Ingeniare. Revista Chilena de Ingeniería
ISSN: 0718-3291
facing@uta.cl
Universidad de Tarapacá
Chile

Uribe-Paredes, Roberto; Cazorla, Diego; Arias, Enrique; Sánchez, José Luis
Un sistema heterogéneo Multicore/GPU para acelerar la búsqueda por similitud en estructuras métricas
Ingeniare. Revista Chilena de Ingeniería, vol. 22, núm. 1, enero, 2014, pp. 26-40
Universidad de Tarapacá
Arica, Chile

Un sistema heterogéneo Multicore/GPU para acelerar la búsqueda por similitud en estructuras métricas

A Multicore/GPU heterogeneous system to accelerate the similarity search over metric structures

Roberto Uribe-Paredes[1] Diego Cazorla[2] Enrique Arias[2] José Luis Sánchez[2]

Recibido 12 de marzo de 2013, aceptado 27 de junio de 2013
Received: March 12, 2013 Accepted: June 27, 2013

RESUMEN

En la actualidad, la búsqueda por similitud en espacios métricos representa una línea de investigación de interés debido a sus múltiples campos de aplicación. Sin embargo, cuando en dichas aplicaciones aparecen grandes volúmenes de datos, se hace más que necesario el poder acelerar la búsqueda de las diferentes consultas en tales cantidades de datos. Una manera de llevar a cabo dicha aceleración pasa por el uso de *clusters*, multiprocesadores o *clusters* de multiprocesadores. En la actualidad, ha surgido con fuerza la posibilidad de utilizar aceleradores gráficos (GPU) como vehículo para acelerar aplicaciones a un muy bajo coste. En estos casos, la relación CPU/GPU no es de par a par y por tanto se denominan sistemas heterogéneos. Para explotar dichos sistemas heterogéneos se requiere una programación también heterogénea que emplee a la vez la CPU y la GPU. En este trabajo se realiza una verdadera programación heterogénea en el que tanto CPU como GPU están trabajando en forma simultánea y por tanto se aprovecha al máximo la arquitectura subyacente. Se presenta la implementación de una estructura genérica adaptada para un sistema multiprocesador con una GPU, mostrando los resultados experimentales en términos de tiempo y *speed-up*. Se muestra experimentalmente las ventajas comparativas al insertar GPU a una plataforma multicore, así como el análisis del consumo energético.

Palabras clave: Búsqueda por similitud, espacios métricos, consultas por rango, plataformas basadas en GPU, plataformas multicore.

ABSTRACT

Nowadays, similarity search on metric spaces is becoming a research field of interest due to the fact of its application to different scientific areas. However, when these applications produce a huge amount of data, it is necessary to accelerate the searching process by means of parallel architectures such as clusters, multiprocessors (multicores) or clusters of multiprocessors (multicores). Currently, graphic accelerators have emerged as a technology that allows for good performance at a low cost. In order to exploit the underlined architecture formed by multicores and a graphic accelerator it is nedeed to carry out heterogeneous programming, where CPUs and GPU are working at the same time taking benefits of the complete architecture. In this paper, a suitable generic structure adapted to the multicore/GPU system is presented and the experimental results, obtained in terms of execution time and speed-up, show the advantages of using this heterogeneous system, as well as a study of power consumption.

Keywords: Similarity search, metric spaces, queries by range, GPU platform, multicore platform.

[1] Departamento de Ingeniería en Computación. Universidad de Magallanes. Avenida Bulnes 01855. Punta Arenas, Chile. E-mail: roberto.uribeparedes@gmail.com
[2] Departamento de Sistemas Informáticos. Universidad de Castilla-La Mancha. Avenida España s/n, 02071. Albacete, España. E-mail: diego.cazorla@uclm.es; enrique.arias@uclm.es; jose.sgarcia@uclm.es

INTRODUCCIÓN

La búsqueda de objetos similares sobre un gran conjunto de datos se ha convertido en una línea de investigación de gran interés debido a que estas técnicas pueden aplicarse a diferentes campos de ciencia y tecnología, como reconocimiento de voz e imagen, problemas de minería de datos, detección de plagios, búsqueda de objetos sobre bases de datos multimedia y muchas otras.

Sin embargo, cuando en dichas aplicaciones se tiene la necesidad de almacenar y procesar grandes volúmenes de datos es necesario aumentar el rendimiento en términos de tiempo de procesamiento, si queremos que las soluciones sean viables. En general, para dar respuesta a las necesidades computacionales o temporales requeridas, el uso del paralelismo se considera una de las técnicas más adecuadas para conseguirlo. En este sentido, los procesadores gráficos (*Graphic Processor Unit, GPU*) de las actuales tarjetas gráficas permiten un alto nivel de paralelismo a un muy bajo coste.

En este artículo se presentan diferentes aportaciones:

1. Una estructura de datos métrica genérica que se adecúa a la plataforma objetivo de este trabajo, esto es, una basada en GPU. Si bien los autores reconocen que la estructura no es novedosa desde el punto de vista de los espacios métricos, sí lo es su adecuación a la plataforma objetivo y, además, introducen una reflexión para simplificar las estructuras existentes de cara a ser llevadas a dicha plataforma que impone grandes restricciones de memoria y en la que hay que cuidar la forma de acceder a los datos si se quieren obtener las mejores prestaciones.
2. Una implementación heterogénea en la que se emplean una GPU y núcleos de cómputo. De esos núcleos o *cores*, uno de ellos se reserva para gestionar la GPU.
3. Análisis de la implementación heterogénea mediante una serie de experimentos que ponen de relieve la bondad de la solución. Este análisis no se hace solo en términos de tiempos de ejecución, sino que además se detalla en qué se invierte el tiempo y cuántas consultas realiza cada dispositivo de cara a obtener mejores conclusiones.

La estructura del artículo está compuesta por una primera sección que corresponde al marco teórico e introduce el concepto de búsqueda por similitud, describe la arquitectura y el modelo de programación de una GPU y resume el trabajo relacionado en esta línea de investigación. En la segunda sección se describen de manera breve la estructura implementada y los algoritmos sobre las plataformas utilizadas. La sección tercera presenta resultados y discusión respecto de las implementaciones sobre las plataformas *multicore*, GPU e híbrida. Por último, las conclusiones y trabajo futuro son presentados en la última sección.

Búsqueda por similitud

La búsqueda por similitud consiste en la búsqueda de objetos similares mediante una búsqueda por rango o de vecinos más cercanos en un espacio métrico. Se puede definir un espacio métrico como un conjunto X con una función de distancia $d:X^2 \rightarrow R^+$, tal que $\forall\ x,y,z \in X$, se deben cumplir las propiedades de: positividad ($d(x,y)>=0$ *y* $d(x,y)=0$ *sii* $x=y$), simetría ($d(x,y)=d(y,x)$) y desigualdad triangular ($d(x,y)+d(y,z)>=d(x,z)$).

Sobre un espacio métrico (X,d) y un conjunto de datos finito $Y \subset X$ se puede realizar una serie de consultas. La consulta básica es la **consulta por rango**. Sea un objeto $x \in X$, y un rango $r \epsilon R$. La consulta de rango r alrededor de x es el conjunto de puntos $y \in Y$, tal que $d(x,y)<= r$.

Un segundo tipo de consulta, que puede construirse usando la consulta por rango, es **los *k* vecinos más cercanos**. Sea un objeto $x \in X$, y un entero k. Los k vecinos más cercanos a x son un subconjunto A de objetos de Y, donde $|A|=k$ y no existe un objeto $y \notin A$ tal que $d(y,x)$ sea menor a la distancia de algún objeto de A a x [1].

