Cálculo en paralelo empleando tarjetas gráficas

Aplicación al algoritmo símplex revisado

Palabras clave: programación matemática, símplex revisado, GPGPU, CUDA.

Resumen:

En este artículo se presenta el cálculo paralelo sobre tarjetas gráficas actuales (que son sistemas SIMD, *Single Instruction Multiple Data*) y se analiza su eficiencia al aplicarlo a un algoritmo matricial común, como es el algoritmo símplex revisado de optimización lineal. La presencia de tarjetas gráficas cada vez más potentes en los entornos domésticos pone al alcance del gran público la posibilidad de llevar a cabo estos cálculos en paralelo. En este artículo se emplea la tecnología CUDA (Compute Unified Device Architecture) de NVidia para desarrollar la herramienta que permite obtener resultados experimentales, a partir de los cuales se pueden extraer conclusiones sobre la efectividad de este enfoque de cálculo paralelo.

Abstract: *mathematical programming, revised símplex, GPGPU, CUDA.*

Abstract:

This article presents a parallel computing approach using current graphic cards (which are SIMD systems, Single Instruction Multiple Data). It studies the efficiency of its application to the revised símplex algorithm, a common mathematical programming algorithm for linear problems. The presence of graphic cards more and more powerful in the domestic computers offers the general public the capability of performing these parallel computations. In this article, CUDA (Compute Unified Device Architecture) technology from NVidia is employed to develop a computer tool that allows obtaining some experimental results. Based on these results, some conclusions are extracted related to the efficiency of this approach to parallel computing.

Alfredo G. Escobar Portillo

Ingeniero Informático del ICAI de la promoción de 2011.

Jesús Mª Latorre Canteli

Ingeniero Industrial del ICAI (1995) y Doctor Ingeniero Industrial del ICAI (2007). Es investigador en el Instituto de Investigación Tecnológica.

Introducción

Existen en el ámbito científico multitud de algoritmos y métodos que se pueden expresar de forma vectorial o matricial. Al poder adoptar esta forma, la ejecución de este tipo de algoritmos en un ordenador es paralelizable en la mayoría de los casos de forma relativamente sencilla. Algunos ejemplos de algoritmos que pueden ser paralelizados con éxito son la simulación de fluidos, los métodos de optimización o el tratamiento de imágenes.

La historia de la paralelización es muy larga, comenzando en los años 70 con la aparición de los primeros sistemas multiprocesador y los procesadores vectoriales. En aquellos momentos, la posibilidad de emplear el cálculo paralelo se podía desarrollar en pocos lugares donde se disponía de los recursos necesarios. Con el rápido desarrollo que ha tenido lugar desde entonces se ha evolucionado hacia los procesadores de varios núcleos y el cálculo distribuido en *grids* o *clouds* (redes de ordenadores conectados trabajando como un único equipo virtual). Todas estas alternativas tienen un coste mucho menor en la actualidad y permiten poder hacer un uso más generalizado de la paralelización, incluso en entornos domésticos.

En los últimos 10 años, la tecnología empleada para la fabricación de las tarjetas gráficas o GPUs (*Graphics Processing Units*) ha evolucionado de una manera mucho más significativa que otras áreas del mundo tecnológico, debido a las cada vez mayores demandas de potencia que solicitan aplicaciones de tipo CAD, el modelado 3D, el diseño gráfico o los juegos. Las tarjetas gráficas de hace años tenían todas las funciones implementadas en el propio hardware, y eran por tanto fijas. Esta situación cambió en 2001 con el lanzamiento de la GeForce 3, primera tarjeta de la compañía NVidia que incorporaba módulos de sombreado programables.

La idea de usar esa potencia para otros menesteres se denomina GPGPU (*General Purpose computation on Graphics Processing Units*) o GPU Computing [GPGPU11]. En el momento en el que las tarjetas gráficas permiten que se programen funciones sobre su hardware [OWENS08] se empieza a hacer uso de GPGPU.

Al principio era necesario utilizar los lenguajes enfocados a la visualización en pantalla (como OpenGL) para realizar otros cálculos no relacionados con los gráficos. Esto implicaba el uso de funciones muy poco flexibles, originalmente diseñadas para otros fines, lo que hacía que la labor de programar para tal fin fuese realmente tediosa y complicada.

