Acelerando la Búsqueda por Rango con un Sistema Híbrido de Memoria Compartida.

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

Resumen— **En la actualidad, la búsqueda por similitud en espacios métricos representa una línea de investigación de interés debido a los múltiples campos de aplicación, tales como, reconocimiento de patrones, minería de datos, recuperación de objetos multimedia, entre muchos otros. Este artículo muestra resultados preliminares del comportamiento de una estructura de datos métrica sobre una plataforma híbrida de memoria compartida. Se presenta la implementación de una estructura genérica adaptada para un sistema multicore con una GPU. Se muestran los resultados experimentales en términos de tiempo y** *speed-up* **al utilizar todos los recursos del sistema, cores y GPU. Se muestra experimentalmente las ventajas comparativas al insertar GPUs a una plataforma multicore.**

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

I. Introducción

La búsqueda de objetos similares sobre un gran conjunto de datos se ha convertido en una línea de investigación de gran interés.

Aplicaciones de estas técnicas pueden ser encontradas en 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, la necesidad de almacenar y procesar grandes volúmenes de datos hace necesario aumentar el rendimiento en términos de tiempo de procesamiento. En general, se utilizan para ello diversas plataformas paralelas. En este sentido, nuevas tecnologías basadas en tarjetas gráficas (*Graphic Processor Unit, GPU*) permiten un alto nivel de paralelismo a un muy bajo coste.

El artículo presenta una estructura de datos métrica implementada sobre una plataforma híbrida. Mediante una serie de experimentos analiza algunos aspectos relevantes a considerar al utilizar CPU-cores y GPUs.

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 Sección II se describen brevemente la estructura implementada y los algoritmos sobre las plataformas utilizadas. La Sección III presenta resultados y discusión respecto de las implementaciones sobre las plataformas multicore, GPU e híbrida. Finalmente, las conclusiones y trabajo futuro son presentados en la Sección IV.

[1]Departamento de Ingeniería en Computación, Universidad de Magallanes, Chile. e-mail: roberto.uribeparedes@gmail.com

[2]Departamento de Sistemas Informáticos, Universidad de Castilla La Mancha, España. e-mail: {diego.cazorla,enrique.arias,jose.sgarcia}@uclm.es

A. Búsqueda por Similitud

La similitud se modeliza en muchos casos interesantes a través de un espacio métrico, y la búsqueda de objetos más similares a través de una búsqueda por rango o de vecinos más cercanos. Un espacio métrico es un conjunto $\mathbb{X}$ con una función de distancia $d : \mathbb{X}^2 \rightarrow \mathbb{R}$, tal que $\forall x, y, z \in \mathbb{X}$, se deben cumplir las propiedades de: positividad ($d(x, y) \geq 0$ *and* $d(x, y) = 0$ *ssi* $x = y$), simetría ($d(x, y) = d(y, x)$) y desigualdad triangular ($d(x, y) + d(y, z) \geq (d(x, z))$.

Sobre un espacio métrico $(\mathbb{X}, d)$ y un conjunto de datos finito $\mathbb{Y} \subseteq \mathbb{X}$, se pueden realizar una serie de consultas. La consulta básica es la *consulta por rango*. Sea un objeto $x \in \mathbb{X}$, y un rango $r \in \mathbb{R}$. La consulta de rango alrededor de x con rango r es el conjunto de puntos $y \in \mathbb{Y}$, tal que $d(x, y) \leq r$. Un segundo tipo de consulta, que puede construirse usando la consulta por rango, es *los k vecinos más cercanos*. Sea un objeto $x \in \mathbb{X}$ *y un entero* k. Los k vecinos más cercanos a x son un subconjunto $\mathbb{A}$ de objetos de $\mathbb{Y}$, donde $|\mathbb{A}| = k$ y no existe un objeto $y \in \mathbb{A}$ tal que $d(y, x)$ sea menor a la distancia de algún objeto de $\mathbb{A}$ a x.

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. Algunas de las estructuras basan la búsqueda en *pivotes* y otras en *clustering*. 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 sólo 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 variates [3], *Spaghettis* y sus variantes [4], *FQA* [5], *SSS-Index* [6] y otros.

