Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXIX, págs. 2951-2967 (artículo completo)
Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (Eds.)
Buenos Aires, Argentina, 15-18 Noviembre 2010

RESOLUCIÓN DE MATRICES TRI-DIAGONALES UTILIZANDO UNA TARJETA GRÁFICA (GPU) DE ESCRITORIO

Pablo Alfaro, Pablo Igounet and Pablo Ezzatti

Centro de Cálculo, Instituto de Computación, Facultad de Ingeniería - Universidad de la Republica, Julio Herrera y Reissig 565, 11.300 Montevideo, Uruguay
{palfaro,proymb,pezzatti}@fing.edu.uy, http://www.fing.edu.uy/inco

Palabras Clave: Matrices tri-diagonales, Reducción Cíclica, GPU, CUDA.

Resumen.

La resolución de sistemas lineales tri-diagonales es una de las etapas más costosas, en cuanto a costo computacional, al abordar diferentes problemáticas. En arquitecturas convencionales se dispone del algoritmo de Thomas para la resolución de este tipo de sistemas lineales. Sin embargo, la estructura del algoritmo es intrínsecamente serial, no siendo una buena opción para aplicar técnicas de programación paralela. En los últimos años, ha crecido enormemente el uso de hardware secundario para acelerar el cómputo de problemas de propósito general, y en particular el uso de las GPUs (procesadores gráficos). En este trabajo se presenta el estudio e implementación del algoritmo de reducción cíclica (Cyclic Reduction) para resolver sistemas tri-diagonales en GPU. Las diferentes estrategias implementadas fueron probadas sobre un computador dual-core a 2.5 GHz y 2 GB de memoria RAM conectado a una GPU NVIDIA 9800 GTX+ mostrando importantes valores de aceleración para la propuesta.

1 INTRODUCCIÓN

La resolución de sistemas lineales tri-diagonales es una de las etapas más costosas, en cuanto a costo computacional, al abordar diferentes problemas. En particular, al realizar simulaciones con modelos numéricos que implican la resolución de ecuaciones diferenciales parciales sobre grillas estructuradas con estrategias ***ADI*** (*Alternating Direction Implicit*) es común tener que resolver varios sistemas tri-diagonales para calcular cada paso de tiempo.

Para la resolución de sistemas tri-diagonales sobre arquitecturas convencionales se dispone del método de Thomas (Conte and De Boor, 1980), una reordenación de la factorización LU que saca provecho del patrón tri-diagonal de la matriz a resolver para conseguir la resolución con un orden de cómputos lineal en la dimensión de la matriz. Sin embargo, para realizar simulaciones, incluso de periodos cortos, es común tener que resolver cientos o incluso miles de estos sistemas. Esto implica que la suma de los tiempos de resolución de los sistemas sea relevante y por ende resulte interesante acelerar dicha etapa aplicando técnicas de computación de alto desempeño.

En los últimos años, las tarjetas gráficas o co-procesadores gráficos (GPU) han experimentado una evolución explosiva. La evolución no solo fue cuantitativa (capacidad de cómputo) sino que en gran medida ha sido cualitativa (han pasado a computar etapas del proceso gráfico que antes las computaba la CPU). Especialmente, desde el lanzamiento de CUDA (*Compute Unified Device Architecture*) (NVIDIA, 2008) por parte de NVIDIA en 2007 las GPUs se han transformado en verdaderos multi-procesadores capaces de ejecutar algoritmos paralelos mediante el paradigma ***SPMD*** (Single Program - Multiple Data). Este hecho, ha motivado que muchos científicos de diversas áreas del conocimiento busquen la utilización de las GPU para atacar problemas generales (bases de datos, computación científica, etc.). En particular, en el área de álgebra lineal numérica (ALN) diversos trabajos se han presentado mostrando importantes aceleraciones a los métodos mediante el uso de GPUs.

El gran éxito que han mostrando diferentes trabajos acelerando el cómputo de operaciones de ALN y el bajo costo económico que representan las GPUs motivan este trabajo, donde se presenta la aceleración de la resolución de sistemas lineales tri-diagonales utilizando una GPU de escritorio. La propuesta se basa en el método de Reducción Cíclica (Cyclic Reduction), en el trabajo se presentan distintas implementaciones del método, utilizando CUDA, realizadas buscando explotar las características del hardware empleado (por ejemplo velocidades de los niveles de memoria, estrategias de cómputo, técnicas híbridas, etc.). La evaluación experimental de las versiones desarrolladas, trabajando con una computadora dual-core a 2.5 GHz y 2 GB de memoria RAM conectado a una GPU NVIDIA 9800 GTX+, muestra interesantes disminuciones en los tiempos de ejecución al comparar con el método de Thomas en CPU.

Lo que resta del artículo se estructura de la siguiente manera. En la Sección 2 se describen los conceptos básicos de la utilización de GPUs para propósito general (GPGPU), con particular detalle en la arquitectura CUDA. La descripción del método de Reducción Cíclica y los trabajos relacionados se pueden encontrar en la Sección 3. En la Sección 4 se presentan y explican las implementaciones desarrolladas, mientras que en la Sección 5 se resumen los distintos experimentos realizados para evaluar las implementaciones. Por último, en la Sección 6, se resumen las conclusiones arribadas durante el trabajo y las líneas de trabajo futuro planteadas.

2 GPGPU

Desde el año 2000, la utilización de GPU para resolver problemas de propósito general ha ido en aumento (en el sitio http://gpgpu.org/ se puede profundizar sobre el tema). Sin embargo, en esa primera etapa la principal limitante para poder explotar las capacidades de las GPUs era la necesidad de conocer el proceso gráfico y la necesidad de mapear los distintos problemas a resolver a conceptos de computación gráfica. Esta limitante fue abatida en gran medida con el lanzamiento de CUDA, en los comienzos del año 2007, por parte de NVIDIA, permitiendo manejar las GPUs como verdaderos multiprocesadores. En lo que resta de la sección se describen las principales características de dicha arquitectura. En el libro de Kird and Mwv (2010) se puede encontrar material adicional sobre la temática.

