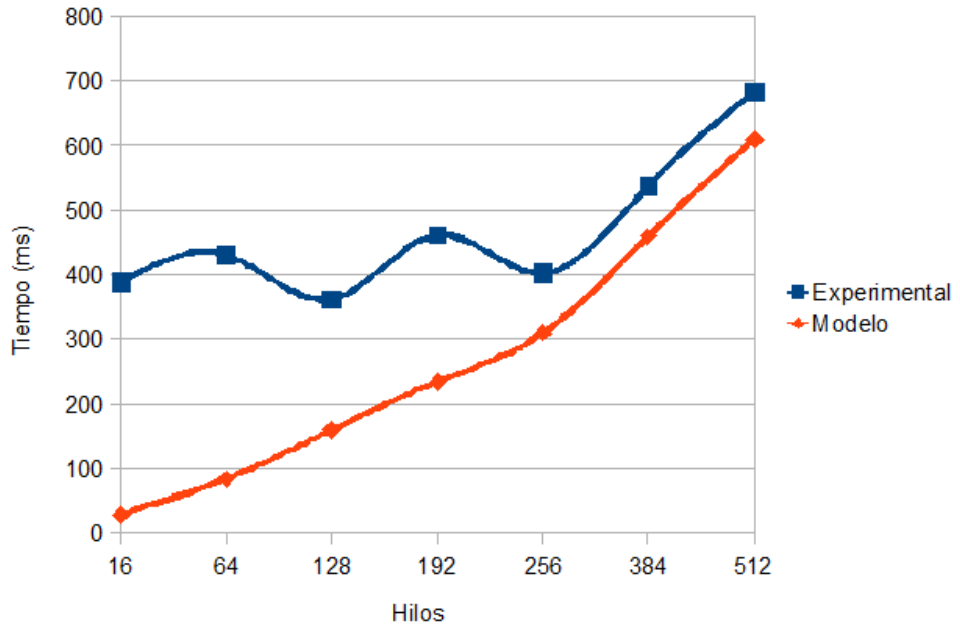

Figura 9.10: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 1.3 - cambio de hilos

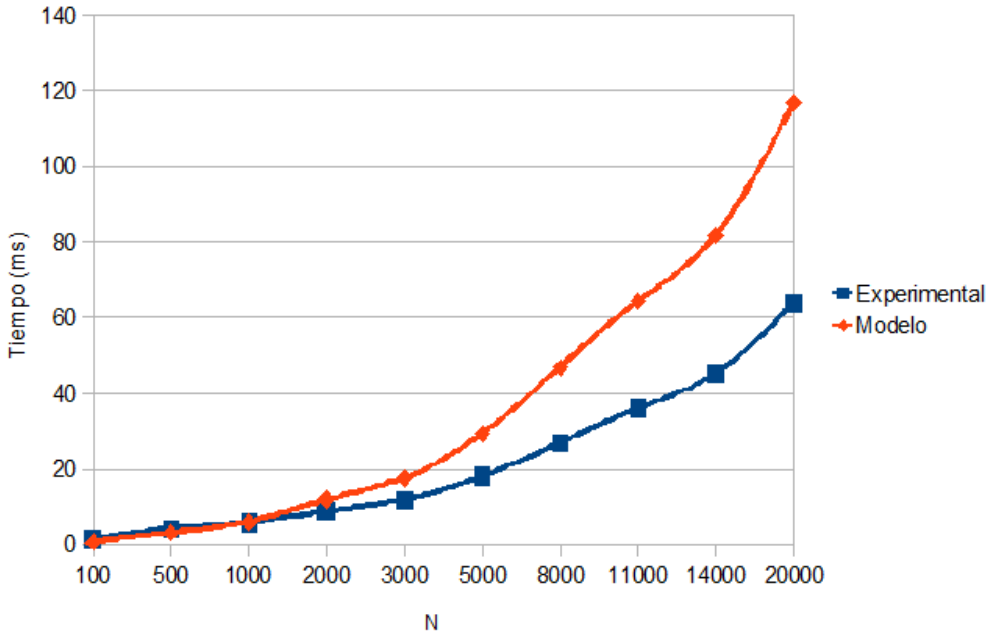

Figura 9.11: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 2.0 - cambio de tamaño

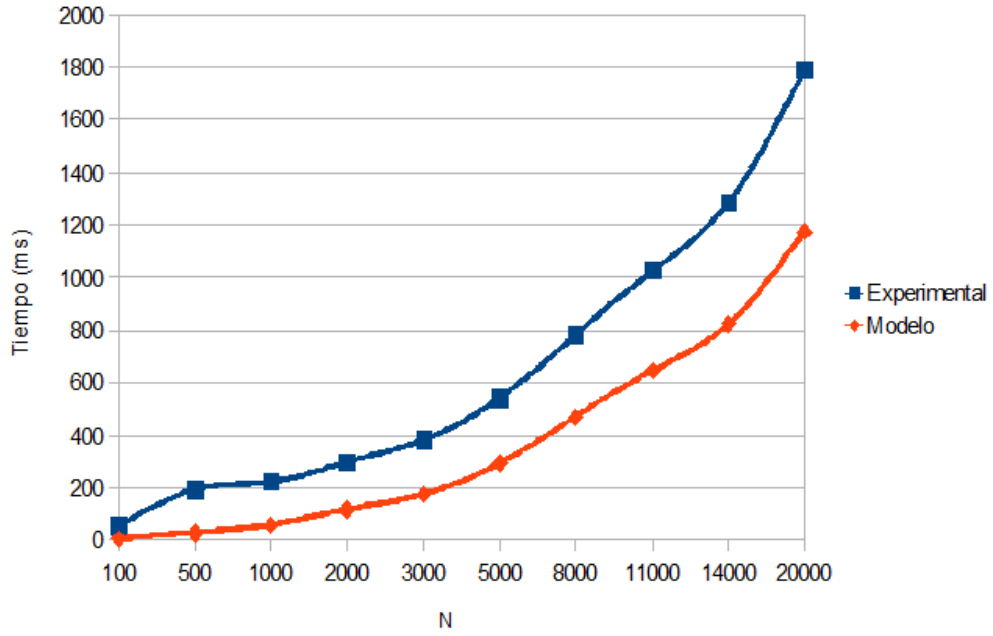

Figura 9.12: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 1.3 - cambio de tamaño