Para facilitar el empleo de las tarjetas gráficas para cualquier uso no vinculado con los gráficos, NVidia desarrolló toda una tecnología alrededor de la tarjeta, que permitía usar la misma para cualquier tarea: CUDA [CUDA11]. ATI, la principal (y actualmente casi única) competidora, un poco más tarde haría lo propio lanzando su propia tecnología: Stream [STREAM11].

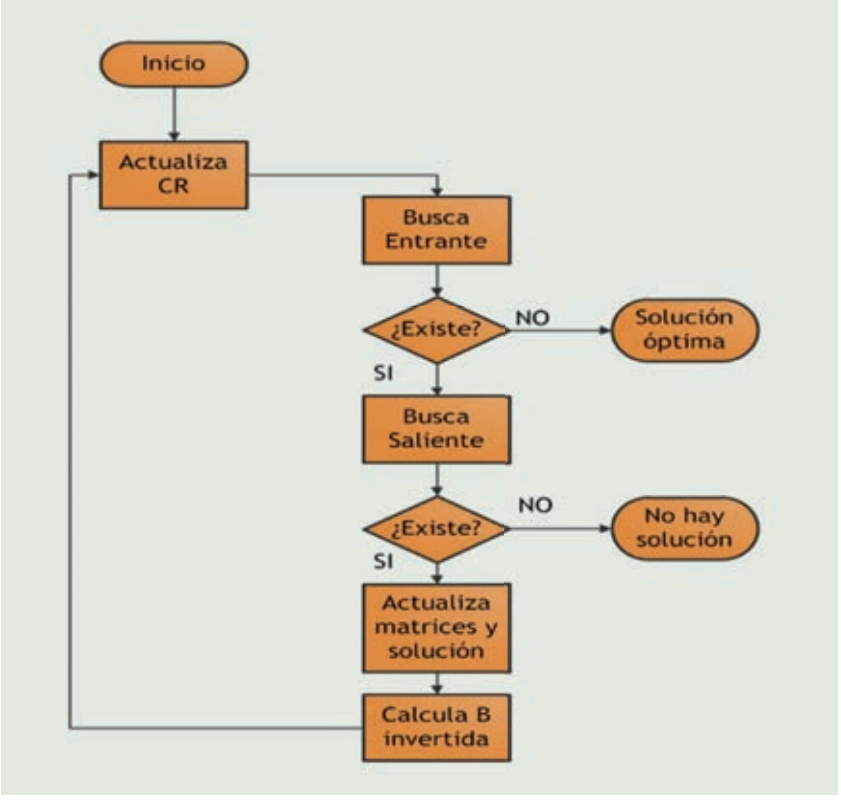
Debido a la mayor madurez de la tecnología CUDA sobre el resto de tecnologías, ésta ha sido la elegida para llevar a cabo este trabajo, con el objetivo de intentar demostrar sus ventajas (e inconvenientes) mediante la comparación de un algoritmo matricial sobre el que se han implementado dos versiones: una que se ejecuta en CPU y otra que se ejecuta en GPU la parte del código que puede ser eficientemente paralelizable.

Se va a presentar un análisis de la resolución en paralelo del algoritmo símplex revisado, método que sirve para optimizar problemas de programación lineal, de tal forma que minimice una función objetivo cuyas variables estén sujetas a restricciones y posiblemente a unas cotas. Estos problemas pueden ser de muy variados tamaños, pudiendo contener desde unas pocas hasta millones de variables y restricciones.

Algoritmo símplex revisado

El algoritmo símplex fue desarrollado por George B. Dantzig en 1947, y desde entonces se ha establecido como uno de los métodos básicos de resolución de problemas de programación matemática lineal [HILLIE01]. Se trata de resolver un problema de maximización o minimización de una función lineal, a la vez que las variables

Figura 1. Estructura general de funcionamiento del símplex



que forman esa función están relacionadas entre sí por restricciones lineales. Un ejemplo sencillo de este tipo de problemas es el de maximización de la producción de una fábrica, problema que está limitado por la disponibilidad de materias primas y el consumo de éstas en los diferentes productos.