2.1 Modelo de Ejecución

CUDA es una arquitectura para el cómputo de propósito general enfocada hacia el cálculo masivamente paralelo utilizando la capacidad de procesamiento de las tarjetas gráficas de NVIDIA. Para lograr explotar de manera óptima los recursos de la GPU, CUDA provee un nuevo conjunto de instrucciones accesibles a través de lenguajes de alto nivel como son FORTRAN y C, y un nuevo modelo de ejecución que se describe en la Figura 1.

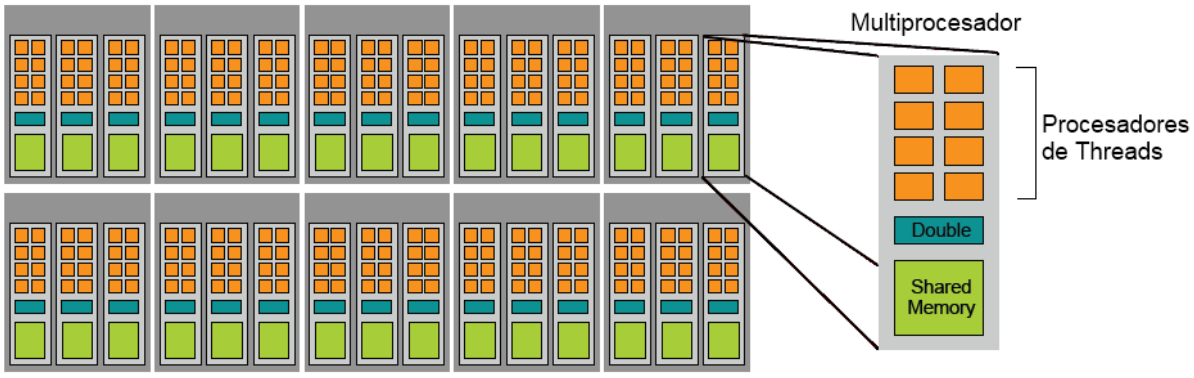

Figura 1: Modelo de Ejecución de CUDA.

Cada GPU cuenta con un conjunto de multiprocesadores compuestos a su vez por un conjunto de procesadores orientados a la ejecución de threads. La noción

fundamental en el nuevo modelo de ejecución es que cada multiprocesador ejecutará en cada uno de sus procesadores de threads el mismo conjunto de instrucciones, pero sobre distintos datos, es decir paralelismo bajo el paradigma de programación ***SPMD***.

Los programas a ejecutar en la GPU se denominan kernels y se organizan en una grilla de threads de ejecución con coordenadas *x, y, z*. A su vez los threads se agrupan en bloques con coordenadas *x* e *y*. Estas coordenadas se utilizaran para identificar el conjunto de datos sobre el que se quiere que actúe cada thread. Cada bloque será ejecutado en un multiprocesador independiente (esto es: no puede haber ni comunicación de datos ni sincronización entre los bloques) y cada thread en un procesador de threads (entre threads si puede haber sincronización y comunicación). Al conjunto de threads que ejecutan en el mismo instante de tiempo en un mismo bloque se lo denomina warp. El tamaño del warp es igual a la cantidad de procesadores de thread en los multiprocesadores. Organizar la ejecución de esta manera permite escalar sin necesidad de recompilar el programa dado que siempre se utilizaran todos los multiprocesadores disponibles para la ejecución de los bloques y todos los procesadores de threads en ellos para la ejecución de los threads.

2.2 Jerarquía de Memoria

El acceso a memoria por parte de los threads es un componente fundamental en los tiempos totales de ejecución de los programas CUDA, incluso pudiendo ser mayor que el tiempo de cálculo propiamente. Las GPUs de NVIDIA accedidas mediante la arquitectura CUDA cuentan con una jerarquía de memoria en la cual los accesos deben ser provistos explícitamente (salvo a los registros), esto es: el programador debe indicar que tipo de memoria desea utilizar en cada momento. En la Figura 2 se presenta un diagrama con los tipos de memoria, su distribución entre las unidades de ejecución y a continuación una breve descripción de las mismas.

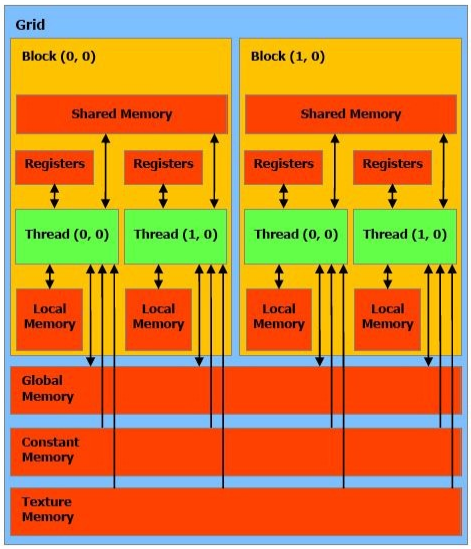

Figura 2: Jerarquía de Memoria de CUDA.

Memoria Global: es una memoria común a todos los bloques de threads que ejecutan dentro de una tarjeta gráfica. Es la memoria de mayor capacidad, y los datos almacenados persisten durante la ejecución de múltiples kernels. La memoria global es la más lenta de las memorias y no posee caché (en tarjetas de compute capability menores a 2.0, este concepto se describe más adelante en el trabajo). Es la única memoria de la tarjeta gráfica que es visible desde el host (CPU). Por lo tanto todos los datos de entrada y salida tienen obligatoriamente que pasar por memoria global. Se puede obtener un mejor rendimiento de esta memoria si se emplea la técnica denominada ***coalescing*** que implica que threads consecutivos accedan a posiciones de memoria consecutivas, realizando de esta forma múltiples accesos en el tiempo de uno solo.

