UNIVERSIDAD DE CHILE
FACULTAD DE FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA POR SIMILITUD EN ESPACIOS MÉTRICOS SOBRE PLATAFORMAS MULTI-CORE (CPU Y GPU)

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS MENCIÓN COMPUTACIÓN

RICARDO JAVIER BARRIENTOS ROJEL

2011

UNIVERSIDAD DE CHILE
FACULTAD DE FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

BÚSQUEDA POR SIMILITUD EN ESPACIOS MÉTRICOS SOBRE PLATAFORMAS MULTI-CORE (CPU Y GPU)

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS MENCIÓN COMPUTACIÓN

RICARDO JAVIER BARRIENTOS ROJEL

PROFESOR GUÍA:
JUAN MAURICIO MARIN CAIHUÁN

PROFESOR CO-GUÍA:
BENJAMIN BUSTOS CÁRDENAS

MIEMBROS DE LA COMISIÓN:
VERÓNICA GIL-COSTA
GONZALO NAVARRO BADINO
FERNANDO RANNOU FUENTES

SANTIAGO DE CHILE
JUNIO 2011

A Telmo e Iris

Agradecimientos

Se me hace muy grato recordar a las personas que de alguna u otra forma colaboraron en la realización de este trabajo.

Por supuesto, debo comenzar por mis padres, Telmo e Iris, los que me permitieron obtener mi primer título como profesional, y posteriormente, depositando su confianza en mí y en mis capacidades, me apoyaron en la decisión de realizar un postgrado.

Son varios los profesores que me han ayudado en distintas etapas de este trabajo. Primeramente, debo agradecer a Mauricio Marin, a quien conozco de hace varios años y por el que tengo un profundo aprecio. Muchas gracias por todo su tiempo invertido en mi formación y preocupación constante a través de los años por mi desarrollo profesional y bienestar personal.

A Verónica Gil-Costa por su siempre buena disposición a cooperar y su valiosa ayuda en el presente trabajo.

Al profesor Roberto Uribe, por todo el apoyo, ánimo y confianza en mi trabajo. Ha sido un apoyo constante, del que estoy muy agradecido.

A los profesores Francisco Tirado, Manuel Prieto, Ignacio Gómez y Christian Tenllado por su valiosa ayuda y conocimiento aportado en la realización de este trabajo. Todos ellos han depositado una gran confianza en mí y en mi trabajo, y han permitido desarrollarme plenamente en mis áreas de investigación en la etapa actual de mi vida.

A Ruth Calisto, mi tía, por su constante preocupación y apoyo incondicional a través del tiempo. Ha hecho mi estadía en Santiago mucho más agradable y amena.

A Angélica Aguirre, jefa de estudios, por su constante apoyo y siempre muy buena disposición a colaborar. A todos mis compañeros y amigos, especialmente a aquellos que he conocido mientras he estado en Santiago, pues han hecho mucho más grata mi estadía.

También debo agradecer al *Departamento de Ciencias de la Computación* de la Universidad de Chile, al *Centro de Investigaciones de la Web* (CIW) y a *Yahoo! Research LA* por las becas otorgadas.

Ricardo J. Barrientos, Junio 2011.

Resumen

Este trabajo de tesis está dedicado al estudio de la gestión eficiente de threads en el procesamiento de consultas sobre índices para espacios métricos. Se proponen algoritmos multi-thread eficientes para procesadores multi-core convencionales y para procesadores gráficos GPU.

Para procesadores multi-core se concluye que la mejor estrategia es mantener un sólo índice compartido por todos los threads y se propone aplicar una estrategia híbrida de procesamiento de consultas, la cual aplica una estrategia distinta dependiendo del nivel de tráfico de consultas que llegan al procesador. La estrategia propuesta alcanza buen rendimiento para diversos tipos de índices métricos. Para tráfico alto de consultas, la estrategia híbrida aplica un método estándar en el que cada thread se hace cargo de procesar completamente una consulta por vez. Para tráfico bajo de consultas, la estrategia híbrida utiliza varios threads para procesar cada consulta utilizando paralelismo sincrónico por lotes. También se concluye que la estrategia intuitiva de particionar el índice en tantos threads como núcleos tenga el procesador, y asignar un thread a cada partición para ejecutar en ella el algoritmo secuencial, no entrega buenos resultados en comparación a la estrategia híbrida propuesta en este trabajo de tesis.

Para procesadores gráficos GPU, fue necesario diseñar tanto una estrategia de distribución del índice en los núcleos como estrategias de procesamiento de consultas que tengan en cuenta el modelo de programación de las GPUs. En este caso, los algoritmos propuestos resultan ser bastante más complejos que los diseñados para procesadores convencionales, pero estos muestran de que es posible alcanzar un rendimiento eficiente y escalable en este hardware. Se observó que sólo los índices métricos que pueden ser representados como matrices permiten alcanzar un buen rendimiento. En particular, un índice basado en clustering de objetos es el que alcanza el mejor rendimiento tanto para consultas por rango como para consultas de vecinos más cercanos.

Índice general

Agradecimientos I

Resumen II

1. Introducción 1
- 1.1. Objetivos . . . 2
- 1.2. Organización de la tesis . . . 2

2. Conocimientos básicos 4
- 2.1. Paralelismo . . . 4
  - 2.1.1. OpenMP . . . 4
  - 2.1.2. Threads vs. Cores . . . 6
  - 2.1.3. GPU (Graphic Process Units) . . . 6
    - 2.1.3.1. Hardware . . . 6
    - 2.1.3.2. Ejecución . . . 7
    - 2.1.3.3. Reducción de lecturas a memoria . . . 9
    - 2.1.3.4. Funciones atómicas . . . 11
- 2.2. Espacios métricos . . . 12
  - 2.2.1. Búsquedas por similitud . . . 12
  - 2.2.2. Indexación . . . 13
  - 2.2.3. Indices métricos . . . 16
    - 2.2.3.1. EGNAT . . . 16
    - 2.2.3.2. M-Tree . . . 18
    - 2.2.3.3. SSS-Index . . . 20
    - 2.2.3.4. SSS-Tree . . . 22
    - 2.2.3.5. Lista de Clusters (LC) . . . 22
- 2.3. Trabajo relacionado . . . 26
  - 2.3.1. Memoria distribuida . . . 26
  - 2.3.2. GPU . . . 26

3. Estrategias de distribución y búsqueda sobre procesadores Multi-core **28**
3.1. Modelo de búsqueda . . . 28
3.2. Descripción de la búsqueda . . . 28
3.2.1. Estrategia Local . . . 29
3.2.2. Estrategia Bulk-Circular . . . 30
3.2.3. Estrategia Bulk-Critical . . . 32
3.2.4. Estrategia Bulk-Local . . . 32
3.3. Resultados experimentales . . . 32
3.3.1. Estrategia híbrida . . . 35
3.3.2. Distribución Local de la base de datos . . . 38
3.4. Conclusiones . . . 39

4. Estrategias de distribución y búsqueda sobre GPU **45**
4.1. Consultas por Rango . . . 46
4.1.1. Fuerza Bruta . . . 46
4.1.2. Lista de Clusters (LC) . . . 47
4.1.3. *SSS-Index* . . . 48
4.2. Consultas de los k vecinos más cercanos (kNN) . . . 52
4.2.1. Búsqueda exhaustiva . . . 53
4.2.1.1. Solución basada en ordenamiento . . . 53
4.2.2. Lista de Clusters (LC) . . . 54
4.2.3. *SSS-Index* . . . 56
4.3. Resultados experimentales sobre GPU . . . 59
4.3.1. Experimentos sobre consultas por rango . . . 61
4.3.2. Experimentos sobre consultas de tipo kNN . . . 63
4.3.3. Solución de consultas en-línea . . . 66
4.4. Conclusiones . . . 67

5. Conclusiones **69**
5.1. Trabajos futuros . . . 70

Índice de figuras

2.1. Plataforma multi-core. 5
2.2. Ejemplo de programa usando OpenMP. 6
2.3. Arquitectura de hardware en una GPU. 8
2.4. Ejemplo de suma de vectores con CUDA. 10
2.5. Ejemplos para ilustrar la sincronización de los threads de un *warp*. 11
2.6. Ejemplos de consultas, por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos en $\mathbb{R}^2$. 13
2.7. *EGNAT*: Construcción del árbol (aridad 4). 17
2.8. *EGNAT*: Inserción de un nuevo objeto. 17
2.9. Descarte de subárboles usando rangos. Se descarta D_x ya que $d(q,p) + r < min\_d(p, D_x)$. 19
2.10. *M-tree*: Construcción de la estructura. 20
2.11. *SSS-Tree*: Estructura después del primer paso de construcción. 22
2.12. Campos de un nodo en el *SSS-Tree*. 23
2.13. *Lista de Cluster*: Zonas abarcadas por 3 centros según este orden: c_1, c_2, c_3. . . . 24
2.14. *Lista de Cluster*: Tres casos de búsqueda. 25
2.15. Particionamiento de las matrices de datos y consultas en submatrices. 27