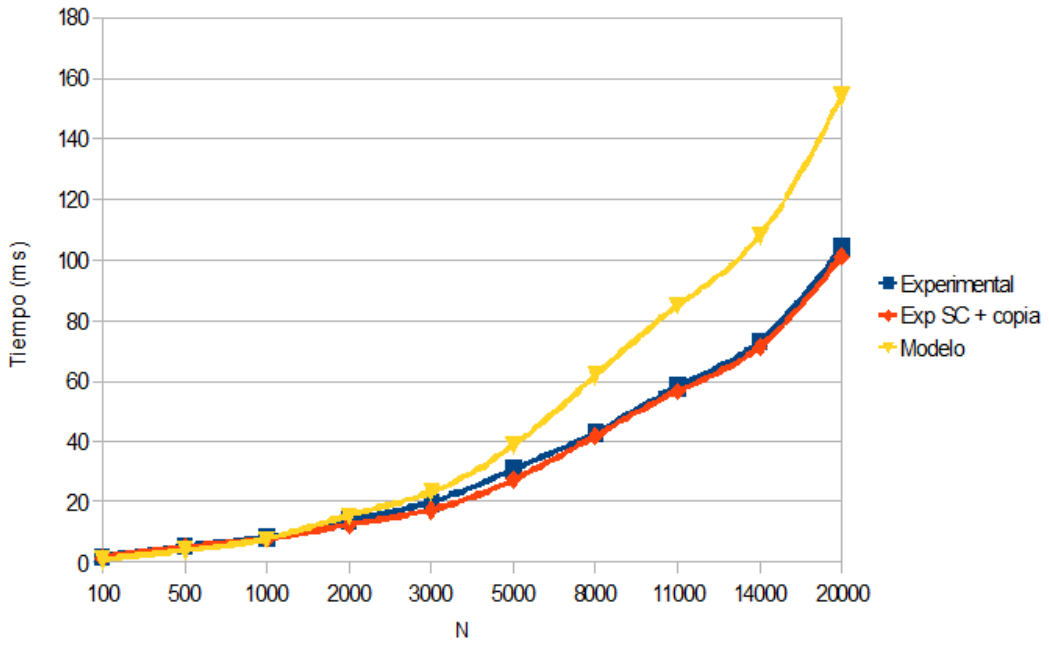

Figura 9.13: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 2.0 - cambio de tamaño (con copia)

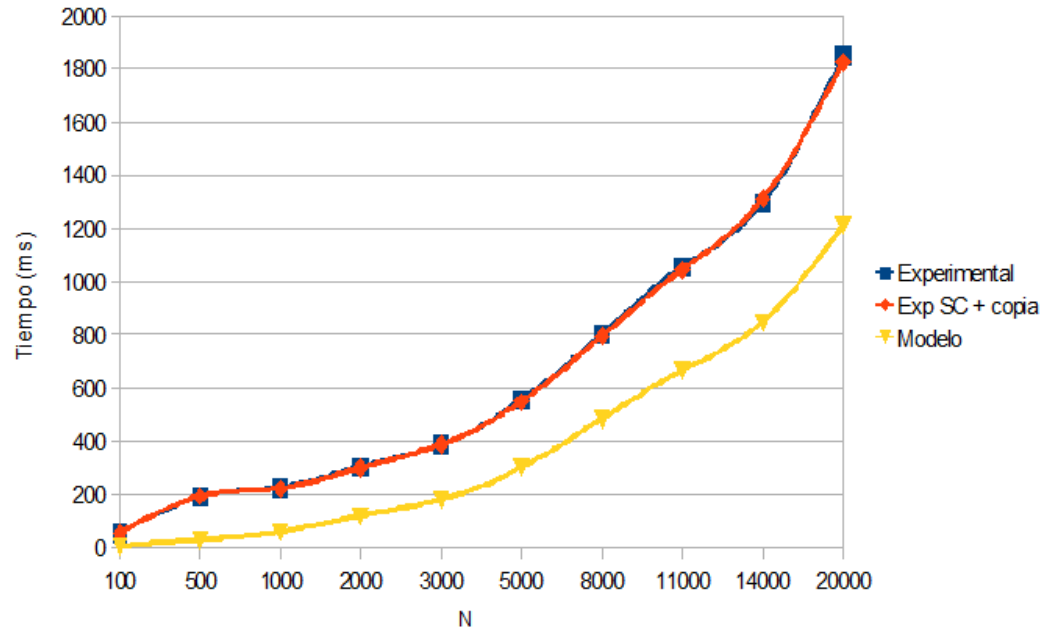

Figura 9.14: Comparativa de la estimación teórica y resultados experimentales del producto matriz-vector por columnas, CC 1.3 - cambio de tamaño (con copia)

Conclusión

Este caso de uso pone de manifiesto que, cuanto más se adecúa a GPU el enfoque usado, más consigue ajustarse el modelo a la realidad. Esto ocurre al asignar un hilo a cada fila. Lo contrario se da en la asignación por bloques de filas pues, aunque en un sistema de paso de mensajes esta agrupación sirve para reducir las comunicaciones, en una GPU provoca un acceso a memoria ineficiente por la falta de coalescencia. Esto se refleja en el modelo como una falta de exactitud bastante notoria.

A pesar de todo, allí donde la estimación no mejora la del modelo original, al menos consigue igualarlo. Por tanto, el modelo muestra un buen comportamiento al tratar algoritmos con características típicas en un *kernel*.

Respecto a las ampliaciones de funcionalidad sobre al modelo original, las operaciones atómicas demuestran ser un aspecto complejo y con comportamiento muy variable entre arquitecturas. En cada una se han obtenido resultados distintos: en la GPU de familia Fermi la estimación se sitúa bastante por encima del tiempo real, mientras que en la familia Tesla es al contrario, pero sigue más fielmente la curva asintótica.

9.3. Caso de uso 3: Descomposición de Cholesky

En esta sección se evalúa la estimación de coste para el enfoque denso estructurado de la descomposición de Cholesky. Aunque en el capítulo 8 se describen varias implementaciones previas, uno de los objetivos de este caso de uso era llegar a esta implementación y por tanto es la más representativa. Por ello, se ha considerado preferible realizar un estudio exhaustivo de ella.

Para que fuese posible realizar la estimación, en lugar de emplear los *kernels* de CUBLAS, muy eficientes pero de código propietario, se reimplementaron los núcleos computacionales básicos de la manera descrita en la sección 8.4.

Evaluación teórica

Por su complejidad y el hecho de implicar el uso de varios *kernels*, es más sencillo realizar la evaluación si se descompone en los siguientes pasos:

1. Ejecución y copia de datos de DSYRK
2. Ejecución y copia de datos de DGEMM
3. Ejecución y copia de datos de DTRSM
4. Algoritmo completo

Ejecución de DSYRK

Este algoritmo se expresa como $B = B - A \times A'$, por lo que se compone de 3 pasos: una trasposición, un producto y una resta de matrices.

Tanto la trasposición como el producto de matrices hacen uso de memoria global y memoria *shared*, por lo que se debe evaluar el coste cada tipo de memoria por separado. Las operaciones que se realizan son únicamente de acceso a memoria: primero se mueve el dato de la celda a trasponer a memoria *shared* y después se copia a memoria global en la posición correspondiente a la traspuesta. Por tanto, se ejecutan 2 instrucciones de acceso a memoria global y 2 de acceso a memoria *shared*. La ejecución en este caso concreto está planificada para bloques de $64 \times 64 = 1024$ hilos, y el algoritmo se estructura en *tiles* de 32×32 elementos.

La estimación de coste realizada para memoria global es la siguiente:

- $comp\_insts = 0$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 0$ ciclos
- $mem\_insts = 2$ instrucciones no cacheables
- $data\_size = 8$ bytes
- $cache\_factor = 1$ datos/petición
- $latency_{gmem} = 600$ ciclos
- $C_{mem} = 600 \times 2 = 1200$ ciclos
- $C_{max} = max(0, 1200) = 1200$ ciclos
- $C_{sum} = 0 + 1200 = 1200$ ciclos
- $N_B(K) = (\frac{64}{32})^2 = 4$ bloques
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{1024}{32} = 32$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 600 \times 32 \times 32 \times 1200 \times \frac{1}{32 \times 4} = 38400$ ciclos
- $T(K) = \frac{38400}{1,15 \times 10^9} = 0,0334 \times 10^{-3}$ s

Para memoria *shared* el esquema es exactamente el mismo exceptuando que C_{mem} se calcula con una latencia igual a 4 ciclos. Teniendo esto en cuenta, $C_{sum}(K) = 256$ ciclos y $T(K) = 0,223 \times 10^{-6}$ s.