Existen estructuras de tipo árbol y de tipo arreglo,

sin embargo, estas últimas son más adecuadas para ser implementadas en plataformas basadas en GPUs [7]. Las estructuras de tipo arreglo aplican directamente los pivotes para descartar elementos, diferenciándose entre ellas, muchas veces, sólo en la forma de reducir el tiempo extra de procesamiento y no en la reducción de los cálculos de distancia.

B. Unidades de Procesamiento Gráfico

Actualmente las unidades de procesamiento gráfico (GPUs) disponen de un alto número de cores con un alto ancho de banda. Este tipo de dispositivos permite aumentar la capacidad de procesamiento respecto de las CPUs [8]. Una tendencia denominada *Computación de Propósito General sobre GPU* o GPGPU ha orientado la utilización de GPUs sobre nuevos tipos de aplicaciones.

En general, las GPUs utilizan todos sus recursos en tener la mayor cantidad de elementos de procesamiento y gran ancho de banda con memoria, lo que posibilita una potencia de cálculo muy superior al de las CPUs. Así, uno de los objetivos de las GPUs es aumentar la capacidad de procesamiento explotando el paralelismo a nivel de datos para problemas paralelos o que se puedan paralelizar.

Existen nuevos lenguajes o extensiones para los lenguajes de alto nivel propuestos por algunos fabricantes, un ejemplo es 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 software C/C++, librerías de funciones, y un mecanismo de abstracción que oculta los detalles 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, lo 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.

C. Trabajo Relacionado

Actualmente, existe una gran variedad de plataformas paralelas sobre las cuales se pueden implementar estructuras métricas. En este contexto, muchos estudios se han enfocado inicialmente a plataformas de memoria distribuida usando bibliotecas de alto nivel como MPI o PVM, y memoria compartida usando directivas de OpenMP [9].

Algunos estudios se han enfocado a la paralelización de diferentes estructuras sobre plataformas de memoria distribuida y a la distribución de los datos y el balance de carga usando MPI o BSP, usualmente plataformas del tipo cluster.

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 en base al nivel de tráfico de consultas.

Al día de hoy 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 GPUs. Algunas soluciones generalistas que utilizan GPUs sólo abordan el problemas de consultas *kNN* sin utilizar estructuras de datos. En general, las GPUs básicamente se utilizan para paralelizar búsquedas exhaustivas por lo que no se utilizan estructuras métricas [10], [11]. En general, en los trabajos anteriores la paralelización es aplicada 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 2 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], [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 secuencial y multicore-CPU, mostrando una mejora notable al usar este nuevo tipo de plataforma.

II. Búsqueda por Rango sobre Sistemas Híbridos de Memoria Compartida

A. Preparando una Estructura de Datos Métrica Basada en Pivotes

Al utilizar una plataforma basada en GPU se decidió utilizar una estructura métrica basada en pivotes y de tipo arreglo. Para ellos se definió una estructura genérica.

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.

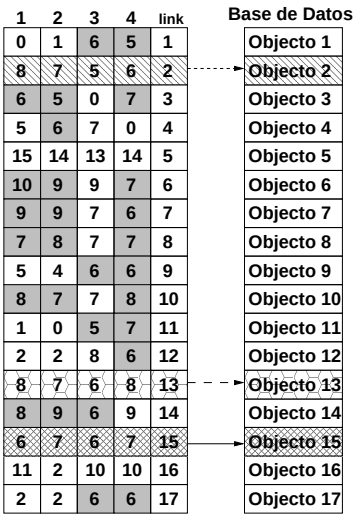

Fig. 1. Búsqueda sobre una estructura genérica construida usando 4 pivotes. Para cada consulta q con distancias a los pivotes $d(q, p_i) = \{8, 7, 4, 6\}$ y un rango de búsqueda $r = 2$, define los siguientes intervalos $\{(6, 10), (5, 9), (2, 6), (4, 8)\}$. Las celdas marcadas en gris oscuro son aquellas que están dentro del intervalo de búsqueda. Las celdas tachadas con líneas son los objetos candidatos.