3.1. Modelo de Búsqueda usado en los experimentos. 29
3.2. Tiempo de Ejecución para la estrategias *Bulk-Circular* y *Bulk-Critical*. 34
3.3. Tiempo de Ejecución para un alto tráfico de consultas. 36
3.4. Tiempo de Ejecución para un bajo tráfico de consultas. 37
3.5. Distribuciones de consultas en un escenario de baja frecuencia. 38
3.6. Comparación de los índices con la estrategia *Local* para alto tráfico de consultas y la *Bulk-Circular* para bajo tráfico. 40
3.7. Speed-up de los índices sobre la BD Spanish con radio 3. 41
3.8. Eficiencia de las estrategias multi-core usando la LC con un tráfico bajo de consultas. 42
3.9. Consultas completamente resueltas por unidad de tiempo. 42
3.10. Tiempos de Ejecución normalizados para las bases de datos **a)***Images* y **b)***Spanish*. 43
3.11. Promedio normalizado de evaluaciones de distancias por consulta para las bases de datos **a)***Images* y **b)***Spanish*. 44

4.1. 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*. 51
4.2. Cantidad de veces en que al menos un thread de un *warp* diverge para la estrategia *Distribución-Pivote* sobre la base de datos *Images*. 53
4.3. Tiempo de ejecución normalizado sobre la base de datos *Images* de un método de rango creciente y decreciente para resolver consulta de tipo kNN usando la *LC* sobre GPU. 55
4.4. 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*. 59
4.5. 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. 62
4.6. Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión secuencial sobre la base de datos *Images* resolviendo consultas por rango. 63
4.7. Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión en OpenMP sobre la base de datos *Images* resolviendo consultas por rango. 64
4.8. 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. 65
4.9. Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión secuencial optimizada sobre la base de datos *Images* resolviendo consultas de tipo kNN. 66
4.10. **a)** Productividad (número de consultas dividido por su correspondiente tiempo de ejecución) resolviendo distintos números de consultas de la *LC* sobre la base de datos *Images*. **b)** Speed-Up de la *LC* resolviendo un conjunto de consultas (de 30 y 100) al mismo tiempo sobre la correspondiente implementación OpenMP con la base de datos *Images*. 68

Índice de tablas

3.1. Características Generales 32

4.1. Tiempos de ejecución (en segundos) y cantidad de lecturas/escrituras a *device memory*. 52

4.2. Tiempos reales de ejecución (en segundos) entre el algoritmo de ordenamiento y la *LC*. 64

Índice de algoritmos

1. *EGNAT*: búsqueda por rango r para la consulta q. 18
2. *M-Tree*: búsqueda por rango r para la consulta q. 21
3. *SSS-Index*: algoritmo de selección de pivotes. 21
4. *SSS-Index*: algoritmo de búsqueda. 21
5. *SSS-Tree*: búsqueda por rango r para la consulta q. 23
6. *Lista de Cluster*: algoritmo de construcción. 24
7. *Lista de Cluster*: búsqueda por rango r para la consulta q. 25
8. Búsqueda para la estrategia Local. 30
9. Búsqueda en la estrategia Bulk-Circular. 31
10. Búsqueda en la estrategia Bulk-Critical. 33
11. Búsqueda por Fuerza Bruta en consultas por rango sobre GPU. 47
12. *Kernel* de búsqueda de la LC para resolver consultas por rango sobre GPU. . . . 49
13. *Kernel* de búsqueda del *SSS-Index* para resolver consultas por rango sobre GPU. 50
14. Funciones auxiliares usadas por los algoritmos de búsqueda de consultas kNN en los índices LC y *SSS-Index* sobre GPU. 57
15. *Kernel* de búsqueda de la LC para resolver consultas de tipo kNN sobre GPU. . 58
16. *Kernel* de búsqueda del *SSS-Index* para resolver consultas de tipo kNN sobre GPU. 60

Capítulo 1

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*, 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 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, bio-informática, reconocimiento de patrones, minería de datos, la Web.

Actualmente los buscadores para la Web indexan docenas de billones de documentos y cientos de millones de otros tipos de objetos complejos tales como audio, video e imágenes. Por ejemplo existen aplicaciones especializadas en imágenes tales como los sistemas que permiten a comunidades de usuarios publicar y compartir fotografías. Al subir una nueva imágen, el usuario debe etiquetarla con un texto pequeño que describe su contenido. Esto le permite al buscador realizar el ranking de imágenes y desplegar las más relevantes para una consulta formulada en texto.

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) con memoria distribuida 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 índices métricos sobre uno de los nodos antes mencionados, el que posee una arquitectura de memoria compartida. Se proponen e implementan estrategias de búsqueda que puedan implementarse sobre cualquier estructura métrica que cumpla ciertos requisitos mínimos. También se analiza su comportamiento bajo distintos tráficos de consultas.

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. Estos han sido optimizados para resolver consultas individuales y no necesariamente mantienen su eficiencia cuando se paralelizan en sistemas de memoria distribuida o compartida.

Las plataformas de hardware 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 T10, 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.

1.1. Objetivos

El objetivo principal de esta tesis es el diseño, implementación y evaluación de estrategias de distribución y procesamiento paralelo de consultas para índices métricos sobre plataformas paralelas de memoria compartida.

Los objetivos específicos son los siguientes:

- Diseñar estrategias de particionado e indexación de objetos para explotar el paralelismo disponible en distintos esquemas de indexación secuencial para espacios métricos.
- Diseñar estrategias de procesamiento de consultas relativamente independientes del tipo de estructura de datos utilizada para indexar los objetos de la base de datos.
- Evaluar el rendimiento de las estrategias desarrolladas sobre procesadores multi-core y procesadores gráficos GPU, utilizando un conjunto de bases de datos de naturaleza distinta respecto de las funciones de distancia entre objetos.

1.2. Organización de la tesis

Los capítulos siguientes de esta tesis están organizados de la siguiente manera:

- En el Capítulo 2 se introduce a las áreas de espacios métricos, sistemas distribuidos, programación multithread con OpenMP sobre procesadores multi-core convencionales, el modelo de programación CUDA de NVIDIA para procesadores gráficos basados en GPUs, y los trabajos previos relevantes sobre estas áreas. No se encontró trabajo relacionado con respecto a paralelización sobre espacios métricos en una plataforma de memoria compartida. Pero sí hay trabajo relacionado sobre memoria distribuida ([15, 16, 17, 26, 27, 28, 29, 30, 31, 46]), los cuales sirvieron como base para el desarrollo de esta tesis.

- En el Capítulo 3 se describe el proceso de búsqueda y las estrategias de distribución propuestas y desarrolladas bajo una plataforma multi-core. Se muestran los algoritmos aplicados con cada estrategia y una solución al problema de acceso a datos compartidos entre threads concurrentes. En la Sección 3.3 se muestran los primeros experimentos, mostrando el desempeño de las estrategias propuestas bajo escenarios de baja y alta frecuencia de consultas entrantes. En la Sección 3.3.2 se evalúa el caso de distribución local del índice entre los threads, de tal forma que cada thread genera su propio índice con datos distinto al del resto y procesa la consulta en cada índice local para luego integrar los resultados. En la Sección 3.4 se muestran las conclusiones del capítulo.

- En el Capítulo 4 se describen los algoritmos desarrollados sobre GPU (Graphic Processor Unit). Se detallan los métodos diseñados para procesar consultas sobre los índices seleccionados para este tipo de plataforma. También se enumeran las dificultades y diferencias con los métodos clásicos en CPU. Debido a las restricciones de la GPU se analizan las consultas de tipo *rango* y kNN de forma separada. En la Sección 4.4 se muestran las conclusiones del capítulo.

- En el Capítulo 5 se dan a conocer las conclusiones finales y globales del presente trabajo.

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), 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), Bordeaux, France, Sept. 2011.

Capítulo 2

Conocimientos básicos

Actualmente son cada vez más frecuentes las arquitecturas de computadores que incluyen varios núcleos en una misma CPU. Algunos ejemplos de procesadores multi-cores son el Intel Xeon X7460 o el AMD Opteron, los cuales son capaces de incluir 6 núcleos en una misma CPU.

Varias librerías 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 ([41], [49]).

A partir de una colección de objetos, es creado un *índice*, que es una estructura de datos que almacena información obtenida de dichos objetos, con la finalidad de disminuir la cantidad de *evaluaciones de distancia* entre los objetos de la base de datos y la consulta. Una evaluación de distancia entre dos objetos es una operación de gran costo, por lo que cada índice implementa distintos métodos de búsqueda para realizar la menor cantidad de evaluaciones de distancia posible.

2.1. Paralelismo

El presente trabajo se desarrolló bajo dos plataformas paralelas de memoria compartida. La primera de éstas la muestra la figura 2.1, en donde hay presente 2 CPU's, cada una de ella 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.

2.1.1. OpenMP

Existen varios modelos y estándares que permiten programación multi-thread, tales como Pthreads([37]), TBB ([40]) u OpenMP ([8]). Para el presente trabajo la biblioteca seleccionada fue OpenMP debido a su gran nivel de abstracción, pues el código resultante difiere poco del

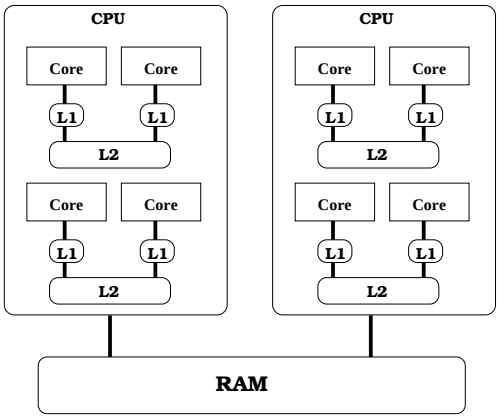


Figura 2.1: Plataforma multi-core.