En el producto, al igual que en la trasposición, se emplean bloques de $64 \times 64 = 1024$ hilos y las *tiles* tienen 32×32 elementos.

Al dividir las operaciones entre memoria global y *shared*, se observa que a la primera se accede 3 veces, 2 para copiar valores a memoria *shared* y otro para copiar el resultado de vuelta a memoria global. El coste es de $C_{sum}(K) = 57600$ ciclos y $T(K) = 0,0501 \times 10^{-3}$ s.

El grueso de la computación se realiza en memoria *shared*, para la que el coste es el siguiente:

- $comp\_insts = 32$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 32 \times 48 = 1536$ ciclos
- $mem\_insts = 2 + 32 \times 2 + 1 = 67$ instrucciones
- $data\_size = 8$ bytes
- $latency_{smem} = 4$ ciclos
- $C_{mem} = 4 \times 67 = 268$ ciclos
- $C_{max} = max(1536, 268) = 1536$ ciclos
- $C_{sum} = 1536 + 268 = 1804$ ciclos
- $N_B(K) = (\frac{64}{32})^2 = 4$ bloques
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{1024}{32} = 32$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 4 \times 32 \times 32 \times 1804 \times \frac{1}{32 \times 4} = 57728$ ciclos
- $T(K) = \frac{57728}{1,15 \times 10^9} = 0,0502 \times 10^{-3}$ s

La operación de resta de matrices es muy similar a la suma ya analizada en 5.1.2. Puesto que el *kernel* siempre se lanza sobre bloques de 64×64 elementos y con 192 hilos, el coste independiente del tamaño es de $C_{sum}(K) = 60984$ ciclos y $T(K) = 0,053 \times 10^{-3}$ s.

Sumando los resultados obtenidos en cada una de las operaciones, el coste total es de $0,1869 \times 10^{-3}$ s.

Copia de datos de DSYRK

Para ejecutar esta operación se necesitan 2 bloques de 64×64 elementos. Para simplificar el código de los *kernel*, en lugar de emplear el *leading dimension* como en BLAS y CUBLAS, se emplean directamente dichos bloques. Por ello, es necesario copiarlos de la matriz completa a una nueva región de memoria mediante un `cudaMemcpyDeviceToDevice`. El ancho de banda en este tipo de transferencias es de aproximadamente 80 GB/s, por lo que para transferir los 8192 elementos que ocupan 65536 bytes se consumen $0,763 \times 10^{-6}$ s.

Como resultado se devuelve una triangular de la matriz B, que consta de 2048 elementos. Para transferir 16384 bytes se consumen $0,191 \times 10^{-6}$ s.

Una vez incluido este tiempo, el coste global de una ejecución de DSYRK es de $0,1879 \times 10^{-3}$ s.

Ejecución de DGEMM

Las operaciones realizadas en DGEMM son exactamente iguales que en DSYRK, por lo que el coste es también de $0,1869 \times 10^{-3}$ s.

Copia de datos de DGEMM

Siguiendo la misma línea que DSYRK, esta operación recibe 3 bloques de 64×64 elementos y devuelve uno, los cuales se deben copiar mediante un `cudaMemcpyDeviceToDevice`. Teniendo en cuenta que el ancho de banda en transferencia dentro del *device* es de 80 GB/s, el tiempo consumido para copiar los bloques de entrada es de $1,144 \times 10^{-6}$ s y para el bloque resultado es de $0,381 \times 10^{-6}$ s.

Sumando ese tiempo al consumido en los *kernels*, una ejecución de DGEMM tiene un coste de $0,1884 \times 10^{-3}$ s.

Ejecución de DTRSM

Este núcleo computacional es totalmente distinto a los anteriores. Se compone de un solo *kernel* que se ejecuta N veces, siendo N el lado del bloque sobre el que trabajar. El coste para una iteración se detalla a continuación:

- $comp\_insts = 3$ instrucciones
- $issue\_cycles = 48$ ciclos
- $C_{comp} = 48 \times 3 = 144$ ciclos
- $mem\_insts = 6$ instrucciones no cacheables
- $data\_size = 8$ bytes
- $latency_{gmem} = 600$ ciclos
- $latency_{smem} = 4$ ciclos
- $latency_{cache} = 4$ ciclos
- $C_{mem} = 600 \times 6 = 3600$ ciclos
- $C_{max} = max(144, 3600) = 3600$ ciclos
- $C_{sum} = 144 + 3600 = 3744$ ciclos
- $N_B(K) = 4$ bloque
- $N_t(K) = 32$ hilos
- $N_w(K) = \frac{1024}{32} = 32$ *warps*
- $N_C = 32$ *cores*
- $D = 4$
- $R = 1,15$ GHz
- $C_{sum}(K) = 4 \times 32 \times 32 \times 3744 \times \frac{1}{32 \times 4} = 119808$ ciclos
- $T(K) = \frac{119808}{1,15 \times 10^9} = 0,1042 \times 10^{-3}$ s

Por tanto, si el tamaño de bloque es 64×64, el coste de 64 iteraciones es $6,6676 \times 10^{-3}$ s.

Copia de datos de DTRSM

Para ejecutar esta operación son necesarios dos bloques de entrada de 64×64 elementos, cuya transferencia consume $0,763 \times 10^{-6}$ s y uno de salida que consume $0,381 \times 10^{-6}$ s.

Incluyendo este tiempo de copia, la ejecución de DTRSM tiene un coste de $6,6687 \times 10^{-3}$ s.

Algoritmo completo

Una vez calculado el coste de ejecución de los *kernels* y de la copia de los datos que emplean como parámetros y resultado, para calcular el coste del algoritmo completo se debe estimar el número de veces que es necesario ejecutar cada núcleo computacional. Las matrices densas estructuradas que se han tomado como ejemplo cumplen las siguientes propiedades:

- Si es la fila 0, el primer bloque con contenido tiene el índice 0. Es el bloque de la diagonal.
- Si el índice de fila i es múltiplo de 3, el primer bloque con contenido tiene el índice $j = i - 1$.
- En el resto de filas, el primer bloque es el $j = 1$.

El número de ejecuciones de cada *kernel* se ha tomado automáticamente (no mediante cálculos manuales) empleando una versión simplificada del algoritmo que únicamente contemplase esta función. También se ha medido de manera experimental el coste de ejecución de DPOTRF en CPU sobre un bloque.

Además, tras ejecutar DSYRK con todos los bloques asociados a otro en la diagonal, se copian todos ellos, incluyendo este último. Tras la primera ejecución se copia 1 bloque, en la segunda 2, en la tercera 3, y así sucesivamente. Por tanto, se llevan a cabo $\frac{N}{64}$ series de copias que en total suponen $\frac{N \times (N+1)}{2}$ elementos. En la copia realizada tras ejecutar DPOTRF se transfiere el mismo número de datos que componen el bloque, es decir, 64×64 elementos.