Memoria Compartida: la memoria compartida es una memoria a la cual solo pueden acceder los threads que corren dentro un mismo bloque, su capacidad es menor a la memoria global y su ciclo de vida es de una ejecución de kernel. La memoria compartida si posee caché y el costo de acceso es del orden de 100 veces menor que el acceso a memorias globales o locales.

Memoria Local: esta memoria es privada de cada thread. Tiene la misma velocidad que la memoria global y su ciclo de vida es de una ejecución de kernel.

Registros: son privados de cada thread. Es la memoria más rápida pero su tamaño es limitado y su uso es determinado por el compilador.

Memoria Constante: esta memoria es accesible por todos los threads. Es de solo lectura para la GPU, pero la CPU puede escribir en ella. Reside en memoria de dispositivo por lo que tiene el mismo tiempo de acceso que la memoria global, pero está memoria si posee caché, logrando disminuir los tiempos de acceso en caso de producirse un caché hit.

Memoria de Texturas: esta memoria es accesible por todos los threads. Está optimizada para aprovechar la localidad espacial de datos en dos dimensiones y posee un caché pero las escrituras a memoria de texturas no se actualizan del caché hasta finalizar la ejecución del kernel en curso, por lo que no se puede acceder a través de memoria de texturas a un dato en el mismo kernel una vez que fue modificado.

2.3 Compute Capabilities

El número de threads totales, el tamaño del warp, los tamaños de las distintas memorias, así como otras características dependen de cada tarjeta gráfica y de la gama a la que esta pertenece. Para clasificar estas cualidades NVIDIA utiliza el concepto de compute capability que se compone por un número mayor y un número menor de la revisión de la arquitectura de la tarjeta. Entre las sucesivas versiones se encuentran mejoras a las características mencionadas anteriormente así como nuevas funcionalidades. Actualmente se agrupan en 3 rangos de compute capabilities, las tarjetas de 1.0-1.1, tarjetas de 1.2-1.3 y las tarjetas de 2.0. Nuestro trabajo se centra en las tarjetas de más baja gama siendo la 9800 GTX+ una tarjeta de compute

capability 1.1. Dentro de las clases 1.2-1.3 se encuentran las GPUs NVIDIA Tesla mientras que las de clase 2.0 se reserva para las recientemente lanzadas GPUs con arquitectura Fermi.

3 REDUCCIÓN CÍCLICA

El método tradicional para resolver matrices tri-diagonales es el método de Thomas, una variante de la factorización LU, optimizada para sacar partido de la estructura tri-diagonal de las matrices sobre las que se trabaja. El método posee un orden lineal en la dimensión de la matriz, lo que lo hace muy eficiente para resolver en arquitecturas tradicionales este tipo de sistemas. Sin embargo, las dependencias de datos que establece el algoritmo dificultan la aplicación de estrategias de paralelismo.

Por las razones expuestas en el párrafo anterior en este trabajo se tomó el algoritmo de Reducción Cíclica como base para los desarrollos.

El método de Reducción Cíclica (RC) resuelve el problema $Ax = b$ reduciendo sucesivamente la matriz A, combinando las filas pares con sus dos adyacentes (la anterior y la posterior) eliminando la mitad de las incógnitas en cada paso, obteniendo una matriz reducida y aplicando nuevamente el método hasta que el problema a resolver sea de un tamaño pequeño. Una vez alcanzado este problema reducido se resuelve el sistema (por ejemplo mediante el método de Thomas) despejando los valores de un conjunto de incógnitas. Luego se sustituye las incógnitas despejadas en las ecuaciones no resueltas, obteniendo el valor de las restantes incógnitas de la ecuación y volviendo a sustituir en el siguiente conjunto de ecuaciones no resueltas hasta obtener el valor de todas las incógnitas del problema.

A continuación se explica en detalle la etapa de reducción del algoritmo para el primer paso de una matriz de 8x8.

	1	2	3	4	5	6	7	8
1	d1	u1						
2	l̂2	d2	û2					
3		l3	d3	u3				
4			l̂4	d4	û4			
5				l5	d5	u5		
6					l̂6	d6	û6	
7						l7	d7	u7
8							l̂8	d8

Figura 3: Eliminación de valores no diagonales en reducción

Primero, se eliminan los valores no diagonales marcados con ^ de las filas pares realizando una combinación lineal de cada fila par i con sus dos filas adyacentes *i-1* e

i+1, según la ecuación (1)[1]:

$$fila_i := fila_i - \frac{l_i}{d_{i-1}} * fila_{i-1} - \frac{u_i}{d_{i+1}} * fila_{i+1} \qquad (1)$$

	1	2	3	4	5	6	7	8
1	d1	u1						
2	0	d2'	0	u2'				
3		l3	d3	u3				
4		l4'	0	d4'	0	u4'		
5				l5	d5	u5		
6				l6'	0	d6'	0	u6'
7						l7	d7	u7
8						l8'	0	d8'

Figura 4: Generación de nuevos valores de banda en reducción

Esta combinación lineal genera los valores $\overline{l'}$ y $\overline{u'}$ apartados de la diagonal y los nuevos valores de la diagonal $\overline{d'}$.

	1	2	3	4
1	d2'	u2'		
2	l4'	d4'	u4'	
3		l6'	d6'	u6'
4			l8'	d8'

Figura 5: Matriz reducida

Con estos valores se construye una nueva matriz tri-diagonal compuesta por la mitad de las incógnitas originales y reduciendo efectivamente el tamaño del problema. Aplicando nuevamente la reducción se obtiene un sistema 2x2 en el que participan x4 y x8 que es resoluble de forma muy sencilla por cualquier método tradicional. Luego, sustituyendo hacia atrás estos valores en las ecuaciones 2 y 6 se obtienen los valores de x2 y x6 y sustituyendo estos 4 valores en las ecuaciones anteriores se obtiene el valor del resto de las incógnitas.

