Búsqueda por similitud en espacios métricos sobre plataformas multi-core (CPU y GPU)

(Tesis presentada y aprobada en la Escuela de Postgrado de la Facultad de Ciencias Físicas y Matemáticas de la Universidad de Chile, Santiago, Chile.)

Autor: Ricardo J. Barrientos
Departamento de Ciencias de la Computación
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile, Santiago, Chile
Email: ribarrie@dcc.uchile.cl

Profesor Guía: Mauricio Marin
Yahoo! Research Latin America, Santiago, Chile.
Email: mmarin@yahoo-inc.com

***Resumen*—Similarity search has been widely studied in the last years, as it can be applied to several fields such as searching by content in multimedia objects, text retrieval or computational biology. These applications usually work on very large databases that are often indexed off-line to enable the acceleration of on-line searches. However, to maintain an acceptable throughput, it is essential to exploit the intrinsic parallelism of the algorithms used for the on-line query solving process, even with indexed databases. Therefore, many strategies have been proposed in the literature to parallelize these algorithms using distributed memory multiprocessor systems.**

This thesis is focused on the study of efficient management of threads in query processing on metric spaces. We propose efficient multi-thread algorithms for conventional multi-core processors, and for NVIDIA's GPU graphic cards. All our proposal are tested using databases of different nature, and we also validate our implementation in the context of real-time systems, when it is not affordable to wait for thousands of queries to fill the system before processing them all in parallel.

On a multi-core platform, we obtained the best results using a hybrid strategy, able to change between different distribution strategies depending on the current query traffic. On a GPU graphic card, the best results is obtained with a clustering based index, due to its good performance on the coalescing of I/O instructions.

***Index Terms*—Similariy search, metric spaces, distributed databases, multi-core algorithms, GPU, kNN, range search.**

I. Introducción

Las operaciones de búsqueda en bases de datos tradicionales son aplicadas a información estructurada, como información numérica o alfabética que es buscada de forma exacta. Es decir, es retornado el número o cadena de texto que es *exactamente igual* a la consulta dada.

Con la evolución de las tecnologías de información y comunicación, han emergido repositorios de información que no pueden ser estructurados de una forma tradicional. Tipos de datos, tales como audio, video o imágenes no pueden ser estructurados bajo tuplas o llaves, pero actualmente poseen la necesidad de ser consultados. Por lo tanto, se hace necesario la creación de nuevos modelos para búsqueda en repositorios no estructurados.

El primer concepto a tener en cuenta para poder entregar una solución, es el de *búsqueda por similitud* [1], es decir, búsqueda de los elementos de la base de datos que son similares o cercanos a la consulta dada. La similitud es medida con una función de distancia que satisface la propiedad de desigualdad triangular y el conjunto de objetos es llamado *espacio métrico*. Debido a que el problema ha aparecido en diversas áreas, las soluciones han provenido de campos tales como estadísticas, geometría computacional, inteligencia artificial, bases de datos, bioinformática, reconocimiento de patrones, minería de datos, la Web.

Una técnica muy difundida en los últimos años para indexar y buscar eficientemente objetos complejos son los llamados *índices* para espacios métricos. Se han propuesto numerosas estructuras de datos para computación secuencial basada en esta técnica, las que pueden alcanzar buena eficiencia frente a búsquedas en espacios multidimensionales que contienen gran número de objetos. No obstante, el diseño de estos índices ha sido orientado a la optimización de consultas individuales.

Actualmente los buscadores para la Web indexan docenas de billones de documentos y cientos de millones de otros tipos de objetos complejos tales como datos multimeda. Por ejemplo, recientemente ha aparecido el primer buscador comercial (*Google Goggles* [2]), que permite entregar una imagen como consulta, y aunque sólo presenta buen funcionamiento con ciertos objetos, es el principio de este tipo de aplicaciones.

Las cargas de trabajo en los grandes buscadores se caracterizan por la existencia de una gran cantidad de consultas siendo resueltas en todo momento sobre un conjunto muy grande de objetos (cientos de millones). En estos sistemas la métrica de interés a ser optimizada es el throughput, el que se define como la cantidad de consultas completamente resueltas por unidad de tiempo.

Para alcanzar altas tasas de respuesta sobre cientos de millones de objetos con miles de consultas por segundo, es necesario utilizar técnicas de computación paralela. En este caso la paralelización se realiza sobre decenas o cientos de *nodos* (procesadores) sobre los cuales se distribuyen uniformemente los objetos e índices, y donde cada nodo puede contener varias CPUs (o GPUs) bajo un entorno de memoria compartida.

La contribución principal de esta tesis está enfocada en la búsqueda eficiente en espacios métricos sobre uno de los nodos antes mencionados, utilizando un entorno de memoria compartida. Se proponen e implementan estrategias de búsqueda sobre diferentes plataformas paralelas. Se utilizaron bases de datos de distinta naturaleza con diferente tamaño. También, se varió la frecuencia de las consultas entrantes, con la finalidad de optimizar la solución de nuestros algoritmos dependiendo de la frecuencia con que arriban las consultas al sistema.

Para el presente trabajo se utilizaron como base índices métricos que ya existían en la literatura, los que son capaces de utilizar algoritmos secuenciales para resolver consultas en espacios métricos de forma eficiente. Como se mencionó anteriormente, estos índices han sido optimizados para resolver consultas individuales, y no para la resolución de un conjunto de ellas en paralelo sobre sistemas de memoria distribuida o compartida.

Las plataformas paralelas utilizadas fueron dos. La primera, un servidor multi-core con 2 CPU, cada una con 4 núcleos. La segunda, una tarjeta GPU (Graphic Processor Unit) modelo NVIDIA Tesla C1060, la que posee un total de 240 núcleos, distribuidos en 30 multiprocesadores. Si bien ambas son plataformas de hardware que utilizan un esquema multi-core, en el presente trabajo se utiliza el término *plataforma multi-core* para referirse al servidor multi-core convencional de 8 núcleos, pues la GPU es una plataforma multi-core que se aleja de lo convencional.

II. Conocimientos Básicos

II-A. Multi-Core

Actualmente son cada vez más frecuentes las arquitecturas de computadores que incluyen varios núcleos en una misma CPU, y son varias las librerías que permiten la programación multi-thread, de modo de poder ejecutar en paralelo varios threads, y cada uno ejecutándose en el núcleo que se desee.

Al trabajar sobre una plataforma de memoria compartida una de las dificultades que se presenta es el problema de la concurrencia, con el que hay que lidiar para permitir una correcta comunicación entre los threads. Con esto lo que se busca es reducir el tiempo de ejecución P veces (P = número de threads), lo cual puede ser difícil de alcanzar al trabajar sobre bases de datos para espacios métricos ([1], [3]).

El presente trabajo se desarrolló bajo dos plataformas paralelas de memoria compartida. La primera, una plataforma multi-core, y la segunda representada por una tarjeta gráfica NVIDIA.

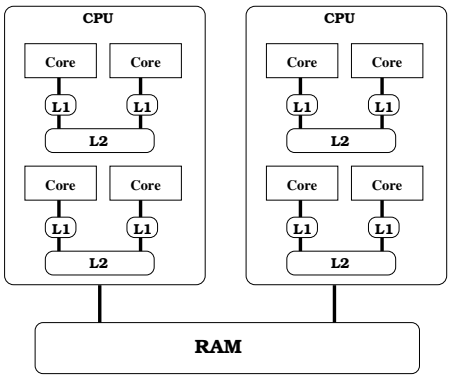


Figura 1: Plataforma multi-core.

La plataforma multi-core está representada por la Figura 1, en donde hay presente 2 CPUs, cada una de ellas con 4 núcleos. Cada núcleo posee una caché L1 y cada 2 núcleos se comparte una caché L2. Todos los núcleos comparten la misma memoria RAM.

Existen varios modelos y estándares que permiten programación multi-thread, tales como Pthreads([4]), TBB ([5]) u OpenMP ([6]). Para el presente trabajo la biblioteca seleccionada fue OpenMP debido a su gran nivel de abstracción, pues el código resultante difiere poco del secuencial, lo cual es muy útil al momento de desarrollar software. También OpenMP se viene perfeccionando desde 1997, por lo tanto los compiladores actuales son bastante robustos. También no está demás destacar la confianza que brinda el grupo que dirige OpenMP, el que está compuesto por personas de diferentes compañías (AMD, Intel, Sun MycroSystems y otros) y no de una en particular.

En todos los experimentos del presente trabajo, con la ayuda de la librería sched.h, se asigna siempre cada thread a un núcleo distinto. Esto asegura que no hay conflicto por recursos en un mismo núcleo.

II-B. GPU (Graphic Process Units)

Todas los experimentos sobre GPU han sido desarrollados usando el modelo de programación CUDA [7] de NVIDIA. Este modelo representa a la GPU como un coprocesador que puede ejecutar *kernels* sobre datos en paralelo, y provee extensiones del lenguaje C para (1) asignar memoria en la GPU, (2) transferir datos entre GPU y CPU, y (3) lanzar *kernels*.

Un *kernel* es una función que se ejecuta sobre la GPU, utilizando un gran número de threads en paralelo. Aquellos threads están organizados en un conjunto de *CUDA Blocks*. Los threads que pertenecen al mismo bloque (hasta 512 threads) pueden cooperar entre ellos mediante (1) compartir eficientemente datos a través de una memoria compartida de muy baja latencia, y (2) usando sincronización mediante barreras. Por otro lado, los threads de diferentes bloques sólo pueden coordinarse

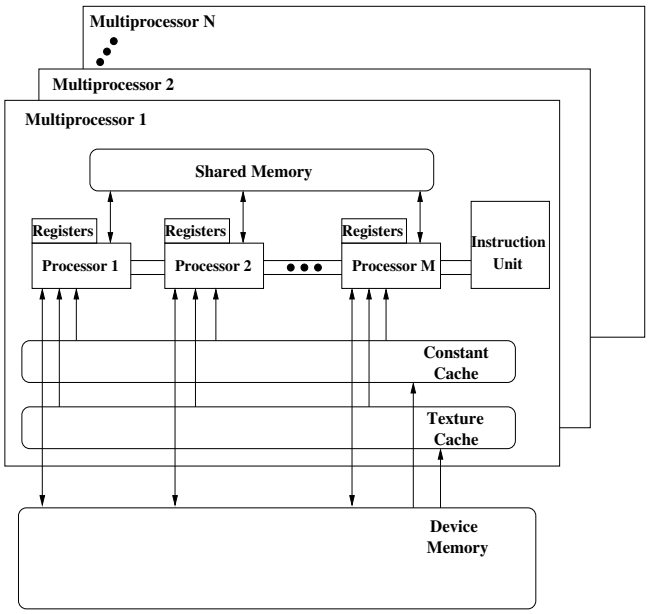


Figura 2: Arquitectura de hardware en una GPU.

mediante accesos a una memoria global de alta latencia (la memoria de la tarjeta gráfica). El programador puede especificar cuántos bloques y cuántos threads por bloque son asignados a la ejecución de un determinado kernel. Cuando un kernel es lanzado, se crean los threads por hardware y enviados a los núcleos de la GPU hasta que la ejecución de todos ellos finalice.

La Figura 2 muestra un esquema del actual hardware de una GPU NVIDIA. Los núcleos de la GPU están organizados en varios *multiprocesadores*. Cada uno de estos núcleos integran sus propias unidades funcionales y un gran número de registro, que permite la ejecución de cientos de threads de forma concurrente (para tolerar la gran latencia asociada con los accesos a la memoria de la tarjeta gráfica *device memory*). Los multiprocesadores integran una unidad de instrucción única, y una memoria compartida localmente (*shared memory*). La jerarquía de memoria también incluye memorias cachés para acelerar accesos a las memorias de texturas y de constantes, las que son compartidas por cada par de multiprocesadores. La abstracción de un CUDA Block está relacionada a esta organización: cada CUDA Block es ejecutado completamente en un único multiprocesador, que dependiendo de la disponibilidad de los recursos, puede ejecutar varios bloques de forma concurrente (actualmente el número de threads concurrentes por multiprocesador está limitado por los recursos asignados).

Como se mencionó anteriormente, la creación de los threads y su distribución es ejecutada completamente por hardware. La unidad mínima de ejecución no es un único thread, sino que un conjunto de ellos denominado *warp*. Los threads de un bloque son asignados a warps por su ID thread. En cada ciclo, se elige en cada multiprocesador el siguiente warp a ser ejecutado, es decir, no threads individuales, sino que warps son intercambiados. Si threads en un warp ejecutan una diferente secuencia de instrucciones, se estará decrementando el rendimiento que es posible alcanzar, dado que sólo los threads que ejecuten las mismas instrucciones pueden ser ejecutados simultáneamente.

Según las especificaciones de NVIDIA, uno de los principales factores de rendimiento es la maximización del *occupancy*, el cual indica la cantidad de warps que se pueden mantener activos en un multiprocesador. Actualmente las GPU NVIDIA soportan un máximo de 1024 threads de forma concurrente por multiprocesador, distribuidos en bloques de igual tamaño.

El otro factor de gran importancia para el rendimiento de la GPU es el uso de memoria. A pesar, que la GPU oculta latencias (con multithreading) sobre las operaciones de lectura/escritura, el tener cientos de threads simultáneamente accediendo a memoria global introduce gran presión sobre el ancho de banda. Reduciendo accesos a memoria, mediante el uso de memoria compartida explotando localidad inter-threads y reusando datos reduce en gran medida los tiempos de ejecución. Además, el patrón de acceso a memoria es relevante para permitir la fusión de instrucciones de lectura/escritura a device memory.