En la tabla 9.3 se muestran el número de ejecuciones y el tiempo (en milisegundos) que se consume en cada una de ellas, hasta llegar al coste total del algoritmo para diferentes tamaños de matriz.

Comparativa con pruebas experimentales

En la gráfica 9.15 se muestra cómo el modelo sigue con bastante fidelidad la evolución a nivel asintótico, mostrando su mejor aproximación en matrices con tamaño en torno a 2000×2000 elementos. En la gráfica 9.16 se puede observar cómo la estimación se mueve en un error inferior al 25 % para tamaños entre 1000 y 2600. El distanciamiento en tamaños mayores es debido a la sobrecarga de las operaciones que no son de acceso a memoria ni cálculo en coma flotante.

Conclusión

Este caso de uso demuestra que, a pesar de la complejidad del algoritmo por la gran cantidad de operaciones y copias de datos que lo componen, es factible realizar un análisis de rendimiento con un margen de error aceptable. Para añadir

Tamaño de problema	512	1024	1280	1536	1920	2560
Nº ejecuciones DSYRK	16	70	120	183	289	494
Tiempo DSYRK	3,006118	13,15176	22,54588	34,38247	54,29800	92,81389
Nº ejecuciones DGEMM	17	190	447	868	1755	3978
Tiempo DGEMM	3,203728	35,80636	84,23919	163,5785	330,7377	749,6722
Nº ejecuciones DTRSM	16	70	120	183	289	494
Tiempo DTRSM	106,6995	466,8104	800,2464	1220,375	1927,260	3294,347
Nº ejecuciones DPOTRF	8	16	20	24	30	40
Tiempo DPOTRF (exper.)	0,048316	0,048316	0,048316	0,048316	0,048316	0,048316
Nº copias tras DSYRK	8	16	20	24	30	40
Tiempo copias DSYRK	0,271797	1,086129	1,696746	2,442996	3,816684	6,784333
Nº copias tras DPOTRF	8	16	20	24	30	40
Tiempo copias DPOTRF	0,007629	0,007629	0,007629	0,007629	0,007629	0,007629
Tiempo total consumido	113,2371	516,9106	908,7841	1420,835	2316,168	4143,674

Cuadro 9.1: Estimación de coste de la descomposición de Cholesky para diferentes tamaños de problema (en milisegundos)

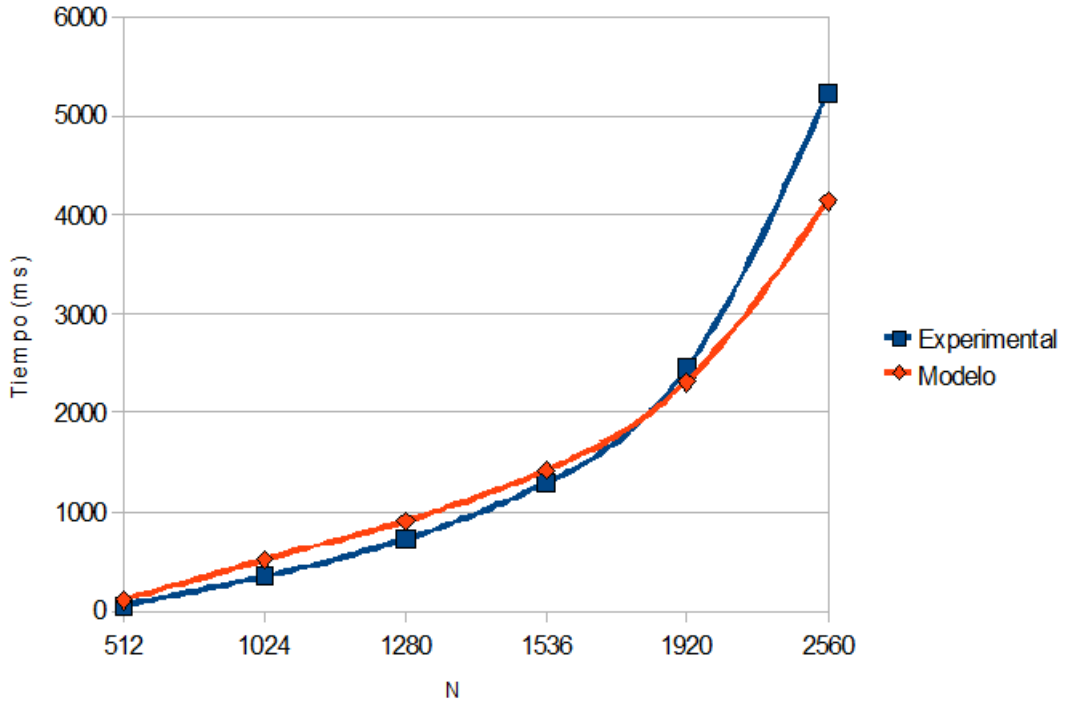

Figura 9.15: Comparativa de la estimación teórica y resultados experimentales de la descomposición de Cholesky

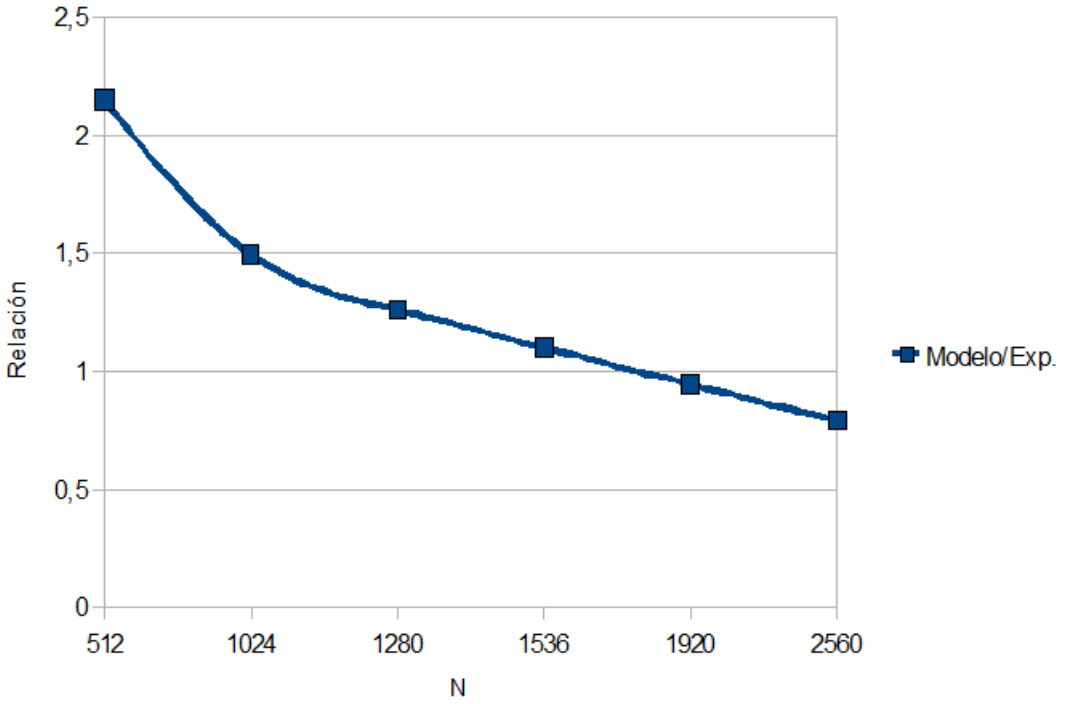

Figura 9.16: Relación entre la estimación teórica y resultados experimentales de la descomposición de Cholesky

más precisión es necesario tener en cuenta las operaciones que no se realizan en coma flotante, como los cálculos con enteros que por ahora se omiten en la estimación.