3.1 Trabajos sobre GPUs relacionados

Desde los primeros trabajos al comienzo de la década, que intentaban explotar ingeniosamente el poder de las tarjetas gráficas para acelerar operaciones de ALN entre los que se destacan los esfuerzos de Larsen and McAllister (2001) para

[1] Para la fila 1 y la fila n, se descartan los términos que incluyen valores que quedan fuera de la matriz.

multiplicar matrices, Krüger and Westermann (2003) para resolver aplicaciones de ALN y el estudio de la factorización LU presentado por Galoppo et al. (2005), hasta los trabajos presentados recientemente que muestran importantes resultados incluso superando el estado del arte en computación de alto desempeño aplicadas a la resolución de problemas de ALN, tanto el hardware como la algoritmia para utilizar las GPUs han evolucionado notoriamente. Algunos de los trabajos más destacados de esta última etapa del desarrollo de la utilización de GPUs para ALN son: el estudio de aceleración de operaciones de BLAS en el trabajo de Barrachina et al. (2008), la multiplicación de matrices y factorización QR presentado por Volkov and Demmel (2008), la inversión de matrices en el trabajo de Benner et al. (2009), así como los esfuerzos de Tomov et al. (2009) por desarrollar una biblioteca que explote las capacidades de las GPUs para resolver problemas de ALN densa.

Además de trabajos en el área de GPU y ALN relacionados en el último año se han presentado algunos esfuerzos centrados directamente en la resolución de matrices tri-diagonales, a continuación se presentan las principales características de dichos trabajos.

En el trabajo de Zhang et al. (2009), se muestran distintas estrategias para resolver simultáneamente 512 matrices de tamaños pequeños (hasta 512 incógnitas), realizando experimentos con implementaciones de Reducción Cíclica y las variantes de dicho algoritmo Reducción Cíclica Paralela (PCR - Parallel Cyclic Reduction) y Recursive Doubling (RD), así como algoritmos híbridos utilizando Reducción Cíclica para los primeros pasos donde el número de ecuaciones es más grande y empleando PCR o RD para resolver los últimos pasos de la iteración, ya que estos permiten explotar mejor el paralelismo en problemas pequeños. En su caso, los mejores resultados se obtuvieron con el algoritmo de Reducción Cíclica y la versión híbrida con PCR.

Por otra parte en el artículo de Sakharnykh (2009), se resume el estudio de la aplicación de los métodos de resolución de matrices tri-diagonales presentados en el trabajo de Zhang et al., exponiendo los beneficios obtenidos a nivel de performance por utilizar accesos a memoria ***coalesced***.

En base al estudio realizado de los trabajos propuestos en el área y los resultados presentados, las diferentes implementaciones desarrolladas en este trabajo se centran en el algoritmo de Reducción Cíclica profundizando en el estudio de aplicación de técnicas híbridas y el aprovechamiento de las características de la arquitectura CUDA. Sin embargo, el objetivo es acelerar la resolución de un único sistema lineal (en cada momento), potencialmente grande, en contraposición con otros enfoques que buscan acelerar la resolución de varios sistemas lineales pequeños al mismo tiempo.

4 IMPLEMENTACIONES

En esta sección se describen los métodos implementados para la resolución de

matrices tri-diagonales utilizando la GPU, se detallan la motivación para cada desarrollo y las prestaciones de la arquitectura que buscan explotar. En todos los casos se intenta alcanzar un máximo aprovechamiento de la arquitectura de la GPU utilizando un método paralelizable como lo es Reducción Cíclica, de forma de explotar los múltiples procesadores que proveen las mismas.

4.1 Reducción Cíclica Simple en GPU (*RC-GPU*)

Esta versión fue una primera aproximación a la resolución de una matriz tridiagonal utilizando Reducción Cíclica en GPU. El enfoque se basó en explotar el paralelismo sin hacer hincapié en las características particulares de CUDA. Al comienzo del algoritmo se copian los vectores del espacio de memoria de CPU a GPU, se resuelve el problema en GPU y posteriormente se copia de GPU a CPU el resultado.

La implementación del algoritmo en GPU se realizó reutilizando el espacio de memoria asignado a la matriz, esto es: todas las filas resultado de los cálculos ocuparán, una vez que sean calculadas, el lugar de una de las filas de origen. Esto se logra realizando operaciones sobre los índices para mantener la estructura lógica de la matriz sobre la misma estructura física.

Se utiliza una sola de las dimensiones de las grillas de threads y bloques. El número de threads utilizados se fijó en 256 de forma de que sea múltiplo del tamaño del warp y lograr aprovechar cada procesador de threads al máximo. Este número es mayor al tamaño del warp permitiendo de esta forma que el programa sea escalable a versiones futuras sin necesidad de ser recompilado. La cantidad de bloques necesaria para resolver el sistema se calcula a partir del tamaño de la matriz y la cantidad de threads en base a la Ecuación (2), donde $n_{subProblema}$ es el número de ecuaciones a reducir de el paso actual.

$$nBloques = \left\lceil {}^{n_{subProblema}}/_{nThreads} \right\rceil \qquad (2)$$

Cada bloque tendrá un conjunto de threads que resolverán una fila de la matriz cada uno. En la etapa de reducción, la fila correspondiente a un thread es aquella fila a la que se le van a eliminar las incógnitas. En la etapa de sustitución, la fila del thread es la fila sobre la que se van a sustituir las incógnitas para resolver el nuevo juego de incógnitas.

Luego la resolución se hace en un único bloque con un solo thread para evitar la copia de memoria entre el GPU y la CPU.

El algoritmo tiene como entradas *l, d, u*, vectores de reales conteniendo las tres bandas de la matriz *A*. En el vector *l* se almacenan los coeficientes de la diagonal inferior, en *d* la diagonal principal y en *u* la diagonal superior y un cuarto vector *b* conteniendo los términos independientes del sistema de ecuaciones. Los cuatro vectores de entrada poseen un tamaño *n* igual a la dimensión de la matriz a resolver. Como salidas el algoritmo tiene el vector *x* con los valores resueltos de las incógnitas.