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.

OpenMP se organiza mediante *directivas* y funciones, algunas de ellas son las siguientes:

Funciones:

***omp_set_num_threads(P)*:** Establece el número de threads de la aplicación (P threads en este caso).

***omp_get_thread_num()*:** Devuelve el identificador del thread.

***omp_get_num_threads()*:** Devuelve la cantidad de thread que están en ejecución.

***omp_get_num_procs()*:** Devuelve la cantidad disponibles de núcleos.

Directivas:

***#pragma omp parallel*:** Esta directiva inicia el paralelismo, en este punto del código se crean P threads, todos los threads ejecutan el mismo código y a cada uno se les asigna un identificador. Para esta directiva existe la opción de indicar las variables que serán globales (compartidas por todos los threads) y cuales privadas.

***#pragma omp critical*:** Esta directiva permite definir una zona de código que debe ser ejecutado por un thread a la vez.

***#pragma omp master*:** Esta directiva define una zona de código que será ejecutada por el thread maestro (el thread con identificador $ID = 0$).

```
#include <stdio.h>
#include <omp.h>
main ()
{
   int id_privado, global=0;
   omp_set_num_threads(4);/* Usamos 4 Threads */
   #pragma omp parallel shared(global) private(id_privado)
   {
      /* En este punto se crean 4 threads.
         Todos ellos ejecutan el siguiente codigo */
      id_privado = omp_get_thread_num();
      printf("Soy el Thread con id=%d\n", id_privado);
      #pragma omp critical
      {
         global++;
      }
      #pragma omp barrier
      #pragma omp master
      {
         printf("variable global = %d", global);
      }
    }  /* fin del bloque paralelo */
    return 0;
}
```

Salida:

```
Soy el Thread con id=0
Soy el Thread con id=2
Soy el Thread con id=1
Soy el Thread con id=3
variable global = 4
```

Figura 2.2: Ejemplo de programa usando OpenMP.

***#pragma omp barrier*:** Un thread al encontrar esta directiva se detiene hasta que todos los threads han alcanzado este punto.

La figura 2.2 muestra un ejemplo de programa con OpenMP y su respectiva salida. En esta figura, la directiva *#pragma omp critical* soluciona el problema de concurrencia al permitir la actualización de la variable *global* a un thread a la vez. La directiva *#pragma omp barrier* garantiza que cuando el thread maestro imprime la variable *global* todas las actualizaciones a esta variable ya fueron realizadas.

2.1.2. Threads vs. Cores

La librería sched.h permite asignar un thread a un núcleo determinado. En todos los experimentos del presente trabajo se asignó cada thread a un núcleo distinto. Esto asegura que no hay conflicto por recursos en un mismo núcleo.

2.1.3. GPU (Graphic Process Units)

2.1.3.1. Hardware

Todas las implementaciones sobre GPU se realizaron usando el modelo de programación CUDA ([1]) de NVIDIA. Este modelo hace una distinción entre el código ejecutado en la CPU

(*host*) con su propia DRAM (*host memory*) y el ejecutado en GPU (*device*) sobre su DRAM (*device memory*). La GPU se representa como un coprocesador capaz de ejecutar funciones denominadas *kernels*, y provee extensiones para el lenguaje C que permiten alojar datos en la memoria de la GPU y transferir datos entre GPU y CPU.

Por lo tanto, la GPU sólo puede ejecutar las funciones declaradas como *kernels* dentro del programa, y sólo puede usar variables alojadas en *device memory*, lo cual significa que todos los datos necesarios deberán ser movidos a esta memoria de forma previa a la ejecución del *kernel*.

La arquitectura de la GPU utilizada en este trabajo usa *Compute Capability 1.3*, por lo que todos los datos mencionados en este trabajo corresponden a este tipo de arquitectura.

La figura 2.3 muestra un esquema del hardware de la GPU, la cual está constituida por una serie de *multiprocesadores*, en donde cada uno de éstos cuenta con 8 núcleos. Cada multiprocesador cuenta con 16384 registros y una *shared memory* de 16 KB, la cual es compartida por todos los núcleos del multiprocesador. Esta última es una memoria de muy baja latencia pues se encuentra próxima a los núcleos del multiprocesador. También cada multiprocesador cuenta con un par de memorias caché compartidas por todos los núcleos del multiprocesador, la primera es la *constant cache*, la cual es una caché de datos de 8 KB de la *constant memory*, la que a su vez es una memoria de sólo lectura de 64 KB. La segunda es la *texture cache*, la cual es una caché de 8 KB de la *texture memory*, la que a su vez es una memoria optimizada para localidad espacial 2D (con un tamaño máximo de matriz de $2^{16} \times 2^{15}$ Bytes). La *constant memory* y *texture memory* están alojadas físicamente en la *device memory*, la cual es de mayor tamaño y es compartida por todos los multiprocesadores.

2.1.3.2. Ejecución

Los threads son organizados en *bloques*, y un bloque se ejecuta completamente en un único multiprocesador. Debido a esto todos los threads de un bloque comparten la misma *shared memory*.

Un *kernel* puede ser ejecutado con un máximo de 512 threads por bloque, y 65535^2 bloques. Los threads de bloques distintos no pueden ser sincronizados, sólo lo pueden hacer los que pertenecen al mismo bloque (con la función *__syncthreads()*).

Para soportar la gran cantidad de threads en ejecución se emplea la arquitectura SIMT (Single-Instruction, Multiple-Thread). La unidad de ejecución no es un thread, sino un *warp*, que es un conjunto de 32 threads (y un *half-warp* son los primeros o segundos 16 threads). La forma en que un bloque es dividido en *warps* es siempre la misma, los primeros 32 threads representan el primer *warp*, los 32 threads consecutivos (según el identificador del thread) representa el segundo *warp* y así sucesivamente.

Algunos datos relevantes con respecto al poder de cómputo de la GPU, son:

- El throughput (por multiprocesador) de las operaciones de suma y multiplicación en punto flotante con precisión simple es de 8 operaciones por ciclo de reloj, el de la división es de 1.6 y el de la raíz cuadrada es de 1 operación por ciclo de reloj.

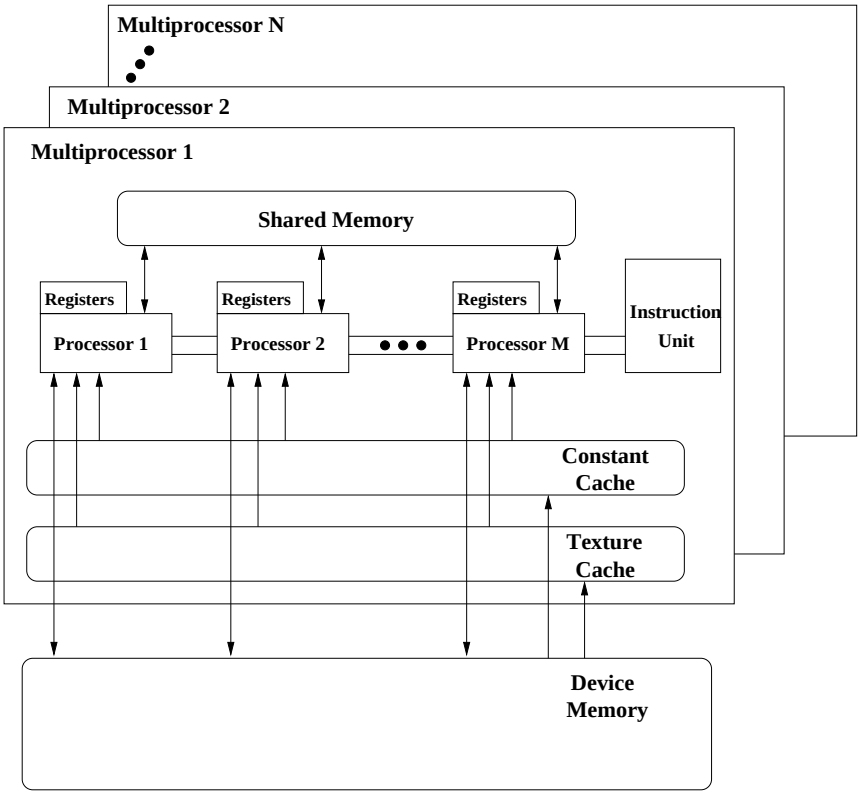


Figura 2.3: Arquitectura de hardware en una GPU.

- El throughput (por multiprocesador) de las operaciones de suma sobre enteros es de 8 operaciones por ciclo de reloj, y de 2 para la multiplicación.
- Una lectura o escritura a *device memory* requiere de 400 a 600 ciclos de reloj. Pero mucha de esta latencia se puede ocultar al existir otras instrucciones aritméticas que se pueden ejecutar mientras se espera por el resultado de una lectura.

Un *warp* ejecuta una instrucción por vez, por lo que el mayor rendimiento se alcanza cuando todos los threads del *warp* ejecutan la misma instrucción. Debido a esto, las instrucciones de flujo que implican una condición pueden disminuir significativamente el rendimiento del programa, pues al haber dos (o más) caminos de ejecución, éstos son serializados, aumentando el número de instrucciones y forzando la ejecución del *warp* sólo con los threads que deben ejecutar la instrucción (deshabilitando el resto).