En cualquier caso, el modelo se aproxima con bastante fidelidad a la curva asintótica, por lo que aún sin ser totalmente exacto en las predicciones, sí permite estudiar con bastante precisión la forma que tomará dicha curva al modificar parámetros como el tamaño del problema.

10

Conclusiones y trabajo futuro

En el presente trabajo se ha desarrollado un modelo para estimar el coste de algoritmos implementados en GPU mediante la tecnología CUDA y abrir el camino para aquellos que siguen un enfoque híbrido.

El primer objetivo planteado en el alcance del proyecto consistía en que el modelo fuese teórico, es decir, que no hubiese que realizar una implementación para poder hacer las estimaciones. Uno de los modelos estudiados obligaba incluso a hacer un análisis del *bytecode* PTX generado por CUDA. El modelo elegido finalmente como base fue el de Mukherjee [1], que resultó ser también el más sencillo de aplicar.

El modelo también debía estar orientado a algoritmos implementados para aprovechar al máximo las capacidades de la CPU y la GPU. A la evaluación exclusiva del coste de un *kernel* que realizaban los modelos estudiados se añade la de la transferencia de datos entre CPU y GPU. Aunque el modelo presenta limitaciones por no haberse integrado con un modelo de CPU, sí resulta útil como primera aproximación y basta con solventar este último punto para que se cubra de manera teórica el coste completo de un algoritmo híbrido.

A las mejoras arriba citadas se une también el análisis profundo de la divergencia entre ramas; el modelo de Baghsorkhi [6] es el único que contemplaba esta característica explícitamente, pero su aplicación era demasiado compleja y fue descartado. Otros añadidos son el modelado básico del funcionamiento de la memoria caché y el estudio de las operaciones atómicas. Aunque estas últimas no pudieron modelarse satisfactoriamente, al menos se cuenta con un estudio al respecto tanto en tarjetas de arquitectura Fermi como Tesla.

En cuanto a las pruebas experimentales, en todos los casos ha mostrado un resultado igual o mejor que el modelo tomado como referencia. En el caso de la reducción ha conseguido mejorarlo en un 15-20 %, dependiendo del tipo de memoria empleado y la inclusión de la copia de datos o no.

En el producto matriz-vector por filas el modelo original realizaba una estimación a la baja, mientras que el modelo desarrollado sigue la curva con total fidelidad. En el enfoque por bloques de filas, si bien la estimación no mejora al original, se mantiene al mismo nivel. En cuanto al enfoque por columnas, el modelo puede servir como una primera aproximación para estudiar la curva asintótica, pero no para tenerlo en cuenta como una medida fidedigna.

Finalmente, la descomposición de Cholesky para matrices densas estructuradas deja patente que para algoritmos de gran complejidad el modelo realiza una estimación bastante adecuada, manteniéndose en un margen de error máximo de aproximadamente el 20 % para los tamaños de problema estudiados.

Ya que el modelo se presenta como una buena base, cabe plantear posibles mejoras para cubrir todo el espectro de aplicaciones desarrolladas en GPU. La primera de ellas integrarlo explícitamente con un modelo de CPU para estimar el coste completo de una aplicación híbrida. Además de calcular el coste del código ejecutado en CPU, es necesario tener en cuenta el solapamiento entre ambas unidades de cómputo. Este es un factor de cálculo bastante complicado y que implica tener en cuenta los *streams*, otra técnica mediante la cual aumentar el paralelismo, permitiendo solapar cálculos y copia de datos, o copias en diferentes sentidos entre CPU y GPU.

Por otra parte, la primera aproximación a las operaciones atómicas no ha resultado todo lo satisfactoria que se esperaba. Otra posible vía de mejora sería continuar su estudio para realizar una estimación más precisa aunque solo fuese en la arquitectura Fermi, la más reciente de las existentes.

Otra característica no contemplada son las funciones de votación en *warps*, que permiten ejecutar cierto código si todos los hilos de un *warp* verifican una condición.

Véase cómo estas mejoras consisten en su mayoría en ampliaciones sobre el modelo ya existente y no contradicen las características ya contempladas. Uniendo esta consideración a los positivos resultados obtenidos de las pruebas experimentales, **se puede concluir que las estimaciones del modelo son útiles** de cara a emplearlo en un estudio previo a una implementación, por la sencillez (en parte derivada del modelo de Mukherjee) de los cálculos y por la exactitud que ha presentado en casos como el matriz-vector por filas.

Además, aunque no se ha contemplado en este trabajo, al tratarse de un modelo de alto nivel no ligado directamente a ningún lenguaje permitiría evaluar algoritmos implementados en OpenCL. Para ello, podría ser necesario modificar valores como el coste de una instrucción o los tiempos de latencia, pero gran parte de las bases conceptuales de esta tecnología son las mismas que en CUDA.

Por último, cabe destacar que el estudio de las arquitecturas heterogéneas y de la biblioteca MAGMA han derivado en la publicación de un artículo [31] y en la elaboración de otro pendiente de ser enviado para su revisión [32].

A

Bibliotecas

A.1. CUBLAS

Como se describe en [33], esta biblioteca es una implementación de BLAS (*Basic Linear Algebra Subprograms*) para CUDA. Aunque permite a los usuarios aprovechar los recursos de una GPU, no realiza la auto-paralelización entre varias GPU.

Para usar CUBLAS, las matrices y vectores deben almacenarse previamente en la memoria de la GPU, pues los algoritmos no disponen de una interfaz para recibir y devolver automáticamente los datos a CPU, aunque sí se proporcionan funciones específicas para realizar estas tareas de forma sencilla.

Para proporcionar la máxima compatibilidad con los entornos en Fortran, CUBLAS utiliza direccionamiento por columnas e índices basados en 1. Puesto que C y C++ usan direccionamiento por filas, las aplicaciones no pueden usar la semántica de *arrays* nativa en los bidimensionales, sino que se deben definir macros o funciones *inline* para implementar matrices sobre *arrays* unidimensionales.

A.2. CUSPARSE

Esta biblioteca contiene un conjunto de subrutinas básicas de álgebra lineal para matrices dispersas y diseñadas para ser invocadas desde C o C++ [34]. Dichas subrutinas se clasifican en equivalentes a los 3 niveles de BLAS pero aplicados a vectores y matrices dispersos, resolutores de sistemas triangulares, y conversión entre diferentes formatos de almacenamiento de matrices. Los formatos aceptados son COO, CSR y CSC.

A.3. MAGMA

Tal como se describe en [35], el objetivo del proyecto MAGMA (*Matrix Algebra on GPU and Multicore Architectures*) es crear una nueva generación de bibliotecas de álgebra lineal que permita obtener soluciones en arquitecturas híbridas/heterogéneas, como los sistemas actuales que combinan *multi-core* y multi-GPU, explo-

tando mediante los algoritmos y *frameworks* todas las posibilidades de cada uno de los componentes de la arquitectura.