Una estructura métrica genérica (*Generic Metric Structure: GMS*) 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 búsqueda por rango sobre esta estructura dada una consulta q y un rango r, el algoritmo sería:

1. Para cada consulta q, se calcula la distancia entre q y todos los pivotes $p_1, \ldots, p_k$. 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 $d(q, y) \leq r$, 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. En este caso, los objetos candidatos $2, 13, 15$ son evaluados directamente con la consulta.

B. Implementaciones Paralelas

La estructura se construye en una etapa de preproceso y es cargada durante la ejecución del programa. El algoritmo de búsqueda varía según el dispositivo donde sea resuelta la consulta.

B.1 Búsqueda sobre una GPU

La implementación sobre la GPU se realizó de la siguiente manera. Los tres pasos del algoritmo de búsqueda descritos en la Sección II-A se realizan en dos partes:

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 $Q \times P$ 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 ésta 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 los pivotes. En [7] este proceso fue realizado en dos kernels.

B.2 Búsqueda sobre una Plataforma Multicore

La implementación del método de búsqueda sobre la plataforma multicore es literal respecto del algoritmo presentado en la Sección II-A. 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, el paso 2 y 3 se resuelven iterando sobre los elementos de la base de datos.

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` de *OpenMP* para distribuir las consultas e inicialmente la planificación utilizada es de tipo `static`.

B.3 Implementación Híbrida

En general, una consulta es resuelta completamente por un solo dispositivo. El esquema utilizado sobre la plataforma distribuye las consultas sobre los cores y replica la estructura de datos y la base de datos. Por lo tanto, la versión híbrida básicamente correponde a la versión multicore donde uno de los cores resuelve su correspondiente consulta sobre la GPU. Las iteraciones son distribuidas utilizando la sentencia `#pragma` de *OpenMP*. Según el tipo de planificación definida en la sentencia `#pragma`, dependerá de 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 utilizó planificación del tipo `dynamic`.

III. Evaluacion Experimental

A. Entorno Experimental

En esta sección se muestran los resultados experimentales para la estructura GMS en sus versiones paralelas. 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 rangos entre $r = 1$ y $r = 4$. El segundo espacio 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 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 indica que esta cantidad es una de las más adecuadas para realizar las implementaciones secuenciales y sobre GPUs en términos de tiempo de ejecución.

Se han elegido estas condiciones para la realización de los tests de prueba ya que son las típicamente usadas 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 v3.2 [14]. La codificación se ha realizado utilizando el lenguaje C (gcc 4.3.4) y librerías de OpenMP.

B. Resultados Preliminares

Para tener una visión más amplia respecto de la utilización de una plataforma multicore y de una basada en GPU, la Figura 2 muestra los tiempos absolutos para las versiones secuencial, para una plataforma con 8 cores y para una plataforma con una GPU.

Como era de esperar, los gráficos de la Figura 2 muestran la enorme diferencia al utilizar una plataforma multicore o manycore sobre una secuencial, observándose una disminución considerable del tiempo de procesamiento.

Para el espacio de histogramas de colores el comportamiento de las distintas versiones mantienen las mismas diferencias a medida que aumenta el radio de búsqueda. En este caso, un versión con 8 cores siempre es mejor que una con sólo una GPU. Para el espacio de palabras esta regularidad en el comportamiento de las estructuras se pierde. Para rangos mayores a 2 la plataforma manycore (1 GPU) tiene un mejor comportamiento que aquella con 8 cores. Esta diferencia se debe básicamente a un conjunto de propiedades del espacio de palabras.

Primero, hay que considerar que al aumentar el rango o radio de búsqueda la cantidad de objetos a descartar disminuye notoramiente. 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. Finalmente, 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 sólo con cores.