El algoritmo símplex refina de forma iterativa una solución inicial, escogiendo en cada paso un cambio que mueve esa solución en el sentido de la optimización. De manera muy simplificada, considera en cada iteración un subconjunto de las variables del problema (que denomina la base) y busca qué variables no incluidas en la base pueden intercambiarse por otra básica mejorando la solución. Continúa de esta manera hasta que en una iteración no se puede encontrar un cambio que mejore el valor de la función objetivo. En ese caso se dice que ha encontrado la solución óptima del problema. También puede ocurrir que al calcular qué variable debe salir de la base, se detecte que el problema es no acotado, es decir, que se puede aumentar el valor de la función objetivo hasta infinito, sin que las variables se vean limitadas por sus cotas o por las restricciones. En la figura 1 se puede ver el esquema básico de las iteraciones del símplex. Para mayor información sobre el algoritmo y el significado de los costes reducidos (CR) o la matriz de la base B, véase la referencia [HILLIE01].

El algoritmo símplex revisado realiza los mismos cálculos que el algoritmo símplex, pero de manera más eficiente desde el punto de vista computacional. Para ello, solamente guarda y recalcula la información que necesita en cada iteración, de la forma más compacta posible.

Como se puede ver, los procesos de búsqueda de las variables entrante y saliente se pueden realizar en paralelo, dividiendo el rango de búsqueda entre los diferentes hilos de ejecución en paralelo y coordinando los resultados de cada uno. De igual manera, las operaciones matriciales implicadas en la actualización de los costes reducidos, la actualización de las matrices y la solución, y el recálculo de la inversa de la matriz de la base son cálculos matriciales que se han paralelizado en el trabajo presentado en este artículo.

Implementación del algoritmo

Con el objetivo de estructurar el desarrollo se ha llevado a cabo un diseño modular de la aplicación, articulado en torno a los siguientes cuatro módulos:

- Módulo de *input* (entrada): se encarga de la lectura de los problemas de optimización desde ficheros en formato MPS, formato habitual para los problemas de optimización.
- Módulo de preproceso y control: controla el flujo principal de la herramienta, llamando a los demás módulos, reservando memoria para el problema y adaptando el problema leído a la estructura que necesitan los cálculos del símplex revisado.
- Módulo de proceso (núcleo del símplex): se encarga de realizar los cálculos de cada iteración del símplex revisado, en las versiones de CPU y GPU, que se describen más adelante.
- Módulo de inversión de matrices: para el cálculo de la inversa de la matriz de la base se ha empleado este módulo. En él se pueden encontrar las funciones que se encargan tanto de la inversión explícita de la matriz, como del uso de la forma producto de la inversa que acelera la ejecución drásticamente y que no ha presentado problemas de estabilidad numérica en las pruebas realizadas.

Como punto de referencia para las pruebas experimentales se ha desarrollado una versión de la aplicación que se ejecuta únicamente en CPU: todos y cada uno de los módulos se ejecutan en la CPU y la aplicación se ha programado en C estándar.

El siguiente paso ha consistido en transformar esa versión en una nueva que se ejecute en GPU. Realmente, no toda la nueva versión se ejecuta en GPU, ya que no es posible portar todos los módulos a CUDA. Los módulos de entrada y de preproceso y control no se han modificado porque no es posible paralelizarlos o no se beneficiarían de tal cambio, mientras que los otros módulos sufren importantes cambios. La estructura de la aplicación, en ambas versiones, se puede ver en la figura 2.

La función principal del módulo de proceso (núcleo símplex) se ejecuta sobre la CPU, que guía todo el proceso. Son las funciones que se van llamando desde esa función principal las que se van ejecutando en la GPU, manteniendo una estructura similar a la versión de la CPU con objetivo de que los tiempos obtenidos en cada una de las fases sean comparables entre versiones.