II-C. Espacios Métricos y Búsquedas por Similitud

Un *espacio métrico* es un conjunto X de objetos válidos, con una función de distancia $d : X^2 \to \mathbb{R}$, tal que $\forall x, y, z \in X$ cumple con las siguientes propiedades:

- Positividad : $d(x, y) \geq 0$, $x \neq y \Rightarrow d(x, y) > 0$.
- Simetría: $d(x, y) = d(y, x)$.
- Desigualdad triangular : $d(x, y) + d(y, z) \geq d(x, z)$.

Entonces, el par (X, d) es llamado *Espacio Métrico*.

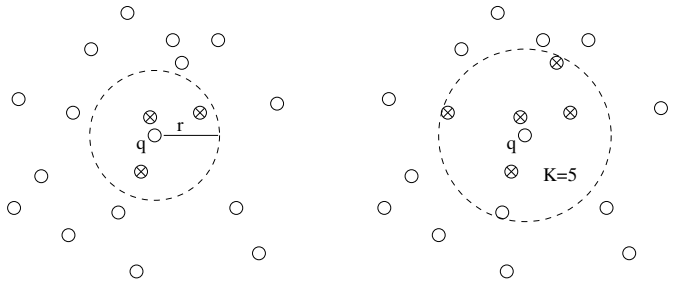
Sea $Y \subseteq X$ el conjunto de objetos que componen la base de datos. El concepto de *búsqueda por similitud* consiste en recuperar todos los objetos pertenecientes a Y que sean parecidos a un elemento de consulta q que pertenece al espacio X.

Las consultas por similitud sobre espacios métricos son básicamente dos:

1. **Consulta por Rango** (q, r) **[8]:** Sea un espacio métrico (X, d), un conjunto de datos finito $Y \subseteq X$, una consulta $q \in X$, y un radio $r \in \mathbb{R}$. La consulta por rango q con radio r es el conjunto de puntos $y \in Y$, tal que $d(q, y) \leq r$.

2. **Los** k **Vecinos más Cercanos** k**NN(**q**) [9]:** Sea un espacio métrico (X, d), un conjunto de datos finito $Y \subseteq X$, una consulta $q \in X$ y un entero k. Los k vecinos más cercanos a q son un subconjunto A de objetos de Y, donde la $|A| = k$ y no existe un objeto $y \in X - A$ tal que $d(y, q)$ sea menor a la distancia de algún objeto de A a q.

En la Figura 3 se ilustran ambos tipos de consulta. Para mayor claridad las consultas están realizadas sobre un conjunto de puntos en $\mathbb{R}^2$(espacio métrico). A la izquierda se muestra una consulta por rango con radio r y a la



(a) Consulta por rango (q, r) (b) Consulta kNN(q)

Figura 3: Ejemplos de consultas.

derecha una consulta por los 5-vecinos más cercanos a q. En este último caso, también se gráfica el radio necesario para encerrar los 5 puntos. Entonces, se observa que dada una consulta q y una cantidad k (en este ejemplo 5), es posible que existan distintas respuestas.

En [8] se ilustran varios métodos para resolver consultas de tipo kNN, los cuales son:

1. **Radio Creciente:** Este algoritmo de búsqueda de los k vecinos más cercanos está basado en un algoritmo de búsqueda por rango de la siguiente forma: Buscar q con radio fijo $r{=}a^i\epsilon$ ($a > 1$, $\epsilon \in codom(d)$), con $i = 0$ al comienzo e incrementarlo hasta que al menos k elementos son abarcados con $r = a^i\epsilon$. Luego, el radio es ajustado entre $r = a^{i-1}\epsilon$ y $r = a^i\epsilon$ hasta que k elementos son alcanzados.

2. **Radio Decreciente:** Este algoritmo de búsqueda comienza con una búsqueda por rango (r) con $r = \infty$, y una vez que se han alcanzado k elementos, el radio es ajustado a $r \leftarrow min(r, d(q, e))$ (siendo e un nuevo elemento con el que se debe comparar la consulta q), con la ayuda de una cola de prioridad.

II-D. Indexación

Teniendo en cuenta que la función de distancia es *computacionalmente costosa* de calcular, la indexación surge como una alternativa a la solución trivial de búsqueda exhaustiva (que toma $O(n)$ para una base de datos con n elementos).

Por lo tanto, se hace necesario preprocesar la base de datos, para lo cual se paga un costo inicial de construcción de un *índice* a fin de ahorrar computaciones de distancia al momento de resolver las búsquedas. Esto último se realiza descartando objetos utilizando la propiedad de desigualdad triangular.

El tiempo total de resolución de una búsqueda puede ser calculado de la siguiente manera:

$$T{=}\#evaluaciones\ de\ d \times complejidad(d) + tiempo\ extra\ de\ CPU + tiempo\ de\ E/S$$

En muchas aplicaciones la evaluación de la distancia es tan costosa que las demás componentes de la fórmula anterior pueden ser despreciadas. Todos los algoritmos de indexación particionan la base de datos Y en subconjuntos. Para particionar la base de datos existen dos grandes enfoques, éstos son: *algoritmos basados en pivotes* y *algoritmos basados en clustering o particiones compactas* [8].

Los algoritmos basados en pivotes realizan una preselección de objetos de la base de datos, estos objetos (o pivotes), se utilizan para descartar objetos usando la propiedad de desigualdad triangular. Algunos ejemplos son *SSS-Index* [10], *FQT* y sus variantes [11], *Spaghettis* y sus variantes [12], [13].

Los algoritmos basados en clustering dividen el espacio en áreas, donde cada área tiene un *centro* o *split*. Se almacena alguna información sobre el área que permita descartarla completamente mediante sólo comparar la consulta con su centro. Algunos ejemplos son *GNAT* [14], *EGNAT* [15], *M-tree* [16] y *Lista de Clusters* [17].

Si bien todos los métodos e índices mencionados anteriormente son eficientes en el descarte de elementos, todos ellos pierden eficiencia al trabajar sobre *espacios de alta dimensión*. Muchas de las técnicas de indexación tradicionales para espacios vectoriales tienen una dependencia exponencial en la dimensión de representación D del espacio, es decir, a medida que la dimensión crece, dichas técnicas se vuelven menos eficientes, y número de objetos que se pueden descartar es menor. La razón es que, a medida que crece la dimensionalidad, se reduce significativamente la capacidad de dividir el espacio de búsqueda que tienen los índices para espacios métricos. Con la intención de intentar medir la dificultad de búsqueda sobre un espacio métrico, en [8] se define el concepto de *dimensionalidad intrínseca* de un espacio métrico como $\rho = \frac{\mu^2}{2\sigma^2}$, donde μ y σ^2, son la media y la varianza del histograma de distancias entre elementos del mismo espacio.

II-E. Indices métricos

En el presente trabajo se seleccionaron y realizaron experimentos utilizando 5 índices distintos, debido a que éstos son ampliamente citados en la literatura técnica y porque todos poseen diferentes características abarcando un gran abanico de modelos de búsqueda. Estos son el *EGNAT* [15], *M-tree* [16], *SSS-Index* [10], *SSS-Tree* [18] y la *Lista de Clusters* (*LC*) [17].

En particular, el SSS-Index y LC también fueron seleccionados para ser utilizados sobre la GPU (Sección V), debido a que las características de su estructura son favorables para la tarjeta gráfica. A continuación se detallan estos dos últimos índices.

II-E1. SSS-Index [10]: Establece una tabla de distancias entre pivotes y todos los elementos de la base de datos. Los pivotes son seleccionados según el algoritmo 1. El resultado de éste es, un conjunto de pivotes que se encuentran al menos a distancia $M\alpha$ unos de otros (M=distancia máxima entre 2 elementos de la base de datos; α=parámetro real). Luego el índice será una tabla

Algoritmo 1 *SSS-Index*: algoritmo de selección de pivotes.

```
1: {Sea D la base de datos}
2: {Sea M la distancia máxima posible entre 2 elementos de D}
3: {Sea α una constante real}
4: PIVOTES ← {x_1}
5: for all x_i ∈ D do
6:     if ∀p ∈ PIVOTES, d(x_i, p) >= Mα then
7:         PIVOTES ← PIVOTES ⋃ {x_i}
8:     end if
9: end for
```

de distancias entre cada pivote y los elementos restantes. El valor óptimo de α observado en [10] fue para valores alrededor de $\alpha = 0{,}4$.

Con respecto a la búsqueda sobre este índice, se deben calcular previamente las distancias entre cada pivote y la consulta, luego con esta información más la tabla de distancias (previamente almacenada) se intenta descartar por desigualdad triangular cada elemento de la base de datos, si esto no es posible entonces se realiza una evaluación de distancia entre la consulta y el elemento no descartado.

*II-E2. Lista de Clusters (*LC*) [17]:* Este índice se basa en clustering y radio cobertor. Particiona la colección de datos en grupos denominados *clusters*, en donde los elementos similares forman parte del mismo conjunto.

Con respecto a su construcción, se debe elegir un centro $c \in D$ (D=colección de datos), y un radio r_c. Una *bola centro* (c,r_c) es el subconjunto de elementos de D que están a lo más a r_c de c. Definamos los siguientes subconjuntos de D:

$I_{D,c,r_c} = \{x \in D - \{c\}, d(c,x) <= r_c\}$

como el cluster de elementos internos que son parte de la bola (c, r_c), y

$E_{D,c,r_c} = \{x \in D, d(c,x) > r_c\}$

como los elementos externos.

Hay 2 formas de particionar el espacio: establecer un radio fijo para cada partición o establecer un tamaño fijo de elementos para cada cluster. En el presente trabajo se utilizó la segunda opción, pues ésta presenta mejores condiciones para lidiar con el paralelismo involucrado.

La selección de centros es de la siguiente manera: el primer elemento de la colección D es elegido como centro, luego el siguiente centro $c \in D$ será el que maximice la suma de distancias a todos los centros anteriores (tomando en cuenta la colección completa de elementos). Cabe notar que el primer centro elegido tiene preferencia sobre el resto cuando hay solapamiento, de este modo los elementos que pertenecen a la bola del primer centro son almacenados sólo en su cluster I, a pesar de que haya intersección con los cluster siguientes.

Durante el proceso de búsqueda de una consulta por rango (q,r) (descrito por el algortimo 2), si el primer cluster está dado por (c_1, r_1, I_1), se evalúa $d(q,c_1)$ y se agrega c_1 al conjunto resultado si $d(q,c_1) \leq r$. Luego, se realiza una

Algoritmo 2 *Lista de Cluster*: búsqueda por rango r para la consulta q.

busquedarango(Lista L, Consulta q, Rango r)

```
1: {El operador ":" es el constructor de lista.}
2: if L is empty then
3:     return;
4: end if
5: L = (c, rc, I):E
6: dist = d(c, q)
7: if dist <= r then
8:     Agregar c a los Resultados
9: end if
10: if dist ≤ rc + r then
11:     Buscar en I exhaustivamente
12: end if
13: if dist ≥ rc − r then
14:     busquedarango(E, q, r)
15: end if
```

búsqueda exhaustiva sobre los objetos en I_1 sólo si el radio de la consulta (q,r) intersecta la bola (c_1, r_1), es decir, sólo si $d(q,c_1) \leq r_1 + r$. Luego, se continúa con los clusters restantes, pero siguiendo el orden en que fueron creados. Sin embargo, si la consulta (q,r) está totalmente contenida en el cluster (c_i, r_i, I_i), es decir, si $d(q,c_i) \leq r_i - r$, entonces no es necesario continuar visitando los demás clusters, pues el proceso de construcción asegura que los elementos resultados están dentro de I_i o en un cluster previo.

Existen 3 casos posibles al momento de realizar una búsqueda. El primero, cuando el radio de búsqueda intersecta con el cluster, entonces se debe buscar exhaustivamente dentro del cluster y continuar con el resto de la lista de clusters. El segundo, cuando el radio de la consulta está completamente contenido dentro del cluster, entonces es posible detener la búsqueda. El tercero, es cuando no existe intersección entre el radio de la consulta y el cluster, y por lo tanto, es posible descartar todos los elementos del cluster.

III. Trabajo relacionado

Los trabajos de paralelismo aplicados a estructuras métricas corresponden a trabajos sobre plataformas de memoria distribuida (utilizando un cluster de procesadores), éstos son [19], [20], [21], [22], [23], [24], [25]. En dichas publicaciones la búsqueda se realiza previa distribución del índice. Estas distribuciones se pueden clasificar como *locales* y *globales*.

Las estrategias locales tienen la característica que cada procesador del cluster puede procesar una consulta de forma independiente e incomunicada del resto, para luego reportar los resultados al procesador *broker* (procesador encargado de presentar los resultados al usuario). Lo anterior implica que la consulta debe hacerse llegar a todos los procesadores, y el resultado final es la unión de los resultados entregados por cada procesador. En cambio, las estrategias globales crean sólo un índice, y éste es distribuido entre los procesadores, y cada consulta se resuelve utilizando un subconjunto de procesadores.

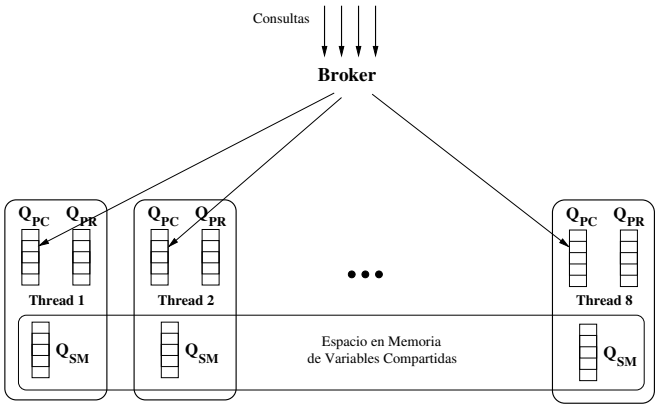


Figura 4: Modelo de Búsqueda usado en los experimentos.

