Métodos no lineales basados en el gradiente conjugado para GPUs

H. Migallón, [1] V. Migallón[2] y J. Penadés[2]

Resumen— **En este artículo se presentan algoritmos paralelos para resolver sistemas no lineales, diseñados para GPUs (Graphics Proccesing Unit), para lo cual se hecho uso de CUDA (Compute Unified Device Architecture). Los algoritmos propuestos están basados tanto en la versión de Fletcher-Reeves del método del gradiente conjugado, como en precondicionadores polinomiales construidos mediante el método por bloques en dos etapas. Se analizan diversas estrategias para la paralelización de dichos algoritmos, así como diferentes formatos de almacenamiento/compresión de las matrices dispersas consideradas en este trabajo. Expondremos resultados numéricos comparando la ejecución en plataformas paralelas de grano fino (GPU) con la ejecución en plataformas paralelas basadas en hilos (multiprocesadores de memoria compartida o multicores).**

Palabras clave—**GPGPU, librerías GPU, gradiente conjugado no lineal, precondicionadores paralelos, factorizaciones ILU, métodos por bloques en dos etapas.**

I. Introducción

SE considera la resolución del sistema no lineal

$$Ax = \Phi(x), \tag{1}$$

donde $A \in \Re^{n\times n}$ es una matriz simétrica y definida positiva, y $\Phi : \Re^n \to \Re^n$ es una función no lineal con ciertas propiedades. Sea $\Psi : \Re^n \to \Re$ una aplicación no lineal, y sea $\langle x, y\rangle = x^T y$ el producto interno en $\Re^n$. El problema de minimización consistente en encontrar $x \in \Re^n$ tal que

$$J(x) = \min_{y\in\Re^n} J(y), \tag{2}$$

donde $J(x) = \frac{1}{2}\langle Ax, x\rangle - \Psi(x)$, es equivalente a encontrar $x \in \Re^n$ tal que $F(x) = Ax - \Phi(x) = 0$, donde $\Phi(x) = \Psi'(x)$.

Un método efectivo para resolver el sistema (1), teniendo en cuenta la conexión con el problema de minimización (2), es la versión de Fletcher-Reeves [1] del método del gradiente conjugado no lineal (NLCG), que se detalla a continuación:

Algoritmo 1: (GC no lineal (Fletcher-Reeves))
Dado un vector inicial $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
$p^{(0)} = r^{(0)}$
Para $i = 0, 1, \ldots,$ hasta convergencia
$\alpha_i =\to$ (ver a continuación)
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i Ap^{(i)}$
Test de convergencia
$\beta_{i+1} = -\frac{\langle r^{(i+1)}, r^{(i+1)}\rangle}{\langle r^{(i)}, r^{(i)}\rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

[1]Dpto. de Física y Arquitectura de Computadores, Universidad Miguel Hernández, e-mail: hmigallon@umh.es.
[2]Dpto. de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, e-mail: violeta,jpenades@dccia.ua.es.

En el Algoritmo 1 la elección de α_i debe minimizar la función asociada J en la dirección $p^{(i)}$. Esto es equivalente a resolver el problema unidimensional de punto cero $\frac{dJ(x^{(i)}+\alpha_i p^{(i)})}{d\alpha_i} = 0$. De la definición de J, se deduce que

$$J(x^{(i)} + \alpha p^{(i)}) =$$

$$\frac{1}{2}\left\langle A(x^{(i)} + \alpha_i p^{(i)}), x^{(i)} + \alpha_i p^{(i)}\right\rangle - \Psi(x^{(i)} + \alpha_i p^{(i)}).$$

Por tanto, diferenciando respecto a α_i se obtiene

$$\frac{dJ(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i} =$$

$$\alpha_i \left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle r^{(i)}, p^{(i)}\right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(x^{(i)} + \alpha_i p^{(i)}), p^{(i)}\right\rangle,$$

donde $r^{(i)} = \Phi(x^{(i)}) - Ax^{(i)}$ es el residuo no lineal.

Por otra parte, puede verse que la segunda derivada respecto a α_i es

$$\frac{d^2 J(x^{(i)} + \alpha_i p^{(i)})}{d\alpha_i^2} =$$

$$\left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle \Phi'(x^{(i)} + \alpha_i p^{(i)})p^{(i)}, p^{(i)}\right\rangle.$$

Por lo tanto, si se usa el método de Newton para resolver el problema de punto cero para α_i, se obtiene $\alpha_i^{(k+1)} = \alpha_i^{(k)} - \delta^{(k)}$, donde (siendo $\gamma = (x^{(i)} + \alpha_i^{(k)} p^{(i)})$)

$$\delta^{(k)} = \frac{dJ(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i}{d^2 J(x^{(i)} + \alpha_i^{(k)} p^{(i)})/d\alpha_i^2} =$$

$$\frac{\alpha_i^{(k)}\left\langle Ap^{(i)}, p^{(i)}\right\rangle - \left\langle r^{(i)}, p^{(i)}\right\rangle + \left\langle \Phi(x^{(i)}) - \Phi(\gamma), p^{(i)}\right\rangle}{\langle Ap^{(i)}, p^{(i)}\rangle - \langle \Phi'(\gamma)p^{(i)}, p^{(i)}\rangle}.$$