La figura 2.4 muestra un código que suma 2 vectores en la GPU. Para esto primero se asigna memoria a los arreglos en *device memory*, luego se copian los valores necesarios a estas variables, y posteriormente se invoca al *kernel* con 1 bloque (dado por la variable *N_BLOQUES*) y 200 threads por bloque (dado por la variable N). Cada thread obtiene su identificador (variable *tid*) usando variables predefinidas por la GPU, ID_{Thread} (que retorna el ID del thread en el bloque), T_{Block} (que retorna la cantidad de threads por cada bloque) y ID_{Block} (que retorna el identificador del bloque). Cada thread realiza la suma del elemento *tid*-ésimo de ambos arreglos (*dev_A* y *dev_B*). Y ya una vez terminado el *kernel* se copia el arreglo resultado a la memoria de CPU para poder imprimir los resultados, pues no se puede invocar una función para imprimir dentro del kernel en la GPU utilizada.

Los threads de un *warp* siempre están sincronizados, es decir, ningún thread se puede adelantar a otro que pertenezca al mismo *warp* en la ejecución de instrucciones. Un ejemplo de esto lo muestra la figura 2.5, donde en el código 2.5(a) se deben realizar instrucciones de sincronización para garantizar el correcto valor del arreglo *result*, mientras que el código 2.5(b) es realizado solamente por los threads de un *warp*, y debido a esto no es necesario utilizar sincronización.

2.1.3.3. Reducción de lecturas a memoria

Los accesos a memoria de un *half-warp* a *device memory* son fusionados en una sola transacción de memoria si las direcciones accedidas caben en el mismo segmento de memoria de tamaño:

- 32 bytes si todos los threads acceden palabras de largo 1 byte.
- 64 bytes si todos los threads acceden palabras de largo 2 bytes.
- 128 bytes si todos los threads acceden palabras de largo 4 u 8 bytes.

Por lo tanto, si los threads de un *half-warp* acceden a n diferentes segmentos, entonces se ejecutarán n transacciones de memoria.

```
#include <stdio.h>
#include <cuda.h>
#define N 200
#define N_BLOQUES 1

/* Se define un kernel llamado 'function' */
__global__ void function(float *A, float *B, float *result)
{
    /* Se obtiene el identificador del thread */
    int tid = ID_Thread + ( T_Block *ID_Block );
    /* Cada thread realiza la suma de un elemento distinto */
    result [tid] = A[tid] + B[tid];
    return;
}

main()
{
    float host_A[N], host_B[N], host_result[N];
    float *dev_A, *dev_B, *dev_result;

   /* Asignamos memoria en device memory */
    cudaMalloc((void **)&dev_A, sizeof(float)*N);
    cudaMalloc((void **)&dev_B, sizeof(float)*N);
    cudaMalloc((void **)&dev_result, sizeof(float)*N);

    inicializar_valores(host_A, host_B, host_result);
    /* Se copian los valores a las variables alojadas en device memory */
    cudaMemcpy(dev_A, host_A, sizeof(float)*N, cudaMemcpyHostToDevice);
    cudaMemcpy(dev_B, host_B, sizeof(float)*N, cudaMemcpyHostToDevice);

    /*Invocamos al kernel con 'N_BLOQUES' bloques y 'N' threads por bloque
       con las variables alojadas en device memory */
    function<<<N_BLOQUES, N>>>(dev_A, dev_B, dev_result);

    /* Se copian los resultados a memoria de la CPU */
    cudaMemcpy(host_result, dev_result, sizeof(float)*N, cudaMemcpyDeviceToHost);

    imprimir_resultados(host_result);
    cudaThreadExit();
    return 0;
}
```

Figura 2.4: Ejemplo de suma de vectores con CUDA.

```
int tid = ID_Thread
...
int ref1 = myArray[tid] * 1;
__syncthreads();
myArray[tid + 1] = 2;
__syncthreads();
int ref2 = myArray[tid] * 1;
result[tid] = ref1 * ref2;
...
```

(a)

```
int tid = ID_Thread
...
if (tid < warpSize) {
int ref1 = myArray[tid] * 1;
myArray[tid + 1] = 2;
int ref2 = myArray[tid] * 1;
result[tid] = ref1 * ref2;
}
...
```

(b)

Figura 2.5: Ejemplos para ilustrar la sincronización de los threads de un *warp*.

Si es posible reducir el tamaño de la transacción, entonces se lee sólo la mitad de ésta, más específicamente se realiza lo siguiente:

- Se establece el tamaño de la transacción igual al tamaño del segmento a leer.
- Si el tamaño de la transacción es de 128 bytes y sólo se utilizan los datos almacenados en la mitad (superior o inferior), se reduce el tamaño de la transacción a 64 bytes.
- Si el tamaño de la transacción es de 64 bytes y sólo se utilizan los datos almacenados en la mitad (superior o inferior), se reduce el tamaño de la transacción a 32 bytes.

2.1.3.4. Funciones atómicas

Existe un grupo de funciones atómicas que realizan operaciones de tipo *read-modify-write* sobre palabras de 32 o 64 bits residentes en *global memory* o *shared memory*. Estas funciones garantizan que serán realizadas completamente sin interferencia de los demás threads.

Cuando un thread ejecuta una función atómica, ésta bloquea el acceso a la dirección de memoria donde reside la palabra involucrada hasta que se complete la operación. Las funciones atómicas se dividen en funciones aritméticas y funciones a nivel de bits, a continuación se muestran algunas de las funciones aritméticas relevantes para el presente trabajo:

atomicAdd `int atomicAdd(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor (*old*+*val*). El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

atomicSub `int atomicSub(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor (*old*-*val*). El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

atomicMin `int atomicMin(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor mínimo entre *old* y *val*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

atomicMax `int atomicMax(int *address, int val)`

Lee la palabra *old* localizada en la dirección *address* y escribe en la misma dirección el valor máximo entre *old* y *val*. El valor retornado es *old*. Estas tres operaciones se realizan en una transacción atómica.

2.2. Espacios métricos

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*.

2.2.1. Búsquedas por similitud

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 [13]:

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

Los k Vecinos más Cercanos $NN_k(q)$**:** 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 2.6 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 derecha una consulta por los 5-vecinos más cercanos a q. En este último caso, también se gráfica el rango necesario para encerrar los 5 puntos. Entonces,

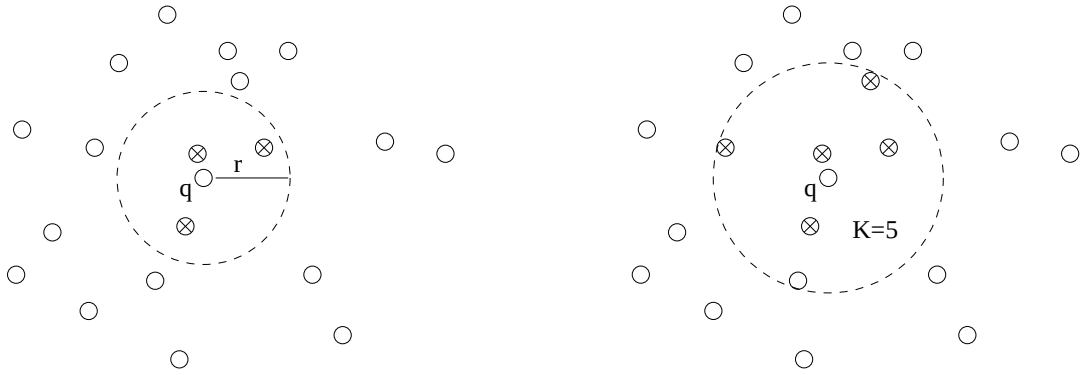


Figura 2.6: Ejemplos de consultas, por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos en $\mathbb{R}^2$.

se observa que dada una consulta q y una cantidad k (en este ejemplo 5), es posible que existan distintas respuestas.

En el Capítulo 3 sólo se abarcan consultas de tipo rango, debido a que éstas son la base para resolver consultas de tipo kNN. En [13] se ilustran varios métodos para resolver consultas de tipo kNN, los cuales son:

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.

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 rango 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.

2.2.2. Indexación

Una manera trivial de responder a consultas por similitud es examinar exhaustivamente la base de datos Y, lo que toma tiempo $O(n)$ para una base de datos con n objetos. Considerando que la función de distancia es *computacionalmente costosa* de calcular, no es práctico resolver una consulta de esta manera.

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.

Un *algoritmo de indexación* es un proceso que construye una estructura de datos denominada *índice*, el que permite realizar búsquedas por similitud evitando, en lo posible, tener que revisar

toda la base de datos, ahorrándose evaluaciones de distancia al momento de realizar las consultas. Todos los algoritmos de búsquedas en espacios métricos utilizan la desigualdad triangular para descartar objetos, esto con el fin de evitar calcular la distancia real con el objeto de consulta. Esto último da una idea de la información que debería contener la estructura de datos.

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. Con la finalidad de evitar dichas evaluaciones de distancia, los métodos de búsqueda en espacios métricos se basan principalmente en dividir el espacio empleando la distancia a uno o más objetos seleccionados.