Una consulta se hace llegar a un procesador que actúa como administrador de la consulta. Luego con la ayuda de información previamente almacenada este procesador determina a qué procesadores se debe enviar la consulta. Finalmente, cada procesador involucrado en la solución le envía sus resultados al procesador administrador, y éste es el que presenta el resultado final.

Con respecto al trabajo relacionado en GPU, existen las publicaciones [26], [27], [28]. Todos ellos sólo resuelven consultas de tipo kNN, y las bases de datos empleadas son de alta dimensionalidad. Las soluciones que presentan requieren siempre calcular las distancias entre todos los elementos de la base de datos y la consulta.

En [26] se propone dividir las matrices que almacenan los elementos y las consultas en pequeñas submatrices. Cada submatriz es distribuida entre los CUDA Blocks, y se calcula una tercera matriz de distancias entre elementos y consultas. Finalmente, los elementos de esta tercera matriz son ordenados utilizando *CUDA-based Radix Sort* [29], y los k primeros elementos son seleccionados como resultado final. En [27] se propone que cada hilo calcule la distancia entre un elemento de la base de datos y la consulta, y el arreglo de distancia resultante es ordenado con *insertion sort*, implementado en CUDA en el mismo trabajo. En [28] sólo se considera el caso particular de kNN con $k = 1$, y la solución se basa en el uso eficiente de la memoria de texturas de la GPU.

IV. Estrategias de Distribución y Búsqueda sobre procesadores multi-core

IV-A. Modelo de Búsqueda

El modelo de búsqueda bajo el que está basado este trabajo está representado por la Figura 4, en el cual las consultas entrantes son distribuidas por una máquina *broker*. La distribución de las consultas para todos los experimentos fue de forma circular entre los threads.

Cada thread posee tres colas, la primera es la *Cola Privada de Consultas* (Q_{PC}), la que mantiene las consultas suministradas por el broker. La segunda es la *Cola Privada de Requerimientos* (Q_{PR}), en donde están los *requerimientos* a ser procesados. Un *requerimiento* es una estructura que indica una cierta cantidad de evaluaciones de distancias a ser ejecutadas por un thread durante la solución de una consulta. La tercera es la *Cola Secundaria de Mensajes* (Q_{SM}), en donde se encuentran los requerimientos destinados a otros threads.

Las colas Q_{PC} y Q_{PR} son privadas a cada thread. En cambio, la cola Q_{SM} es global y todos los threads tienen acceso a ella, esto es para permitir el paso de mensajes (como se explica en la Sección IV-D). Debido a la variación que puede sufrir el tamaño de la cola Q_{PR}, sus elementos son creados dinámicamente.

IV-B. Descripción de la Búsqueda

La búsqueda está basada en el procesamiento de *requerimientos*. Un requerimiento es una estructura formada por los siguientes campos:

1. **Región:** Indica la región del índice que se debe acceder para procesar la consulta. En el caso de los índices basados en árboles este campo es un puntero a un nodo.
2. **Indice:** Indica el elemento del nodo donde se debe comenzar a buscar. Esto es usado para retomar una búsqueda interrumpida por haber alcanzado R evaluaciones de distancia.
3. **Consulta:** La consulta misma.
4. **Campo extra:** Datos específicos requeridos por la búsqueda sobre el índice.

De esta forma cada requerimiento implica realizar L evaluaciones de distancia, donde $1 <= L <= N_{region}$ (N_{region} es la cantidad de elementos almacenados en la zona de memoria apuntada por el campo *Región*). En el caso de los índices basados en árboles N_{region} es la cantidad de elementos de un nodo.

Según lo anterior, cuando una consulta se obtiene desde la Q_{PC}, se genera un *requerimiento inicial*, que es agregado a la Q_{PR} para comenzar su búsqueda. Dependiendo del índice, el procesar un requerimiento puede generar más requerimientos, los cuales al ser procesados generarán aún más, y así sucesivamente hasta completar todos los requerimientos que exige la consulta.

A continuación se describen las estrategias multi-core implementadas. En todas ellas no se realiza particionado del índice, es decir, se mantiene sólo un índice global en memoria al que acceden todos los threads.

IV-C. Estrategia Local

En esta estrategia los threads obtienen una consulta desde Q_{PC}, de donde se obtiene el requerimiento inicial, y éste se agrega a Q_{PR}. Todos los requerimientos generados se almacenan en la misma Q_{PR}, que es local y privada a cada thread. La cola Q_{SM} no es usada en esta estrategia.

Lo anterior implica que cada thread resuelve una consulta completamente y de forma aislada. Es decir, cada

Algoritmo 3 Búsqueda para la estrategia Local.

{*executeRequirement*(*req*): procesa el requerimiento *req* y retorna la lista de requerimientos generados (para los índices que corresponda).}

```
ThreadQueryProcessing()
 1: while true do
 2:    if empty(Q_PR) then
 3:       initial_requirement ← nextQuery(Q_PC);
 4:       Q_PR.insert(initial_requirement);
 5:    end if
 6:    requirement ← nextRequirement(Q_PR);
 7:    requirementList ← executeRequirement(requirement);
 8:    for all task ∈ requirementList do
 9:       Q_PR.insert(task);
10:    end for
11: end while
```

thread es capaz de resolver sus consultas de forma incomunicada del resto de los threads, evitando instrucciones de sincronización y de paso de mensajes.

El Algoritmo 3 describe el proceso de búsqueda sobre esta estrategia.

IV-D. Estrategia Bulk-Circular

Esta estrategia procesa consultas utilizando el modelo BSP (Bulk Synchronous Parallel) [30], el que define un comportamiento sincrónico para los threads involucrados. Para implementar el modelo BSP, y poder crear una secuencia de pasos (denominados *supersteps*) se utilizó la directiva de OpenMP *#pragma omp barrier*.

En esta estrategia cada thread usa las tres colas Q_{PC}, Q_{PR} y Q_{SM}. Cada thread almacena en su propia Q_{SM} los *mensajes* que son destinados a los demás threads. Un *mensaje* es un requerimiento más el identificador del thread destino.

Lo anterior implica que los requerimientos generados no se almacenan solamente en Q_{PR} como en la estrategia Local, sino que los requerimientos destinados a otros threads se almacenan en forma de mensaje en Q_{SM}. Cada thread selecciona el destino de un mensaje siguiendo una distribución circular ($destino = i \,\%P$, siendo P el número de threads).

El Algoritmo 4 describe el proceso de búsqueda para esta estrategia, en donde se establecen dos *supersteps*. El primero inicializa la cola Q_{SM} borrando todos sus elementos, luego se obtienen requerimientos de la Q_{PR} y se procesan hasta completar un máximo de R evaluaciones de distancia (si Q_{PR} está vacía entonces se obtiene una consulta nueva desde la Q_{PC} y se inserta el requerimiento inicial de la consulta en Q_{PR}). Cuando se alcanzan R evaluaciones de distancia se continúa con el segundo superstep, donde cada thread lee las Q_{SM} de los demás y rescata los mensajes que están destinados para él. Todos estos mensajes son insertados en forma de requerimiento en la Q_{PR}, y luego se continúa con el primer superstep y así sucesivamente. Cabe destacar que la escritura y lectura a las colas Q_{SM} se realizan en diferentes supersteps, lo que

Algoritmo 4 Búsqueda en la estrategia Bulk-Circular.

{*tid*: ID del Thread.}
{P: Cantidad total de Threads.}
{*executeRequirement*(*req*): procesa el requerimiento *req* y retorna la lista de requerimientos generados (para los índices que corresponda).}

```
ThreadQueryProcessing(tid)
 1: while true do
 2:    while limit < R do
 3:       if Q_PR.empty() == true then
 4:          initial_requirement ← nextQuery(Q_PC);
 5:          Q_PR.insert(initial_requirement);
 6:       end if
 7:       requirement ← nextRequirement(Q_PR);
 8:       requirementList ← executeRequirement(requirement);
 9:       for all task ∈ requirementList do
10:          if task.targetThread == tid then
11:             Q_PR.insert(task);
12:          else
13:             Q_SM[tid].insert(task);
14:          end if
15:       end for
16:    end while
17:    #pragma omp barrier
18:    for i = 0; i < P; i++ do
19:       if i != tid then
20:          for j = 0; j < Q_SM[i].size(); j++ do
21:             if Q_SM[i].getTask(j).targetThread == tid then
22:                Q_PR.insert(Q_SM[i].getTask(j));
23:             end if
24:          end for
25:       end if
26:    end for
27:    #pragma omp barrier
28:    Q_SM[tid].clear();
29: end while
```

implica que no hay problemas de conflictos entre lecturas y escrituras concurrentes.

IV-E. Estrategia Bulk-Critical

Esta estrategia es muy similar a la Bulk-Circular, pero usa *regiones críticas* de OpenMP para implementar el paso de mensajes. Una región crítica corresponde a una porción de código que se ejecutará por sólo un thread a la vez. Es decir, si un thread desea ejecutar instrucciones de código que se encuentran en una región crítica, sólo lo podrá hacer si ningún otro thread se encuentra ejecutando instrucciones de dicha región. En caso contrario, el thread esperará hasta que la región quede disponible.

Esta estrategia, al utilizar regiones críticas, tiene la ventaja de evitar instrucciones de sincronización, también la de utilizar menos memoria al no utilizar colas de mensajes. Pero posee la desventaja que el acceso secuencial a las regiones críticas puede actuar como cuello de botella.

IV-F. Estrategia Bulk-Local

Esta estrategia es similar a la Bulk-Circular pero con la diferencia que el destino es siempre el mismo thread, por lo que no existe Q_{SM}, lo que implica que no hay intercambio de mensajes. Debido a esto último, ésta es una

Tabla I: Características Generales

Processor	2xIntel Quad-Xeon (2.66 GHz)
L1 Cache	8x32KB + 8x32KB (inst.+data) 8-way associative, 64byte per line
L2 Unifed Cache	4x4MB (4MB shared per 2 procs) 16-way associative, 64 byte per line
Memory	16GBytes (4x4GB) 667MHz DIMM memory 1333 MHz system bus
Operating System	GNU Debian System Linux kernel 2.6.22-SMP for 64 bits
Compiler	icc (v.11.0)
Flag Optimizations	-O3 -march=pentium4 -xW -ip -ipo

estrategia local donde cada thread resuelve completamente una consulta, pero realiza procesamiento por lotes de forma síncrona.

IV-G. Resultados experimentales utilizando procesadores multi-core

Todos los experimentos fueron realizados en una máquina con 2 CPU's Intel Quad-Xeon, cada una con 4 núcleos. Las características generales se muestran en la Tabla I.

Los experimentos se realizaron con 2 bases de datos:

1. ***Spanish*** **:** Diccionario español con 51589 palabras. Se usó la *distancia de edición* (o *distancia de Levenshtein*) [31] con radio 1, 2 y 3, pues éstos son radios usados en trabajos previos [15], [23], [32]. Esta distancia entrega la cantidad mínima de inserciones, eliminaciones o reemplazos para que una palabra sea igual a otra. Las consultas para esta base de datos fue un archivo de 40000 consultas, obtenidas desde la Web Chilena en el dominio todo.cl.

2. ***Images*** **:** Esta base de datos fue creada a partir de una colección de 40701 vectores que representan imágenes de la NASA, y éstas se usaron como una distribución de probabilidades para generar vectores aleatorios hasta completar 120000 imágenes de dimensión 20. Se usó la *distancia euclidiana* para medir la similitud entre los objetos. Los radios utilizados fueron los que permiten recuperar el 0.01 %, 0.1 % y 1 % de la base de datos. Estos son valores usados en trabajo previos [15], [17], [32]. El 80 % de la base de datos se usó para la construcción del índice y el 20 % restante como archivo de consultas.

Los experimentos fueron normalizados al mayor valor observado (esto para apreciar mejor las diferencias reportadas por las diferentes estrategias). Se utilizaron escenarios de alto y bajo tráfico de consultas entrantes al sistema.

La Figura 5 muestra el tiempo de ejecución para las estrategias *Bulk-Circular* y *Bulk-Critical*. El experimento se realizó bajo un alto tráfico de consultas y usando el *EGNAT*. *Bulk-Critical* presenta tiempos de ejecución muy altos, debido a que las regiones críticas son muy accedidas, y como es sabido ([6]), esto degrada mucho el rendimiento del programa. Por este motivo esta estrategia fue descartada para los experimentos siguientes. Un comportamiento similar se observó con los demás índices.

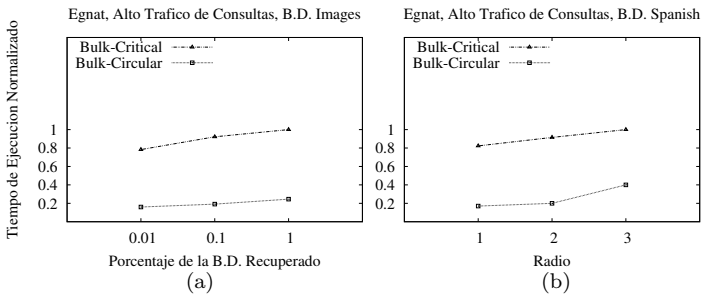


Figura 5: Tiempo de Ejecución para la estrategias *Bulk-Circular* y *Bulk-Critical*.