Hay que remarcar que para obtener $\delta^{(k)}$, los productos internos $\langle Ap^{(i)}, p^{(i)}\rangle$ y $\langle r^{(i)}, p^{(i)}\rangle$ pueden computarse únicamente en la iteración inicial del método de Newton. Además $Ap^{(i)}$ ha sido calculado en la iteración correspondiente del método del gradiente conjugado.

El objetivo del precondicionamiento es mejorar el número de condición (cond) de la matriz del sistema a resolver. Supongamos que M es una matriz simétrica y definida positiva que aproxima a A, y fácilmente invertible. Entonces, podemos resolver indirectamente el sistema $Ax = \Phi(x)$ resolviendo el sistema $M^{-1}Ax = M^{-1}\Phi(x)$. Si $\mathrm{cond}(M^{-1}A) << \mathrm{cond}(A)$, mediante un método iterativo, podemos resolver el sistema $M^{-1}Ax = M^{-1}\Phi(x)$ más rápido que el sistema original. De este modo obtenemos el siguiente método del gradiente conjugado precondicionado para sistemas no lineales (NLPCG).

Algoritmo 2: (CG Precondicionado no lineal)
Dado un vector inicial $x^{(0)}$
$r^{(0)} = \Phi(x^{(0)}) - Ax^{(0)}$
Resolver $Ms^{(0)} = r^{(0)}$
$p^{(0)} = s^{(0)}$
Para $i = 0, 1, \ldots,$ hasta convergencia

$\alpha_i =\rightarrow$ ver algoritmo 1
$x^{(i+1)} = x^{(i)} + \alpha_i p^{(i)}$
$r^{(i+1)} = r^{(i)} - \Phi(x^{(i)}) + \Phi(x^{(i+1)}) - \alpha_i A p^{(i)}$
Resolver $Ms^{(i+1)} = r^{(i+1)}$
Test de convergencia
$\beta_{i+1} = -\frac{\langle s^{(i+1)}, r^{(i+1)} \rangle}{\langle s^{(i)}, r^{(i)} \rangle}$
$p^{(i+1)} = r^{(i+1)} - \beta_{i+1} p^{(i)}$

Dado que el sistema auxiliar $Ms = r$ ha de resolverse en cada iteración del algoritmo, está solución ha de poderse obtener rápidamente. Además, para que el precondicionador sea efectivo es necesario que M sea una buena aproximación de A. El precondicionamiento mediante series truncadas [2] es una técnica común de precondicionamiento para resolver sistemas lineales, que consiste en considerar una partición de la matriz A tal que,

$$A = P - Q \tag{3}$$

y realizar m iteraciones del proceso iterativo definido por esta partición, buscando la solución de $As = r$, y tomando $s^{(0)} = 0$. Es bien conocido que la solución del sistema auxiliar $Ms = r$ es $s = (I + R + R^2 + \ldots + R^{m-1})P^{-1}r$, donde $R = P^{-1}Q$ y la matriz de precondicionamiento es $M_m = P(I + R + R^2 + \ldots + R^{m-1})^{-1}$ (ver [2]).

Si además suponemos que A está dividida en $p \times p$ bloques, con bloques diagonales de orden n_j, $\sum_{j=1}^{p} n_j = n$, tal que el sistema (1) puede escribirse como:

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1p} \\ A_{21} & A_{22} & \cdots & A_{2p} \\ \vdots & \vdots & & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pp} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \Phi_1(x) \\ \Phi_2(x) \\ \vdots \\ \Phi_p(x) \end{bmatrix}, \tag{4}$$

donde x y $\Phi(x)$ están particionados en función del tamaño de los bloques de A. Si consideramos la partición (3) estando P compuesta por los bloques diagonales de A en la ecuación (4), es decir

$$P = \mathrm{diag}(A_{11}, \ldots, A_{pp}), \tag{5}$$

realizar m iteraciones del proceso iterativo definido por la partición (3) para obtener una aproximación de $As = r$, corresponde a realizar m iteraciones del método de Jacobi por bloques. Por lo tanto, en cada iteración l, $l = 1, 2, \ldots$, del método de Jacobi por bloques, ha de resolverse p sistemas lineales independientes del tipo,

$$A_{jj} s_j^{(l)} = (Qs^{(l-1)} + r)_j, \;\; 1 \le j \le p. \tag{6}$$

Por tanto, los sistemas lineales (6) pueden ser resueltos por procesos distintos. Sin embargo, cuando el tamaño de los bloques diagonales A_{jj}, $1 \le j \le p$, es grande, es aconsejable utilizar un proceso iterativo para obtener una aproximación de las soluciones, utilizando por tanto el método en dos etapas; ver por ejemplo [3]. Formalmente, consideramos las particiones