Su funcionalidad, métodos de almacenamiento de datos (por columnas) e interfaces son similares a LAPACK, lo que facilita portar las aplicaciones que ya empleaban dicha biblioteca. El criterio de nombrado de los algoritmos se deriva de LAPACK, con el prefijo "magma" y el sufijo "gpu" dependiendo de la interfaz que se esté utilizando.

La biblioteca incluye las descomposiciones LU, QR y Cholesky en cuatro precisiones (simple, doble, compleja simple y compleja doble) con dos interfaces estilo LAPACK. Una de ellas recibe la entrada y devuelve la salida a CPU, mientras que la otra trabaja con los datos directamente en GPU. También incluye resolutores basados en dichas descomposiciones. A partir de estas descomposiciones se han implementado también varios resolutores.

Finalmente, se proporciona también MAGMA BLAS, un complemento a CUBLAS para mejorar el rendimiento de las rutinas de MAGMA.

A.4. Thrust

Como se indica en su guía rápida [36], Thrust es una biblioteca en C++ para CUDA basada en la STL que permite implementar aplicaciones paralelas usando una interfaz de alto nivel y totalmente interoperable con CUDA C.

Thrust proporciona dos contenedores de vectores denominados `host_vector` y `device_vector`, almacenados en la memoria del *host* o *device*, respectivamente. Estos contenedores son similares al `std::vector` de la STL de C++, por lo que son genéricos y se pueden redimensionar dinámicamente. Así mismo, se puede acceder a sus elementos mediante iteradores que, aunque son similares a punteros, contienen otra información como el dispositivo donde se está almacenando la información.

Thrust también proporciona un gran número de algoritmos paralelos típicos, muchos de los cuales tienen un equivalente en la STL. Estos algoritmos disponen de interfaces para interactuar con datos que se encuentran en CPU o GPU. Entre estos algoritmos se encuentran:

- Transformaciones, que aplican una operación a cada elemento de un conjunto de entrada.
- Reducciones, que emplean una operación binaria para reducir una secuencia de entrada a un único valor.
- Funciones de ordenación o reorganización de datos según cierto criterio.

B

Tablas de resultados

Este apéndice contiene los resultados de rendimiento representados de manera tabular, para poder estudiarlos de manera más precisa que mediante las gráficas distribuidas a lo largo de esta documentación.

B.1. Suma de vectores

Tamaño	Hong y Kim	Mukherjee (SUM)	Mukherjee (caché)	Nugteren	Experimental
784	0,014330	0,009423	0,000863	0,002087	0,007131
4096	0,014386	0,043075	0,003944	0,007161	0,011058
16384	0,038189	0,172299	0,015777	0,050138	0,013203
262144	0,611023	2,756786	0,252438	0,476311	0,068171
524288	1,222046	5,513572	0,504877	0,927552	0,127867
1048576	2,444092	11,027144	1,009753	1,855105	0,246114
4194304	9,776368	44,108577	4,039012	7,345213	0,962570

Cuadro B.1: Resultados absolutos: Estudio de modelos mediante suma de vectores

Tamaño	Hong y Kim/Exp.	Mukherjee/Exp.	Nugteren/Exp.
784	2,009442	0,120991	0,292646
4096	1,300959	0,356696	0,647577
16384	2,892376	1,194955	3,797375
262144	8,963049	3,702997	6,986969
524288	9,557142	3,948441	7,254020
1048576	9,930731	4,102786	7,537584
4194304	10,156527	4,196071	7,630835

Cuadro B.2: Relación experimental/modelo: Estudio de modelos mediante suma de vectores

B.2. Producto matriz-vector

B.2.1. Estudio de modelos existentes

Hilos	Hong y Kim	Mukherjee (SUM)	Mukherjee (caché)	Nugteren	Experimental
96	2,003913	44,621453	6,067174	9,404667	6,196514
128	2,671739	43,913176	5,970870	12,541101	6,425123
192	4,007391	44,621453	6,067174	12,541101	9,055195
256	4,007391	45,329730	6,163478	16,723014	8,129817
384	4,007391	42,496622	5,778261	12,541101	8,684034
448	4,007391	44,621453	6,067174	14,632058	7,456060
512	4,007391	45,329730	6,163478	16,723014	7,774731

Cuadro B.3: Resultados absolutos: Estudio de modelos mediante producto matriz-vector asignando una fila por hilo - cambio de hilos

Hilos	Hong y Kim/Exp.	Mukherjee (caché)/Exp.	Nugteren/Exp.
96	0,323394	0,979127	1,517735
128	0,415827	0,929300	1,951885
192	0,442552	0,670021	1,384962
256	0,492925	0,758132	2,056998
384	0,461467	0,665389	1,444156
448	0,537468	0,813724	1,962438
512	0,515438	0,792758	2,150944

Cuadro B.4: Relación experimental/modelo: Estudio de modelos mediante producto matriz-vector asignando una fila por hilo - cambio de hilos

Hilos	Hong y Kim	Mukherjee (SUM)	Mukherjee (caché)	Nugteren	Experimental
96	4,007391	60,160435	0,414783	12,527188	6,193514
128	5,343043	58,337391	0,553043	16,709101	9,498195
192	6,010870	65,629565	0,829565	25,072928	10,689940
256	5,343043	58,337391	1,106087	33,436754	13,953180
384	6,010870	65,629565	1,659130	50,164406	21,808970
448	7,012609	76,567826	1,935652	58,528232	25,363610
512	5,343043	58,337391	2,212174	66,892058	28,967880

Cuadro B.5: Resultados absolutos: Estudio de modelos mediante producto matriz-vector asignando un bloque de filas por hilo - cambio de hilos

Hilos	Hong y Kim/Exp.	Mukherjee (caché)/Exp.	Nugteren/Exp.
96	0,647030	0,066971	2,022630
128	0,562532	0,058226	1,759187
192	0,562292	0,077602	2,345469
256	0,382927	0,079271	2,396354
384	0,275615	0,076076	2,300173
448	0,276483	0,076316	2,307567
512	0,184447	0,076366	2,309180

Cuadro B.6: Relación experimental/modelo: Estudio de modelos mediante producto matriz-vector asignando un bloque de filas por hilo - cambio de hilos

Tamaño	Hong y Kim	Mukherjee (SUM)	Mukherjee (caché)	Nugteren	Experimental
100	2,170543	2,734565	0,288913	6,268232	2,261811
500	2,170793	8,203696	0,866739	6,268232	2,503722
1000	2,087690	16,407391	1,733478	6,268232	3,122608
2000	2,087941	30,080217	3,178043	6,268232	6,077603
3000	2,087941	43,753043	4,622609	12,541101	9,687424
5000	2,087941	73,833261	7,800652	12,541101	15,291070
8000	2,672070	114,851739	12,134348	18,813971	16,359990
11000	3,562311	158,604783	16,756957	31,359710	22,229440
14000	4,452630	199,623261	21,090652	37,632580	30,766460
20000	6,233368	287,129348	30,335870	50,178319	54,370950

Cuadro B.7: Resultados absolutos: Estudio de modelos mediante producto matriz-vector asignando una fila por hilo - cambio de tamaño