En la Figura 6 se expone un pseudocódigo del algoritmo.

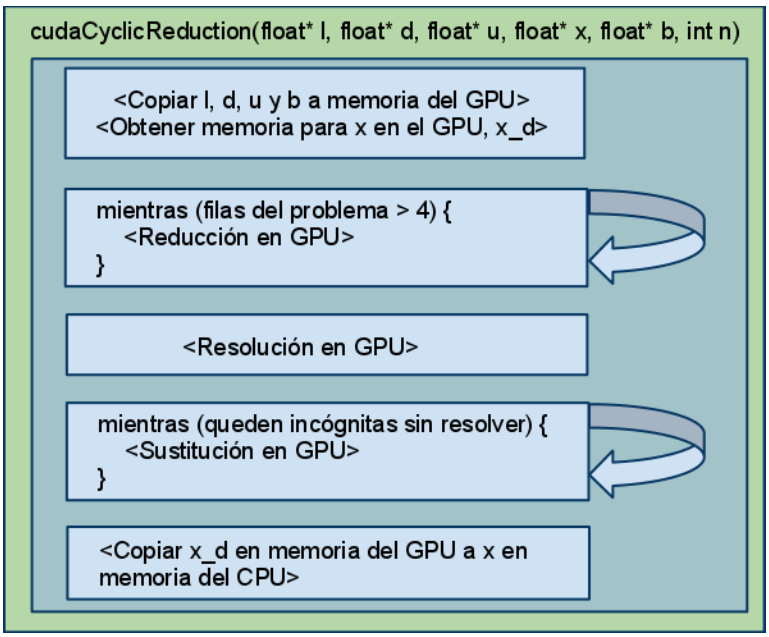

Figura 6: Pseudocódigo del método de reducción cíclica en GPU.

La reducción y la sustitución se hacen sobre múltiples filas en cada paso de la iteración, de esta forma se resuelve el sistema tri-diagonal utilizando un método paralelo que aprovecha la arquitectura de la GPU.

4.2 Reducción Cíclica Híbrido en GPU-CPU (*RC-GPU-H*)

En los últimos pasos de la etapa de reducción (y los primeros de sustitución) en el algoritmo de Reducción Cíclica no se logra un buen nivel de paralelismo (hay menos filas por reducir/sustituir que procesadores de threads) no aprovechándose al máximo la arquitectura de la GPU. Por esta razón se intentó utilizar la CPU, cortando la reducción cuando la matriz reducida alcanza una dimensión dada y utilizando el método de Thomas para resolver el problema reducido.

El nuevo algoritmo consta de los siguientes pasos (en la Figura 7 se presentan en forma gráfica):

1. Reducción mediante RC en GPU hasta que la matriz reducida alcance un tamaño prefijado (aquel tamaño en que no se aprovechan todos los multiprocesadores de la GPU).
2. Copiar a CPU los datos calculados en GPU (dado que estos datos son dispersos, la copia de todo el bloque de memoria es muy costosa, pero CUBLAS provee las funciones *setVector* y *getVector* para la copia de valores espaciados)
3. Resolución utilizando el algoritmo de Thomas implementado en CPU para la matriz resultante de los primeros pasos del RC.
4. Copiar resultados del cálculo de Thomas desde CPU hacia el GPU

5. Sustitución mediante RC hasta finalizar de resolver todas las incógnitas

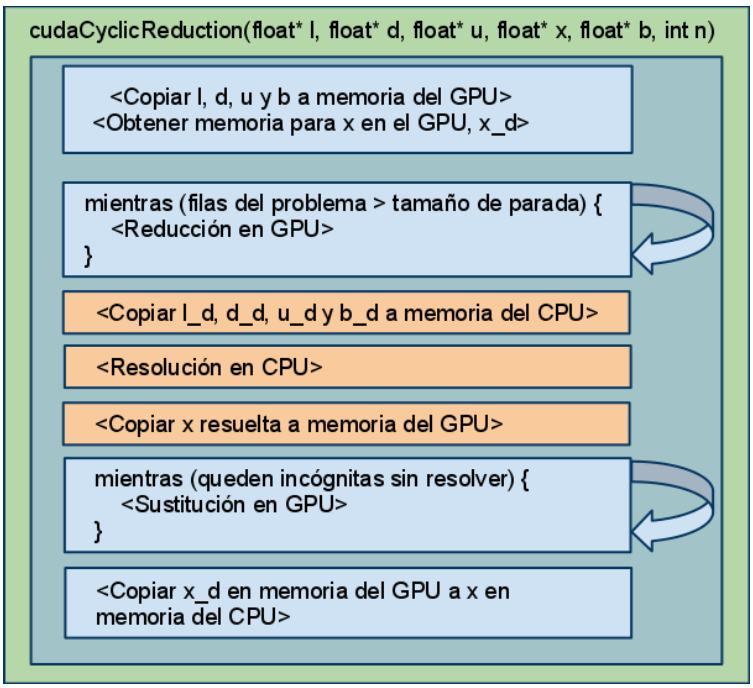

Figura 7: Pseudocódigo del método de reducción cíclica hibrido en GPU/CPU.

De esta forma la CPU resuelve el problema reducido cuando el tamaño de este no permite explotar el paralelismo de la GPU ya que el número de ecuaciones a reducir es menor que las que pueden ser procesadas paralelamente.

4.3 Reducción Cíclica en GPU con Memoria Compartida (*RC-GPU-S*)

Dado que una de las mayores limitante en el uso de la GPU es el tiempo de acceso a memoria parece razonable aprovechar al máximo las memorias más rápidas. En el caso de la arquitectura de CUDA, como ya se presentó en la Sección 2, las memorias con menor penalización para el acceso son los registros y la memoria compartida. Los primeros son manejados por el compilador, pero la memoria compartida debe ser explícitamente manejada por el programador.