En la literatura existe una serie de estructuras de datos métricas e índices cuyo objetivo es reducir las evaluaciones de distancias durante la búsqueda, y con ello disminuir el tiempo de procesamiento. Estas estructuras se basan bien en **pivotes** o bien en **clustering** [1].

Los algoritmos basados en clustering dividen el espacio en áreas o planos, donde cada área tiene un centro y se almacena alguna información sobre el área que permita descartarla completa solo con

comparar la consulta con su centro. Ejemplos de estos son **BST, GHT, M-Tree, GNAT, EGNAT** y muchos otros [1].

En los métodos basados en pivotes se selecciona un conjunto de pivotes y se precalculan las distancias entre los pivotes y todos los elementos de la base de datos. Los pivotes sirven para descartar objetos durante la búsqueda utilizando la desigualdad triangular. Algunas estructuras de este tipo son: **LAESA** [2], **FQT** y sus variantes [3], **Spaghettis** y sus variantes [4], **FQA** [5], **SSS-Index** [6], entre otras.

Otra posible clasificación agrupa en estructuras de tipo árbol y de tipo arreglo. Estas últimas son más adecuadas para ser implementadas en plataformas basadas en GPU [7] y son las que se considerarán como punto de partida para este trabajo. Las estructuras de tipo arreglo aplican directamente los pivotes para descartar elementos, diferenciándose entre ellas, muchas veces, solo en la forma de reducir el tiempo extra de procesamiento y no en la reducción de los cálculos de distancia.

Unidades de Procesamiento Gráfico

Hoy en día, las unidades de procesamiento gráfico (GPU) disponen de un alto número de núcleos o *cores* con un alto ancho de banda con memoria. Este tipo de dispositivos permite aumentar la capacidad de procesamiento respecto de las CPU [8]. De cara a explotar esa arquitectura inherentemente paralela a un coste muy reducido, ha surgido una tendencia denominada **Computación de Propósito General sobre GPU** o GPGPU que ha orientado la utilización de GPU sobre nuevos tipos de aplicaciones. De hecho, uno de los objetivos de las GPU es aumentar la capacidad de procesamiento explotando el paralelismo a nivel de datos para problemas paralelos o que se puedan paralelizar.

Para la implementación de programas en este tipo de plataformas, los principales fabricantes de GPU (AMD-ATI y NVIDIA) han proporcionado herramientas de programación que hacen esta más amigable. En este trabajo hemos utilizado CUDA. El modelo de Programación CUDA de NVIDIA considera a la GPU como un dispositivo capaz de ejecutar un alto número de hilos en paralelo. Este modelo hace una distinción entre el código ejecutado en la CPU (*host*) con su propia DRAM (*host memory*) y el ejecutado en GPU (*device*) sobre su DRAM (*device memory*). CUDA incluye herramientas de desarrollo de software C/C++, librerías de funciones, y un mecanismo de abstracción que oculta los detalles de hardware al desarrollador.

En CUDA, los hilos son organizados en bloques del mismo tamaño y los cálculos son distribuidos en una malla o *grid* de bloques. Estos hilos ejecutan el código de la GPU, denominado **kernel**. Un kernel CUDA ejecuta un código secuencial en un gran número de hilos en paralelo.

Los hilos en un bloque (hasta 1024 para la arquitectura Fermi) pueden trabajar juntos eficientemente, intercambiando datos localmente en una memoria compartida y sincronizando a baja latencia en la ejecución mediante barreras de sincronización. Por el contrario, los hilos ubicados en diferentes bloques pueden compartir datos solamente accediendo a la memoria global de la GPU o *device memory*, que es una memoria de más alta latencia.

Trabajo relacionado

Como se comentaba en la introducción de este artículo, existe una gran variedad de plataformas paralelas sobre las cuales se pueden implementar estructuras métricas. Tradicionalmente, las arquitecturas paralelas se han clasificado en arquitecturas de memoria distribuida o multicomputadores (*cluster*), arquitecturas de memoria compartida o multiprocesadores (actualmente multicores), o una mezcla de ambas, esto es, un *cluster* de multiprocesadores.

Para la programación en dichas arquitecturas se utilizaba diferentes bibliotecas de alto nivel como MPI o PVM y memoria compartida usando directivas de OpenMP [9].

Así pues, en el caso de una plataforma de tipo *cluster,* la paralelización consistía en distribuir, de la manera más balanceada o equilibrada (equitativa) posible los datos entre los diferentes nodos que configuran la máquina utilizando librerías como MPI o BSP.

Menos son los trabajos orientados hacia aplicaciones de memoria compartida. En general, algunos estudios analizan la distribución de consultas y datos sobre nodos *multicores*. Otras investigaciones proponen combinar procesamiento de consultas multihilo

totalmente asíncronas con masivamente síncronas basándose en el nivel de tráfico de consultas.

Actualmente, la mayoría del trabajo realizado se lleva a cabo sobre plataformas clásicas de memoria distribuida y compartida, existiendo pocos estudios en torno a plataformas basadas en GPU. Cabe destacar algunas soluciones generalistas que utilizan GPU y que solo abordan el problema de consultas *kNN* sin utilizar estructuras de datos. En general, las GPU básicamente se utilizan para paralelizar búsquedas exhaustivas por lo que no se utilizan estructuras métricas [10-11]. En dichos trabajos la paralelización se lleva a cabo en dos etapas. La primera consiste en construir la matriz de distancias y la segunda en el proceso de ordenamiento para obtener los resultados.

En [12] se presenta una variante de lo anterior. Este trabajo compara dos estrategias, la primera, al estilo de los trabajos anteriores. Sin embargo, la segunda estrategia, llamada *Heap Based Reduction*, propone resolver una consulta por cada bloque. Después de haber calculado todas las distancias para una consulta (exhaustivamente), envía en cada lanzamiento de kernel un solo bloque, manteniendo un *heap* por cada hilo del bloque. Cada *heap* de tamaño k se utiliza para almacenar los k vecinos más cercanos a partir de las distancias entre los elementos de la base de datos y la consulta.

En [12] y [7] se utilizan estructuras métricas sobre una GPU y comparan sus resultados con versiones secuenciales. En [7] se utiliza una estructura métrica y se comparan los resultados obtenidos para la búsqueda por rango con versiones secuenciales y multicore-CPU, mostrando una mejora notable al usar este nuevo tipo de plataforma.

En este trabajo se va un paso más adelante, mediante la realización de una implementación heterogénea que aproveche al mismo tiempo la plataforma *multicore* y la tarjeta gráfica.

BÚSQUEDA POR RANGO SOBRE SISTEMAS HETEROGÉNEOS DE MEMORIA COMPARTIDA

Como se comentó con anterioridad, al utilizar una plataforma basada en GPU se decidió emplear una estructura métrica basada en pivotes y de tipo arreglo, ya que es más conveniente para este tipo de plataformas. Además, y debido a la limitación de memoria de la GPU y a que ciertas operaciones son costosas en dicha plataforma, se decide simplificar las estructuras métricas existentes, como SSS-Index o Spaghetti, a una estructura métrica de tipo arreglo más sencilla en la que no sea necesario hacer selección de pivotes tal y como lo hace el método SSS-Index, ni reordenación (Spaghetti). Así pues, se define una estructura genérica, denominada *Generic Metric Structure (GMS*). Con el empleo de esta estructura métrica la elección de pivotes es aleatoria, con lo que se puede elegir el número de pivotes idóneo para, por un lado, obtener buenas prestaciones en cuanto a tiempo de ejecución y, por otro lado, no saturar la memoria de la GPU. Además, no necesita reordenación de la estructura, operación costosa, en cuanto a tiempos de ejecución, de realizar en GPU.