Resultados experimentales

Para probar las dos versiones y observar las diferencias en cuanto a tiempos de ejecución entre ellas, se han realizado unas pruebas sobre una

Figura 2. Estructura de la aplicación desarrollada

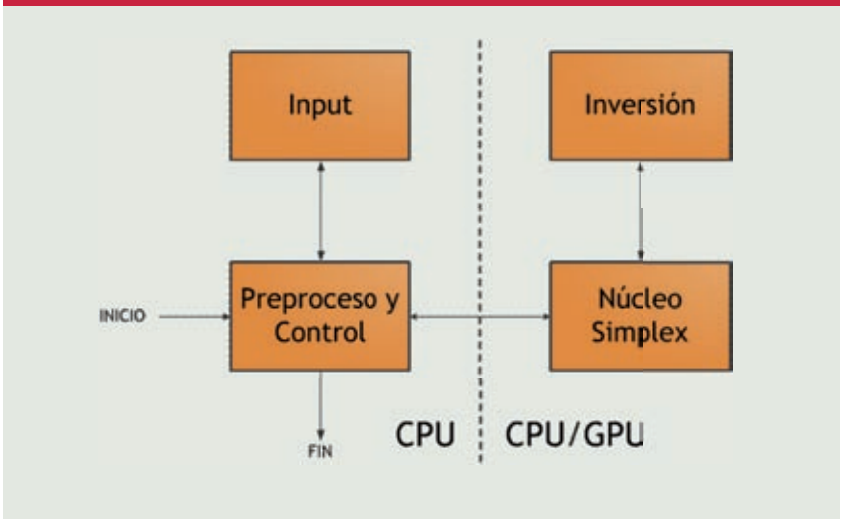


Figura 3. Resultados para problemas de pequeño tamaño

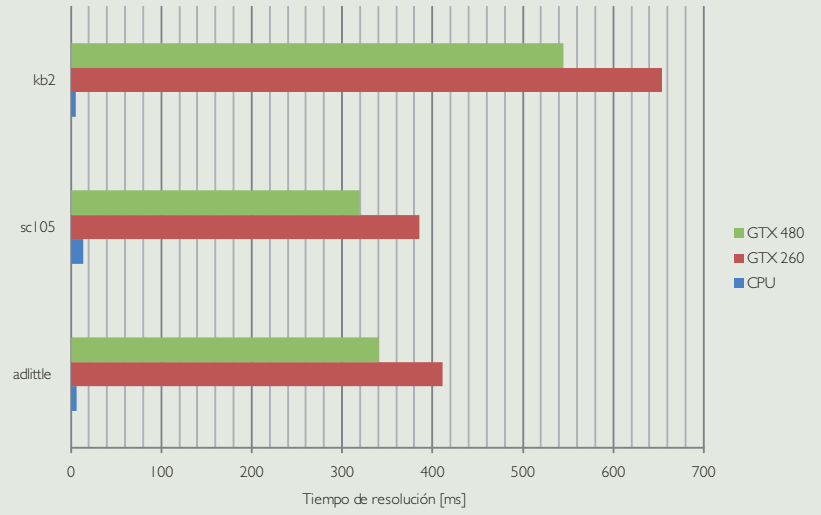
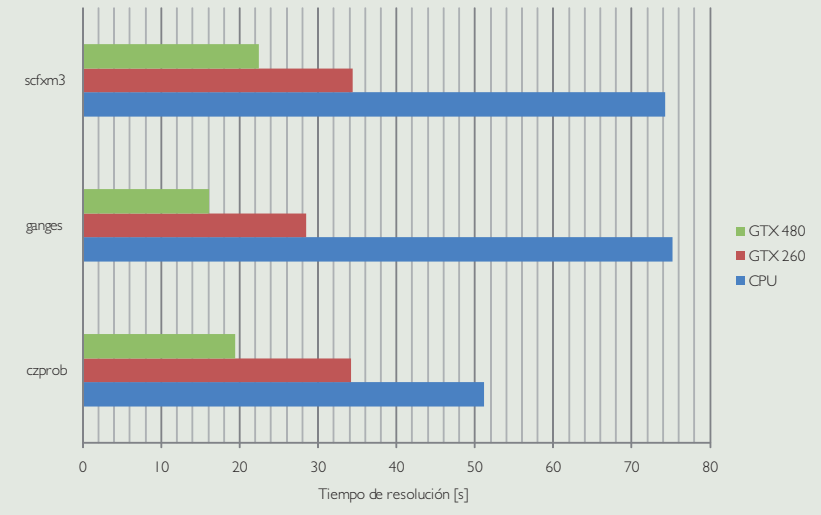


Figura 4. Resultados para problemas de mayor tamaño



serie de problemas procedentes de los repositorios de Netlib [NETLIB11]. En concreto, se han seleccionado 36 problemas de diversos tamaños para tomar una muestra significativa que proporcione resultados útiles para evaluar las capacidades de cálculo de ambas versiones.