$$A_{jj} = B_j - C_j, \;\; 1 \le j \le p, \tag{7}$$

y en la l-ésima iteración se realizan, para cada j, $1 \le j \le p$, $q(j)$ iteraciones del proceso iterativo definido por las particiones (7) para obtener una aproximación de la solución de (6). Por tanto, para resolver el sistema auxiliar $Ms = r$ del algoritmo 2, se realizan m pasos de la iteración $s^{(l)} = Ts^{(l-1)} + W^{-1}r$, $l = 1, 2, \ldots, m$, tomando $s^{(0)} = 0$, donde

$$T = H + (I - H)P^{-1}Q, \;\; W = P(I - H)^{-1}, \tag{8}$$

estando P definido en (5) y $H = \mathrm{diag}((B_1^{-1}C_1)^{q(1)}, \ldots, (B_p^{-1}C_p)^{q(p)})$; ver por ejemplo [4]. El siguiente algoritmo muestra el método utilizado para aproximar el sistema lineal $As = r$ (ver [3]).

Algoritmo 3: (Método paralelo por bloques en dos etapas)
Dado un vector inicial $s^{(0)} = \left((s_1^{(0)})^T, (s_2^{(0)})^T, \ldots, (s_p^{(0)})^T\right)^T$, y una secuencia de número de iteraciones internas $q(j)$, $1 \le j \le p$
Para $l = 1, 2, \ldots$, hasta convergencia
En el proceso j, $j = 1, 2, \ldots, p$
$y_j^{(0)} = s_j^{(l)}$
Para $k = 1$ hasta $q(j)$
$B_j y_j^{(k)} = C_j y_j^{(k-1)} + (Qs^{(l-1)} + r)_j$
$s^{(l)} = \left((y_1^{(q(1))})^T, (y_2^{(q(2))})^T, \ldots, (y_p^{(q(p))})^T\right)^T$

Hay que remarcar, que el vector obtenido tras m iteraciones del algoritmo 3, siendo $s^{(0)} = 0$, viene dado por $s^{(m)} = (I + T + T^2 + \ldots + T^{m-1})W^{-1}r$ donde T y W están definidos en (8). Por tanto, el precondicionador obtenido mediante el método por bloques en dos etapas es $M_m = W(I + T + T^2 + \ldots + T^{m-1})^{-1}$. Para obtener las particiones internas, en el método NLPCG, se ha hecho uso de factorizaciones incompletas LU, en [5] puede verse una descripción detallada del algoritmo NLPCG.

II. Programación paralela con CUDA

La arquitectura de una GPU (Graphics Processing Unit), está formada por un conjunto de multiprocesadores denominados "streaming multiprocessors (SM)", cada uno de los cuales está compuesto por un conjunto de procesadores denominados "streaming processors (SP)". CUDA es el modelo de programación utilizado para explotar el paralelismo de las GPUs, el cual es un modelo heterogéneo que hace uso tanto de la CPU como de la GPU. En el modelo de programación de CUDA (ver por ejemplo [6] y [7]), una aplicación consiste en un programa secuencial principal (o "*host*") ejecutado en la CPU, el cual puede lanzar programas, conocidos como "*kernels*", en el dispositivo paralelo, es decir en la GPU. Pese a que la CPU sobre la que se ejecuta el programa *host* puede ser un multiprocesador de memoria compartida, con capacidad para ejecutar programas paralelos, desarrollados por ejemplo con OpenMP, sólo un procesador (o "*core*") puede lanzar un *kernel*, es decir las llamadas a los *kernels* deben serializarse. La ejecución de los *kernels* son de tipo SPMD (Single Program Multiple Data), que, además, puede utilizar un gran número de "*threads*" o hilos. Cada hilo de un *kernel* ejecuta el mismo programa secuencial, siendo el programador el que debe organizar los hilos de un *kernel* en bloques, formando estos bloques lo que se conoce como "*grid*". Los hilos de un bloque pueden cooperar entre ellos, sin-

cronizando su ejecución mediante barreras. Las memorias disponibles en una GPU son: la memoria global que es la de mayor latencia, la memoria de sólo lectura ("constant"), la memoria de "textura", la memoria compartida y los registros. La memoria compartida lo es para un bloque, y los registros son propios de cada hilo. Aunque, tanto la memoria "constant" como la memoria "texture" disponen de caché, no han sido utilizadas por la naturaleza de nuestro problema.