La Figura 6 muestra los experimentos realizados con todas las estrategias bajo una situación de alto tráfico de consultas, para las bases de datos *Spanish* e *Images*. En cambio la Figura 7, muestra los mismos experimentos pero con un bajo tráfico de consultas. Los resultados indican que con un alto tráfico la estrategia *Local* obtiene un mejor rendimiento en todos los índices. Esto es debido al costo que implica la sincronización y el paso de mensajes para el caso de la *Bulk-Circular*, pero esta última toma ventaja con un bajo tráfico de consultas. Esta ventaja se debe principalmente a que la estrategia *Bulk-Circular* reduce los tiempos de ociosidad de los threads que se encuentran en espera de una consulta. Para tráfico alto la estrategia *Bulk-Local* alcanza un rendimiento similar al de la estrategia *Local* para la mayoría de los índices estudiados.

Para ilustrar de mejor manera lo que sucede en un escenario de baja frecuencia de consultas, la Figura 8 muestra un ejemplo de dos distribuciones de consultas con baja frecuencia, procesadas usando 4 threads. La primera, es cuando el arribo de consultas corresponde a la mejor distribución posible, es decir, las consultas se distribuyen de tal forma que todos los threads harán (aproximadamente) la misma cantidad de evaluaciones de distancia, y la segunda, es una distribución desfavorable, en donde el primer thread procesa las consultas ($q1$, $q5$, $q9$) que implican más evaluaciones de distancia. En ambos casos la estrategia *Bulk-Circular* obtiene el mayor throughput (consultas resueltas por unidad de tiempo). La frecuencia de arribo de consultas para este ejemplo (Figura 8) fue: en la primera unidad de tiempo arriban 2 consultas, en la segunda 1 consulta y en la tercera 1 consulta, y luego se repite este ciclo.

La Figura 9 muestra la comparación entre los índices usando la estrategia *Local* para alta frecuencia de consultas y *Bulk-Circular* para baja frecuencia. Esto con la finalidad de observar cuál índice se adapta mejor a la implementación de las estrategias. El índice con un

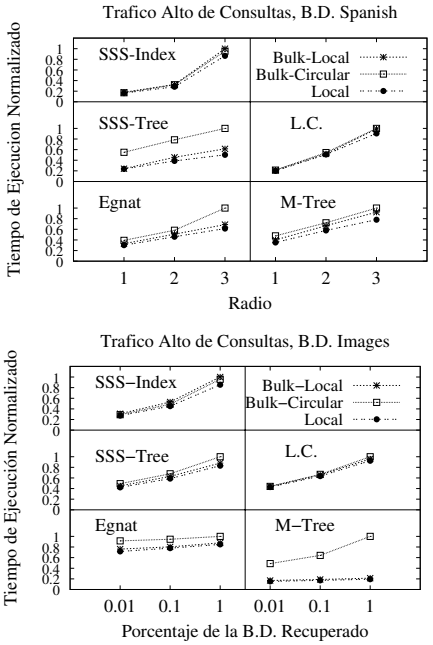


Figura 6: Tiempo de Ejecución para un alto tráfico de consultas.

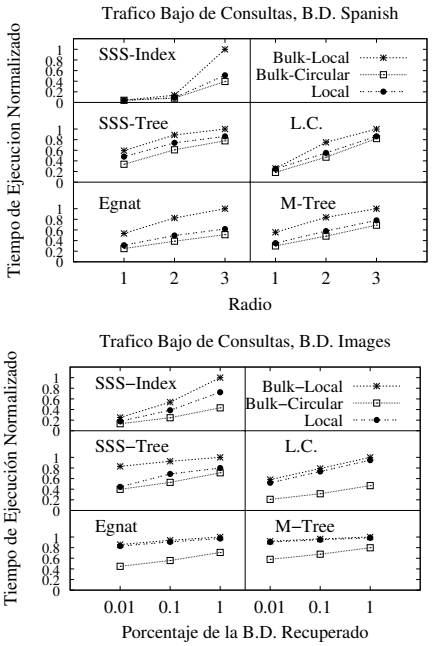


Figura 7: Tiempo de Ejecución para un bajo tráfico de consultas.

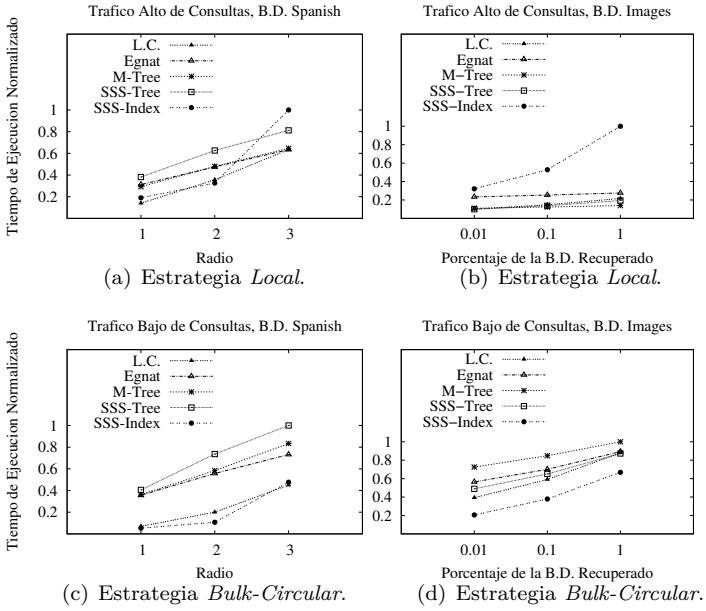
Distribucion Favorable

Tiempo	Local	Bulk–Circular
	T1 T2 T3 T4	**T1 T2 T3 T4**
1	q1 q2	q1 q1 q1 q1
2	q1 q2 q3	q2 q2 q2
3	q1 q2 q3 q4	q3 q3 q4
4	q1 q5 q6	**q5 q5 q5 q5**
5	q5 q6 q7	q6 q6 q6 q7
6	q8 q5 q6 q7	q7
7	q9 q5q10	q8 q10 q10 q10
8	q9 q10 q11	q9 q9 q9 q9
9	q9q12 q10 q11	q11 q12 q11
10	q9	
11		
12		

Distribucion Desfavorable

Tiempo	Local	Bulk–Circular
	T1 T2 T3 T4	**T1 T2 T3 T4**
1	q1 q2	q1 q1 q1 q1
2	q1 q2 q3	q2 q2 q2
3	q1 q2 q3 q4	q3 q3 q4
4	q1 q6	q5 q5 q5 q5
5	q5 q6 q7	q6 q6 q6
6	q5 q6 q7 q8	q7 q7 q8
7	q5 q10	q9 q9 q9 q9
8	q5 q10 q11	q10 q10 q10
9	**q9** q10 q11 q12	q11 q11q12
10	**q9**	
11	**q9**	
12	**q9**	

Figura 8: Distribuciones de consultas en un escenario de baja frecuencia.



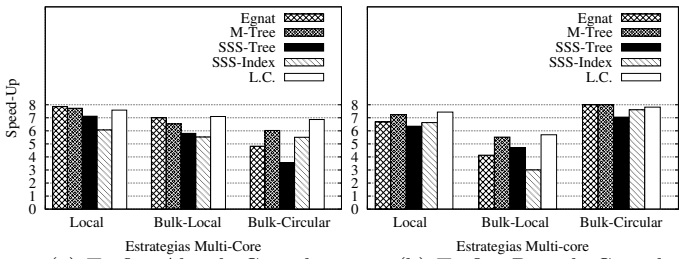
(a) Estrategia *Local*. (b) Estrategia *Local*.

(c) Estrategia *Bulk-Circular*. (d) Estrategia *Bulk-Circular*.

Figura 9: Comparación de los índices con la estrategia *Local* para alto tráfico de consultas y la *Bulk-Circular* para bajo tráfico.

buen desempeño tomando en cuenta ambos escenarios de frecuencia fue la *LC*. Una de las razones de esto es que el requerimiento de este índice implica una cantidad fija de evaluaciones de distancia, ya que éste encierra a un cluster completo, y esto favorece el balance de carga.

La Figura 10 muestra el speed-up de los índices sobre la base de datos *Spanish* con radio 3. El speed-up S está dado por $S = T_s/T_m$, donde T_s es el tiempo de ejecución de la aplicación secuencial y T_m el tiempo de la estrategia multi-core. El mayor speed-up, usando alta frecuencia de consultas, lo obtiene la estrategia Local, mientras que con baja frecuencia de consultas la estrategia *Bulk-Circular* obtiene la ventaja.



(a) Trafico Alto de Consultas (b) Trafico Bajo de Consultas

Figura 10: Speed-up de los índices sobre la base de datos *Spanish* con radio 3.

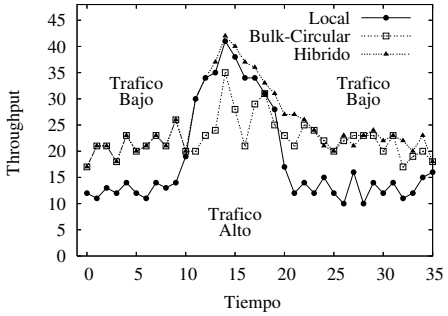


Figura 11: Consultas completamente resueltas por unidad de tiempo.

IV-H. Estrategia híbrida

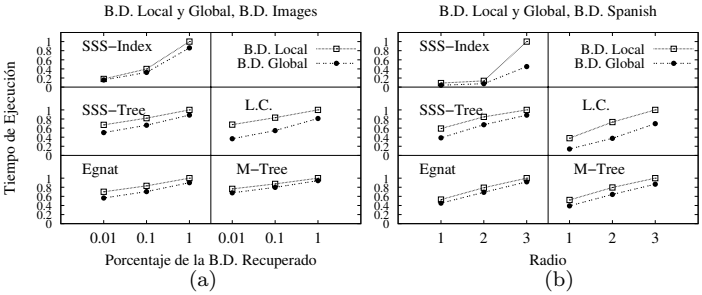
Esta estrategia es capaz de aplicar un intercambio entre la estrategia *Local* y *Bulk-Circular* dependiendo del tráfico de consultas entrantes. Cuando el número de consultas en espera C_p satisface $C_p > P * C_{max}$ (P: cantidad de threads, C_{max}: Número de consultas que indica el límite entre una situación de tráfico bajo y una de tráfico alto) se comienza a procesar los requerimientos según la estrategia *Local* y cuando el número de consultas en espera es suficientemente bajo se cambia a la estrategia *Bulk-Circular*

La Figura 11 muestra el throughput de las consultas en el sistema, es decir, la cantidad de consultas resueltas por unidad de tiempo. En esta figura se muestran las estrategias *Local*, *Bulk-Circular* e *Híbrida* utilizando la *LC*, variando la frecuencia de consultas entrantes. La estrategia *Híbrida* es la que muestra el mayor throughput, pues es la que explota las ventajas de ambas estrategias.

IV-I. Distribución Local de la base de datos

Hasta ahora todos los experimentos han sido realizados con una base de datos global y compartida, es decir, se tiene un único índice en memoria principal al cual acceden todos los threads para resolver sus consultas.

En esta sección se propone distribuir el índice en P partes (P = número de threads), de esta forma cada thread



(a) (b)

Figura 12: Tiempos de Ejecución normalizados para las bases de datos **a)***Images* y **b)***Spanish*.

puede resolver sus consultas accediendo a su propio índice. Para esto se distribuyó cada elemento de la base de datos de forma circular entre los threads, y luego cada thread con sus elementos crea su propio índice. Esta distribución tiene la desventaja que cada thread tiene un índice que representa una porción de la base de datos completa, y por lo tanto, cada consulta debe ser procesada por todos los threads. Pero posee la ventaja que cada thread realiza el proceso de búsqueda sobre un índice reducido en elementos. Para evitar operaciones de sincronización, cada thread reporta sus resultados escribiéndolos en una posición distinta de memoria.

Las Figuras 12 y 13 muestran el tiempo de ejecución y la cantidad de evaluaciones de distancia respectivamente. Los valores están normalizados al mayor valor observado en el experimento, sobre las bases de datos *Spanish* e *Images* usando un alto tráfico de consultas con la estrategia Local. Estos experimentos muestran que al distribuir los elementos de la base de datos para generar índices locales (estrategia denominada *B.D. Local*), aumentan las evaluaciones de distancia implicando un aumento en el tiempo de ejecución. Esto se debe principalmente a que los centros (o pivotes) seleccionados en el método *B.D. Global* son de mejor calidad [33] y más representativos de la base de datos. Esto permite una mejor discriminación y poda de elementos, debido a que los centros (o pivotes) globales son seleccionados teniendo en cuenta todos los elementos de la base de datos.

IV-J. Conclusiones acerca de la distribución y búsqueda sobre procesadores multi-core

En esta sección se han propuestos algoritmos sobre una plataforma multi-core para resolver consultas en espacios métricos. Se utilizaron varios índices representativos y muy usados en la literatura con el propósito de mostrar lo genérico de los algoritmos propuestos. Estos algoritmos implementan procesamiento multi-thread asíncrono (estrategia *Local*) y bulk-sincrónico (estrategias *Bulk-Critical*, *Bulk-Local* y *Bulk-Circular*), siendo este último

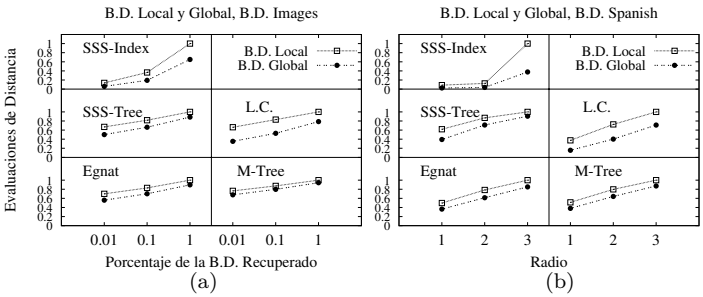


Figura 13: Promedio normalizado de evaluaciones de distancias por consulta para las bases de datos **a)***Images* y **b)***Spanish*.

una implementación del modelo BSP.