Todos los algoritmos de indexación particionan la base de datos Y en subconjuntos. El índice permitirá determinar una lista de subconjuntos Y_i candidatos potenciales a contener objetos relevantes a la consulta. A cada subconjunto Y_i podría aplicársele el mismo método de particionamiento, en forma recursiva. Durante la búsqueda, se recorre el índice para obtener los conjuntos relevantes y luego se examina cada uno de ellos (en forma exhaustiva si no se ha continuado recursivamente la indexación). Todas estas estructuras trabajan sobre la base de descartar elementos usando la desigualdad triangular.

Para particionar la base de datos existen dos grandes enfoques que determinan los algoritmos de búsqueda en espacios métricos generales. Estos son: *algoritmos basados en pivotes* y *algoritmos basados en clustering o particiones compactas* [13].

Los algoritmos basados en pivotes realizan una preselección de objetos de la base de datos, estos objetos (o pivotes) sirven para filtrar objetos en una consulta utilizando la desigualdad triangular, sin medir realmente la distancia entre el objeto consulta y los objetos descartados.

- Sea $\{p_1, p_2, ..., p_k\} \in X$ un conjunto de pivotes. Se almacena para cada elemento x de la base de datos Y, su distancia a los k pivotes $(d(x, p_1), ..., d(x, p_k))$. Dada una consulta q, se calcula su distancia a los k pivotes $(d(q, p_1), ..., d(q, p_k))$.

- Si para algún pivote p_i se cumple que $|d(q, p_i) - d(x, p_i)| > r$, entonces por desigualdad triangular conocemos que $d(q, x) > r$, y por lo tanto no es necesario evaluar explícitamente $d(x, q)$. Todos los objetos que no se puedan descartar por esta regla deben ser comparados directamente con la consulta q.

Algunos algoritmos hacen una implementación directa de este concepto, y se diferencian básicamente en su estructura extra para reducir el costo de CPU de encontrar los puntos candidatos (puntos no descartados), pero no en la cantidad de evaluaciones de distancia. Ejemplos de éstos son: *SSS-Index* [4], *SSS-Tree* [5], *AESA* [48], *LAESA* [33], *Spaghettis* y sus variantes [9, 36], *FQT* y sus variantes [2] y *FQA* [10].

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.

Existen dos criterios para delimitar las áreas en las estructuras basadas en clustering, *área de Voronoi* o *hiperplanos* y *radio cobertor* (*covering radius*). El primero divide el espacio usando hiperplanos y determina la partición a la cual pertenece la consulta según a qué zona corresponde. El criterio de radio cobertor divide el espacio en esferas que pueden intersectarse y una consulta puede pertenecer a más de una esfera.

Criterio de partición de Voronoi: Se selecciona un conjunto de puntos y se coloca cada punto restante dentro de la región con centro más cercano. Las áreas se delimitan con hiperplanos y las zonas son análogas a las regiones de Voronoi en espacios vectoriales.

- **Definición (*Diagrama de Voronoi*)**: considérese un conjunto de puntos $\{c_1, c_2, \ldots, c_m\}$ (centros). Se define el diagrama de Voronoi como la subdivisión del plano en m áreas, una por cada c_i. La consulta q pertenece al área c_i si y sólo si la distancia euclidiana $d(q, c_i) \leq d(q, c_j)$ para cada c_j, con $j \neq i$.

  Durante la búsqueda se evalúa $d(q, c_1), ..., d(q, c_m)$, se elige el centro c_i más cercano y se descartan todas las zonas c_j que cumplan con $d(q, c_j) > d(q, c_i) + 2r$, dado que su área de Voronoi no puede intersectar la *bola de consulta* con centro q y radio r. Se entiende por bola de consulta a la bola cerrada con centro q y radio r.

Criterio de Radio Cobertor: El radio cobertor $rc(c_i)$ es la distancia entre el centro c_i y el objeto más alejado dentro de su zona. Entonces, se puede descartar la zona c_i si $d(q, c_i) - r > rc(c_i)$.

Algunas estructuras combinan estas técnicas, como el caso del *GNAT* [3] y *EGNAT* [35][45]. Otras sólo utilizan radio cobertor, como son los *M-trees* [14] y la *Lista de Clusters* [11]. Algunas que utilizan hiperplanos son GHT y sus variantes [44, 39] y los Voronoi-trees [18, 38].

Si bien todos los métodos e índices mencionados anteriormente son eficientes en el descarte de elementos, reducción de las evaluacioens de distancia y reducción del tiempo de ejecución, 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. 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.

La dificultad de la búsqueda en espacios de alta dimensión radica en la baja cantidad de objetos que se pueden descartar. En [13] y [12] 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. De esta forma, un espacio de alta dimensión intrínseca posee su histograma de distancias concentrado, es decir, la *media* es alta y la *varianza* pequeña, lo que indicaría que los objetos están todos más o menos a la misma distancia entre sí. Por lo tanto, pueden existir espacios cuya dimensión de representación es alta, pero su dimensión intrínseca es baja.

2.2.3. Indices métricos

Son varios los índices métricos existentes en la literatura. En el presente trabajo se seleccionaron 5, 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* ([35][45]), *M-tree* ([14]), *SSS-Index* ([4]), *SSS-Tree* ([5]) y la *Lista de Clusters* (*LC*) ([11]). A continuación detallaremos brevemente cada índice.

2.2.3.1. EGNAT

El *EGNAT* ([35][45]) es una estructura de tipo árbol optimizada para memoria secundaria, incluye un método de eliminación denominado *planos fantasmas* y posee 2 tipos de nodos: (a)*nodos gnat* (nodos internos) y (b) *nodos bucket* (nodos hojas). Los elementos no se almacenan sólo en las hojas, sino que también en los nodos internos.

Para la construcción del *EGNAT* se seleccionan k puntos claves denominados *centros* o *splits* para particionar el espacio $\{p_1, p_2, \ldots, p_k\}$, y cada punto restante es asignado al centro más cercano, definiéndose así el subárbol de influencia de cada centro. Cada subárbol es particionado recursivamente.

El algoritmo básico de la construcción del *EGNAT* es como sigue:

1. Se seleccionan k puntos (*centros*), $p_1, \ldots, p_k$ de la base de datos que se va a indexar.
2. Se asocia cada punto restante del conjunto de datos al centro más cercano a él. El conjunto de puntos asociados al split p_i se denota como D_{p_i}.
3. Para cada par de centros (p_i, p_j), se calcula el rango $range\left(p_i, D_{p_j}\right) = \left[min\_d\left\{\left(p_i, D_{p_j}\right)\right\}, max\_d\left\{\left(p_i, D_{p_j}\right)\right\}\right]$, la mínima y máxima distancia $d\left(p_i, x\right)$ donde $x \in D_{p_j} \cup \{p_j\}$.
4. El árbol se construye recursivamente para cada subconjunto D_{p_i}.

Cada conjunto D_{p_i} va a representar un subárbol cuya raíz es p_i, o lo que es lo mismo, cada D_{p_i} va a corresponder a la región de Voronoi cuyo centro es p_i. En la figura 2.7 se muestra un ejemplo de construcción del primer nivel de un *EGNAT* con *k=4*, también se muestra la tabla de rangos que debe ser almacenada para cada centro p_i. En este ejemplo se insertaron los puntos en orden al valor numérico que tienen. Si se insertase un nuevo objeto p_{16} y quedara en el plano cuya raíz es p_4, entonces, esto crearía un nuevo nodo y haría aumentar el nivel del árbol como se muestra en la figura 2.8. Nótese que p_4 ya no es parte de estos subplanos.

El algoritmo 1 representa la búsqueda por rango sobre el *EGNAT*. Esta búsqueda se realiza recursivamente como sigue:

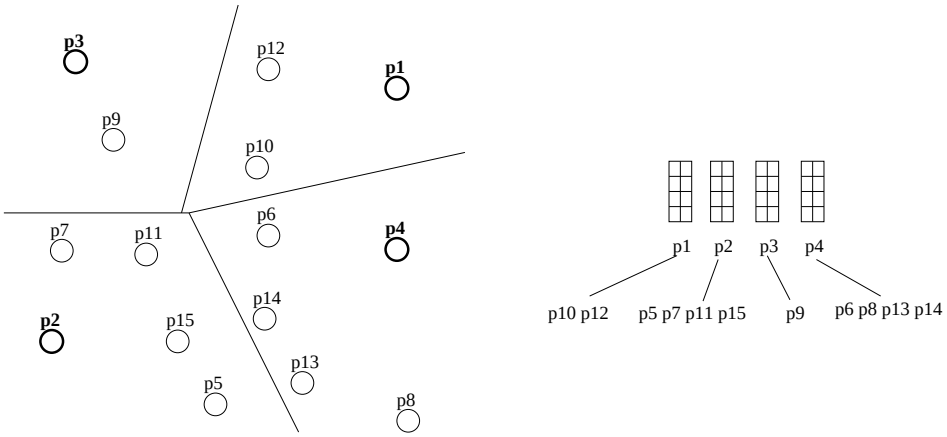


Figura 2.7: *EGNAT*: Construcción del árbol (aridad 4).