El funcionamiento del proceso de búsqueda es el que sigue. Durante la construcción de la estructura, se selecciona un conjunto de pivotes $p_1,...,p_k$, los cuales pueden o no pertenecer a la base de datos a indexar. De hecho, una estructura métrica genérica puede considerarse como una tabla de distancias entre los pivotes y todos los elementos de la base de datos. Es decir, cada celda almacena la distancia $d(y_i,p_j)$, siendo y_i un elemento de la base de datos.

Durante la fase de búsqueda por rango sobre esta estructura dada una consulta q y un rango r, el algoritmo consistiría en las siguientes acciones:

1. Para cada consulta q se calcula la distancia entre q y todos los pivotes $p_1,...,p_k$. Con esto se obtienen k intervalos de la forma $[a_1,b_1],...,[a_k,b_k]$, donde $a_i=d(p_i,q)-r$ y $b_i=d(p_i,q)+r$.
2. Los objetos, representados en la estructura por sus distancias a los pivotes, son candidatos a la consulta q si todas sus distancias están dentro de todos los intervalos.
3. Para cada candidato y se calcula la distancia $d(q,y)$, y si $d(q,y)<=r$, entonces el objeto y es solución a la consulta q.

La Figura 1 representa una estructura de datos métrica genérica (GMS) construida con 4 pivotes. La búsqueda por rango sobre esta estructura dada una consulta q y un rango $r=2$, es de la siguiente manera:

1. En la primera fase del algoritmo se calculan las distancias a los pivotes *d(q,pi)={8,7,4,6}* y considerando el rango, generará los intervalos: *{(6,10), (5,9), (2,6), (4,8)}*.
2. En la segunda fase, por ejemplo para el pivote *1*, todos los objetos cuyas distancias (almacenadas en las celdas) están dentro del intervalo *(6,10)*, no pueden ser descartados. Si está fuera del intervalo se descarta de inmediato. El proceso se repite con los pivotes restantes.
3. La tercera fase del algoritmo se aplica sobre aquellos objetos que no pudieron ser descartados durante el paso 2. En este caso, los objetos candidatos *2*, *13*, *15* son evaluados en forma directa con la consulta.

Implementaciones paralelas

La estructura genérica introducida en la sección anterior se construye en una etapa de preproceso y es cargada durante la ejecución del programa. Por tanto, esta etapa es independiente del dispositivo en el que se realice la búsqueda. No ocurre lo mismo con el algoritmo de búsqueda, que varía según el dispositivo donde sea resuelta la consulta.

1	2	3	4	link	Base de Datos
0	1	6	5	1	Objeto 1
8	7	5	6	2	Objeto 2
6	5	0	7	3	Objeto 3
5	6	7	0	4	Objeto 4
15	14	13	14	5	Objeto 5
10	9	9	7	6	Objeto 6
9	9	7	6	7	Objeto 7
7	8	7	7	8	Objeto 8
5	4	6	6	9	Objeto 9
8	7	7	8	10	Objeto 10
1	0	5	7	11	Objeto 11
2	2	8	6	12	Objeto 12
8	7	6	8	13	Objeto 13
8	9	6	9	14	Objeto 14
6	7	6	7	15	Objeto 15
11	2	10	10	16	Objeto 16
2	2	6	6	17	Objeto 17

Figura 1. Búsqueda sobre una estructura genérica construida usando cuatro pivotes. Las celdas marcadas en gris oscuro son aquellas que están dentro del intervalo de búsqueda. Las celdas tachadas con líneas son los objetos candidatos.

Búsqueda sobre una GPU

La implementación sobre la GPU se realiza siguiendo los tres pasos del algoritmo de búsqueda descritos en la sección anterior. Estos pasos se realizan en dos funciones de GPU o *kernels*:

1. La primera parte consiste en calcular las distancias entre el conjunto de consultas, *Q*, y el conjunto de pivotes, *P*. Para aprovechar al máximo las ventajas del dispositivo GPU, estas distancias se calculan en un solo *kernel* y se almacenan en una estructura de tipo matriz de tamaño *Q x P* para su utilización en los pasos posteriores. El *kernel* lanzará un conjunto de hilos equivalente a la cantidad de consultas realizadas, es decir, cada hilo se hace cargo de una consulta y procesará la distancia entre esta y todos los pivotes. Para este primer *kernel* cada hilo accederá a todos los pivotes, es decir, los pivotes serán datos compartidos por todas las consultas. Por esta razón, los pivotes son llevados a la memoria compartida de la GPU, optimizando así la utilización de este tipo de memoria y aprovechando su menor latencia. La estructura bidimensional que almacena las distancias será almacenada en la memoria global de la GPU.
2. La segunda y tercera parte del algoritmo son resueltas en un solo *kernel* que se ejecuta por cada consulta. En este caso, cada *kernel* determina si un objeto es candidato y posteriormente si es o no solución. La cantidad de hilos lanzados por el *kernel* corresponde a la cantidad de elementos (filas) de la estructura, equivalente al número de objetos en la base de datos. Cada hilo determina si un objeto es solución a la consulta. En primera instancia cada hilo accede a las distancias entre la consulta y los pivotes para determinar si es o no candidato. Seguidamente, si el objeto y_i es candidato, los hilos acceden a la consulta para calcular $d(q,y_i)$. Entonces, todos los hilos leerán las distancias de la consulta actual a los pivotes y, en caso de haber un objeto candidato, se accederá a la consulta. Por lo anterior, el proceso se optimiza al llevar desde memoria global a memoria compartida tanto la consulta como sus distancias a los pivotes. En [7] este proceso fue realizado en dos *kernels*.

Para ejemplificar el funcionamiento interno de la GPU, en la Figura 2 se puede observar el segundo *kernel* mostrando qué datos o conjuntos de datos están dentro de la memoria compartida y dentro de la memoria global. También se representa los distintos accesos que hacen los hilos a datos o estructuras ubicados en dichos bancos de memoria.

En este punto hay que destacar que la reducción en el número de *kernels* redunda en una mejora en las prestaciones de la implementación.

Búsqueda sobre una Plataforma Multicore
La implementación del método de búsqueda sobre la plataforma *multicore* es equivalente al algoritmo presentado en la sección de definición de la estructura. La primera parte del algoritmo se resuelve para todas las consultas, es decir, se construye una matriz auxiliar con las distancias entre pivotes y consultas distribuyendo el trabajo sobre todos los *cores* disponibles. Posteriormente, los pasos 2 y 3 se resuelven iterando sobre los elementos de la base de datos.

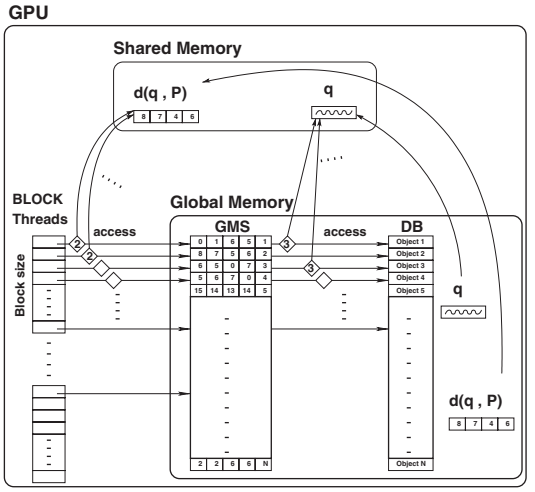


Figura 2. Funcionamiento del segundo *kernel* del algoritmo en GPU.