La estrategia *Bulk-Critical* es la que muestra el peor rendimiento, debido principalmente al costo de acceso secuencial a las *regiones críticas*.

La estrategia *Local* es la de mejor rendimiento en un escenario de alta frecuencia de consultas. Esta estrategia distribuye todos los *requerimientos* de una consulta al mismo thread. Con esto se consigue que cada thread resuelva una consulta completamente y de forma independiente del resto. Es decir, cada thread puede resolver sus consultas de forma aislada e incomunicada con el resto de los threads.

La estrategia *Bulk-Circular* es la de mejor rendimiento en un escenario de bajo tráfico de consultas. En esta estrategia cada thread distribuye los requerimientos de sus consultas entre todos los demás threads, lo que implica que todos los threads participen en la solución de todas las consultas. Esta estrategia obtiene la ventaja en un tráfico bajo de consultas debido principalmente a que disminuye el tiempo de ociosidad de los threads.

También se propone una estrategia *híbrida*, la que es capaz de cambiar su modo de procesamiento entre las estrategias *Local* y *Bulk-Circular* dependiendo del tráfico de consultas observado.

En la Subsección IV-I se comparan dos distribuciones de la base de datos. La primera distribuye los elementos de la base de datos entre los threads, y cada uno de éstos construye su propio índice. La segunda mantiene sólo un índice global en memoria principal. Esta última es la que muestra el mejor rendimiento en evaluaciones de distancia y tiempo de ejecución debido a la calidad que presentan los centros (o pivotes) globales, mejorando la discriminación y poda de elementos.

V. Estrategias de Distribución y Búsqueda sobre GPU

En esta sección se proponen y comparan algoritmos de búsqueda de índices métricos sobre GPU (Graphic Processor Unit) basados en CUDA [7]. Los índices seleccionados para ser implementados en GPU fueron la *LC* y *SSS-Index*, debido a que ambos almacenan su índice en matrices, y por lo mencionado en la Subsección II-B, esto favorece al rendimiento en GPU, pues la localidad espacial de los datos es un factor de gran importancia en este tipo de plataforma.

Debido a la complejidad y restricciones de la GPU, se analizan los dos tipos de consultas (consultas por rango y consultas kNN) de forma separada. Para ambos tipos de consultas, las estrategias empleadas y dificultades encontradas fueron diferentes.

En todas las implementaciones siguientes cada bloque de threads se encarga de resolver una consulta completamente, pues de esta forma se pueden resolver varias consultas en sólo un lanzamiento de *kernel*, dado que el lanzamiento sucesivo de éstos degrada el rendimiento. Además, de esta forma los threads encargados de resolver una misma consulta pueden usar sincronización, la que está habilitada sólo para los threads pertenecientes al mismo bloque, evitando de esta forma conflictos de concurrencia. Es decir, se explota paralelismo a dos niveles: (i) *Paralelismo de grano grueso* al resolver un conjunto de Q consultas en paralelo en sólo un lanzamiento de *kernel*, y (ii) *Paralelismo de grano fino* al resolver cada consulta con un conjunto de threads.

Por lo anterior cada *kernel* es lanzado con Q bloques (Q=número de consultas a resolver) maximizando el número de threads por bloque. Si Q sobrepasa el máximo permitido de bloques, entonces se deben hacer sucesivos lanzamientos de *kernels* hasta resolver todas las consultas. En los experimentos del presente trabajo, es necesario sólo un lanzamiento de *kernel*. La consulta a resolver por el bloque, siempre se almacena en *shared memory* (Subsección II-B), debido al frecuente acceso a ésta por parte de los threads.

Resulta complejo definir el costo de un algoritmo en GPU, debido principalmente a que no es sabido la cantidad de núcleos usados por un multiprocesador al ejecutar una secuencia de instrucciones. Esto último depende de cuán paralelo es la ejecución de instrucciones por parte de los threads de un *warp*. Son varios los factores que agregan divergencia a la secuencia de instrucciones de los threads del mismo *warp*, tales como: (i) toda sentencia de código que implique una condición de salto, (ii) accesos no contiguos a memoria, (iii) accesos al mismo banco de *shared memory*.

En esta sección se aplica el método de balance de carga Round-Robin. Este método es típicamente usado en sistemas operativos y básicamente consiste en atender todos los procesos pendientes sin ningún tipo de discriminación. La forma de implementar este método en el presente trabajo es siguiendo una distribución circular.

La organización de esta sección es como sigue. En la Subsección V-A se describen los algoritmos sobre GPU para resolver consultas por rango. En la Subsección V-B

se muestran los resultados experimentales para resolver consultas por rango. En la Subsección V-C se describen los algoritmos para resolver consultas kNN. En la Subsección V-D se muestran los resultados experimentales para resolver consultas kNN. En la Subsección V-E, se muestran los experimentos al resolver consultas en-línea sobre GPU. Finalmente el la Subsección V-F se dan a conocer las conclusiones acerca de la distribución y búsqueda sobre GPU.

V-A. Consultas por Rango

V-A1. Fuerza Bruta: El propósito de esta implementación es que cada thread de un bloque resuelva la evaluación de distancia entre un elemento de la base de datos y la consulta, evitando acceder a una estructura de datos intermedia o índice.

Previamente se almacenó la base de datos en una matriz de $D \times SIZE_{BD}$ (D=dimensión de los elementos[1], y $SIZE_{BD}$= número de elementos de la base de datos), en donde cada columna representa un elemento. El hecho de almacenar los datos por columna es con la finalidad que threads consecutivos accedan a posiciones contiguas de memoria al leer datos desde *device memory*.

Este algoritmo está dividido en dos etapas delimitadas por la invocación de una función de sincronización (la que es aplicable sólo a los threads del mismo bloque). En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory*. En la segunda etapa, los elementos de la base de datos se asignan a los threads siguiendo una distribución Round-Robin, y cada thread realiza las evaluaciones de distancia entre sus elementos y la consulta. En caso que la distancia entre el elemento y la consulta sea menor que el radio de búsqueda, el elemento es agregado al conjunto resultado.

El reportar un elemento como resultado puede ser implementado de varias maneras dependiendo de las necesidades de la aplicación. Pero, para fines del presente trabajo, esta función sólo aumenta un contador, llevando la cuenta de los elementos encontrados por cada thread.

*V-A2. Lista de Clusters (*LC*):* La estructura de datos usada para implementar la *LC* consistió en 3 matrices, denotadas como *CENTROS*, *RC* y *CLUSTERS* en el Algoritmo 5. *CENTROS* es una matriz de $D \times SIZE_{Centros}$ ($SIZE_{Centros}$=cantidad de centros de clusters), donde cada columna representa un centro de cluster. *RC* es un arreglo de largo $SIZE_{Centros}$ con los radios cobertores de cada cluster. *CLUSTERS* es una matriz de $D \times SIZE_{Clusters}$ ($SIZE_{Clusters}$=cantidad de elementos en todos los clusters), donde cada columna representa un elemento de cluster, con la característica que los elementos de un mismo cluster se encuentran en columnas contiguas.

[1]En el caso de la base de datos *Spanish D* es el tamaño máximo de una palabra del diccionario

Al igual que el algoritmo de fuerza bruta, el hecho de almacenar los datos por columnas es para favorecer la fusión de instrucciones de lecturas (Subsección II-B).

El Algoritmo 5 muestra la búsqueda realizada por el *kernel* (sobre la base de datos de vectores) usando la *LC*. Este algoritmo está divido en etapas delimitadas por la función de sincronización `__syncthreads()`.

En la primera etapa los threads de cada bloque colaboran para copiar en *shared memory* la consulta que le corresponde resolver (línea 6). En la segunda etapa, los centros de clusters se asignan a los threads siguiendo una distribución Round-Robin, y cada thread realiza la evaluación de distancia entre la consulta y un subconjunto de centros (línea 12). Si el centro está dentro del radio de búsqueda (línea 13), éste se agrega al conjunto respuesta. En *shared memory* (en la variable *buscar*) se almacenan los clusters sobre los que se debe realizar una búsqueda exhaustiva (línea 16). Y si la consulta está completamente contenida en un cluster, entonces se actualiza la variable *minC* para acotar la búsqueda (línea 17). Finalmente, en la tercera etapa, los elementos de los clusters son asignados a los threads siguiendo una distribución Round-Robin (línea 24). Los elementos de clusters no descartados son comparados contra la consulta (línea 26).

V-A3. SSS-Index*:* Este índice se representó por 3 matrices denominadas *PIVOTES*, *DISTANCIAS* y *BD*. *PIVOTES* es una matriz de $D \times SIZE_{piv}$ ($SIZE_{piv}$=cantidad de pivotes), que almacena en cada columna un pivote. *DISTANCIAS* es una matriz de $SIZE_{piv} \times SIZE_{BD}$ ($SIZE_{BD}$=cantidad de elementos de la BD), que almacena las distancias entre los pivotes y los elementos de la base de datos. *BD* es una matriz de $D \times SIZE_{BD}$ que almacena en cada columna un elemento de la base de datos.

Este algortimo está dividido en etapas delimitadas por una función de sincronización. En la primera etapa los threads del mismo bloque colaboran para copiar la consulta que le corresponde resolver a *shared memory*. En la segunda etapa, los threads del mismo bloque (siguiendo una distribución Round-Robin) obtienen la distancia entre todos los pivotes y la consulta. Esta distancia es almacenada en *shared memory*. Finalmente, en la tercera etapa, cada elemento es asignado a un thread siguiendo una distribución Round-Robin. Cada thread intenta descartar su elemento mediante desigualdad triangular, y en caso de no ser posible, el mismo thread realiza la evaluación de distancia entre el elemento y la consulta.

En el artículo [10], los autores encontraron empíricamente que la cantidad de pivotes creadas con valores alrededor de $\alpha = 0{,}4$ produce el óptimo para este índice. Sin embargo, si observamos la Figura 14, el rendimiento óptimo para el *SSS-Index* sobre GPU se consigue con $\alpha = 0{,}6$ (1 pivote) para la base de datos de vectores. Esta figura muestra tres gráficos correspondientes al tiempo de

Algoritmo 5 *Kernel* de búsqueda de la LC para resolver consultas por rango sobre GPU.

{Cada fila de *Queries* representa una consulta.}
{Sea D la dimensión de cada elemento de la BD}
{Sea B_{Size} la cantidad de elementos de cada cluster.}
{Sea $SIZE_{Clusters}$ la cantidad de elementos en los clusters}
{Sea $SIZE_{Centros}$ la cantidad de centros de clusters}

___global___ busquedarango(float ***CENTROS*, float ***RC*, float ***CLUSTERS*, float ***Queries*, float *radio*)

```
1: ___shared___ float query[D]
2: ___shared___ int buscar[SIZE_Centros]
3: ___shared___ int minC=SIZE_Centros
4:
5: for (i = ID_Thread; i <D; i+=T_Block) do
6:     query[i] = Queries[ID_Block][i]
7: end for
8:
9: ___syncthreads()
10:
11: for (j = ID_Thread; j <minC; j+=T_Block) do
12:     dist = distancia(CENTROS, j, query)
13:     if dist <= radio then
14:         encontrado()
15:     end if
16:     buscar[j] = dist <= RC[j] + radio
17:     if dist < RC[j] - radio then
18:         atomicMin(&minC, j)
19:     end if
20: end for
21:
22: ___syncthreads()
23:
24: for (j = ID_Thread; j < SIZE_Clusters && j/B_Size <= minC;
    j+=T_Block) do
25:     if buscar[j/B_Size] == 1 then
26:         if distancia(CLUSTERS, j, query) <= radio then
27:             encontrado()
28:         end if
29:     end if
30: end for
```

ejecución, la cantidad de lecturas de 32, 64 y 128 bytes en *device memory* y el promedio de evaluaciones de distancia por consulta, para el *SSS-Index* sobre la base de datos *Images* utilizando distintos valores de α. Los valores fueron normalizados al mayor valor observado en el experimento.

Como se esperaba, mientras mayor es el α, mayor es la cantidad de evaluaciones de distancia realizadas. Pero el mejor rendimiento en cuanto a tiempo de ejecución se consigue con $\alpha = 0{,}66$ (1 pivote), a pesar de realizar 17.7 veces más evaluaciones de distancia que usando $\alpha = 0{,}5$ (73 pivotes). La respuesta a este comportamiento lo tiene el gráfico de operaciones de lecturas/escrituras. Cuando se utilizan más pivotes, los threads de un *warp* son más propensos a divergir. Por lo mismo, el patrón de acceso a memoria es más irregular impidiendo la fusión de operaciones de lectura/escritura.

Esto último significa que el realizar menos evaluaciones de distancia no compensa el costo causado por la divergencia en la secuencia de instrucciones de threads del mismo

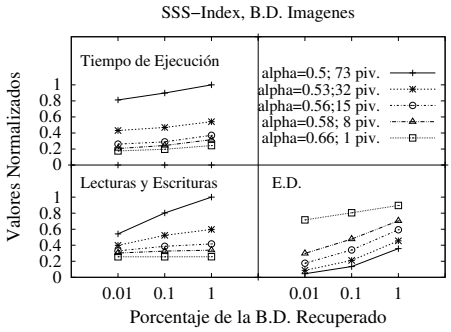


Figura 14: Valores normalizados del tiempos de ejecución, cantidad de lecturas/escrituras (de 32, 64 o 128 bytes) a *device memory* y del promedio de evaluaciones de distancia por consulta del *SSS-Index* sobre GPU para la base de datos *Images*.

warp y la irregularidad en los accesos a memoria.