El hardware se ocupa de la organización, creación y manejo de los hilos. Por ejemplo, la GPU GTX 280 dispone de 30 multiprocesadores, pudiendo trabajar hasta con $30K$ hilos. Para manejar eficientemente esta cantidad de hilos, la GPU utiliza una arquitectura SIMT (Single Instruction Multiple Thread), ver por ejemplo [6] y [8], en la cual los hilos de un bloque se ejecutan en grupos de 32 hilos, llamados "*warps*". Un *warp* en un momento dado ejecuta una única instrucción en todos sus hilos. No obstante, los hilos de un *warp* pueden seguir su propia ejecución, es decir, la ejecución en cada hilo puede ser diferente, siendo mucho más eficiente que todos los hilos realicen la misma ejecución.

III. Formatos de almacenamiento de matrices dispersas

El producto de una matriz dispersa por un vector (SpMV) es una de las operaciones básicas en los algoritmos vistos en la sección I. Para optimizar esta operación hemos considerado varios formatos de almacenamiento de matrices dispersas. En concreto, se ha usado el formato CRS (Compressed Row Storage), el formato ELLPACK [9] (o ITPACK), y el formato propuesto en [10] denominado ELLPACK-R. Existen multitud de posibles representaciones de matrices dispersas, cada una con diferentes requisitos de almacenamiento, diferentes características de computación y distintas formas de acceder y manipular los elementos de la matriz. Hemos considerado únicamente formatos de almacenamiento de matrices dispersas de uso común y que además presentan un buen comportamiento al computar la operación SpMV en una GPU.

El formato CRS (o CSR), muy común y de propósito general, no presupone nada respecto al patrón de dispersión de la matriz, y no almacena ningún elemento no necesario. El formato CRS almacena en posiciones contiguas de memoria los elementos no nulos de la matriz. Este formato utiliza tres vectores, uno de "*floats*" y dos de enteros. El primero almacena los elementos no nulos de la matriz A agrupados por filas. En el primer vector de enteros se almacena la columna de los elementos no nulos. El otro vector de enteros almacena la posición en la que empieza cada fila en los otros dos vectores, por tanto el último elemento de este vector corresponde al número de elementos no nulos (NNZ) o a NNZ+1 si el primer elemento es 1 en lugar de 0.

El formato ELLPACK [9] fue diseñado para resolver sistemas lineales de gran tamaño en arquitecturas vectoriales. Hay que hacer notar que existen ciertas similitudes entre una arquitectura vectorial y la arquitectura de una GPU. El formato ELLPACK, también denominado ITPACK, usa dos vectores, el primero, de *floats*, almacena los elementos no nulos de la matriz; y el segundo, de enteros, almacena el número de columna de cada uno de los elementos no nulos almacenados en el primer vector. La dimensión de ambos vectores es, al menos, $N * MaxEntriesbyRows$, donde N es el número de filas y $MaxEntriesbyRows$ es el número máximo de elementos no nulos por fila en la matriz. Por tanto, en este formato todas las filas se almacenan ocupando el mismo tamaño, aquellas filas con un número de elementos no nulos inferior a $MaxEntriesbyRows$, son rellenadas con ceros. Teniendo en cuenta esta estructura, el formato ELLPACK almacena en una estructura regular, similar a una matriz densa, una matriz dispersa. Esta estructura regular, como se ha dicho anteriormente, es apropiada para realizar operaciones con matrices dispersas en arquitecturas vectoriales. Sin embargo, si el porcentaje de elementos nulos es alto y el patrón de dispersión de la matriz es irregular, respecto al número de elementos no nulos por fila, el rendimiento del formato ELLPACK disminuye, además de aumentar el tamaño de memoria necesario para almacenar la matriz respecto a otros formatos.

La variante del formato ELLPACK, denominada ELLPACK-R [10], fue diseñada con el objetivo de optimizar el producto de una matriz dispersa por un vector en GPUs. El formato ELLPACK-R está formado por los mismos dos vectores del formato ELLPACK, más un tercer vector de enteros de tamaño N, que almacena el número de elementos no nulos de cada fila, descartando, por tanto, los elementos nulos con los que se han rellenado las filas con un número de elementos no nulos inferior a $MaxEntriesbyRows$. En este caso es necesario que los elementos de cada fila se almacenen por orden creciente de columna. La ventajas que presenta el formato ELLPACK (ver [10]) para su uso en GPUs son: proporciona un acceso coalescente a la memoria global, para ello es necesario que las filas estén ordenadas en orden creciente de número de columna; al igual que los otros dos formatos vistos, permite una ejecución sin procesos de sincronización entre hilos; reduce los tiempos de espera entre los hilos de un *warp*; además, la computación realizada por los hilos de un *warp* es homogénea.

IV. Operaciones Básicas

Según la descripción del algoritmo 1 y del algoritmo 2, las operaciones básicas para implementar dichos métodos son:

- El producto de una matriz dispersa por un vector (SpMV).
- Operaciones vectoriales básicas (incluidas en el nivel 1 de la librería BLAS [11]).
- El producto interno.
- La resolución de un sistema LU (método incluido en SPARSKIT [12]), utilizado únicamente en el método NLPCG.