En general, el conjunto de consultas es distribuido sobre los *cores* resolviéndose completamente cada consulta sobre un *core*. Para resolver una consulta, cada *core* accede a la estructura y a la base de datos que se ubica en la memoria global del sistema. Se utiliza la sentencia *#pragma parallel for* de **OpenMP** para distribuir las consultas, e inicialmente la planificación utilizada es de tipo *static*.

Implementación Híbrida
En la implementación híbrida, una consulta es resuelta completamente por un solo dispositivo. Esto quiere decir que las consultas se van a intentar distribuir entre los *cores* y la GPU, replicando la estructura de datos y la base de datos. Así pues, la versión híbrida básicamente corresponde a la versión *multicore* donde uno de los *cores* resuelve su correspondiente consulta sobre la GPU. Las iteraciones son distribuidas utilizando la sentencia *#pragma parallel for* de **OpenMP**. El tipo de planificación definida en la sentencia *#pragma parallel for* determinará la cantidad de iteraciones asignada a cada core y la forma de distribuir las consultas sobre el conjunto de *cores.* Para el sistema híbrido se utiliza planificación del tipo *dynamic*, de manera que cuando un dispositivo finaliza una consulta, toma otra. Evidentemente, y debido a que la GPU es mucho más rápida que los *cores,* se espera que la GPU procese más consultas que los *cores* y que además, cuanto mayor sea el número de consultas resueltas por la GPU, mejores prestaciones se obtendrán.

EVALUACIÓN EXPERIMENTAL

Entorno experimental

En esta sección se muestran los resultados experimentales obtenidos al emplear la estructura GMS en sus versiones paralelas, esto es, tanto GPU y *multicores*, como híbrida. Para los casos de estudio se han considerado dos conjuntos de datos: el primero es un diccionario con palabras en español con 86.061 elementos. La función de distancia utilizada es la **distancia de edición**, definida como el mínimo número de inserciones, eliminaciones o sustituciones de caracteres necesarios para que una palabra sea convertida en otra. Para este espacio, los rangos de búsqueda son discretos y con valores entre r=1 y r=4. El segundo espacio considerado como caso de estudio corresponde a un conjunto de 112.682 histogramas de colores (vectores de dimensión 112) de una base de datos de imágenes. Para este espacio se seleccionó la distancia euclidiana. Los radios utilizados son aquellos que permiten recuperar el 0,01%, 0,1% y 1% de la base de datos.

Para ambos espacios, obtenidos de www.sisap.org, se construye la estructura con el 90% de los datos, dejando el restante 10% para consultas. Para construir la estructura GMS se utilizaron 32 pivotes

seleccionados aleatoriamente. En [13] se presenta un estudio en el que se indica que esta estructura y esta cantidad de pivotes son las más adecuadas para realizar las implementaciones secuenciales y sobre GPU en términos de tiempo de ejecución.

Se han elegido estas condiciones para la realización de los casos de prueba, ya que son las típicamente empleadas para este tipo de experimentos.

La plataforma utilizada corresponde a un 2 Quadcore Xeon E5530 a 2.4GHz y 48GB de memoria principal con una tarjeta Nvidia Tesla C1060 de 240 cores a 1.3GHz y 4 GB de memoria global, usando CUDA SDK v4.2 [14]. La codificación se ha realizado utilizando el lenguaje C (gcc 4.3.4) y librerías de OpenMP.

Resultados preliminares: *Multicore* versus GPU

Para tener una visión más amplia respecto de la utilización de una plataforma *multicore* y de una basada en GPU, la Figura 3 muestra los tiempos absolutos para la versión secuencial para una plataforma con 8 *cores* y para una plataforma con una GPU. Como era de esperar, los gráficos de la Figura 3 muestran la enorme diferencia al utilizar una plataforma *multicore* o GPU sobre una secuencial, observándose una disminución considerable del tiempo de procesamiento.

De hecho, para el caso de estudio de diccionario de Español (Figura 3, superior) y para rango r=4 (caso computacionalmente más costoso) la versión *multicore* es casi ocho veces más rápida que la versión secuencial, y la versión de GPU hasta casi 32 veces más rápida que la secuencial. Así pues, estamos ante un programa muy paralelizable en el que la versión *multicore* alcanza casi el máximo de ganancia de velocidad teórica que puede alcanzar (ocho). Respecto de la comparativa entre la versión *multicore* y de GPU se observa que la GPU es casi cuatro veces más rápida que la implementación *multicore*.

Para el espacio de histogramas de colores el comportamiento de las distintas versiones mantiene las mismas diferencias a medida que aumenta el radio de búsqueda, respecto de la implementación secuencial, aunque no ocurre lo mismo en la comparativa entre multicore y GPU. En este caso, una versión con ocho *cores* siempre es mejor que una con solo una GPU. Para el espacio de palabras esta regularidad en el comportamiento de las estructuras se pierde. De hecho, para rangos mayores a dos la plataforma GPU tiene un mejor comportamiento que aquella con ocho *cores*.

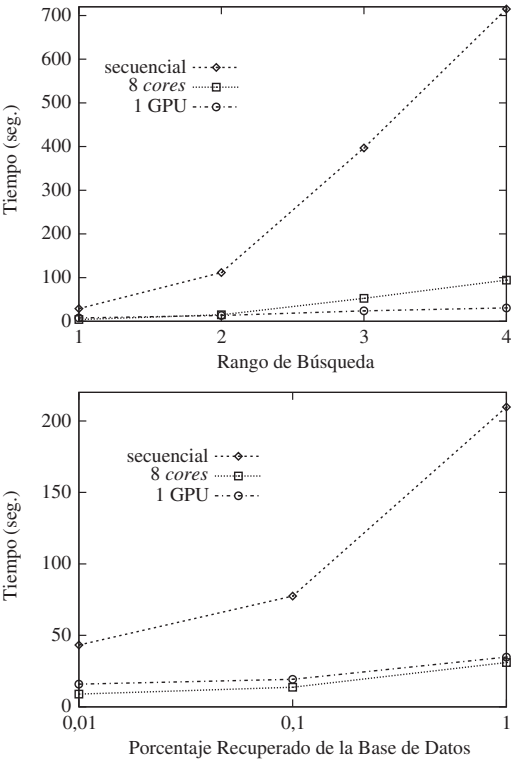


Figura 3. Tiempos de ejecución absolutos para las versiones secuencial (1 *core*), *multicore* de ocho núcleos y para la versión GPU utilizando la memoria compartida. Diccionario español de 86.061 objetos (superior) e histogramas de colores de 112.682 vectores (inferior).

Este comportamiento tiene una explicación totalmente racional. Primero, hay que considerar que al aumentar el rango o radio de búsqueda la cantidad de objetos a descartar disminuye notoriamente. Por ejemplo, usando los 16 pivotes, con r=1 se puede descartar más del 99,9% de la base de datos, sin embargo, con r=4 no se puede llegar a descartar más allá del 39,7%. Esto es diferente en el espacio de vectores donde estos valores van desde el 97% al 90,1% para el menor y mayor radio respectivamente, es decir, la estructura es más estable en este último espacio. Influye en el espacio de palabras el hecho de que la distancia sea discreta. Con lo anterior, se puede inferir que para descartar un objeto es necesario

utilizar la mayoría de los pivotes en el caso de los rangos mayores. Además, se puede decir que para el espacio de palabras es sumamente relevante la cantidad de hilos disponible para resolver una consulta, por lo cual, para rangos grandes el comportamiento de la GPU supera al de la plataforma solo con *cores*. Finalmente, se puede apuntar que la diferencia de comportamiento observada entre ambos espacios métricos, considerando el porcentaje descartado para el menor y mayor radio utilizado, también puede deberse a que los porcentajes de recuperación de las bases de datos con dichos radios son significativamente diferentes.