La estrategia de distribución que se ha descrito fue llamada *Distribución-Elemento*, pero también se implementó una segunda estrategia denominada *Distribución-Pivote*. Esta última consistió en distribuir (Round-Robin) los pivotes entre los threads utilizando una matriz de distancias de dimensión $SIZE_{BD} \times SIZE_{piv}$. De esta manera, cada thread intenta descartar (usando desigualdad triangular) cada elemento de la base de datos, utilizando el pivote que se le ha sido asignado. Para esto fue necesario almacenar en *shared memory* una variable por cada elemento de la base de datos, indicando si el elemento ha sido descartado por algún thread. Luego, los elementos que no han sido descartados se distribuyen (Round-Robin) entre los threads, y cada thread realiza la evaluación de distancia entre sus elementos y la consulta. Por las limitaciones de espacio de *shared memory*, este proceso se debió repetir sucesivas veces con un subconjunto de elementos por vez.

La Tabla II muestra el tiempo de ejecución real y la cantidad de lecturas/escrituras entre las dos estrategias (*Distribución-Pivote* y *Distribución-Elemento*) sobre la base de datos *Images*.

Los resultados muestran un mal rendimiento para la estrategia *Distribución-Pivote*, alcanzando un tiempo de ejecución de hasta 40 veces más que *Distribución-Elemento* al recuperar el 0,01 % de los elementos de la base de datos. Esto se explica debido a la gran cantidad de lecturas/escrituras realizadas. En este caso particular puede resultar un comportamiento anómalo la disminución del tiempo de ejecución al aumentar el porcentaje de elementos recuperados. Esto es debido a la divergencia en la secuencia de instrucciones de los threads de un *warp*, a medida que crece el radio de búsqueda, también crece la regularidad en la ejecución de instrucciones. Para la base de datos *Spanish* los resultados fueron similares. Debido a estos resultados, en todos los experimentos posteriores, sólo se utiliza la estrategia *Distribución-Elemento* sobre el

Tabla II: Tiempos de ejecución (en segundos) y cantidad de lecturas/escrituras a *device memory*.

Tiempo de Ejecución (segs.)		
% Recuperado	*Distribución-Pivote*	*Distribución-Elemento*
0,01	88,8	2,2
0,1	82,8	2,4
1	76,4	3,0
Cantidad de Lecturas/Escrituras (x10000)		
% Recuperado	*Distribución-Pivote*	*Distribución-Elemento*
0,01	455551	29699
0,1	461104	29740
1	468319	29806

SSS-Index.

V-B. Resultados experimentales sobre GPU para resolver consultas por rango

La GPU utilizada fue una NVIDIA Tesla C1060, con 30 multiprocesadores, 8 núcleos por multiprocesador y 16K de *shared memory*. El tamaño de *device memory* es de 4GB. Los experimentos sobre OpenMP y versiones secuenciales fueron ejecutados sobre la plataforma descrita en la Tabla I.

Los gráficos de lectura/escritura mostrados en esta sección representan la suma total de estas operaciones. Recordemos que la lectura (o escritura) mínima permitida es de 32 bytes (Subsección II-B), y las de 64 o 128 bytes son instrucciones de lecturas (o escritura) de 32 bytes fusionadas. Estos gráficos fueron agregados debido a que las operaciones de lecturas/escrituras en GPU poseen una gran latencia.

A continuación se presentan los experimentos comparativos entre los índices *LC*, *SSS-Index* y *Fuerza Bruta* para resolver consultas por rango. En los experimentos se ajustaron los parámetros de la *LC* y *SSS-Index* seleccionando las versiones de mejor rendimiento en cuanto a tiempo de ejecución para cada base de datos. En el caso de la *LC* se compararon diferentes valores de B_{Size} (B_{Size}=cantidad de elementos en un cluster). Para la base de datos *Images* se seleccionó $B_{Size} = 64$, y para *Spanish* $B_{Size} = 32$. En el caso del *SSS-Index*, para la base de datos *Images* se seleccionó la versión mostrada en la Subsección V-A3, la que utiliza $\alpha = 0{,}66$ (versión que genera 1 pivote) y para la base de datos *Spanish* $\alpha = 0{,}5$ (64 pivotes).

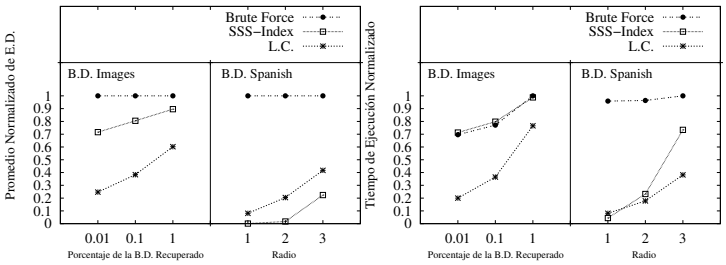
La Figura 15 muestra tres gráficos correspondientes al (i) tiempo de ejecución, (ii) cantidad de lecturas/escrituras a *device memory* y (iii) el promedio de evaluaciones de distancia por consulta de los algoritmos *Fuerza Bruta*, *LC* y *SSS-Index*. Todos los valores fueron normalizados al mayor valor observado en el experimento.

Con respecto al número de evaluaciones de distancia (Figura 15(a)), en ambas bases de datos se presenta un comportamiento esperado. Ambos índices decrementan significativamente la cantidad de evaluaciones de distancia al compararse con el algoritmo *Fuerza Bruta*. El *SSS-Index* tiende a acercarse al algoritmo de *Fuerza Bruta*, debido a que la implementación de este índice corresponde a la versión utilizando 1 pivote. Los gráficos de tiempo de ejecución (Figura 15(b)) no se coinciden con el gráfico de evaluaciones de distancia, pues por ejemplo el algoritmo de *Fuerza Bruta* supera al *SSS-Index* en la base de datos *Images* y la *LC* supera al *SSS-Index* en la base de datos *Spanish* con radios 2 y 3. Esto se explica con el gráfico de lecturas/escrituras (Figura 15(c)). *Fuerza Bruta* presenta un mejor patrón de acceso a memoria que el *SSS-Index* en la base de datos *Images*. El alineamiento en el acceso a memoria por threads consecutivos influye en gran medida en el rendimiento de las actuales GPUs. Como se mencionó en la Subsección II-B, cuando un *warp* realiza accesos no consecutivos en memoria, el hardware no es capaz de fusionar dichos accesos y un simple acceso a memoria se puede convertir en 32 accesos separados. La *LC* muestra el mejor resultado en este aspecto en ambas bases de datos, lo que explica su superioridad en cuanto a tiempo de ejecución.

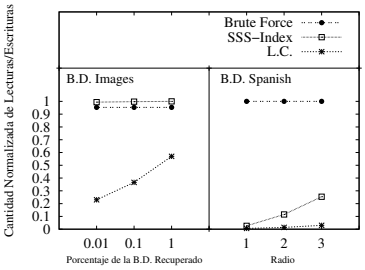
La única excepción a lo mencionado anteriormente se presenta con la base de datos *Spanish* con $radio = 1$, en donde a pesar de hacer un número mayor de lecturas/escrituras, el *SSS-Index* aventaja en tiempo a la *LC*. Esto es debido a que la cantidad de lecturas/escrituras para esta base de datos con $radio = 1$ por parte de la *LC* y *SSS-Index* fue suficientemente pequeño como para que otros factores influyan en el tiempo de ejecución, factores como que la cantidad de registros usados por threads en el *SSS-Index* fue un 12 % menor, y debido a esto la cantidad de *warps* que se puede mantener activos por multiprocesador es mayor. También el número total de instrucciones ejecutadas por el *SSS-Index* fue un 2 % menor y el número de *warps* que tuvieron que serializar su acceso a *shared memory* (por conflictos de acceso al mismo banco) fue 34 % menor en el *SSS-Index*. Pero estos factores se vuelven irrelevantes al momento de aumentar la cantidad de lecturas/escrituras, debido al costo (en ciclos de reloj) asociado a esta instrucción.

La Figura 16 muestra el speed-up de la *LC* y *SSS-Index* para resolver consultas por rango, sobre la base de datos *Images*. Se comparó el speed-up de las versiones en GPU, con las versiones sobre una plataforma multi-core utilizando la estrategia *Local* (Subsección IV-C). Todos los speed-ups fueron calculados sobre el algoritmo de fuerza bruta secuencial.

Las versiones en GPU superan ampliamente las versiones sobre una plataforma multi-core, lo que era esperado dada la capacidad superior de la GPU. Se puede observar que mientras más pequeño es el radio, mayor es el speed-up para todas las versiones, llegando a alcanzar 100.6x el índice *LC*, y 38.5x el *SSS-Index*. Esto es debido a que mientras mayor sea el radio, mayor será la cantidad de evaluaciones de distancia, y mayor la cantidad de operaciones de lectura/escritura sobre *device memory*. Las versiones OpenMP escalan peor que las versiones en GPU cuando se comparan con sus respectivas versiones



(a) Evaluaciones de Distancia (b) Tiempos de Ejecución



(c) Lectura/Escrituras a Memoria

Figura 15: Valores normalizados del **a)** promedio de evaluaciones de distancia por consulta, **b)** tiempo de ejecución, y **c)** cantidad de lecturas/escrituras a *device memory* de los algoritmos de Fuerza Bruta, *SSS-Index* y *LC* sobre GPU.

secuenciales, obteniendo speed-ups de 4.4x, 4.0x y 3.8x para la *LC* y 3.4x, 3.6x y 3.9x para el *SSS-Index*. Para el servidor multi-core, el controlador de memoria común es un cuello de botella, pues los 8 cores son atendidos de forma concurrente. Luego, los conflictos de acceso a memoria son serializados, decrementando el rendimiento. La GPU intenta superar esta limitación ofreciendo un enorme ancho de banda entre elementos procesados y memoria DDR. Para explotar esto último, es crucial la fusión de instrucciones en los accesos a memoria. Además, el multi-threading de grano fino que presenta la GPU ayuda a ocultar parcialmente las latencias durante los accesos a memoria.

También se observa en la misma figura que el speed-up de las versiones multi-core no alcanzan 8x sobre su correspondiente version secuencial. Esto ocurre al aplicar optimizaciones agresivas en tiempo de compilación, las que favorecen a la versiones en 1 CPU. La razón es que luego de dichas optimizaciones el sistema de memoria se vuelve aún más crítico dado que la densidad de accesos se incrementa. Como se dijo anteriormente, el controlador de memoria común en la plataforma multi-core es un cuello de botella, y además se incrementa el número de fallos de la caché L2, que es compartida por cada dos núcleos.

Para algunos lectores, el speed-up mostrado por las versiones en GPU pueden resultar no muy grande, dado que la GPU utilizada posee 30 multiprocesadores,

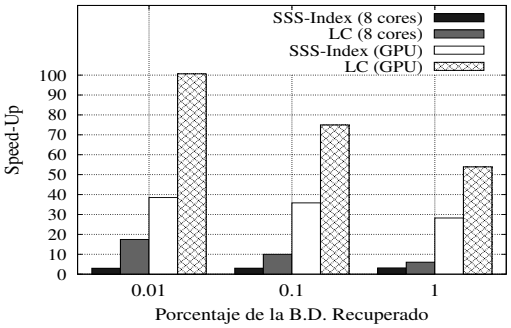


Figura 16: Speed-Up de las versiones sobre una plataforma multi-core y GPU de la *LC* y *SSS-Index* sobre el algoritmo secuencial de fuerza bruta, utilizando la base de datos *Images*, resolviendo consultas por rango.

comparados con la plataforma multi-core de 8 núcleos Xeon. Sin embargo, es importante destacar que cada multiprocesador NVIDIA es muchísimo más simple que las CPUs Intel basadas en micro-arquitectura Core/Nehalem.

V-C. Consultas de los k vecinos más cercanos (kNN)

Los algoritmos de consultas por rango se utilizaron como base para resolver el tipo de consultas kNN. A pesar de aquello, se encontraron varias dificultades distintas a la anteriormente vistas, debido a las particularidades del hardware de la GPU. Todo lo anterior dió pie a una interesante exploración acerca del comportamiento de la GPU frente a este tipo de consultas. A continuación se describen los algoritmos de búsqueda exhaustiva, *LC* y *SSS-Index*, implementados para resolver consultas kNN.

V-C1. Búsqueda exhaustiva: Este método se basa en ordenar las distancias de la consulta a todos los elementos de la base de datos. Para esto, es necesario el parámetro de entrada δ, donde $\delta[i]$ es la distancia entre el elemento i-ésimo de la base de datos y la consulta. Para obtener este arreglo, se ejecuta un *kernel* previamente con todos los bloques necesarios para que cada thread realice la evaluación de distancia entre un elemento de la base de datos y la consulta.

Este método está basado en trabajos del estado del arte (Sección III). Es decir, este algoritmo de búsqueda exhaustiva ordena el arreglo de distancias δ. Los trabajos previos [26] y [27] utilizan como método de ordenamiento el *radix sort* [34] e *insertion sort*, pero el método utilizado en este algoritmo usa el *GPU-Quicksort* [35], el cual tiene mejor rendimiento que los algoritmos anteriores.

*V-C2. Lista de Clusters (*LC*):* En el caso de consultas de tipo kNN, este índice se representó con las mismas estructuras de datos usadas para resolver consultas por rango (Subsección V-A2).