Esta sección está dedicada a describir las diferentes opciones para realizar las operaciones básicas reseñadas y analizarlas en el contexto de nuestro trabajo.

A. Producto matriz dispersa por vector

El objetivo de utilizar diferentes formatos de almacenamiento de matrices dispersas, descritos en la sección III, es optimizar el producto de una matriz dispersa por un vector. El código del *kernel* que implementa la operación SpMV usando el formato CRS no ha sido optimizado. Para optimizar este código existen dos vías, la primera es utilizar un formato de almacenamiento que optimice dicho cálculo, opción que sí será considerada; y la segunda vía consiste en modificar la estructura de hilos con el objetivo de optimizar el acceso a la memoria global de la GPU. Sin embargo, en este segundo caso, las optimizaciones que podrían aplicarse se centran en matrices con un patrón de dispersión mayor al del ejemplo utilizado en nuestro trabajo (ver la sección V). En el ejemplo utilizado en nuestro trabajo el número típico de elementos no nulos por fila es de 7, y, por ejemplo, las optimizaciones propuestas en [13] necesitan más de 32 elementos no nulos por fila para ser aplicadas. Por tanto, la vía utilizada para optimizar la operación SpMV será la utilización de los formatos de almacenamiento de matrices dispersas ELLPACK y ELLPACK-R, las cuales mejoran el patrón de acceso a memoria respecto al formato CRS.

Por otra parte, se ha utilizado la librería CUSPARSE [14] para calcular la operación SpMV. La librería CUSPARSE es una reciente librería para la ejecución en GPUs de operaciones entre elementos (matrices o vectores) dispersos, y entre elementos dispersos y elementos densos. Actualmente, sólo soporta el formato CRS en el producto de una matriz dispersa por un vector.

B. Operaciones vectoriales

Los operaciones vectoriales, que no implican un proceso de reducción, incluidas en los algoritmos 1 y 2, son la copia de vectores, el producto de un escalar por un vector, la operación *axpy* (o similar) y el cálculo de la función no lineal propia del sistema. La computación de estas operaciones en una GPU ya está optimizada por la propia arquitectura de la GPU, no obstante intentamos optimizarlas agrupando varias de ellas en un único *kernel*. Por otra parte, se ha utilizado CUBLAS [15], versión de la librería BLAS para su uso con CUDA. El uso de CUBLAS impide la agrupación de varias operaciones en único *kernel*, que era la primera vía de optimización propuesta.

C. Producto interno

El proceso de reducción que implica el producto interno hace que éste sea una operación especial en CUDA. En la computación en GPUs es necesario mantener ocupados a todos los SM (*streaming multiprocessors*), además para trabajar con vectores de gran tamaño es necesario usar muchos bloques de hilos. Para conseguir estos dos propósitos cada bloque realiza el proceso de reducción de una porción del vector. Dado que CUDA no proporciona mecanismos globales de sincronización que permitan comunicar resultados parciales entre bloques, se calculan *VectorN* vectores de *ElementN* elementos, tal y como se propone en la *NVIDIA CUDA C SDK*. El número de elementos (*ElementN*) de cada porción de vector debe ser múltiplo del tamaño de un *warp*, para mantener las restricciones de alineamiento de la memoria y el acceso coalescente. Un bloque calcula la reducción de una o más porciones del vector. Además, para evitar procesos de sincronización se trabaja con la memoria compartida, la cual actúa como un acumulador. El tamaño de esta memoria compartida (*ACCUM_N*) deber ser potencia de dos y si es posible múltiplo del tamaño de un *warp*. Por tanto, cada hilo calcula un elemento del acumulador trabajando con elementos del vector separados en *ACCUM_N* elementos. El *kernel* finaliza realizando un proceso de reducción tipo árbol de los elementos almacenados en el acumulador y donde sí es necesario realizar procesos de sincronización entre hilos. Hay que remarcar que la CPU debe finalizar la operación trabajando con los resultados parciales obtenidos, lógicamente el número de resultados parciales obtenidos es igual al número de porciones de vector (*VectorN*) con las que se ha trabajado. En nuestros algoritmos hemos agrupado varios productos internos en único *kernel*.

Por otra parte, la librería CUBLAS proporciona también la función que permite el cálculo del producto interno, y teniendo en cuenta que hemos trabajado con la versión optimizada de CUBLAS incluida en el *CUDA Toolkit 3.2 RC*, no se han considerado otras optimizaciones.