La plataforma de pruebas consiste en un equipo que consta de una CPU Intel Core 2 Duo E8500 con una frecuencia de trabajo de 3.16 GHz y 6 MB de memoria caché, y 4 Gb de memoria RAM a 800 MHz en configuración de doble canal. Este procesador servirá para comprobar los tiempos de ejecución de la versión en CPU.

Para probar la versión en GPU se ha dotado al mismo sistema anterior de dos GPUs NVidia, ambas compatibles con CUDA: una GeForce GTX 260 de 192 núcleos y una GTX 480 de 480 núcleos. Esto permite realizar pruebas con las dos tarjetas de forma independiente, obteniendo al final de las pruebas tres series de resultados comparables entre ellos.

Cada uno de los problemas se ha ejecutado diez veces, para después realizar la media entre los tiempos de las diez ejecuciones. Esto se ha hecho tanto para la versión en CPU como para la versión en GPU. Se pretende de esta forma que los resultados no estén influidos por circunstancias ajenas al proceso de cálculo que pudieran alterarlos, y así obtener resultados más representativos.

A continuación se muestran ejemplos de los tiempos de resolución de tres problemas de pequeño tamaño, en los cuales el uso de la GPU no compensa en los tiempos totales de ejecución. Como se puede ver en la figura 3, los tiempos de ejecución son pequeños, menores de un segundo, porque se trata de problemas de entre 40 y 110 restricciones. En estos tamaños, aunque el tiempo de lectura y preproceso de los datos es esencialmente el mismo en ambas versiones, el tiempo que se necesita para coordinar la ejecución de la CPU y la GPU sobrepasa con creces el que se necesita para resolver el problema directamente en la CPU.

Por otra parte, se han analizado problemas de mayor tamaño, como aquéllos cuyos resultados se muestran en la figura 4. Es importante resaltar que en este caso los tiempos de resolución ya no se miden en milisegundos sino en segundos. También resulta interesante constatar el cambio de tendencia que se muestra, ya que en estos casos de mayor tamaño los tiempos totales empleando la GPU son bastante menores que en la versión CPU.

Además, puede verse cómo la tarjeta más potente (GTX 480) obtiene mejores resultados que la tarjeta menos potente (GTX 260), como era de esperar, aunque la relación entre los tiempos de ejecución es inferior a la proporción al número de núcleos que posee cada tarjeta.

Esta relación no es posible verla entre las GPUs y la CPU, teniendo en cuenta la relación entre el número de núcleos y la velocidad de los mismos: la CPU es en teoría entre 4 y 6 veces más rápida que las GPUs empleadas, sin considerar el efecto del acceso a memoria, que en las GPUs es de más altas prestaciones que la memoria principal de la CPU, aunque un núcleo de la CPU es más potente y complejo que uno de la GPU. Esto puede ser debido

a que el proceso de paralelización llevado a cabo en las tarjetas no permite explotar del todo el potencial de las mismas. Este punto queda como una futura línea de investigación que merece la pena estudiar con más detalle para poder aprovechar al máximo la potencia de cálculo de las GPUs.

Para problemas pequeños, una razón para no obtener rendimientos mejores con las GPUs es que con pocas tareas paralelas la GPU no puede ocultar las latencias de acceso a memoria o a registros (que habitualmente se gestionan pasando la ejecución a otra tarea pendiente mientras se completa la operación de acceso a la memoria). Por otro lado, en estos problemas pequeños incluso el tiempo de inicialización de la tarjeta gráfica es significativo. Obviamente esta inicialización solamente se lleva a cabo en la versión GPU y en cuanto los problemas crecen en tamaño, este tiempo se convierte en un dato prácticamente irrelevante comparado con el resto de tiempos de ejecución, ya que la inicialización es del orden de pocos milisegundos (entre 30 y 50, dependiendo de problema y de la tarjeta).

Esta relación directa entre mejora de tiempo de GPU frente a tiempo de CPU y número de restricciones se puede ver de forma clara en la figura 5, donde se encuentran representados estos valores para todos los problemas que se han probado. En la figura se puede ver cómo al aumentar el número de restricciones del problema, las GPUs son capaces de obtener cada vez mejores rendimientos.