C. Sistema Híbrido

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). Las Figuras 3 y 4 muestran los diferentes resultados obtenidos para una planificación del tipo `dynamic`. En los gráficos, el eje X se representa el porcentaje de iteraciones asignadas a cada core (equivalente al número de consultas). Los valores van del 1% al 12.5% y algunos valores intermedios. También se graficó la información para una sola

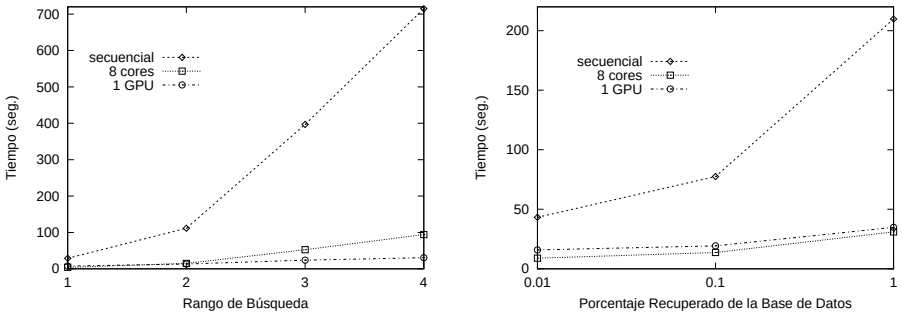

Fig. 2. Tiempos de ejecución absolutos para las versiones secuencial (1 core), multicore de 8 núcleos y para la versión GPU utilizando la memoria compartida. Diccionario español de 86.061 objetos (izquierda) e histogramas de colores de 112.682 vectores (derecha).

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 el sistema resulta mucho más eficiente. 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 tomar más iteraciones aprovechadose de su potencial. En el sistema híbrido propuesto este comportamiento resultó extremo, es decir, el mayor rendimiento se alcanzó cuando el número de iteraciones asignadas cada vez, a cada core, fue de 1. 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.

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 5. En estos gráficos la versión híbrida incluida es la de mejor rendimiento ($1q$).

IV. Conclusión y Trabajo Futuro

En este trabajo se ha llevado a cabo una implementación híbrida en la que se aprovecha el potencial computacional existente en la plataforma subyacente. En este caso, el trabajo ha sido repartido entre los 8 cores y la GPU, 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 los `#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 sólo se asigna una única consulta.

Con respecto a los resultados experimentales, la implementación híbrida es entre $9,5$ y $30,5$ veces más rápida que la versión secuencial paara el espacio de palabras, y para el caso del espacio de vectores es entre $5,5$ y $10,5$ veces más rápida.

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. Adicionalmente, como trabajo futuro, queda el realizar análisis de escalabilidad del sistema y observar el comportamiento de éste al aumentar el tamaño de la base de datos y del número consultas a resolver.

Agradecimientos

Este trabajo ha sido cofinanciado por el MICINN de España con subvención TIN2009-14475-C04-03 y por la Universidad Nacional de la Patagonia Austral, proyecto 29/A274-1, Argentina.

Referencias

[1] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, 2001, pp. 33(3):273–321.

[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.," in *5th Combinatorial Pattern Matching (CPM'94)*, 1994, LNCS 807, pp. 198–212.

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

[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, no. 2, pp. 113–135, 2001.

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

[7] R. Uribe-Paredes, P. Valero-Lara, E. Arias, J.L. Sanchez, and D. Cazorla, "Similarity search implementations for

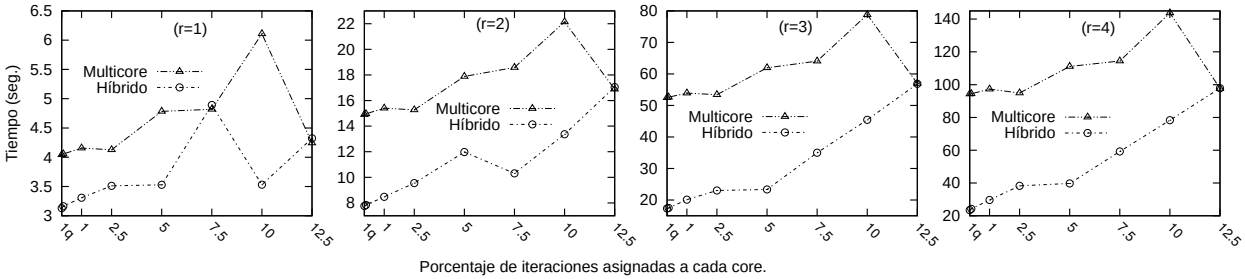

Fig. 3. Sistema Híbrido (Multicore + GPU): Comportamiento para el espacio de espacio de palabras en español (n=86.061 palabras). Planificación dinámica asignando diferentes cantidades de iteraciones a los cores, comparativa entre la versión multicore de 8 núcleos y la versión híbrida de 8 núcleos mas una GPU.