D. Solucionador LU

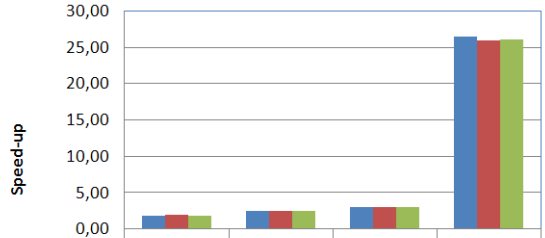
En el proceso para resolver un sistema LU cada elemento computado de la solución es utilizado para el cálculo del siguiente elemento. Por tanto este proceso no dispone de paralelismo inherente de grano fino. Hemos desarrollado diferentes algoritmos con diversas estrategias para resolver el sistema LU en la GPU, pero ninguna de ellas ha obtenido buenos resultados. Por lo tanto, la mejor opción ha sido resolver el sistema LU haciendo uso de la arquitectura multicore de la CPU, lo cual no ha sido del todo efectivo al verse penalizado el algoritmo por el aumento de comunicaciones entre la CPU y la GPU. En la sección V presentaremos resultados haciendo uso de CUDA y de OpenMP conjuntamente. En este ámbito hay que remarcar que en el método NLCG, las comunicaciones entre GPU y CPU se reducen a algunos escalares y los vectores necesarios para completar los procesos de reducción.

V. Resultados numéricos

Para analizar el comportamiento de los métodos propuestos, NLCG y NLPCG, se ha utilizado el multicore Intel Core 2 Quad Q6600, 2.4 GHz, con 4 GB de RAM y 8 MB memoria caché L2, denominado SULLI, con sistema operativo Ubuntu 9.04 (Jaunty Jackalope) para sistemas de 64 bits. La GPU disponible en SULLI es una NVIDIA GeForce GTX 280. Los códigos de CUDA han sido compilados con el compilador de NVIDIA (nvcc) proporcionado por el *CUDA Toolkit 3.2 RC*.

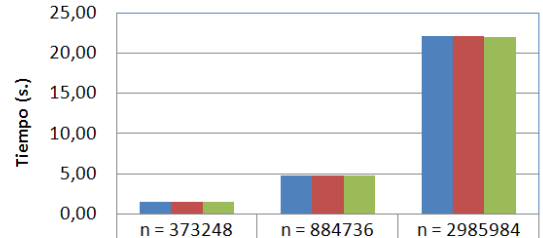
El ejemplo utilizado en nuestros experimentos es una ecuación en derivadas parciales, no lineal y elíptica, conocida como el problema de Bratu. Este problema tridimensional viene dado por

$$\nabla^2 u - \lambda e^u = 0, \tag{9}$$



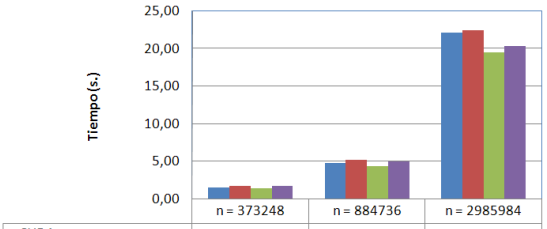
	2 Proc	3 Proc	4 Proc	GPU
n=373248	1,84	2,51	3,04	26,49
n=884736	1,88	2,44	2,95	25,89
n=2985984	1,85	2,46	2,94	26,05

Fig. 1. Speed-up del método NLCG.



	n = 373248	n = 884736	n = 2985984
CSR	1,48	4,71	22,05
ELLPACK	1,49	4,77	22,08
ELLPACK-R	1,46	4,70	21,92

Fig. 2. Método NLCG vs formato de almacenamiento.



	n = 373248	n = 884736	n = 2985984
CUDA	1,48	4,71	22,05
CUDA + CUBLAS	1,73	5,13	22,43
CUDA + CUSPARSE	1,40	4,35	19,49
CUDA + CUBLAS + CUSPARSE	1,68	4,94	20,34

Fig. 3. Uso de CUBLAS y/o CUSPARSE en el método NLCG.

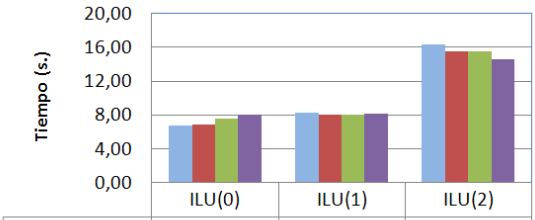
donde u es la temperatura y λ es una constante conocida como el parámetro de Frank-Kamenetskii; ver por ejemplo [16]. Hay dos posibles soluciones para este problema dado un valor de λ. Una de las soluciones, sencilla de obtener, es cercana a $u = 0$. Para converger a la otra solución es necesario partir de un vector inicial cercano a dicha solución. En nuestro modelo se considera un dominio cúbico 3D de longitud unidad y $\lambda = 6$. Para resolver la ecuación (9) usando el método de diferencias finitas, consideramos un mallado del dominio formado por d^3 nodos. La discretización da lugar a un sistema no lineal de la forma $Ax = \Phi(x)$, donde $\Phi : \Re^n \rightarrow \Re^n$ es una aplicación diagonal y no lineal, en la cual la componente i-ésima Φ_i de Φ depende únicamente de la componente i-ésima de x. La matriz A es una matriz dispersa de orden $n = d^3$, siendo el número típico de elementos no nulos por fila de siete, con menos elementos no nulos en aquellos puntos que corresponden a la frontera del dominio físico.