Tamaño	Hong y Kim/Exp.	Mukherjee (caché)/Exp.	Nugteren/Exp.
100	0,959648	0,127735	2,771333
500	0,867026	0,346180	2,503565
1000	0,6685738	0,555138	2,007371
2000	0,343547	0,522911	1,031366
3000	0,215531	0,477176	1,294575
5000	0,136546	0,510144	0,820159
8000	0,163330	0,741709	1,149999
11000	0,160252	0,753818	1,410729
14000	0,144724	0,685508	1,223169
20000	0,114645	0,557943	0,922888

Cuadro B.8: Relación experimental/modelo: Estudio de modelos mediante producto matriz-vector asignando una fila por hilo - cambio de tamaño

Tamaño	Hong y Kim	Mukherjee (SUM)	Mukherjee (caché)	Nugteren	Experimental
100	1,002174	10,687826	0,905217	25,072928	6,339721
500	1,002174	10,687826	0,905217	25,072928	9,334363
1000	2,003913	21,375652	1,810435	25,072928	10,103960
2000	3,005652	32,063478	2,715652	25,072928	10,152790
3000	4,007391	42,751304	3,620870	25,072928	10,366120
5000	7,012609	74,814783	6,336522	25,072928	10,797290
8000	8,014348	117,566087	9,957391	25,072928	22,794230
11000	8,014348	160,317391	13,57826	50,164406	24,602710
14000	8,014348	203,068696	17,19913	50,164406	28,226190
20000	8,014348	288,571304	24,44087	50,164406	37,609210

Cuadro B.9: Resultados absolutos: Estudio de modelos mediante producto matriz-vector asignando un bloque de filas por hilo - cambio de tamaño

Tamaño	Hong y Kim/Exp.	Mukherjee (caché)/Exp.	Nugteren/Exp.
100	0,158079	0,142785	3,954894
500	0,107364	0,096977	2,686089
1000	0,198329	0,179181	2,481495
2000	0,296042	0,267478	2,469560
3000	0,386585	0,349298	2,418738
5000	0,649479	0,586862	2,322150
8000	0,351595	0,436838	1,099968
11000	0,325751	0,551901	2,038979
14000	0,283933	0,609332	1,777229
20000	0,213095	0,649864	1,333833

Cuadro B.10: Relación experimental/modelo: Estudio de modelos mediante producto matriz-vector asignando un bloque de filas por hilo - cambio de tamaño

B.2.2. Comparativa con el modelo desarrollado

Tamaño	Modelo (nocopy)	Experimental (nocopy)	Modelo (copy)	Experimental (copy)
100	0,288913	2,261811	0,665374	2,678189
500	0,964957	2,503722	2,833293	4,394597
1000	1,929913	3,122608	5,662861	6,196980
2000	3,538174	6,077603	11,000344	10,76836
3000	5,146435	9,687424	16,337828	17,40883
5000	8,684609	15,29107	27,334447	30,19086
8000	13,509391	16,35999	43,346897	45,93815
11000	18,655826	22,22944	59,680999	65,50686
14000	23,480609	30,76646	75,693450	85,76202
20000	33,773478	54,37095	108,36166	126,0465

Cuadro B.11: Comparativa con el modelo desarrollado en el producto matriz-vector por filas

Tamaño	Modelo (nocopy)	Experimental (nocopy)	Modelo (copy)	Experimental (copy)
100	1,033239	6,339721	1,221470	6,853849
500	1,033239	9,334363	1,967407	10,25561
1000	2,066478	10,10396	3,932952	12,04139
2000	3,099717	10,15279	6,830803	14,01587
3000	4,132957	10,36612	9,728653	16,10488
5000	7,232674	10,79729	16,557593	22,31002
8000	11,365630	22,79423	26,284383	39,05645
11000	15,498587	24,60271	36,011174	47,18826
14000	19,631543	28,22619	45,737964	58,82610
20000	27,897457	37,60921	65,191545	76,15183

Cuadro B.12: Comparativa con el modelo desarrollado en el producto matriz-vector por bloques de filas

Tamaño	Modelo (CC2.0)	Experimental (CC2.0)	Modelo (CC1.3)	Experimental (CC1.3)
16	12,946087	15,736030	29,323077	389,5327
64	15,784348	14,679180	85,446154	430,6875
128	19,568696	14,272690	160,27692	362,8528
192	23,353043	14,401000	235,10769	461,8774
256	27,137391	14,148320	309,93846	402,3986
384	34,706087	13,339910	459,60000	537,7562
512	42,274783	14,280110	609,26154	681,8846
640	49,843478	16,765700	N/A	N/A
768	57,412174	15,919190	N/A	N/A
992	70,657391	22,493800	N/A	N/A
1024	72,549565	21,083190	N/A	N/A

Cuadro B.13: Comparativa con el modelo desarrollado en el producto matriz-vector por columnas - cambio de hilos

Tamaño	Modelo (CC2.0)	Experimental (CC2.0)	Modelo (CC1.3)	Experimental (CC1.3)
100	0,583826	1,261219	5,877692	56,48224
500	2,919130	4,042973	29,388462	193,2181
1000	5,838261	5,661517	58,776923	221,6833
2000	11,676522	8,578236	117,553846	297,3744
3000	17,514783	11,64135	176,330769	382,9080
5000	29,191304	17,92935	293,884615	541,2787
8000	46,706087	26,74893	470,215385	780,8120
11000	64,220870	35,93764	646,546154	1025,195
14000	81,735652	45,08520	822,876923	1285,296
20000	116,76521	3,825281	1175,53846	1788,835

Cuadro B.14: Comparativa con el modelo desarrollado en el producto matriz-vector por columnas - cambio de tamaño

Tamaño	Modelo (CC2.0)	Experimental (CC2.0)	Modelo (CC1.3)	Experimental (CC1.3)
100	0,772057	1,780250	6,065923	56,77772
500	3,852833	5,181331	30,322164	193,4808
1000	7,703803	8,193027	60,642466	227,2291
2000	15,405744	13,782930	121,283069	304,8934
3000	23,107685	19,767040	181,923672	388,6203
5000	38,511567	30,766630	303,204878	554,5961
8000	61,617389	42,791690	485,126687	803,2687
11000	84,723212	58,054640	667,048496	1055,412
14000	107,829034	73,077810	848,970305	1297,369
20000	154,040679	103,954300	1212,813923	1848,911

Cuadro B.15: Comparativa con el modelo desarrollado en el producto matriz-vector por columnas - cambio de tamaño, con copia

B.3. Reducción

Tamaño	Memoria global	Modelo mem. global	Memoria shared	Modelo mem. shared
500000	0,802527	0,603270	0,694057	0,325228
1000000	1,648662	1,148518	1,448330	0,642894
5000000	7,692883	5,509847	6,876297	3,183569
10000000	15,35641	10,96168	13,78206	6,359583
30000000	46,00338	32,76866	41,28767	19,06330
60000000	91,99040	65,47930	82,59605	38,11905
100000000	153,5262	109,0934	137,7516	63,52666

Cuadro B.16: Comparativa con el modelo desarrollado en la reducción sin incluir la copia de datos

Tamaño	Memoria global	Modelo mem. global	Memoria shared	Modelo mem. shared
500000	2,030358	1,534594	1,827776	1,256552
1000000	4,030442	3,011165	3,454988	2,505541
5000000	24,33442	14,823075	17,47449	12,49680
10000000	37,32257	29,588130	32,42767	24,98604
30000000	99,25737	88,648014	95,18718	74,94266
60000000	200,9679	177,23801	189,1961	149,8778
100000000	349,4570	295,35795	321,8676	249,7912