La Figura 17 muestra el tiempo de ejecución de la LC usando un método de *radio decreciente* y *radio creciente* sobre la base de datos *Images*. El método de radio decreciente inicializa el radio igual ∞, y éste decrece en dos instancias: (i) luego de obtener la distancia entre los centros y la consulta q, y (ii) al momento de realizar una evaluación de distancia entre un elemento de un cluster y q. El ajuste del radio se realiza usando una función atómica. Por otro lado, el método de radio creciente, establece un radio inicial, y se realiza una búsqueda iterativa, aumentando en un Δ el radio de búsqueda en cada iteración, hasta encontrar al menos K elementos resultado.. Se utilizó el 1 % de los elementos de la base de datos como elementos de entrenamiento para establecer los valores del radio inicial y Δ de forma previa a la búsqueda.

Esta figura muestra un peor rendimiento para el método de radio decreciente. Esto se debe principalmente a que los threads de un bloque deben realizar en paralelo el proceso de búsqueda con el mismo radio para luego actualizarlo, es decir, el radio es actualizado cada T_{Block} elementos visitados (T_{Block}=número de threads de un bloque), lo que no permite reducir el radio suficientemente rápido como lo haría un algoritmo secuencial (reduciendo el radio por cada elemento visitado). Para la base de datos *Spanish* los resultados fueron similares.

Con respecto al método de radio creciente, se deben realizar I iteraciones para resolver una consulta ($1 \leq I$). Debido a esto (y teniendo en cuenta que cada bloque de threads resuelve una consulta completamente), la carga de trabajo de cada bloque varía según el valor de I en cada consulta, y esto produce un desbalance de carga significativo entre los multiprocesadores, degradando el rendimiento. Para lidiar con esto, se decidió realizar dos lanzamientos de *kernels*, en el primero se resuelven todas las consultas con $radio = r_{ini}$ y las que necesitan más iteraciones ($I > 1$) se dejan en una cola pendientes, para ser resueltas en el segundo lanzamiento de *kernel*. Esto implica que los bloques de threads realizarán la misma cantidad de trabajo en el primer *kernel*, mientras que en el segundo sólo habrá una diferencia con aquellos bloques de threads que resuelvan consultas que requieran $I \geq 3$ (estas consultas representan un porcentaje menor), y al ser menor la cantidad de bloques distribuidos entre los multiprocesadores, mejora el rendimiento. Los elementos que requieren $I \geq 2$ fueron alrededor de 40 % para las base de datos *Images* y 10 % para *Spanish*. No se reusaron evaluaciones de distancia realizadas en una iteración anterior, debido a que esto incrementa considerablemente la irregularidad en los threads, lo que empeora el rendimiento.

Para que cada thread de un bloque pueda almacenar sus K elementos más cercanos que ha encontrado hasta el momento, se utiliza un *heap* [36] por cada thread. Una vez que cada thread obtiene sus K elementos más cercanos a la consulta (que corresponden a los K elementos de su heap), se realiza una reducción de éstos utilizando *shared memory*. Dicha reducción es realizada por el primer *warp* del bloque y posteriormente por el primer thread del bloque.

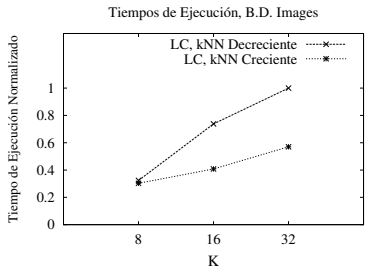


Figura 17: Tiempo de ejecución normalizado sobre la base de datos *Images* de un método de radio creciente y decreciente para resolver consulta de tipo kNN usando la LC sobre GPU.

Los valores de K (8, 16 y 32) usados en los experimentos del presente trabajo permitieron almacenar los heaps de un *warp* en *shared memory*. Estos valores de K se seleccionaron debido a que son usados en papers previos [26], [27], [28].

El conjunto de heaps que reside en *device memory* es almacenado en una matriz de $K \times T_{Block}$. Cada columna representa un heap, de esta forma la i-ésima columna tendrá los elementos correspondientes al heap del i-ésimo thread.

Se utilizó el método de radio creciente.

Las etapas del algoritmo están delimitadas por la función de sincronización `syncthreads()`, y éstas se describen a continuación.

- En la primera etapa se establece previamente el radio inicial. Posteriormente, los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory*.
- En la segunda etapa, cada thread (siguiendo una distribución Round-Robin) obtiene la distancia entre un centro de cluster y la consulta, y la almacena en *shared memory*.
- En la tercera etapa, los elementos de los clusters son asignados a cada thread siguiendo una distribución Round-Robin. Cada elemento que pertenece a un cluster que no puede ser descartado usando desigualdad triangular es comparado contra la query. Y si el elemento está dentro del radio actual de búsqueda, entonces éste se inserta en el heap del thread almacenado en *device memory*. Luego, los centros son distribuidos (Round-Robin) entre los threads y cada centro que se encuentre dentro del radio actual de búsqueda es insertado en el heap del thread.
- En la cuarta etapa, el primer *warp* del bloque de threads accede a los elementos de los heaps de la etapa anterior. Cada thread del *warp* almacena, si corresponde, los elementos en su heap almacenado en *shared memory*.

- Finalmente en la última, etapa el primer thread de cada bloque se encarga de recorrer y almacenar los elementos de los $SIZE_{Warp}$ heaps de la etapa anterior en *shared memory*.Si no se han encontrado, al menos, K elementos resultado, entonces el proceso se repite con un incremento en el radio de búsqueda.

V-C3. SSS-Index*:* Para resolver consultas de tipo kNN, este índice se representó con las mismas estructuras de datos usadas para resolver consultas por rango (Subsección V-A3).

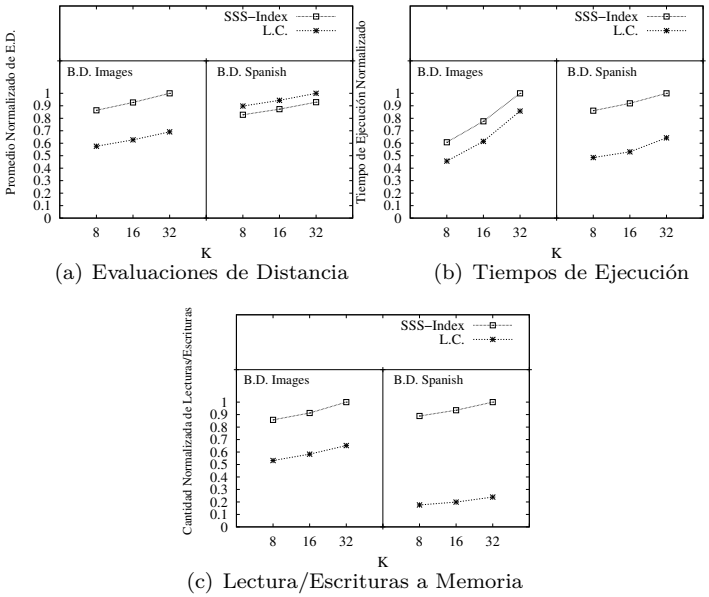
Al igual que en la LC (y por las mismas razones), se utilizó el método de radio creciente y las consultas son resueltas en dos lanzamientos de *kernels*. También, cada thread utiliza como estructura auxiliar un heap, para después realizar una reducción de éstos, utilizando el primer *warp* del bloque y posteriormente el primer thread del bloque.

Con respecto a la cantidad de pivotes, se encontró un resultado similar a lo ocurrido con el algoritmo de búsqueda por rango, el mejor rendimiento se obtiene con 1 pivote ($\alpha = 0{,}66$) para la base de datos de vectores. Las razones de este comportamiento son las mismas que las descritas en la Subsección V-A3.

El algoritmo de búsqueda se dividió en etapas delimitadas por una función de sincronización. En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory*. En la segunda etapa, cada thread (siguiendo una distribución Round-Robin) obtiene la distancia entre un pivote y la consulta, y la almacena en *shared memory*. En la tercera etapa, al igual que la búsqueda por rango, cada elemento es asignado a un thread siguiendo una distribución Round-Robin. Cada thread intenta descartar el elemento mediante desigualdad triangular, y en caso de no ser posible, el mismo thread realiza la evaluación de distancia entre el elemento y la consulta. Cada thread, si corresponde, almacena los elementos en su heap en *device memory*. En la cuarta etapa, el primer *warp* accede a los elementos de los heaps de la etapa anterior. Cada thread del *warp* almacena los elementos en su heap almacenado en *shared memory*. Finalmente en la última etapa, el primer thread de cada bloque se encarga de recorrer y almacenar los elementos de los $SIZE_{Warp}$ heaps de la etapa anterior en *shared memory*. Si no se han encontrado, al menos, K elementos resultado, el proceso se repite.

V-D. Resultados experimentales sobre GPU para resolver consultas kNN

A continuación se presentan los experimentos comparativos entre el algoritmo de búsqueda exhaustiva basado en ordenamiento y los índices LC y *SSS-Index* para resolver consultas de tipo kNN. La GPU y bases de datos utilizadas



(a) Evaluaciones de Distancia (b) Tiempos de Ejecución

(c) Lectura/Escrituras a Memoria

Figura 18: Valores normalizados del **a)** promedio de evaluaciones de distancia por consulta, **b)** tiempo de ejecución, y **c)** cantidad de lecturas/escrituras a *device memory* de los índices *SSS-Index* y LC sobre GPU.

fueron las mismas que en los experimentos de consultas por rango (Subsección V-B).

La Figura 18 muestra tres gráficos correspondientes a: (i) el promedio de evaluaciones de distancia por consulta, (ii) el tiempo de ejecución, y (iii) la cantidad de lecturas/escrituras a *device memory*. Todos los valores están normalizados al mayor valor observado en el experimento con la finalidad de apreciar mejor la diferencia entre los índices.

Se observa una correspondencia entre los gráficos de tiempo de ejecución y de lecturas/escrituras a memoria, a diferencia del gráfico de evaluaciones de distancia y el tiempo de ejecución sobre la base de datos *Spanish*. Esto se debe al costo asociado a una instrucción de lectura (o escritura) y al poder de cómputo de la GPU. El índice LC es el que presenta una menor cantidad de operaciones de lecturas/escrituras, y debido a esto, es el que consigue el mejor rendimiento en cuanto a tiempo de ejecución, sobre ambas bases de datos.

Cabe destacar que las instrucciones de escritura sólo se realizan al acceder a los heaps almacenados en *device memory*.

La Figura 19 muestra el speed-up de los índices LC y *SSS-Index* sobre GPU y sobre una plataforma multi-core de 8 núcleos, resolviendo consultas kNN. El speed-up se calculó sobre un algoritmo secuencial de búsqueda exhaustiva que utiliza un heap como estructura auxiliar. Las versiones multi-core se basaron en la estrategia *Local* (Subsección IV-C), utilizando los parámetros óptimos

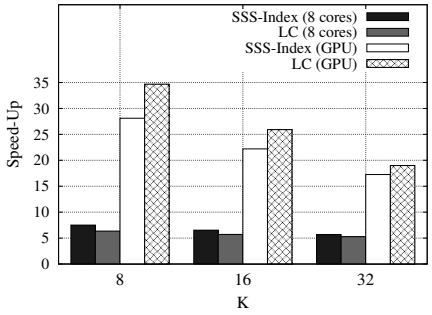


Figura 19: Speed-Up de las versiones sobre una plataforma multi-core y GPU de la *LC* y *SSS-Index* sobre un algoritmo secuencial de búsqueda exhaustiva, sobre la base de datos *Images* resolviendo consultas de tipo kNN.

Tabla III: Tiempos reales de ejecución (en segundos) entre el algoritmo de ordenamiento y la *LC*. Sobre la base de datos *Images*, resolviendo consultas kNN.

K	Algoritmo de Ordenamiento	Lista de Clusters
8	154,8	2,5
16	154,8	3,3
32	154,8	4,5

para cada índice. Los speed-ups de las versiones en GPU obtienen una clara ventaja, alcanzando hasta 34.7x en la *LC* y 28.1x en el *SSS-Index*.

La Tabla III muestra los tiempos de ejecución entre la *LC* y el algoritmo de búsqueda exhaustiva basado en ordenamiento (Subsección V-C1). Siendo este último ampliamente superado por la *LC*. Al variar el valor de K en el algoritmo de ordenamiento, éste no sufre variación, pues este parámetro sólo interviene al momento de seleccionar los primeros K elementos del arreglo de distancias ordenado.

V-E. Solución de consultas en-línea

Los experimentos en GPU hasta ahora han sido asumiendo un alto tráfico de consultas, es decir, el conjunto de consultas por resolver es conocido con anticipación. Pero esta asunción puede ser inasequible en sistemas de tiempo real en-línea, como motores de búsqueda web [33].

Para evaluar si los índices sobre GPU serían eficientes sobre este contexto, se ejecutó un experimento de productividad en función del número de consultas disponibles en paralelo para resolver consultas por rango. Asumiendo que no se sabe con certeza la cantidad de consultas que se pueden llegar a tener en un determinado momento, el experimento consistió en resolver consultas por lotes. La Figura 20(a) muestra los resultados para la *LC*. El eje x indica cuantas consultas son lanzadas en paralelo, es decir el tamaño del lote de consultas (empezando desde 5), que se lanza por vez hasta completar el total de consultas. El eje y muestra la productividad del sistema medido en el número de consultas procesadas por segundo. No es

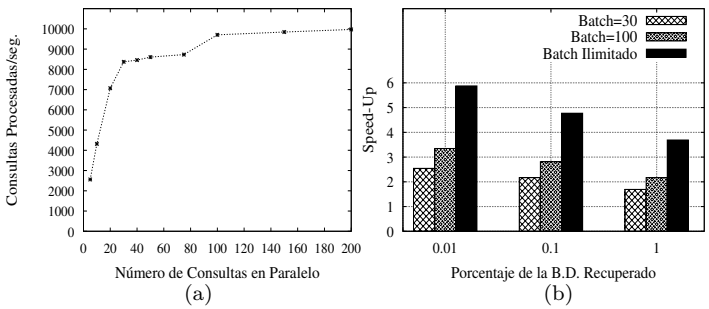


Figura 20: Utilizando la *LC* para resolver consultas kNN sobre la base de datos *Images*, se muestra **a)** la productividad (número de consultas resueltas por segundo) resolviendo el total de las consultas por lotes de distinto tamaño, y el **b)** Speed-up resolviendo las consultas por lotes (de 30 y 100), sobre su correspondiente implementación multi-core.

sorpresa que la productividad crezca rápidamente al punto donde se alcanzan 30 consultas en paralelo, dado que la GPU usada posee 30 multiprocesadores. A partir de 30, la productividad continúa creciendo pero más lentamente. Lanzando consultas en lotes de 200 se alcanza casi la máxima productividad.