La relación entre la diferencia de rendimiento entre versiones y el número de restricciones vista en el punto anterior se justifica porque el número de restricciones marca el tamaño de la matriz de la base usada en los cálculos (número de restricciones al cuadrado), entre otras matrices y vectores que intervienen en el algoritmo. Esta matriz se usa en muchos de los cálculos, por lo que al aumentar mucho su tamaño la versión CPU tenderá a tardar mucho más tiempo, mientras que la versión en GPU, al aprovechar la paralelización, no se ve tan afectada por el tamaño de la matriz de la base.

También se puede apreciar la diferente tendencia que marcan las dos GPUs, notándose la mejora en la 480 GTX debido a su mayor número de núcleos de cálculo. Por último, también puede comprobarse lo anteriormente indicado: para un número de restricciones menor de unas 600, no compensa emplear la versión en GPU porque no se obtiene ganancia de tiempo.

Para intentar ver un poco más en detalle dónde reside el cuello de botella que limita la paralelización de este algoritmo se van a analizar los tiempos de ejecución de los bloques de la aplicación que

Figura 5. Ganancia de tiempos totales con el empleo de GPU frente a CPU

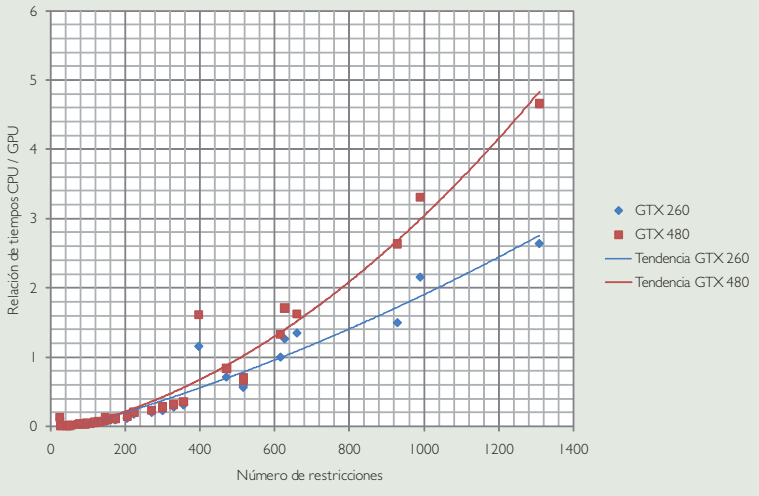
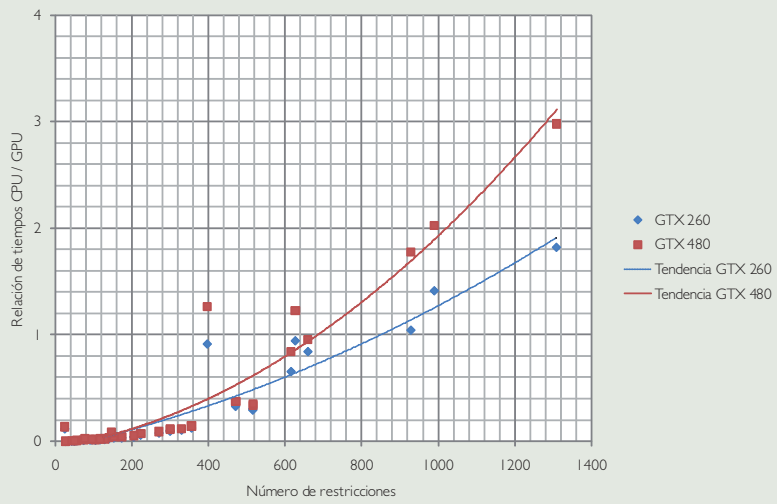


Figura 6. Ganancia de tiempos en el módulo de proceso con el empleo de GPU frente a CPU



se ven en la figura 2. Tanto el módulo de entrada como el módulo de preproceso y control realizan las mismas tareas en ambas versiones, y por tanto sus tiempos son prácticamente iguales. Por tanto, se van a analizar los tiempos de ejecución de los módulos de proceso (figura 6) y de cálculo de la inversa (figura 7), respecto al número de restricciones. En estas gráficas se puede ver cómo la ganancia en tiempos de resolución viene marcada por el módulo de proceso, que marca también el carácter general del crecimiento. Por otra parte, el módulo de inversión de la matriz de la base puede alcanzar unas ganancias mayores, de hasta 30 veces con respecto a la versión en CPU. Pero como supone un tiempo de ejecución de entre un 2% y un 6% del tiempo total, por mucho que se mejore en este apartado, no se puede transmitir apenas esa ganancia al tiempo total.