Cuadro B.17: Comparativa con el modelo desarrollado en la reducción incluyendo la copia de datos

B.4. Descomposición de Cholesky

Tamaño	Experimental	Modelo
512	52,635742	113,23710
1024	345,74356	516,91061
1280	720,73986	908,78417
1536	1290,4357	1420,8357
1920	2450,3530	2316,1685
2560	5235,5161	4143,6741

Cuadro B.18: Comparativa con el modelo desarrollado en la descomposición de Cholesky de una matriz densa estructurada

B.5. Operaciones atómicas

N	Secuencial (registro)	Secuencial (global)	Atomics (C2070)	Atomics (C1060)
16	0,003283	0,003683	0,003584	0,009104
64	0,003518	0,007231	0,003696	0,010378
128	0,003258	0,011891	0,004790	0,045645
192	0,003628	0,016571	0,005661	0,066586
256	0,003283	0,021258	0,006656	0,087530
384	0,003597	0,030621	0,008541	0,129315
512	0,003254	0,039997	0,010518	0,171152
610	0,003542	0,049366	0,012371	N/A
768	0,003248	0,058736	0,014256	N/A
992	0,003547	0,075136	0,017616	N/A
1024	0,003280	0,077469	0,018058	N/A

Cuadro B.19: Comparativa del coste de una suma secuencial con su equivalente atómico

Bibliografía

[1] Kishore Kothapalli y Rishabh Mukherjee y M. Suhail Rehman y Suryakant Patidar y P. J. Narayanan y Kannan Srinathan. A performance prediction model for the cuda gpgpu platform. In Rajeev Muralidhar y Viktor K. Prasanna Yuanyuan Yang, Manish Parashar, editor, *HiPC*, pages 463–472. IEEE, 2009.

[2] J.J. Rodenas y C. Corral y J. Mas y F. Olmeda y J. Albelda. A parallel direct solver for a hierarchical h-adaptive finite element code. In *ECT*, 2010.

[3] Steven Fortune y James Wyllie. Parallelism in random access machines. In *STOC*, pages 114–118. ACM, 1978.

[4] Charles E. Leiserson y Bruce M. Maggs. Communication-efficient parallel algorithms for distributed random-access machines. *Algorithmica*, 3:53–77, 1988.

[5] Sunpyo Hong y Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In Stephen W. Keckler y Luiz André Barroso, editor, *ISCA*, pages 152–163. ACM, 2009.

[6] Sara S. Baghsorkhi y Matthieu Delahaye y Sanjay J. Patel y William D. Gropp y Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In David A. Padua y Mary W. Hall R. Govindarajan, editor, *PPOPP*, pages 105–114. ACM, 2010.

[7] C. Nugteren. Gpu model. Tech. report, Electrical Engineering Department, Eindhoven University of Technology, 2010. http://parse.ele.tue.nl/system/attachments/6/original/cnugteren_doc1.pdf.

[8] NVIDIA Corporation. Nvidia cuda, 2006-2012. URL: http://www.nvidia.com/object/cuda_home_new.html.

[9] Alexey L. Lastovetsky. *Parallel Computing on Heterogeneous Networks*. Wiley-Interscience, 2003.

[10] Electrical Engineering and University of Tennessee Computer Science Department. Blas library, 2009-2012. URL: http://icl.cs.utk.edu/magma.

[11] NVIDIA Corporation. Nvidia cuda c programming guide 4.0, 2011. Disponible en: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.

[12] Alex Fit-Florea Nathan Whitehead. Precision & performance: Floating point and ieee-754 compliance for nvidia gpus. Tech. report, NVIDIA, 2011. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf.

[13] NVIDIA Corporation. Nvidia cuda compute unified device architecture: Programming guide 1.0, 2007. Disponible en: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[14] Khronos Group. Opencl standard, 2008-2012. URL: http://www.khronos.org/opencl/.

[15] Microsoft Corporation. Directcompute application programming interface, 2009-2012. URL: http://www.nvidia.es/object/directcompute_es.html.

[16] Yossi Matias y Vijaya Ramachandran Phillip B. Gibbons. The qrqw pram: Accounting for contention in parallel algorithms. In *SODA*, pages 638–648, 1994.

[17] Yossi Matias y Vijaya Ramachandran Phillip B. Gibbons. The queue-read queue-write asynchronous pram model. In Anne Mignotte y Yves Robert Luc Bougé, Pierre Fraigniaud, editor, *Euro-Par, Vol. II*, volume 1124 of *Lecture Notes in Computer Science*, pages 279–292. Springer, 1996.

[18] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[19] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.

[20] David E. Culler y Richard M. Karp y David A. Patterson y Abhijit Sahay y Klaus E. Schauser y Eunice E. Santos y Ramesh Subramonian y Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *PPOPP*, pages 1–12, 1993.

[21] NVIDIA Corporation. Cuda c best practices guide, 2011. Disponible en: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

[22] NVIDIA Corporation. Ptx: Parallel thread execution isa version 2.3, 2011. Disponible en: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/ptx_isa_2.3.pdf.

[23] Maryam Sadooghi-Alvandi y Andreas Moshovos Henry Wong, Misel-Myrto Papadopoulou. Demystifying gpu microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246. IEEE Computer Society, 2010.

[24] John L. Hennessy y David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.

[25] Jared Hoberock y Nathan Bell. Thrust library, 2009-2011. Disponible en: http://code.google.com/p/thrust/.

[26] Mark Harris. Optimizing parallel reduction in cuda. Tech. report, NVIDIA, 2007. http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[27] Jared Hoberock y Nathan Bell. Thrust library, 2009-2011. Disponible en: http://www.cplusplus.com/reference/std/functional/binary_function/.

[28] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley-Interscience, 2002.

[29] Varios autores. Blas (basic linear algebra subprograms), 1979-2012. URL: http://netlib.org/blas/.

[30] Greg Ruetsch y Paulius Micikevicius. Optimizing matrix transpose in cuda. Tech. report, NVIDIA, 2009.

[31] V. M. Garcia y A. González y C. Gonzalez y F. J. Martínez-Zaldívar y C. Ramiro y S. Roger y A. M. Vidal. The impact of gpu/multicore in signal processing: A quantitative approach. *Waves*, 3:96–106, 2011.

[32] C. González y A. M. Vidal. A performance estimation model for parallel heterogeneous architectures. Artículo, -, 2012. en proceso de elaboración.

[33] NVIDIA Corporation. Cuda toolkit 4.0: Cublas library, 2005-2011. Disponible en: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUBLAS_Library.pdf.

[34] NVIDIA Corporation. Cuda cusparse library, 2005-2011. Disponible en: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUSPARSE_Library.pdf.

[35] P. Du y J. Dongarra S. Tomov, R. Nath. Magma users' guide version 0.2. Users' guide, Electrical Engineering and Computer Science Department, University of Tennessee, 2009. http://icl.cs.utk.edu/projectsfiles/magma/docs/magma-v02.pdf.

[36] NVIDIA Corporation. Cuda toolkit 4.0: Thrust quick start guide, 2011. Disponible en: http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/Thrust_Quick_Start_Guide.pdf.