La Figura 20(b) muestra el speed-up para la versión en GPU de la *LC* sobre su correspondiente implementación en la plataforma multi-core, resolviendo consultas kNN. La barra etiquetada como *Batch Ilimitado* corresponde al speed-up de la *LC* teniendo todas las consultas disponibles desde el inicio. Las barras etiquetadas como *Batch=30* y *Batch=100* corresponden al escenario donde las consultas se resuelven por lotes de 30 y 100 respectivamente, hasta completar el total de las consultas. La implementaciones *Batch=30* y *Batch=100* implicaron 795 y 239 invocaciones de *kernels* respectivamente. Aún si la productividad no está en el punto más alto, las versiones en GPU siempre superan las versiones sobre la plataforma multi-core. Cabe destacar que en este experimento la implementación multi-core asume el mejor caso, es decir, se conocen todas las consultas desde el comienzo.

Un speed-up más alto que 2.1x (en promedio) se alcanza cuando se resuelven 30 consultas en paralelo, lo que representa un escenario de tráfico de consultas muy bajo. Debido a esto, se puede afirmar que los índices basados en GPU pueden ser usados para procesamiento de consultas en-línea en espacios métricos como una alternativa de bajo coste a implementaciones multi-CPU.

V-F. Conclusiones acerca de la distribución y búsqueda sobre GPU

En esta sección, se han propuestos y comparado algoritmos basados en CUDA sobre GPU para resolver consultas en espacios métricos. Se utilizaron los índices *LC* y *SSS-Index* debido a que almacenan su índice en matrices, lo

que es una característica conveniente al utilizar una GPU. Los parámetros óptimos para ambos índices son diferentes a los usados en computación secuencial, en particular, con el *SSS-Index* el óptimo se alcanza utilizando sólo un pivote con la base datos de vectores.

Se abarcaron las consultas por rango y kNN por separado, pues debido a las características de la GPU, las soluciones de ambos tipos de consultas difieren considerablemente al igual que los problemas encontrados en cada una de ellos. En particular, en consultas kNN, los índices se implementaron utilizando un conjunto de *heaps*, y se muestra la eficiencia de utilizar esta cola de prioridad sobre GPU al almacenar sus elementos por columnas. Los elementos de los heaps son almacenados en *device memory*, y luego se realiza una reducción de éstos a *shared memory* para obtener los K elementos resultados. También en este tipo de consultas el método de *radio creciente* presenta un mejor rendimiento que el de *radio decreciente*, que es lo opuesto a lo que ocurre en computación secuencial.

En ambos tipos de consultas, la LC presentó el mejor rendimiento, debido a que presenta un buen alineamiento en los accesos a memoria y regularidad en la ejecución de instrucciones por parte de threads consecutivos. Este índice muestra un speed-up de hasta 100.6x sobre el algortimo secuencial de fuerza bruta, y de 5.9x sobre su versión en una plataforma multi-core.

En el contexto de sistemas de tiempo real, se muestra que el índice LC sobre GPU puede ser usado para procesamiento de consultas en-línea debido a su buena productividad con una baja frecuencia de consultas.

VI. Conclusiones

En el presente trabajo se han desarrollado algoritmos sobre dos plataformas paralelas distintas. La primera de ellas, una plataforma multi-core de 2 CPUs, cada una con 4 núcleos, donde cada uno de éstos posee un gran poder de procesamiento y autonomía. La segunda, una GPU con 30 multiprocesadores, y cada uno de ellos con 8 núcleos, donde cada núcleo tiene un poder de procesamiento menor y comparte una memoria reducida en tamaño con el resto de núcleos del mismo multiprocesador.

En la Sección IV (correspondiente al trabajo realizado sobre una plataforma multi-core), se desarrollaron algoritmos que usan distintas estrategias de gestión de threads diseñadas para mejorar la tasa de consultas en espacios métricos utilizando como herramienta OpenMP.

Los algoritmos propuestos combinan procesamiento asíncrono multi-thread con procesamiento bulk-sincrónico de consultas en un entorno de memoria compartida. El cambio entre un modo y otro es realizado según el tráfico de consultas observado. Estas estrategias propuestas son relativamente independientes del índice métrico utilizado. La estrategia propuesta con mejor rendimiento es la estrategia *híbrida*, que es capaz de cambiar su modo de operación según el tráfico entrante de consultas. También el mantener sólo un índice global muestra un mejor rendimiento frente a un índice distribuido con datos locales.

En la Sección V (correspondiente al trabajo realizado sobre GPU) se proponen implementaciones de los índices LC y *SSS-Index* para resolver consultas en espacios métricos utilizando GPUs. Los parámetros óptimos para cada índice son diferentes a los usados en computación secuencial, en particular, con el *SSS-Index* el óptimo se alcanza utilizando sólo un pivote sobre la base de datos de vectores. En la misma sección, se abarcan las consultas por rango y kNN por separado, pues debido a las características de la GPU, las soluciones de ambos tipos de consultas difieren considerablemente, al igual que los problemas encontrados en cada una de ellos. En particular, en consultas kNN, los índices se implementaron utilizando un conjunto *heaps*, y se demuestra lo eficiente que resulta el uso de esta cola de prioridad sobre GPU al almacenar los datos por columnas. Los elementos de los heaps son almacenados en *device memory*, y luego se realiza una reducción de éstos a *shared memory*, para obtener finalmente los K elementos resultado. También en este tipo de consultas el método de *rango creciente* presenta un mejor rendimiento que el de *rango decreciente*, que es lo opuesto a lo que ocurre en computación secuencial.

El índice LC es el que mejor rendimiento muestra sobre ambas plataformas, alcanzando un speed-up de hasta 7.8x en la plataforma multi-core de 8 núcleos, sobre la versión secuencial. Este mismo índice sobre GPU, supera ampliamente a la versión multi-core, alcanzando hasta 100.6x sobre el algoritmo secuencial de fuerza bruta, y de 5.9x sobre su versión multi-core.

Bajo un contexto de sistemas de tiempo real, se muestra que la GPU puede ser usada para procesamiento de consultas en-línea en espacios métricos, como una buena alternativa de bajo coste (monetario) frente a implementaciones multi-CPU tradicionales.

Según nuestro conocimiento, en las publicaciones derivadas de la presente tesis (que se muestran en la Subsección VI-B), se da a conocer la primera publicación internacional que utiliza indexación para resolver consultas por rango y kNN en espacios métricos sobre GPU.

VI-A. Trabajo Futuro

En el área de distribución y búsqueda sobre una plataforma multi-core:

- Diseñar estrategias específicas para algún índice, que explote los distintos niveles de caché, con la finalidad de obtener un mejor rendimiento que la estrategia *Local*.
- Integrar las estrategias desarrolladas sobre una plataforma de memoria distribuida.
- Implementar el dinamismo en-línea de un índice métrico definiendo una política de reinserción y eliminación.

En el área de distribución y búsqueda sobre GPU:

- Diseñar algoritmos que hagan un buen uso de las cachés de constantes y de texturas.
- Explorar lo que ocurriría al trabajar con funciones de distancia de gran complejidad que impliquen pocas lecturas (y escrituras), pero muchas operaciones aritméticas.
- Resolver consultas de mayor dimensionalidad, verificando si en este tipo de plataforma es eficiente el uso de índices frente a una implementación de fuerza bruta que tiene la ventaja de poseer mayor uniformidad en la ejecución de instrucciones.

VI-B. Contribuciones

El presente trabajo ha dado lugar a las siguientes publicaciones:

- "Scheduling Metric-Space Queries Processing on Multi-Core Processors", In 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (Euro-PDP 2010), IEEE CS, pages 187-194, Pisa, Italy, Feb. 2010.
- "kNN Query Processing in Metric Spaces using GPUs", In 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), ser. LNCS, vol. 6852, pages 380-392, Bordeaux, France, Sept. 2011.

Referencias

[1] P. Zezula, G. Amato, V. Dohnal, and M. Batko, *Similarity Search - The Metric Space Approach*, ser. Advances in Database Systems. Springer, 2006, vol. 32.

[2] "Google goggles," in *http://www.google.com/mobile/goggles/*.

[3] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. New York: Morgan Kaufmann, 2006.

[4] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[5] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[6] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[7] "Cuda: Compute unified device architecture," in *©2007 NVIDIA Corporation*.

[8] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," in *ACM Computing Surveys*, September 2001, pp. 33(3):273–321.

[9] T. Cover and P. Hart, "Nearest neighbor pattern classification," *Information Theory, IEEE Transactions on*, vol. 13, no. 1, pp. 21–27, 1967.

[10] N. R. Brisaboa, A. Fariña, O. Pedreira, and N. Reyes, "Similarity search using sparse pivots for efficient multimedia information retrieval," in *ISM*. IEEE Computer Society, 2006, pp. 881–888.

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

[12] 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)*. Los Alamitos, USA: IEEE CS Press, 1999, pp. 38–46.

[13] S. Nene and S. Nayar, "A simple algorithm for nearest neighbor search in high dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, pp. 989–1003, 1997.

[14] S. Brin, "Near neighbor search in large metric spaces." in *the 21st VLDB Conference*. Morgan Kaufmann Publishers, 1995, pp. 574–584.

[15] G. Navarro and R. Uribe-Paredes, "Fully dynamic metric access methods based on hyperplane partitioning," *Information Systems*, vol. 36, no. 4, pp. 734 – 747, 2011.

[16] P. Ciaccia, M. Patella, and P. Zezula, "M-tree : An efficient access method for similarity search in metric spaces." in *the 23st International Conference on VLDB*, 1997, pp. 426–435.

[17] E. Chávez and G. Navarro, "An effective clustering algorithm to index high dimensional metric spaces," in *The 7th International Symposium on String Processing and Information Retrieval (SPIRE'2000)*. IEEE CS Press, 2000, pp. 75–86.

[18] N. R. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe, "Clustering-based similarity search in metric spaces with sparse spatial centers," in *34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, ser. LNCS, vol. 4910. Slovakia: Springer, 2008, pp. 186–197.

[19] V. G. Costa and M. Marin, "Distributed sparse spatial selection indexes," in *PDP*. IEEE Computer Society, 2008, pp. 440–444.

[20] V. G. Costa, M. Marin, and N. Reyes, "An empirical evaluation of a distributed clustering-based index for metric space databases," in *ICDE Workshops*. IEEE CS, 2008, pp. 386–393.

[21] M. Marin, "Range queries on distributed spatial approximation trees," in *IASTED International Conference on Databases and Applications*. Innsbruck, Austria: IASTED/ACTA Press, February 2005, pp. 140–144.

[22] M. Marin and V. G. Costa, "(sync|async)$^+$ mpi search engines," in *14th European PVM/MPI User's Group Meeting (PVM/MPI 2007)*, ser. LNCS, vol. 4757. Paris: Springer, 2007, pp. 117–124.

[23] M. Marin, V. G. Costa, and C. Bonacic, "A search engine index for multimedia content," in *14th International European Conference on Parallel and Distributed Computing (Euro-Par 2008)*, ser. LNCS, vol. 5168. Spain: Springer, August 2008, pp. 866–875.

[24] M. Marin, R. Uribe, and R. Barrientos, "Searching and updating metric space databases using the parallel egnat," in *International Conference on Computational Science (1)*, 2007, pp. 229–236.

[25] R. Uribe, G. Navarro, R. Barrientos, and M. Marin, "An index data structure for searching in metric space databases," in *6th International Conference on Computational Science (ICCS 2006)*, ser. LNCS, vol. 3991. UK: Springer, 2006, pp. 611–617.

[26] Q. Kuang and L. Zhao, "A practical gpu based knn algorithm," Huangshan, China, 2009, pp. 151–155.

[27] V. Garcia, E. Debreuve, and M. Barlaud, "Fast k nearest neighbor search using gpu," *Computer Vision and Pattern Recognition Workshop*, vol. 0, pp. 1–6, 2008.

[28] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Computational Science (ICCS)*, vol. 3994. Springer, 2006, pp. 196–199.

[29] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–10, 2009.

[30] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[31] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[32] G. Navarro, "Searching in metric spaces by spatial approximation," *The Very Large Databases Journal (VLDBJ)*, vol. 11, no. 1, pp. 28–46, 2002.

[33] M. Marin, V. Gil-Costa, C. Bonacic, R. Baeza-Yates, and I. D. Scherson, "Sync/async parallel search for the efficient design and construction of web search engines," *Parallel Computing*, vol. 36, no. 4, pp. 153 – 168, 2010.

[34] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore GPUs," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 1–10.

[35] D. Cederman and P. Tsigas, "Gpu-quicksort: A practical quicksort algorithm for graphics processors," *J. Exp. Algorithmics*, vol. 14, pp. 1.4–1.24, 2009.

[36] D. E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 1973, vol. 3.