Para determinar que variables es útil mantener en memoria compartida, se realizó un estudio del código del método *RC-GPU*, de forma de obtener los accesos a cada posición de memoria. Para cada fila i a la que se aplica la reducción/sustitución, los accesos a las bandas inferior (l), diagonal (d) y superior (u), a los términos independientes (b) y al vector de incógnitas (x) se pueden ver en las tablas 1 y 2. Cada paso de la resolución, involucra la fila a reducir (fila i) y sus dos filas adyacentes, por lo que se analiza el número de lecturas y escrituras para los valores de la fila i, $i+1$ e $i-1$ en cada paso.

Acción	*l[i]*	*l[i-1]*	*l[i+1]*
lectura	1	1	1
escritura	1		
	d[i]	*d[i-1]*	*d[i+1]*
lectura	1	1	1
escritura	1		
	u[i]	*u[i-1]*	*u[i+1]*
lectura	1	1	1
escritura	1		
	b[i]	*b[i-1]*	*b[i+1]*
lectura	1	1	1
escritura	1		

Tabla 1: Accesos/reúso de variables en la etapa de reducción de RC para la fila *i*.

Acción	*x[i]*	*x[i+1]*	*x[i-1]*
lectura	1	1	1
escritura	1		
	l[i]	*d[i-1]*	
lectura	1	1	
escritura			
	b[i]	*u[i-1]*	
lectura	1	1	
escritura			

Tabla 2: Accesos/reúso de variables en la etapa de sustitución de RC para la fila *i*.

Las filas *i+1* e *i-1* solo son accedidas en forma de lectura. Como a su vez los valores de la filas *i+1* e *i-1* son compartidos en la resolución de las ecuaciones reducidas por los threads contiguos (los que resuelven las filas *i+2* e *i-2*) los accesos a estos valores se realizan dos veces, por lo que es útil moverlos a memoria compartida. De esta forma se reducen los accesos a memoria global, sustituyendo 2 accesos a memoria global por un acceso a memoria global, dos lecturas de memoria compartida y una escritura a la misma. Teniendo en cuenta que los accesos a memoria compartida son del orden de 100 veces menores, se disminuye el tiempo de acceso a memoria a prácticamente la mitad.

Esta versión, al igual que las versiones anteriores, al comienzo de su ejecución copia los vectores del espacio de memoria de la CPU al de la GPU (memoria global) y luego se sigue una estrategia similar a la utilizada en la versión Simple, pero, en base al estudio anterior, antes de sustituir cada thread copia las filas *i* e *i+1*, y espera que el thread anterior (fila *i-2*) copie la fila *i-1*. Al finalizar su copia cada thread debe

sincronizarse con los demás threads del bloque antes de empezar a realizar los cálculos para asegurar que todos los datos están disponibles en la memoria compartida.

4.4 Reducción Cíclica en GPU con Memoria Compartida y Coalescing (*RC-GPU-C*)

Si bien con la versión anterior se logra una mejora en el tiempo de acceso a memoria, este costo sigue siendo determinante en el costo total del algoritmo. Por esta razón la siguiente versión busca utilizar la técnica de *coalescing* para mejorar la relación tiempo de cómputo/tiempo de acceso a memoria.

Este método trabaja sobre la base del algoritmo *RC-GPU-S*, pero reordenando las filas de salida de cada operación (reducción/sustitución) para que en las sucesivas iteraciones la lectura y escritura de los datos pueda aprovechar el *coalescing* a medida que las filas que le corresponden a un thread se van separando.

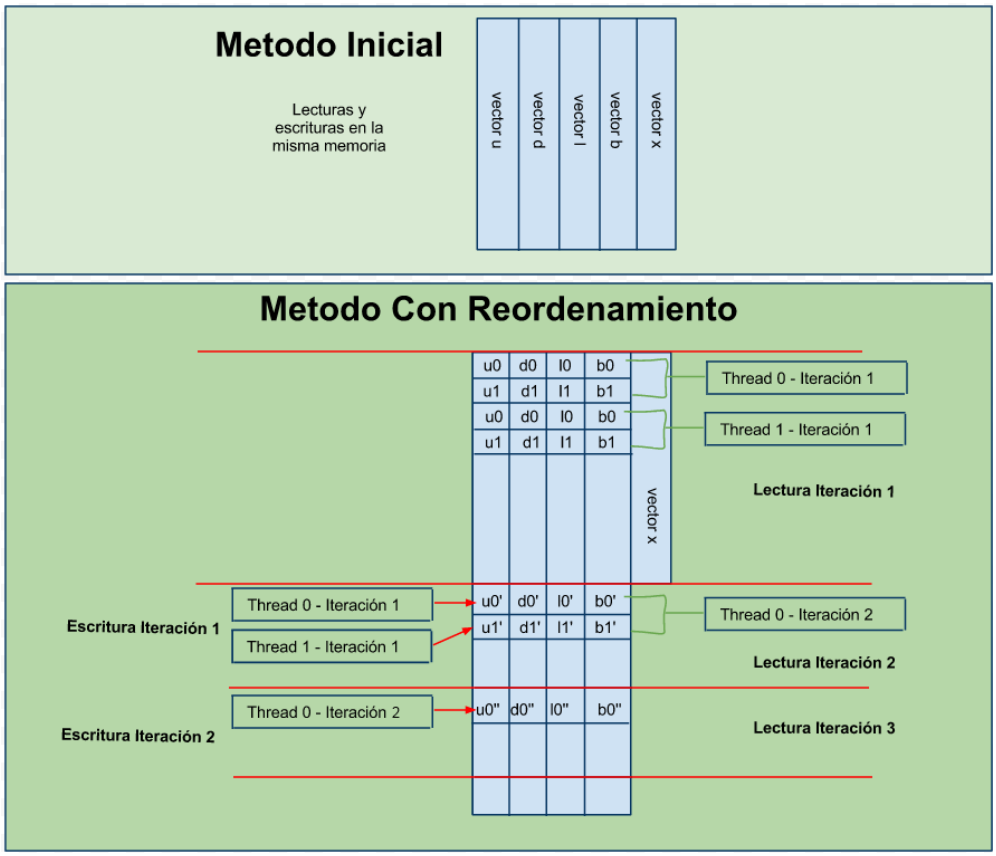

Figura 8: Patrón de acceso a memoria.