Sistema híbrido: *Multicore* + GPU

A la vista de los resultados obtenidos con anterioridad, se plantea que si la plataforma *multicore* funciona muy bien y la de GPU también, ¿cuánto de bien funcionaría si combinamos lo mejor de cada una? En esta subsección se muestran los resultados obtenidos sobre un sistema híbrido que combina el uso de los *cores* y de la GPU en forma conjunta. Al utilizar una sentencia *#pragma* para distribuir las consultas sobre la plataforma *multicore* y un esquema de planificación *static* no se obtienen los rendimientos adecuados debido a que uno de los *cores* (el que administra la GPU) procesa mucho más rápido una consulta que un *core* normal (sin GPU). Por tanto, en lugar de optar por una distribución a priori, se deja que los dispositivos vayan cogiendo las consultas conforme las van finalizando. Para ello se hace necesario implementar un tipo de planificación dinámica, que en **OpenMP** se denomina *dynamic*. En este tipo de planificación se puede indicar un tamaño de bloque de consultas que irá asignado dinámicamente a los dispositivos. Las Figuras 4 y 5 muestran los diferentes resultados obtenidos para esta planificación del tipo *dynamic*.

En los gráficos de las Figuras 4 y 5, el eje X representa el porcentaje de iteraciones asignadas a cada *core* (equivalente al número de consultas). Los valores van del 1% al 12,5%. También se graficó la información para una sola consulta (*1q*) para así tener una perspectiva más amplia del comportamiento del sistema. Para una mejor visualización de los resultados los diferentes rangos han sido separados en gráficos individuales.

Las gráficas demuestran que a menor número de iteraciones asignadas (menor tamaño de bloque)

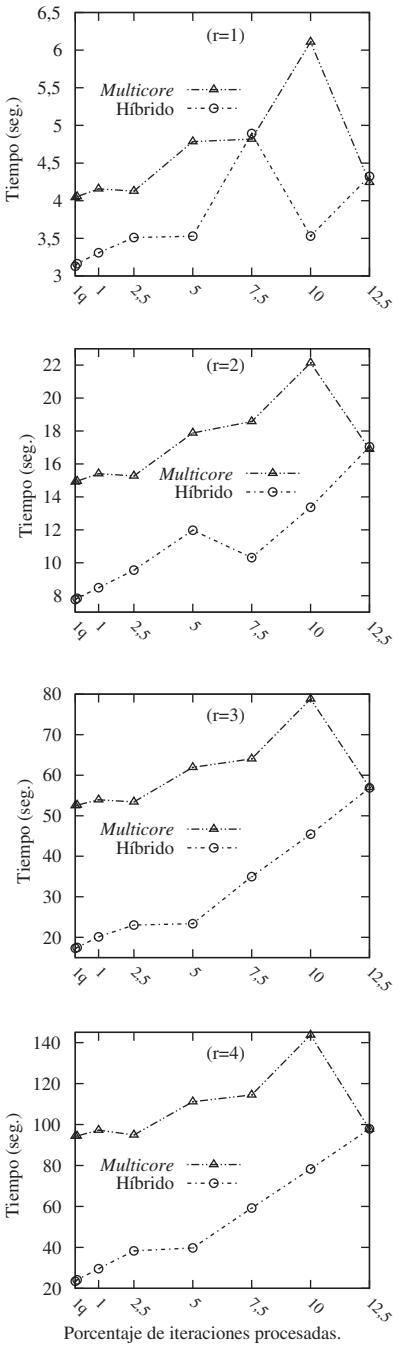


Figura 4. Sistema Híbrido (*Multicore* + GPU): Comportamiento para el espacio de palabras en español. Planificación dinámica asignando diferentes cantidades de iteraciones a los *cores*, comparativa entre la versión *multicore* de ocho núcleos y la versión híbrida de ocho núcleos más una GPU.

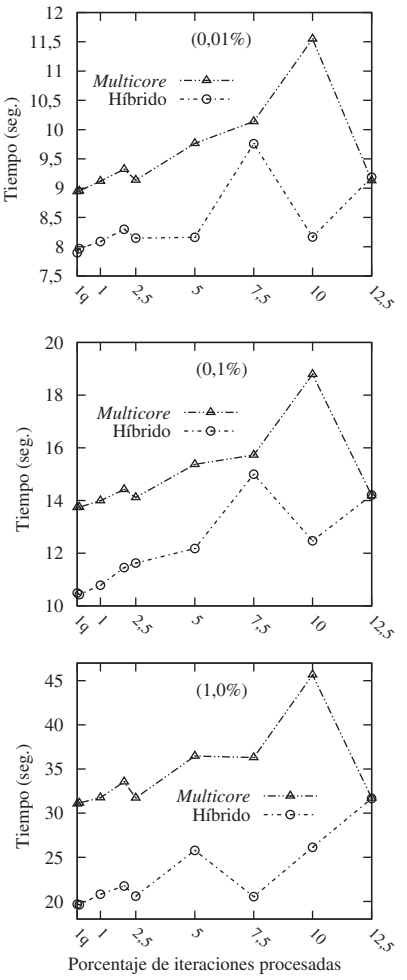


Figura 5. Sistema Híbrido (*Multicore* + GPU): Comportamiento para el espacio de histograma de color. Planificación dinámica asignando diferentes cantidades de iteraciones a los *cores*, comparativa entre la versión *multicore* de ocho núcleos y la versión híbrida de ocho núcleos más una GPU.

el sistema resulta mucho más eficiente, de hecho el mejor resultado se obtiene para el caso de una consulta. Lo anterior es debido a que a menor número de iteraciones fijas asignadas, el *core* que administra la GPU, que es mucho más rápido que el resto de los *cores*, puede llegar a procesar mayor número de iteraciones aprovechándose de su potencial. Si se considera que en cada iteración se resuelve una consulta, es decir, un dispositivo (*core* o GPU) resuelve en forma completa la consulta asignada, se justifica este comportamiento debido a que dos consultas diferentes no demoran lo mismo al ser resueltas, aunque sea utilizando el mismo dispositivo. Además, procesando consulta a consulta los *cores* no suponen ningún cuello de botella a las prestaciones que se pueden alcanzar con la GPU.

Se observan los valores extremos en las Figuras 4 y 5 debido a que las cantidades de bloques de consultas no corresponden a múltiplos de ocho (número de *cores*). Un ejemplo es la asignación de bloques con el 10% de consultas (10 bloques), en este caso, el sistema procesa el conjunto total en dos vueltas (caso *multicore*), pero en la segunda, solo trabajan dos de los ocho cores, produciéndose un cuello de botella. Levemente diferente es el comportamiento con la inclusión de una GPU, en este caso, a mayor número de consultas por bloque, más probabilidad existe de que la GPU quede en espera de la finalización de los cores.

Una visión más amplia de las ventajas al utilizar al máximo todos los recursos de una plataforma híbrida se obtiene al observar los gráficos de la Figura 6. En estos gráficos la versión híbrida incluida es la de mejor rendimiento (1q).

En todos los casos, se observa que las mejores prestaciones en términos de tiempo de ejecución se obtienen cuando se combinan *cores* y GPU. De hecho, se puede observar que la ganancia de velocidad obtenida respecto de la implementación secuencial es aproximadamente la suma de las prestaciones de la implementación *multicore* y la implementación GPU.