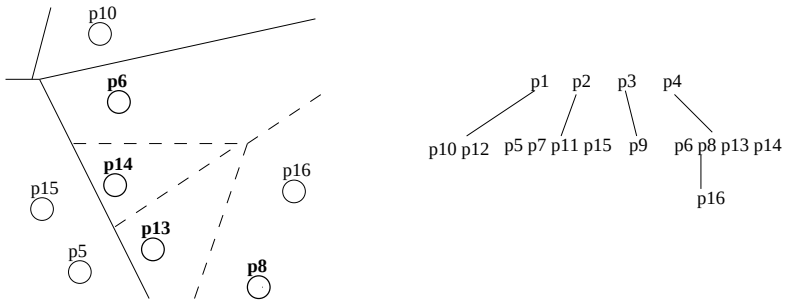


Figura 2.8: *EGNAT*: Inserción de un nuevo objeto.

Algoritmo 1 *EGNAT*: búsqueda por rango r para la consulta q.

busquedarango(Nodo P, Consulta q, Rango r)

```
1: {Sea R el conjunto resultado}
2: R ← ∅
3: d ← dist(p_0, q)
4: if d <= r then
5:    se reporta p_0
6: end if
7: range(p_0, q) ← [d − r, d + r]
8: for all x ∈ P do
9:    if range(p_0, q) ∩ range(p_0, D_{p_x}) ≠ ∅ then
10:      se agrega x a R
11:      if dist(x, q) <= r then
12:         se reporta x
13:      end if
14:   end if
15: end for
16: for all p_i ∈ R do
17:   busquedarango(D_{p_i},q,r)
18: end for
```

1. Se supone que se desea buscar todos los puntos con distancia $d \leq r$ a la consulta q. Sea P la representación del conjunto de centros del nodo actual (inicialmente la raíz del *EGNAT*) el que posiblemente contiene un vecino cercano a q. Inicialmente P contiene todos los puntos centros del nodo actual.

2. Se toma un punto p en P, se calcula la distancia *d(q,p)*. Si $d(q,p) \leq r$, se agrega p al resultado.

3. $\forall\, x \in P$, si $[d(q,p) - r, d(q,p) + r] \cap range(p, D_x)$ es vacío, entonces se elimina x de P.

4. Se repiten los pasos 2 y 3 hasta procesar todos los puntos en P.

5. Para todos los puntos $p_i \in P$, se procede recursivamente sobre D_{p_i}.

La razón que permite descartar subárboles durante la búsqueda en el paso 3 del algoritmo es lo siguiente:

Sea un punto $y \in D_x$. Si $d(y,p) < d(q,p) - r$, entonces, por desigualdad triangular, se tiene que $d(q,y) + d(y,p) \geq d(q,p)$, por lo que se deduce que $d(q,y) > r$. Análogamente, si $d(y,p) > d(q,p) + r$, se puede usar desigualdad triangular $d(y,q) + d(q,p) \geq d(y,p)$, para deducir $d(q,y) > r$ (figura 2.9).

2.2.3.2. M-Tree

La primera estructura de datos métrica dinámica que apareció en la literatura fue el *M-tree* [14]. Es un árbol balanceado y paginado, es decir, usa páginas de tamaño fijo con cantidad de nodos u objetos variables. Es una estructura similar a un B-Tree y su crecimiento es del tipo bottom-up.

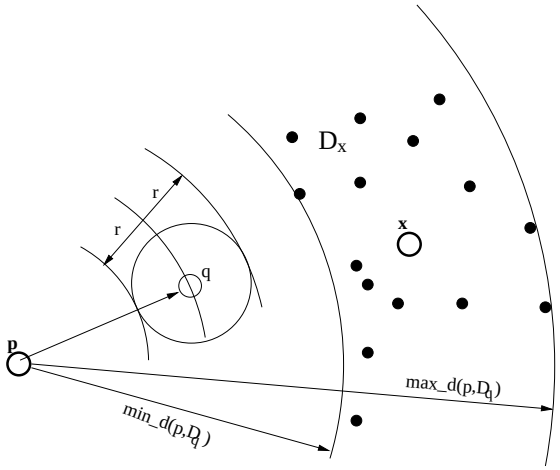


Figura 2.9: Descarte de subárboles usando rangos. Se descarta D_x ya que $d(q,p) + r < min\_d(p, D_x)$.

La implementación del M-Tree se encuentra disponible en la Web, implementado con un paquete llamado *GiST* (*Generalized Search Tree*) [21], el que presenta innumerables problemas de compilación y ejecución. Debido a esto se implementó una versión del M-Tree en lenguaje C, el que está disponible en *http://kataix.umag.cl/∼ribarrie/Programas.html*.

El M-Tree se construye eligiendo un conjunto de centros, los cuales almacenarán su radio cobertor. El máximo de elementos en el nodo es el necesario para abarcar una página de disco (usualmente 4096 bytes). Cada nuevo objeto se inserta en el mejor subárbol, el que se define como aquél en donde crece menos su radio cobertor. En el caso de desborde de un nodo (overflow) se usaron las alternativas *mM_RAD* y *Generalized Hyperplane*, las cuales presentaron los mejores resultados según [14]. *mM_RAD* establece qué par de elementos (O_1 y O_2) serán promovidos al nodo padre, para esto se buscan la mejor combinación posible (lo que implica costo $\mathcal{O}\left(n^2\right)$), tales que se minimice el máximo radio producido por la promoción de los elementos. Una vez que ya se produjo la promoción de O_1 y O_2, *Generalized Hyperplane* establece que por cada elemento restante se calcula $d_1 = d(O_1, O_j)$ y $d_2 = d(O_2, O_j)$, luego si $d_1 < d_2$ el elemento es asignado al nodo apuntado por O_1, sino al apuntado por O_2.

Los nodos internos del M-Tree son distintos a los nodos hojas. Los primeros mantienen la ruta a los objetos y son llamados *objetos ruteadores* (*routing objects*) y los nodos hojas almacenan los objetos de la base de datos.

La información general almacenada para un objeto ruteador es:

- O_r: objeto ruteador (o valor característico).
- $T(O_r)$: subárbol de O_r.
- $ptr(T(O_r))$: puntero a la raíz de $T(O_r)$.
- $rc(O_r)$: radio cobertor de O_r.

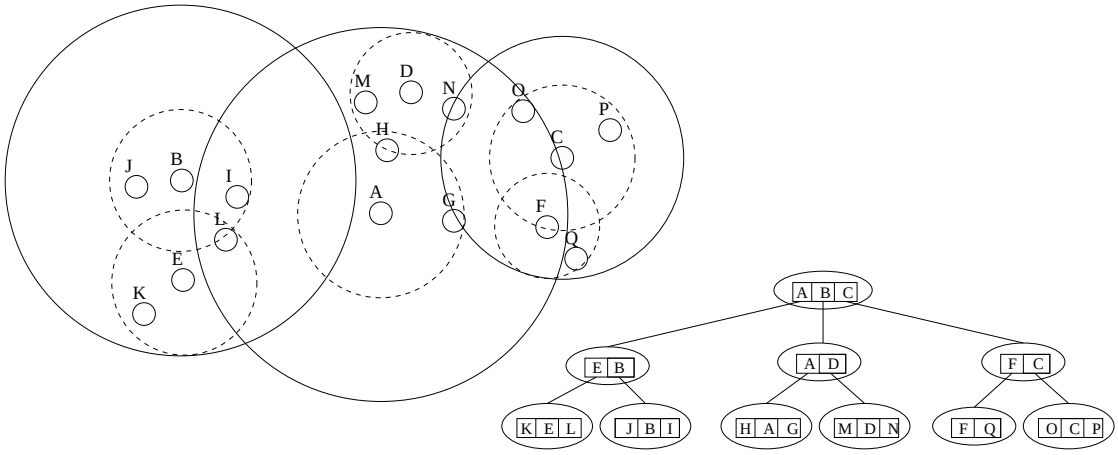


Figura 2.10: *M-tree*: Construcción de la estructura.

- $d(O_r, P(O_r))$: distancia de O_r a su padre.

Para los nodos hojas se almacena:

- O_j: objeto almacenado (o su valor característico).
- $Oid(O_j)$: identificador del objeto O_j.
- $d(O_j, P(O_j))$: distancia de O_j a su padre.

En la figura 2.10 se ilustra una representación gráfica de la estructura de un M-tree y los radios cobertores de cada objeto ruteador.

El algoritmo 2 muestra la búsqueda por rango para el M-Tree. Se comienza por el nodo raíz y recursivamente se visitan todos los hijos que no pueden ser descartados por desigualdad triangular.

2.2.3.3. SSS-Index

El SSS-Index [4] establece una tabla de distancias entre pivotes y todos los elementos de la base de datos. Los pivotes son seleccionados según el algoritmo 3. 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 de distancias entre cada pivote y los elementos restantes. El valor óptimo de α observado en [4] fue para valores alrededor de $\alpha = 0{,}4$.

El algoritmo 4 muestra la búsqueda por rango en el SSS-Index. Para esto se calculan 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 BD, si esto no es posible entonces se realiza una evaluación de distancia entre la consulta y el elemento no descartado de la BD.