Conclusiones

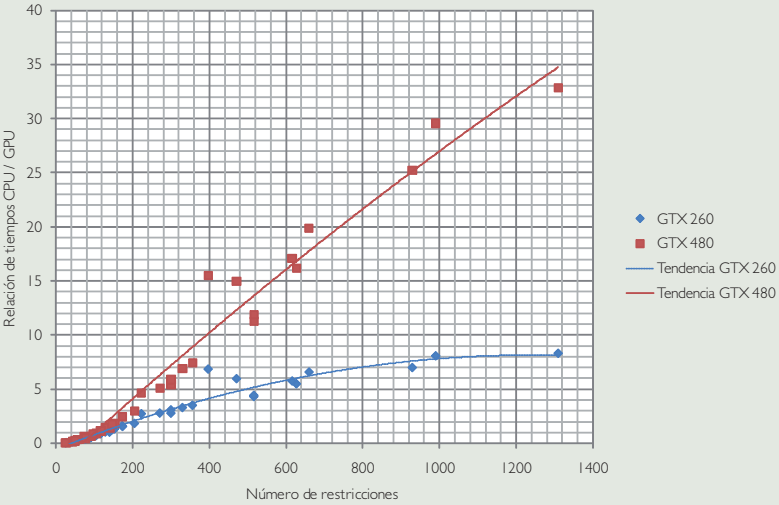
En este artículo se ha presentado un sistema de cálculo paralelo empleando tarjetas gráficas (GPUs) convencionales, que se encuentran presentes habitualmente en los ordenadores domésticos. De esta forma se abre esta nueva posibilidad de cálculo, empleando recursos que no requieren una gran inversión económica.

Se ha presentado de manera breve una aplicación que se ha desarrollado empleando la tecnología CUDA de NVidia, y se han hecho ejecuciones comparando los resultados obtenidos en una CPU y en dos GPUs distintas: una con 192 núcleos y otra con 480 núcleos.

Se han analizado los resultados de paralelizar el algoritmo símplex revisado, y se ha podido concluir que los beneficios de la paralelización se pueden empezar a observar cuando el tamaño del problema supera un umbral mínimo. En los resultados prácticos obtenidos se ha estimado ese tamaño mínimo como de 600 restricciones, por debajo del cual no merece la pena emplear GPUs.

Sin embargo, no se han podido alcanzar los incrementos de velocidad esperables por el gran incremento del número de núcleos (teniendo en cuenta la diferente velocidad de proceso de las dos plataformas, CPU y GPU). Esto es probablemente debido a que es posible mejorar la paralelización del algoritmo, por lo que todavía no se ha podido alcanzar el máximo rendimiento de las GPUs, quedando como una futura línea de investigación. ∎

Figura 7. Ganancia de tiempos en el módulo de inversión de la matriz con el empleo de GPU frente a CPU



Bibliografía

[CUDA11]: CUDA Zone, página de NVidia sobre la tecnología CUDA y sus aplicaciones: http://www.nvidia.es/object/cuda_home_new_es.html (última visita en febrero de 2011).

[GPGPU11]: GPGPU.org, página de análisis y últimos desarrollos en el cálculo con GPUs: http://gpgpu.org/ (última visita en febrero de 2011).

[HILLIE01]: Hillier, F. Investigación de Operaciones. Ed. McGraw Hill, 7ª Edición. 2001.

[NETLIB11]: The Netlib repository. Colección de problemas lineales en http://www.netlib.org/lp/data/ (última visita en febrero de 2011).

[OWENS08]: Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E. y Phillips, J.C.;"GPU Computing," *Proceedings of the IEEE*, vol. 96, no.5, pp.879-899, May 2008.

[STREAM11]: ATI Stream Technology, página de ATI sobre la tecnología Stream y sus aplicaciones: www.amd.com/stream (última visita en febrero de 2011).