El análisis que presentamos se basa en la comparación de los tiempos de ejecución utilizando como plataforma de computación la GPU GeForce GTX 280, con los tiempos de ejecución en el multicore SULLI. En primer lugar presentamos resultados para resolver sistemas de varios tamaños usando el método NLCG. En la figura 1 se muestra el speed-up utilizando, por un lado OpenMP con diferente número de cores, y por otro la GPU GeForce GTX 280. Lógicamente la GPU es controlada por uno de los cores de SULLI. En esta figura podemos observar que se obtiene un buen speed-up usando los cores disponibles de SULLI, pero que dicho speed-up no es comparable al obtenido con la GPU, en la cual el valor es superior a 25. Este resultado confirma la expectativa de una muy buena interacción entre el algoritmo NLCG y la GPU.

En la figura 2 se puede ver el comportamiento de los diferentes formatos de almacenamiento de matrices dispersas descritos en la sección III. Hay que tener en cuenta, que el formato utilizado modifica el *kernel* que calcula la operación SpMV. Los mejores resultados se obtienen utilizando el formato ELLPACK-R, ya que este algoritmo no incluye instrucciones de control de flujo que provoquen la serialización de la ejecución de los diferentes hilos de un *warp* y, además, permite el acceso coalescente a los elementos de la matriz. No obstante, hay que remarcar que el formato ELLPACK-R es el que más memoria requiere de los formatos vistos, debido al uso del tercer vector para almacenar el número de elementos no nulos de cada fila, además del rellenado de algunas filas con ceros. Hemos analizado el comportamiento del algoritmo NLCG en función tanto del número de hilos por bloque, como del tamaño del acumulador implementado en la memoria compartida para calcular el producto interno. La figura 2 presenta resultados utilizando los valores óptimos para estos dos parámetros, es decir 256 hilos por bloque y un tamaño de *ACCUM_N* igual a 128.

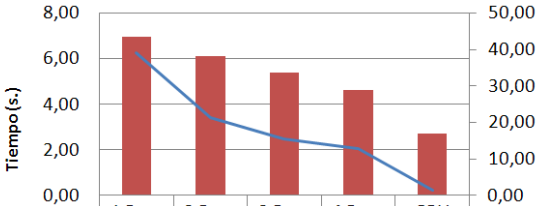
Por último, respecto al método NLCG, en la figura 3 se muestran resultados haciendo uso de las librerías CUBLAS y CUSPARSE, librerías incluidas en el *CUDA Toolkit 3.2 RC*. En dicha figura se analiza el uso de CUBLAS y de CUSPARSE por separado y también el uso de ambas librerías conjuntamente. En primer lugar, podemos observar que el uso de CUBLAS presenta peores resultados que si se utiliza únicamente el API de CUDA. Esto es debido a las optimizaciones realizadas agrupando varias operaciones en un único *kernel*, proceso que no puede llevarse a cabo si se usa CUBLAS. Al contrario, el uso de CUSPARSE obtiene una pequeña mejora. Hay que remarcar que el uso conjunto de CUBLAS y CUSPARSE obtiene resultados aceptables con la ventaja que esconde al usuario la elección de parámetros tales como el número de hilos por bloque y el tamaño del acumulador para el cálculo de operaciones de reducción.

En [5] se detalla el comportamiento del algoritmo NLPCG en una plataforma multicore, todos los experimentos realizados haciendo uso de la GPU muestran que ese comportamiento no difiere al cambiar la plataforma de computación a una GPU. En resumen, el valor óptimo del número de iteraciones internas y el valor óptimo del número de iteraciones externas es un valor pequeño. Respecto al nivel de llenado de la factorización ILU, se con-



	ILU(0)	ILU(1)	ILU(2)
1 Core + GPU	6,76	8,23	16,26
2 Cores + GPU	6,93	8,04	15,49
3 Cores + GPU	7,57	8,09	15,51
4 Cores + GPU	8,09	8,16	14,55

Fig. 4. Método NLPCG, $m = 1$, $q = 1$ y $n = 884736$.



	1 Proc	2 Proc	3 Proc	4 Proc	GPU
NLPCG	6,97	6,13	5,41	4,61	2,70
NLCG	39,20	21,27	15,60	12,92	1,48

Fig. 5. Comparación de los métodos NLCG y NLPCG en CPU y GPU, orden del sistema $n = 373248$.

cluye que el valor óptimo es 0 o 1. Trabajando con dichos valores óptimos, en la figura 4 presentamos resultados del método NLPCG haciendo uso de ambas plataformas de computación conjuntamente. En dicha figura podemos observar que el algoritmo NLPCG no se adapta bien al paralelismo ofrecido por la GPU, y que el uso de ambas plataformas tampoco obtiene buenos resultados al verse penalizado, como se ha comentado anteriormente, por el incremento de las comunicaciones entre GPU y CPU. Respecto al resto de parámetros vistos en el algoritmo NLCG, podemos extender las conclusiones obtenidas en dicho método al método NLPCG.