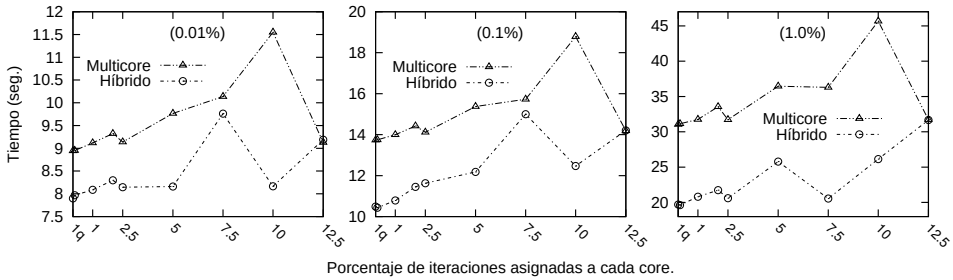

Fig. 4. Sistema Híbrido (Multicore + GPU): Comportamiento para el espacio de espacio de palabras en español (n=112.682 vectores). Planificación dinámica asignando diferentes cantidades de iteraciones a los cores, comparativa entre la versión multicore de 8 núcleos y la versión híbrida de 8 núcleos mas una GPU.

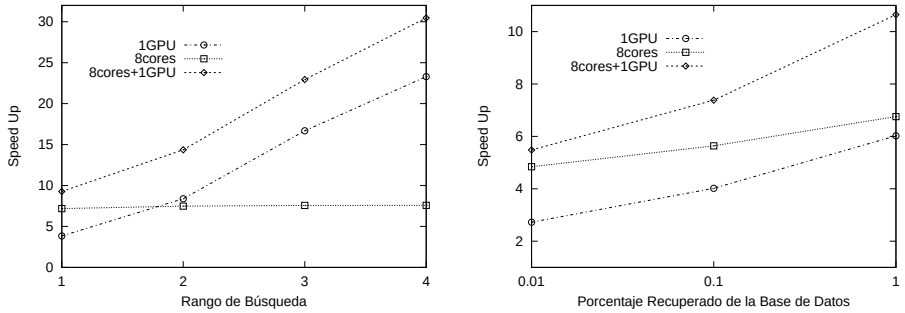

Fig. 5. Comparación de speed-ups para un sistema híbrido (8cores+1GPU) versus un sistema con sóla una GPU y versus un sistema multicore con 8 cores. Diccionario español de 86.061 objetos (izquierda) e histogramas de colores de 112.682 vectores (derecha).

multi-core and many-core processors," in *International Conference on High Performance Computing and Simulation (HPCS)*, 2011, pp. 656–663.

[8] Wu-Feng and Dinesh Manocha, "High-performance computing using accelerators," *Parallel Computing*, vol. 33, pp. 645–647, 2007.

[9] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta, *Introduction to Parallel Computing (2nd Edition)*, Addison Wesley, 2 edition, 2003.

[10] Quansheng Kuang and Lei Zhao, "A practical GPU based kNN algorithm," *International Symposium on Computer Science and Computational Technology (ISCSCT)*, pp. 151–155, 2009.

[11] Vincent Garcia, Eric Debreuve, and Michel Barlaud, "Fast k nearest neighbor search using GPU," *Computer Vision and Pattern Recognition Workshop*, vol. 0, pp. 1–6, 2008.

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

[13] Roberto Uribe-Paredes, Diego Cazorla, José L. Sánchez, and Enrique Arias, "A comparative study of different metric structures: Thinking on gpu implementations.," in *To appear: International Conference of Computational Statistics and Data Engineering (ICCSDE'12)*, London, England, July 2012.

[14] *NVIDIA CUDA C Programming Guide, Version 4.0*, NVIDIA, 2011, http://developer.nvidia.com/object/gpucomputing.html.