Algoritmo 2 *M-Tree*: búsqueda por rango r para la consulta q.

busquedarango(Nodo N, Consulta q, Rango r)

```
1: {Sea O_p el objeto padre del nodo N}
2: if N es un nodo interno then
3:    for all O_r ∈ N do
4:       if |d(O_p, q) − d(O_r, O_p)| <= r + rc(O_r) then
5:          Calcular d(O_r, q);
6:          if d(O_r, q) <= r + rc(O_r) then
7:             busquedarango(ptr(T(O_r)), q, r);
8:          end if
9:       end if
10:   end for
11: else
12:   for all O_j ∈ N do
13:      if |d(O_p, q) − d(O_j, O_p)| <= r then
14:         Calcular d(O_j, q);
15:         if d(O_j, q) <= r then
16:            agregar oid(O_j) al resultado;
17:         end if
18:      end if
19:   end for
20: end if
```

Algoritmo 3 *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
```

Algoritmo 4 *SSS-Index*: algoritmo de búsqueda.

busquedarango(Consulta q, Rango r)

```
1: {Sea BD la base de datos}
2: {Sea DIST la tabla de distancias}
3: {Sea PIVOTES el conjunto de Pivotes}
4: i = 0;
5: for all x_i ∈ PIVOTES do
6:    arrD[i++] = d(x_i, q);
7: end for
8: for i = 0; i < size(BD); i++ do
9:    descartado = false
10:   for j = 0; j < size(PIVOTES); j++ do
11:      if arrD[j] < DIST[i][j] − r  ||  arrD[j] > DIST[i][j] + r then
12:         descartado = true;
13:         break;
14:      end if
15:   end for
16:   if !descartado then
17:      if d(BD[i], q) <= r then
18:         agregar BD[i] a los Resultados.
19:      end if
20:   end if
21: end for
```

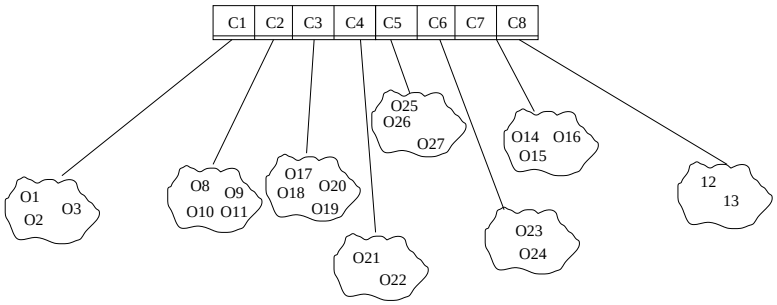


Figura 2.11: *SSS-Tree*: Estructura después del primer paso de construcción.

2.2.3.4. SSS-Tree

El *SSS-Tree* [5] es una estructura de tipo árbol, cuyo método de indexación se basa en clustering. Los centros de cluster de cada nodo interno del árbol se seleccionan aplicando la técnica SSS (algoritmo 3). En cada nodo se particiona el subespacio que le corresponde en varias particiones o clusters, y para cada uno de ellos se almacena el radio cobertor, el que se utiliza durante la operación de búsqueda.

El proceso de construcción comienza con todos los elementos en un *nodo bucket*. Sea D el conjunto de elementos de dicho nodo, M la distancia máxima entre 2 elementos de D y sea α una constante real, entonces se aplica SSS sobre D. Es decir, el primer elemento de D es seleccionado como pivote, y luego por cada elemento restante $x_i \in D$ si se cumple que $\forall p \in PIVOTES, d(x_i, p) >= M\alpha$, entonces x_i es un nuevo pivote, de lo contrario es asignado al cluster del pivote más cercano. Para esto último, cada pivote posee un puntero a un cluster de elementos. La figura 2.11 ilustra un ejemplo del índice de esta estructura después de haber elegido todos los pivotes correspondientes al *nodo bucket* inicial. Luego el proceso se ejecuta recursivamente sobre cada cluster. La recursión termina cuando la cantidad de elementos del cluster es menor o igual a una constante K.

Los nodos internos y hojas tienen la misma composición. La figura 2.12 muestra los campos de un nodo para esta estructura.

El algoritmo 5 muestra la búsqueda por rango en el *SSS-Tree*, en donde el radio cobertor es usado para realizar descarte de subárboles por desigualdad triangular.

2.2.3.5. Lista de Clusters (LC)

La *Lista de Clusters* ([11]) 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 la construcción de la LC, 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:

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

```
struct Nodo{
  unsigned int ncentros;//Cant. de centros del nodo
  struct Centro *centro;//arreglo contenedor de centros
};
struct Centro{
  TipoElemento info;//Elemento de la B.D.
  int tipo;//El tipo es Nodo Interno o Nodo Hoja
  TipoDist radiocovertor;//Radio Covertor
  struct Nodo *hijo;//Puntero a un nodo interno
};
```

Figura 2.12: Campos de un nodo en el *SSS-Tree*.

Algoritmo 5 *SSS-Tree*: búsqueda por rango r para la consulta q.

busquedarango(Nodo N, Consulta q, Rango r)

```
1: {Sea x un centro del nodo N}
2: for i = 0; i < N.ncentros; i++ do
3:     dist = d(x.info, q)
4:     if dist <= r then
5:         Agregar x.info a los Resultados.
6:     end if
7:     if x.hijo != NULL then
8:         if dist - r <= x.radiocovertor then
9:             busquedarango(x.hijo, q, r);
10:        end if
11:    end if
12: end for
```

Algoritmo 6 *Lista de Cluster*: algoritmo de construcción.

Construccion(D)

1: {El operador ":" es el constructor de lista.}
2: {$kNN(c)$ entrega los k elementos más cercanos a c.}
3: **if** $D==\emptyset$ **then**
4: Retornar lista vacía.
5: **end if**
6: Seleccionar un centro $c \in D$
7: $I \leftarrow kNN(c), i \in \{D - \{c\}\}\ \forall i \in I$
8: $r_c \leftarrow max(d(x, c)), x \in I$
9: $E \leftarrow D - I$
10: Retornar (c, r_c, I):Construccion(E)

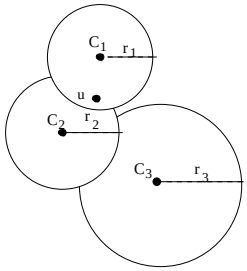


Figura 2.13: *Lista de Cluster*: Zonas abarcadas por 3 centros según este orden: c_1, c_2, c_3.

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.

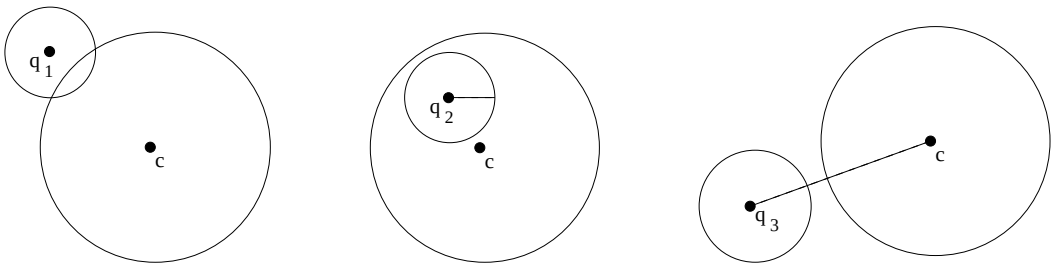
El algoritmo de construcción se muestra en el algoritmo 6, el que retorna una lista (c_i, r_i, I_i) (centro, radio, bucket), como se muestra en la figura 2.13. Cabe notar que el primer centro elegido tiene preferencia sobre el resto cuando hay solapamiento, de este modo los elementos que pertenecen a la bola del primer centro son almacenados sólo en su cluster I, a pesar de que haya intersección con los cluster siguientes (figura 2.13). 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).

El algoritmo de búsqueda se muestra en el algoritmo 7 y la figura 2.14 muestra 3 casos posibles de búsqueda dado un cluster (c, r_c), en donde para la consulta $q1$ es necesario buscar sobre el cluster y continuar la búsqueda en el resto de la lista de clusters, en cambio para $q2$ es posible podar la búsqueda debido a que la consulta está completamente contenida dentro del cluster, y para $q3$ es posible evitar la búsqueda sobre el cluster por la propiedad de *desigualdad triangular*.

Algoritmo 7 *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
```



(a) (b) (c)

Figura 2.14: *Lista de Cluster*: Tres casos de búsqueda.

2.3. Trabajo relacionado

En la literatura el trabajo relacionado previo relevante para esta tesis incluye publicaciones de paralelismo aplicado a estructuras métricas sobre una plataforma de memoria distribuida, y también de soluciones al problema de los k vecinos más cercanos (kNN) sobre GPU.

2.3.1. Memoria distribuida

Los trabajos de paralelismo aplicados a estructuras métricas sobre una plataforma de memoria distribuida (usando un cluster de procesadores), son [15, 16, 26, 27, 28, 29, 46].

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). Un ejemplo de estrategia local es el siguiente: Se tiene un cluster con P procesadores y un diccionario de N palabras en español. Dicho diccionario es distribuido entre los procesadores de forma que cada uno posea N/P palabras. Luego cada uno crea su propio índice con las palabras que posee, de esta forma cada procesador tendrá un índice con una porción de la base de datos completa.

Lo anterior implica que la consulta debe hacerse llegar a todos los procesadores. El resultado final es la unión de los resultados entregados por cada procesador.