No obstante, se hace preciso hacer un estudio más detallado de los tiempos de ejecución para saber en qué se invierte el tiempo. Esto tiene especial relevancia en el caso de la GPU, ya que, necesariamente, hay que realizar transferencias de datos entre el dispositivo *host* (*core*) y la GPU.

En las Figuras 7 y 8 se muestra con más detalle cuál es el reparto de las consultas entre los diferentes recursos así como en qué se invirtió el tiempo para el caso de estudio del diccionario de español. En esta figura la información mostrada corresponde al núcleo del sistema, es decir, solo al proceso de búsqueda, parte 3 del algoritmo original.

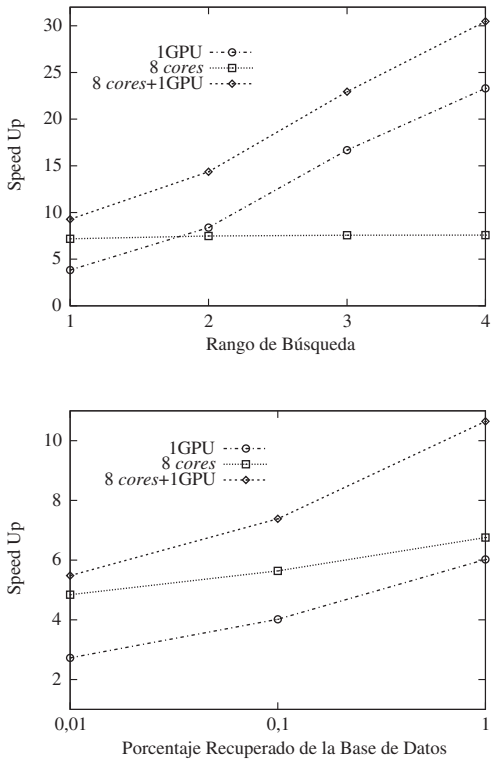


Figura 6. Comparación de *speed-ups* para un sistema híbrido (8 *cores*+1GPU) versus un sistema con solo una GPU y versus un sistema *multicore* con ocho *cores*. Diccionario español (superior) e histogramas de colores (inferior).

En las Figuras 9 y 10 se detallan los mismos resultados, pero para el caso de histogramas de color. Respecto del reparto de las consultas se observa, para todos los casos, que se obtienen los mejores tiempos de ejecución cuando la GPU es capaz de procesar mayor cantidad de consultas, esto es, se aprovecha mejor la capacidad de cómputo del acelerador gráfico.

Esto es lo que se esperaba según el tipo de planificador utilizado, que recordemos es dinámico.

En referencia al reparto del tiempo, los tiempos empleados en transferencia de datos hacia y desde la GPU, así como el tiempo empleado en la reserva de estructuras se mantienen prácticamente constantes, esto es, prácticamente no tiene influencia el tamaño de bloque. Sí que se reduce el tiempo de cálculo

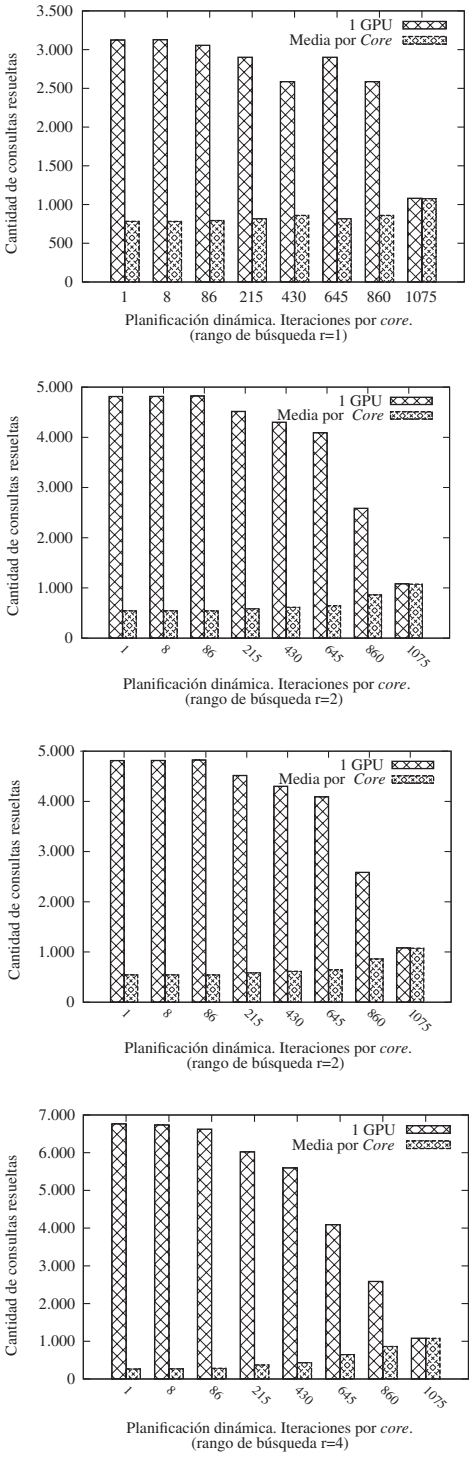


Figura 7. Número de consultas resueltas por la GPU y por los *cores* (en media) para el caso de estudio de diccionario de Español.

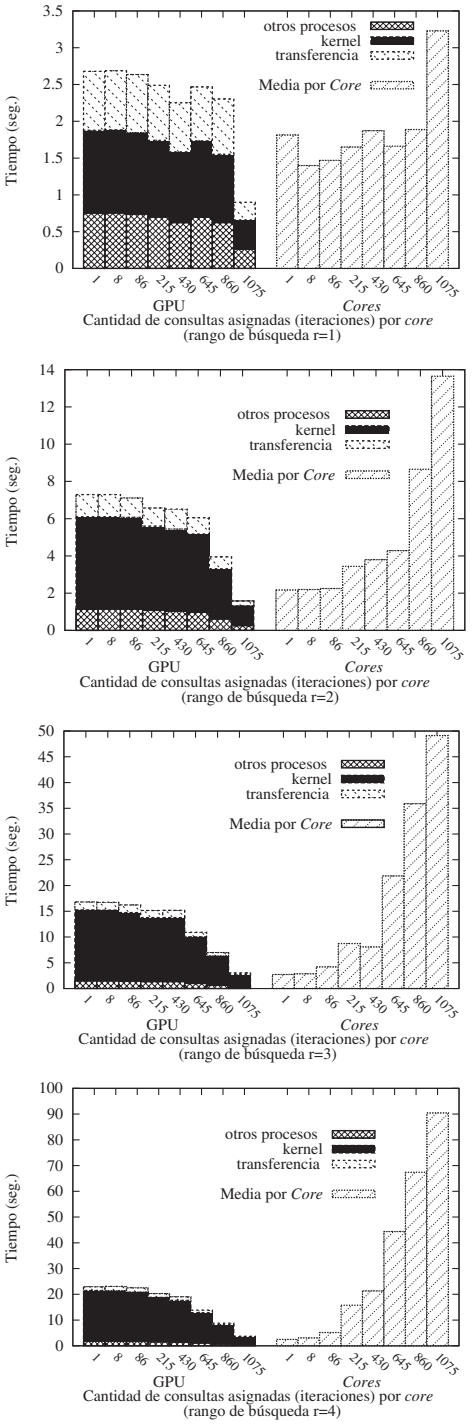


Figura 8. Detalle de en qué invierte el tiempo de ejecución la GPU y del tiempo que invierten los *cores* (en media) para el caso de estudio de diccionario de Español.