Por último, si analizamos la figura 5 deducimos que el speed-up obtenido usando la GPU siempre es mayor que usando los cuatro cores disponibles en SULLI. De este modo podemos concluir que utilizando una GPU el método NLCG ofrece mejores prestaciones, pero no así usando un multicore, en cuya caso las mejores prestaciones las ofrece el método NLPCG.

VI. Conclusiones

Haciendo uso del modelo de computación GPGPU (General-purpose computing on graphics processing units) hemos desarrollado la versión de Fletcher-Reeves del método del gradiente conjugado no lineal, y hemos aplicado, a dicho método, un precondicionador de tipo polinomial basado en los métodos en dos etapas. Se han comparado los métodos desarrollados con los mismos métodos desarrollados para OpenMP, y en el caso del método precondicionado se ha utilizado un modelo de programación mixto para explotar el paralelismo ofrecido por la GPU y el paralelismo ofrecido por el multicore. Hemos identificado las operaciones básicas de nuestros algoritmos, con el objetivo de optimizarlas y de experimentar algunas librerías disponibles. Hemos podido concluir que librerías como CUBLAS y CUSPARSE pueden ofrecer un buen rendimiento. Respecto al formato de almacenamiento de matrices dispersas se concluye que debe ser seleccionado en función de las características de la plataforma paralela, siendo ELLPACK-R el formato más eficiente para la ejecución en una GPU. Por último, se han mostrado las diferencias de adaptación de ambos métodos a las dos arquitecturas paralelas utilizadas, obteniendo en ambos casos mejores resultados trabajando con la GPU que trabajando con el multicore, y además en este caso el método NLCG explota mejor el paralelismo, obteniendo mejores resultados que el método NLPCG.

Agradecimientos

El presente trabajo ha sido financiado por el Ministerio de Ciencia e Innovación mediante el proyecto TIN2008-06570-C04-04.

Referencias

[1] R. Fletcher y C. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, pp. 149–154, 1964.

[2] L. Adams, "M-step preconditioned conjugate gradient methods," *SIAM Journal on Scientific and Statistical Computing*, vol. 6, pp. 452–462, 1985.

[3] R. Bru, V. Migallón, J. Penadés, y D.B. Szyld, "Parallel, synchronous and asynchronous two-stage multisplitting methods," *Electronic Transactions on Numerical Analysis*, vol. 3, pp. 24–38, 1995.

[4] V. Migallón y J. Penadés, "Convergence of two-stage iterative methods for hermitian positive definite matrices," *Applied Mathematics Letters*, vol. 10, no. 3, pp. 79–83, 1997.

[5] H. Migallón, V. Migallón, y J. Penadés, "Parallel nonlinear conjugate gradient algorithms on multicore architectures," in *Proceedings of the International Conference on Computational and Mathematical Methods in Science and Engineering*, pp. 689–700. 2009.

[6] J. Nickolls, I. Buck, M. Garland, y K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[7] NVIDIA Corporation, "NVIDIA CUDA C programming guide," Version 3.2, `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`, 2010.

[8] E. Lindholm, J. Nickolls, S. Oberman, y J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[9] D.R. Kincaid y D.M. Young, "A brief review of the ITPACK project," *Journal of Computational and Applied Mathematics*, vol. 24, no. 1–2, pp. 121–127, 1988.

[10] F. Vázquez, J.J. Fernández, y E.M. Garzón, "A new approach for sparse matrix vector product on NVIDIA GPUs," *Concurrency and Computation: Practice and experience*, 2010, DOI: 10.1002/cpe.1658.

[11] C.L. Lawson, R.J. Hanson, D. Kincaid, y F.T. Krogh, "Basic linear algebra subprograms for FORTRAN usage," *ACM Transactions on Mathematical Software*, vol. 5, pp. 308–323, 1979.

[12] Y. Saad, "SPARSKIT: A basic tool kit for sparse matrix computation," `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[13] M. Manikandan y R. Bordawekar, "Optimizing sparse matrix-vector multiplication on GPUs," Tech. Rep. RC24704, IBM, 2008.

[14] NVIDIA Corporation, "CUDA CUSPARSE Library," Tech. Rep. PG-05329-032_V01, 2010, `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUSPARSE_Library.pdf`.

[15] NVIDIA Corporation, "CUDA CUBLAS Library," Tech. Rep. PG-05326-032_V01, 2010, `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUBLAS_Library.pdf`.

[16] B.M. Averick, R.G. Carter, J.J. More, y G. Xue, "The MINPACK-2 test problem collection," Tech. Rep. MCS-P153-0692, Mathematics and Computer Science Division, Argonne.