Como se muestra en la Figura 8 se reserva el doble de espacio en memoria y en cada iteración se lee de una matriz y se escribe en otra, de modo de poder escribir las filas resultantes de cada iteración en memoria contigua, aprovechando esto también en la lectura de la siguiente iteración. De esta forma se consigue que en la ejecución de un warp todas las lecturas y escrituras a memoria sean consecutivas (en el caso de

la lectura con huecos de una palabra, pero de todas formas logrando coalescing en tarjetas con compute capabilities 1.2 o superior).

Para ilustrar la diferencia de como son realizadas las lecturas en los dispositivos de distintos rangos de compute capabilities en la Figura 9 se muestra cómo se aplica coalescing a una lectura en el caso del algoritmo propuesto.

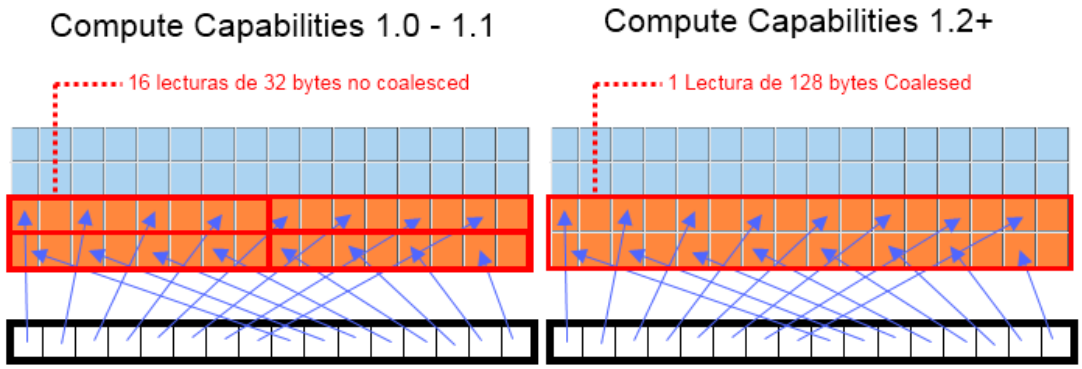

Figura 9: Coalescing según compute capability.

En el caso de compute capabilities 1.0 y 1.1 como los accesos de threads consecutivos se encuentran separados por una palabra no se aplica coalescing y las lecturas se serializan en 16 lecturas individuales (el tamaño mínimo de una lectura es de 32 bytes). En el caso de compute capabilities 1.2 o superiores los accesos a memoria de threads consecutivos no precisan ser contiguos, basta con que se encuentren en un mismo segmento de 32, 64 o 128 bytes. En el caso del algoritmo propuesto la lectura es de 16 valores, uno por cada thread del warp, de 4 bytes separados 4 bytes entre si, por lo que trayendo el segmento entero se resuelven todas las lecturas en un único acceso a memoria de 128 bytes.

5 EXPERIMENTOS

5.1 Plataforma de experimentación

La plataforma de experimentación consistió en una computadora con procesador Intel dual-core a 2.5 GHz y 2 GB de memoria RAM conectada a una tarjeta gráfica NVIDIA 9800 GTX+ (128 procesadores a 738 MHz y 512 MB de memoria).

Se trabajó sobre el sistema operativo Linux y se utilizaron los compiladores gcc versión 4.3.1 y NVCC 2.1. Se utilizó la versión 2.1 de CUDA.

5.2 Experimentos

Todos los experimentos se realizaron sobre matrices generadas con coeficientes aleatorios y utilizando el equipamiento en forma dedicada. Además los tiempos de ejecución evaluados incluyen en todos los casos el tiempo de trasferencia de los datos a GPU y la obtención de los resultados desde la GPU.

Para la evaluación de los algoritmos desarrollados además de las cuatro versiones presentadas anteriormente, se implementó una versión del algoritmo de Thomas en

CPU (*Thomas*) y una versión del método de Reducción Cíclica sobre CPU utilizando OpenMP para explotar el paralelismo en arquitecturas multicore (*RC-CPU*).

En la Figura 10 se presentan los tiempos de ejecución de las distintas versiones implementadas para la resolución de sistemas lineales tri-diagonales de diferentes dimensiones expresados en *Ks* (1024). Para la versión RC-GPU-H primero se estudió el mejor valor para cortar la resolución en GPU y pasar a computar en CPU, el mejor valor encontrado fue 256. Sin embargo, como se puede apreciar en la Figura 10, la utilización de este tipo de estrategias híbridas no reportó mejoras con respecto a la versión original (*RC-GPU*).