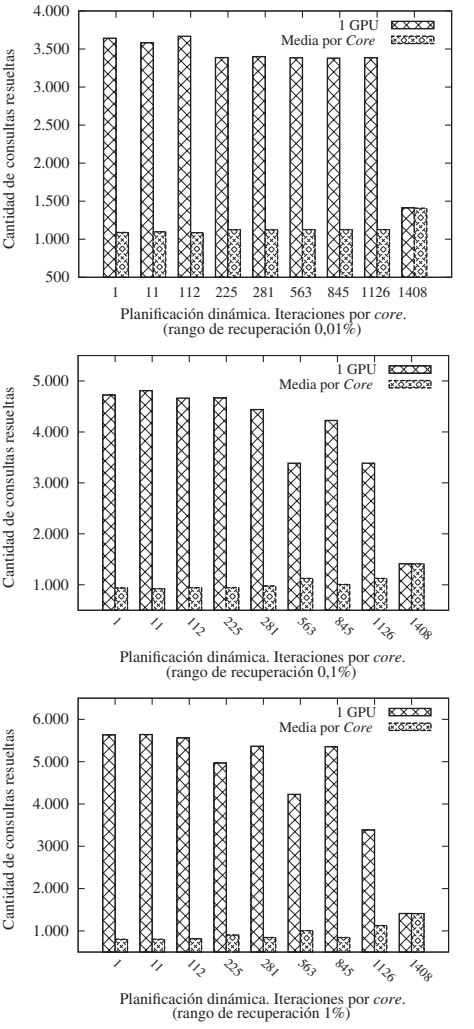


Figura 9. Número de consultas resueltas por la GPU y por los *cores* (en media) para el caso de estudio de histogramas de color.

cuando la GPU procesa un número de consultas mayor, que corresponde con el caso en que el tamaño de bloque es el mínimo, esto es, una consulta. A pesar de que en estas gráficas parece que el tiempo que la GPU emplea en procesado es mayor, el tiempo de ejecución global es menor puesto que la GPU está trabajando más que los *cores,* que son más lentos. El caso contrario es cuando el reparto es totalmente equitativo, esto es, para el caso en el que todos los dispositivos procesan el mismo número de consultas, ya que los *cores* representan un cuello de botella a la potencia de la GPU, equiparando la GPU a los otros dispositivos más lentos.

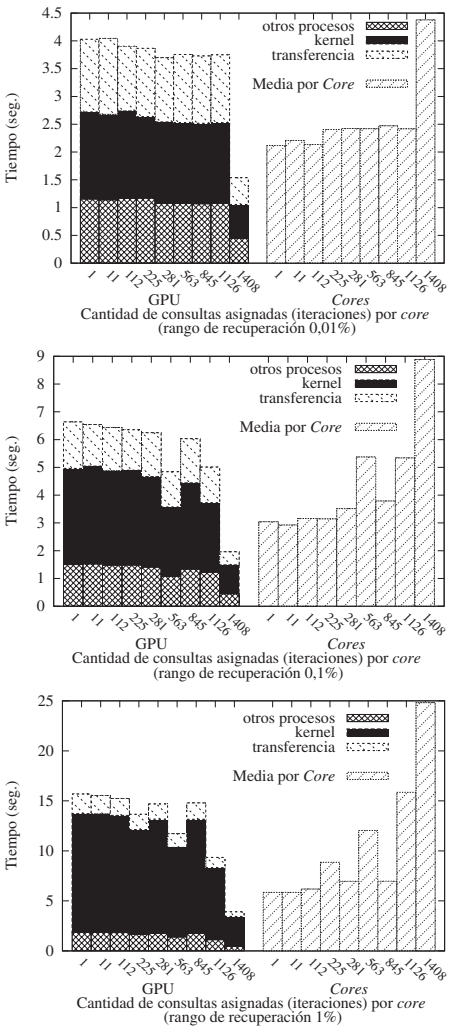


Figura 10. Detalle de en qué invierte el tiempo de ejecución la GPU y del tiempo que invierten los *cores* (en media) para el caso de estudio de histogramas de color.

Respecto del comentario sobre que el tamaño de bloque no tiene influencia en cuanto al tiempo de transferencia, no parece muy lógico. El primer pensamiento sería el que a mayor tamaño de bloque menor número de transferencias y por lo tanto se debería ahorrar tiempo. Sin embargo, no es así. ¿Por qué? Analizando la manera en que **OpenMP** realiza el reparto del tamaño de bloque, realmente lo que hace es enviar a un dispositivo un conjunto consecutivo de consultas, pero una a una, por lo tanto, no se gana nada en tener bloques. La opción sería engañar al sistema de reparto de **OpenMP** para que la asignación de bloques fuese real.

CONSUMO ENERGÉTICO

Hoy en día despierta gran interés no solo realizar implementaciones eficientes desde el punto de vista del tiempo de ejecución, sino que además resulta muy interesante que dicha implementación invierta la menor cantidad de energía posible. Esto es así, por la tendencia mundial a dirigir la computación hacia un nivel de Exaescala (*Exascale computing*).

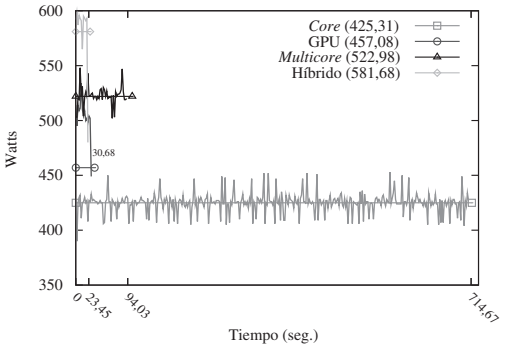
Por tanto, adicionalmente al estudio de prestaciones en términos de tiempo de ejecución y ganancia de velocidad o *speed-up*, se ha realizado un estudio para analizar el coste energético de la implementación propuesta. El coste energético se mide en términos de vatios (*watts*) consumidos por la plataforma, y en julios (*joules*) en el que se relaciona el consumo con el tiempo empleado en resolver el problema dado. Las medidas han sido tomadas utilizando el analizador de potencia PZ4000 de la casa YOKOGAWA [15].

En la Figura 11 se observa cómo el introducir una GPU produce un aumento del consumo de energía, pero por otro lado implica una reducción drástica del tiempo de ejecución. En la Figura 11 se muestran los resultados para los valores de *r=4* para el diccionario español y recuperación de *1%* para el histograma de colores. Cada línea en la gráfica representa a una plataforma y el consumo en watts (potencia) en los intervalos de toma de muestras durante la ejecución del programa. Entre parentesis, a cada plataforma se le agrega el promedio de la potencia consumida.

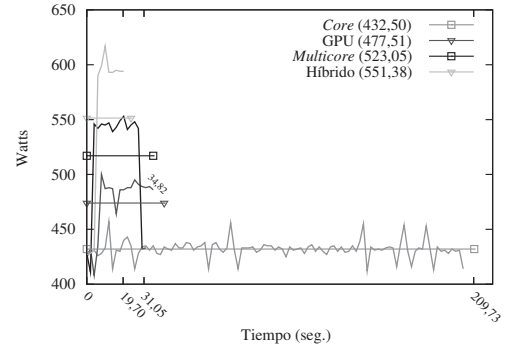
En aras de la claridad de los resultados, en las Tablas 1 y 2 se relaciona el consumo de potencia (watts) con el tiempo, esto es, se presentan los joules. En particular, la Tabla 1 presenta los resultados en joules para el caso de estudio del Diccionario de Español, mientras que la Tabla 2 presenta los resultados para el caso de estudio de histograma de color. Indicar que los joules se han calculado como el promedio del consumo de potencia multiplicado por el tiempo invertido por la implementación.

Además, se muestra el ahorro de consumo entre la plataforma paralela y el secuencial (1 *core*).

En general, se observa que al introducir una plataforma paralela, bien los multicores, una GPU o híbrido, el consumo de potencia, evidentemente, es mayor que el secuencial. No obstante, al reducir



(a) Diccionario Español



(b) Histograma de color

Figura 11. Consumo energético en watts para el caso de estudio de Diccionario de Español (a) e histograma de color (b), considerando las diferentes plataformas.

Tabla 1. Estudio de consumo de potencia para el caso de estudio de Diccionario de Español.

Plataforma	Tiempo	Watts	Joules	%
1 core	714,67	425,31	151.409,5	–
8 cores	94,03	522,98	49.176,1	68
1 GPU	30,68	457,08	14.023,2	90,7
Híbrido	23,85	581,68	13.873,1	90,8

Tabla 2. Estudio de consumo de potencia para el caso de estudio de histograma de color.

Plataforma	Tiempo	Watts	Joules	%
1 core	209,73	432,50	90.709,05	–
8 cores	31,05	523,05	16.240,70	82,1
1 GPU	34,82	477,51	16.626,74	81,7
Híbrido	19,70	551,38	10.862,19	88,02

drásticamente el tiempo de ejecución, los joules también se reducen drásticamente. Así pues, para el caso de estudio de Diccionario en Español, la reducción de consumo energético para el caso de la implementación híbrida, objetivo de este trabajo, es de más del 90%, mientras que para el caso de histograma de color la reducción es cercana al 88%. Por tanto, se desprende que si bien el consumo energético puntual es mayor al emplear el paralelismo (es evidente que al emplear más recursos se consuma más energía) al estar activos durante mucho menos tiempo, el consumo global es radicalmente menor.

CONCLUSIONES Y TRABAJO FUTURO

En este trabajo se ha llevado a cabo una implementación híbrida en la que se aprovecha la potencia computacional existente en la plataforma subyacente. En este caso, el trabajo ha sido repartido entre los ocho *cores* y la GPU de la plataforma de computación, donde uno de los *cores* se encargará de gestionar la GPU.

Puesto que la implementación se ha llevado a cabo en un entorno de memoria compartida, se ha utilizado la directiva *#pragma* de **OpenMP** para que, en función de la planificación indicada en la directiva, se asigne más o menos consultas en cada iteración a cada uno de los *cores* y, por ende, a la GPU. Para poder realizar el reparto de manera dinámica se ha optado por el planificador *dynamic* de **OpenMP**. Con este planificador se observa que a menor número de consultas asignadas a cada uno, mayor es el rendimiento del sistema. A priori, este resultado parece lógico, ya que permite que a la GPU se le asigne mayor cantidad de consultas y así aprovechar su potencial. Esta consecuencia se lleva al límite, cuando tan solo se asigna una única consulta.

Respecto de los resultados experimentales, la implementación híbrida es entre 9,5 y 30,5 veces más rápida que la versión secuencial para el espacio de palabras, y para el caso del espacio de vectores es entre 5,5 y 10,5 veces más rápida. Y todo ello con reducciones de consumo energético cercanas al 90%.

En relación con el trabajo futuro, el más inmediato vendría desde el punto de vista del enfoque de la implementación intentando pasar de dos *kernels* a un único *kernel* que permita mejorar las prestaciones de la implementación de la GPU.

El resultado de este trabajo abre la puerta a realizar implementaciones híbridas con diferentes niveles de paralelismo, para lo que se utilizará **MPI** y **OpenMP** en combinación con **CUDA**, y en plataformas multiGPU. Adicionalmente, como trabajo futuro, queda el realizar análisis de escalabilidad del sistema y observar el comportamiento de este al aumentar el tamaño de la base de datos y del número de consultas a resolver.

Por otro lado, y a la vista de los resultados obtenidos en detalle respecto del tiempo de ejecución, sería deseable realizar una implementación a bloques real, aunque sea burlando el sistema en que **OpenMP** realiza el reparto del bloque que se establece en la directiva.

AGRADECIMIENTOS

Este trabajo ha sido cofinanciado por el MICINN de España con subvención TIN2009-14475-C04-03 y por la Dirección de Investigación de la Universidad de Magallanes, Chile. Asimismo, agradecer la colaboración del Dr. Juan Antonio Villar, del Grupo de Redes y Arquitecturas de Altas Prestaciones del Instituto de Investigación en Informática de Albacete, por su colaboración en lo concerniente al consumo energético.

REFERENCIAS

[1] E. Chávez, G. Navarro, R. Baeza-Yates and J.L. Marroquín. "Searching in metric spaces". ACM Computing Surveys. Vol. 33, Issue 2, pp. 273-321. 2001.

[2] L. Micó, J. Oncina and E. Vidal. "A new version of the nearest-neighbor approximating and eliminating search (AESA) with linear preprocessing-time and memory requirements". Pattern Recognition Letters. Vol. 15, pp. 9-17. 1994.

[3] R. Baeza-Yates, W. Cunto, U. Manber and S. Wu. "Proximity matching using fixedqueries trees". 5th Combinatorial Pattern Matching (CPM'94). LNCS Vol. 807, pp. 198-212. 1994.

[4] E. Chávez, J. Marroquín and R. Baeza-Yates. "Spaghettis: An array based algorithm for similarities queries in metric spaces". The 6th International Symposium on String Processing and Information Retrieval (SPIRE'99). IEEE CS Press, pp. 38-46. 1999.

[5] E. Chávez, J. Marroquín and G. Navarro. "Fixed queries array: A fast and economical data structure for proximity searching". Multimedia Tools and Applications. Vol. 14, Issue 2, pp. 113-135. 2001.

[6] O. Pedreira and N.R. Brisaboa. "Spatial selection of sparse pivots for similarity search in metric spaces". The 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2007), Harrachov, Czech Republic. LNCS Vol. 4362, pp. 434-445. Springer, 2007.

[7] R. Uribe-Paredes, P. Valero-Lara, E. Arias, J.L. Sánchez and D. Cazorla. "Similarity search implementations for multi-core and many-core processors". The International Conference on High Performance Computing and Simulation (HPCS), pp. 656-663. 2011.

[8] W. Feng and D. Manocha. "High-performance computing using accelerators". Parallel Computing. Vol. 33, pp. 645-647. 2007.

[9] A. Grama, G. Karypis, V. Kumar and A. Gupta. Introduction to Parallel Computing. Addison Wesley. 2 edition. 2003.

[10] Q. Kuang and L. Zhao. "A practical GPU based kNN algorithm". The International Symposium on Computer Science and Computational Technology (ISCSCT), pp. 151-155. 2009.

[11] V. García, E. Debreuve and M. Barlaud. "Fast k nearest neighbor search using GPU". Computer Vision and Pattern Recognition Workshop, pp. 1-6. 2008.

[12] R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto and M. Marín. "kNN query processing in metric spaces using GPUs". The 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), Bordeaux, France. LNCS Vol. 6852, pp. 380-392. Springer. 2011.

[13] R. Uribe-Paredes, D. Cazorla, J.L. Sánchez and E. Arias. "A comparative study of different metric structures: Thinking on GPU implementations". The Lecture Notes in Engineering and Computers Sciences: Proccedings of The World Congress on Engineering (WCE 2012). London, England. 2012.

[14] NVIDIA CUDA C Programming Guide, Version 4.2. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. Fecha de consulta: 21/05/2013.

[15] PZ4000 Power Analizer. URL: http://tmi.yokogawa.com. Fecha de consulta: 21/05/2013.