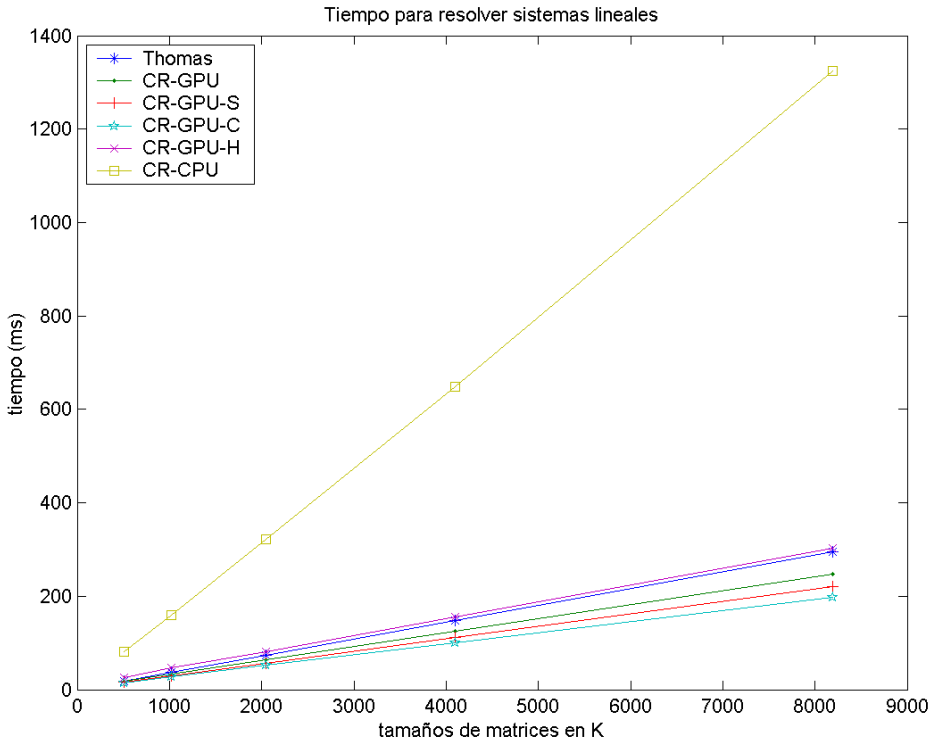

Figura 10: Tiempos de ejecución de las distintas versiones para resolver sistemas lineales.

Al observar los resultados se puede apreciar el buen desempeño del método de Thomas al ejecutar en CPU superando el obtenido por Reducción Cíclica también en CPU. Además se puede constatar los beneficios de utilizar la GPU para implementar el método de Reducción Cíclica ya que todas las versiones que utilizan GPU superan notoriamente la versión de Reducción Cíclica en CPU, inclusive superando a la implementación del método de Thomas en CPU para las distintas matrices evaluadas. En este sentido, la Tabla 3 resume el speed up alcanzado por las distintas versiones de Reducción Cíclica sobre GPU al compararlas con el algoritmo de Thomas en CPU.

ks	***RC-GPU***	***RC-GPU-S***	***RC-GPU-C***
512	1.052	1.193	1.245

1024	1.122	1.238	1.362
2048	1.164	1.294	1.432
4096	1.182	1.320	1.467
8192	1.193	1.335	1.489

Tabla 3: Speed up de las versiones de Reducción Cíclica en GPU.

Otro aspecto que se desprende de los resultados es las diferencias en el desempeño de las distintas versiones que emplean la GPU, pudiéndose constatar los beneficios de utilizar la memoria compartida así como las capacidades de coalescing, ya que esto permitió una mejora de hasta un 12% en la versión *RC-GPU-S* y del 25% para la versión *RC-GPU-C* con respecto a la primera versión implementada sobre GPU (*RC-GPU*).

6 CONCLUSIONES Y TRABAJOS FUTUROS

La principal conclusión del trabajo es que las GPUs son un plataforma altamente útil para acelerar el cómputo de operaciones de álgebra lineal, inclusive para la resolución de sistemas tri-diagonales que posee una baja relación cómputos/datos (lineal), ya que se obtuvieron mejoras de hasta un 1,5*x* utilizando hardware gráfico de bajo costo.

En cuanto a las prestaciones de las GPUs se puede destacar la importancia de explotar el uso de la memoria compartida y de las técnicas de coalescing. Esto permitió acelerar en un 12% y en un 25% el algoritmo desarrollado originalmente.

Como líneas de trabajo futuro se plantea evaluar el desempeño de versiones en doble precisión, ya sea trabajando directamente con números en doble precisión o empleando técnicas de refinamiento iterativo. Evaluar los desarrollos en GPUs de mejores prestaciones en particular sobre modelos Tesla y Fermi.

Por último, es de interés acoplar los métodos desarrollados para resolver sistemas tri-diagonales en modelos numéricos de mayor porte, en particular en modelos de volúmenes finitos que emplean técnicas ADI para la resolución de las ecuaciones diferenciales parciales.

REFERENCIAS

Barrachina, S., Castillo, M., Igual, F., Mayo, R., Quintana-Ortí, E. and Quintana-Ort, G. Evaluation and tuning of the level 3 CUBLAS for graphics processors. *Technical report, Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Campus de Riu Sec, s/n 12.071 - Castellón, España*, 2008.

Benner, P., Ezzatti P., Quintana-Ortí, E. and Remón, A., Using Hybrid CPU-GPU Platforms to Accelerate the Computation of the Matrix Sign Function. *Euro-Par Workshops 2009*: 132-139, 2009.

Conte, S. D. and Boor, C. W. Elementary Numerical Analysis: an Algorithmic Approach. *3rd. McGraw-Hill Higher Education,* 1980.

Galoppo, N., Govindaraju, N. K., Henson, M., and Manocha, D., LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. *In SC 05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing, page 3, Washington, DC, USA. IEEE Computer Society*.

Kirk D. and Hwu W., Programming Massively Parallel Processors: A Hands-on Approach. *Morgan Kaufmann, First edition*, February 2010.

Krüger, J. and Westermann, R., Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22:908-916, 2003.

Larsen, E. S. and McAllister, D., Fast matrix multiplies using graphics hardware. *In Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (Cdrom) (Denver, Colorado, November 10 - 16, 2001). Supercomputing '01. ACM, New York, NY, 55-55*.

NVIDIA, NVIDIA CUDA Programming Guide 2.0. 2008

Sakharnykh, N., Tridiagonal Solvers on the GPU and Applications to Fluid Simulation. *GPU Technology Conference, 2009*.

Tomov, S., Dongarra, J., and Baboulin, M., Towards dense linear algebra for hybrid GPU accelerated manycore systems. *MIMS EPrint 2009.7, Manchester Institute for Mathematical Sciences, University of Manchester, Manchester, UK*.

Volkov, V. and Demmel, J., LU, QR and Cholesky factorizations using vector capabilities of GPUs. *Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley*, 2008.

Zhang, Y., Owens, J.D. y Cohen, J., Fast Tridiagonal Solvers on the GPU. *GPU Technology Conference, 2009*.