Las estrategias locales poseen la ventaja que para procesar una consulta no necesitan ocupar recursos de comunicación con el resto, sin embargo posee la desventaja que todos los P procesadores deben procesar la misma consulta.

En cambio las estrategias globales poseen sólo un índice común a todos los procesadores, y cada consulta se resuelve usando R procesadores, donde $1 <= R <= P$. Un ejemplo es el siguiente: Se tiene un cluster con P procesadores y un diccionario de N palabras en español. El procesador 0 crea un índice con las N palabras, supongamos que el índice corresponde a una estructura de tipo árbol. La raíz del árbol es replicada en todos los procesadores. Luego, las ramas del árbol son distribuidas de forma circular entre los procesadores, de esta forma cada procesador tiene una porción del índice original. También cada procesador conoce a qué procesador pertenece cada rama. Luego, para realizar una búsqueda, la consulta debe hacerse llegar a un procesador (cualquiera). Luego éste determina por qué ramas debería descender la consulta y a qué procesadores pertenecen dichas ramas, para enviar la consulta en forma de mensaje a los procesadores correspondientes.

2.3.2. GPU

El trabajo relacionado sobre GPU existente sólo aborda la solución a las consultas de tipo kNN, ellos son [24], [19], y [6], pero éste último sólo aborda el caso para $k = 1$. Todos estos trabajos abordan el caso en que los elementos de la base de datos poseen una gran dimensión (con una gran

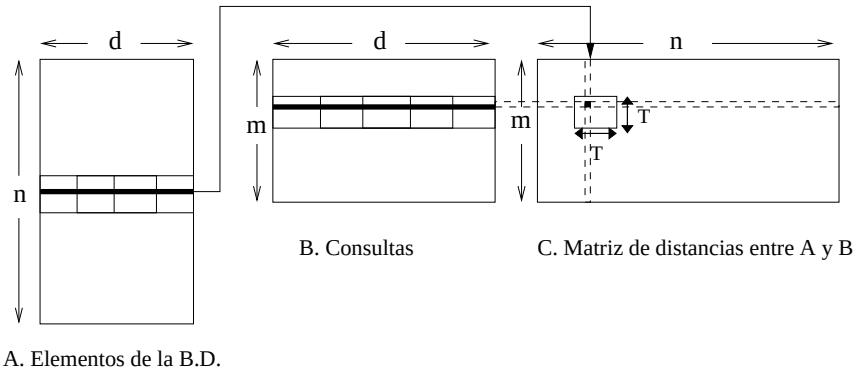


Figura 2.15: Particionamiento de las matrices de datos y consultas en submatrices.

dimensión intrínseca), lo cual implica poder descartar muy pocos elementos (usando desigualdad triangular), y en donde la búsqueda secuencial es competitiva frente a usar indexación.

En [24] se propone dividir la base de datos de elementos (matriz A de dimensión $n \times d$) y la de consultas (matriz B de dimensión $m \times d$) en submatrices de tamaño $T \times T$ (ver figura 2.15). La matriz resultante C es una matriz donde cada elemento representa la distancia entre un elemento de A y uno de B. Cada submatriz de C es resuelta por un bloque, para lo cual cada bloque carga a *shared memory* cada submatriz de A y B para poder escribir las distancias resultantes en la submatriz correspondiente. Esto implica que cada bloque hará $\left(\frac{d}{T}\right)^2 T^2 = d^2$ lecturas a *device memory*, donde T es una constante limitada por el tamaño de *shared memory*, y el número de bloques necesarios es $(m/T) \times (n/T)$. Luego, teniendo la matriz de distancias resultante, ésta se ordena usando el *CUDA-based Radix Sort* ([43]) para posteriormente seleccionar los K primeros elementos como resultado final.

En [19] se propone que cada thread resuelva la distancia de un elemento de la BD contra la consulta, para luego ordenar el arreglo resultante con el *insertion sort* implementado para CUDA en el mismo trabajo.

Capítulo 3

Estrategias de distribución y búsqueda sobre procesadores Multi-core

3.1. Modelo de búsqueda

El modelo de búsqueda bajo el que está basado este trabajo está representado por la figura 3.1, 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 cual 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 información 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. Todos los threads tienen acceso a todas las Q_{SM}, esto es para permitir el paso de mensajes (como se explica en la Sección 3.2.2). Debido a la variación que puede sufrir el tamaño de la cola Q_{PR}, sus elementos son creados dinámicamente.

3.2. Descripción de la búsqueda

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

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.

índice 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.

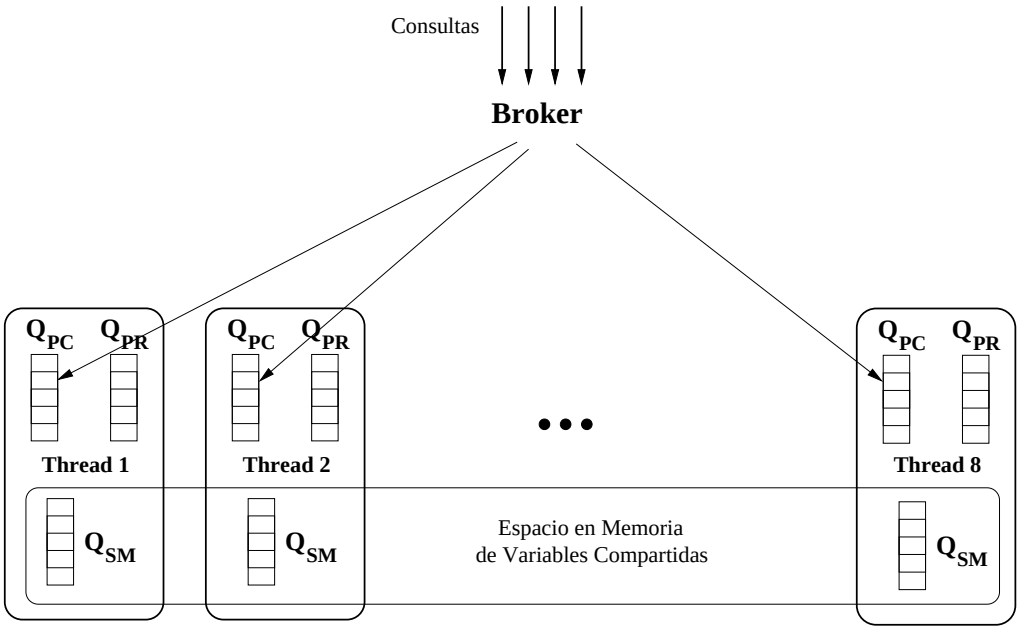


Figura 3.1: Modelo de Búsqueda usado en los experimentos.

consulta La consulta misma.

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 generan 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.

3.2.1. Estrategia Local

En esta estrategia los threads obtienen una consulta de la Q_{PC}, de donde se obtiene el requerimiento inicial, y éste se agrega a la 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 thread es capaz de resolver sus consultas de forma incomunicada del resto de los threads, evitando instrucciones de sincronización y de paso de mensajes.

Algoritmo 8 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
```

El algoritmo 8 describe el proceso de búsqueda sobre esta estrategia.

3.2.2. Estrategia Bulk-Circular

Esta estrategia procesa consultas utilizando el modelo BSP (Bulk Synchronous Parallel) [47], el que define un comportamiento sincrónico para los threads involucrados. Para implementar el modelo BSP, y poder crear una secuencia de pasos (*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} (esta última denominada *Cola Secundaria de Mensajes*). Cada thread almacena en su propia Q_{SM} los *mensajes* que son destinados a los demás threads. Q_{SM} es una cola compartida globalmente (mediante la opción *shared()* de la directiva *#pragma omp parallel* de OpenMP), de esta forma todos los threads pueden ver las Q_{SM} del resto. Un mensaje es un requerimiento más el identificador del thread destino.

Lo anterior implica que los requerimientos generados no se almacenan solamente en la Q_{PR} como en la estrategia Local, sino que los requerimientos destinados a otros threads se almacenan en forma de mensaje en la 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 9 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 la 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 implica que no hay problemas de conflictos entre lectores y escritores concurrentes.

Algoritmo 9 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
```

Tabla 3.1: 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

3.2.3. Estrategia Bulk-Critical

Esta estrategia es muy similar a la Bulk-Circular, pero usa *regiones críticas* (Sección 2.1.1) 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.

El hecho de usar 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.

El algoritmo 10 muestra la implementación de esta estrategia implementando las regiones críticas con la directiva *#pragma omp critical*.

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

3.3. Resultados experimentales

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 3.1.

Los experimentos se realizaron con 2 bases de datos:

- ***Spanish*** **:** Diccionario español con 51589 palabras. Se usó la *distancia de edición* (o *distancia de Levenshtein*) [25] con radio 1, 2 y 3, pues éstos son radios usados en trabajos previos [35, 28, 34]. Esta distancia entrega la cantidad mínima de inserciones, eliminaciones

Algoritmo 10 Búsqueda en la estrategia Bulk-Critical.

{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:            #pragma omp critical (exchange){
14:            Q_SM[tid].insert(task);
15:            }
16:         end if
17:      end for
18:   end while
19:   #pragma omp critical (exchange){
20:   for i = 0; i < P; i++ do
21:      if i != tid then
22:         for j = 0; j < Q_SM[i].size(); j++ do
23:            if Q_SM[i].getTask(j).targetThread == tid then
24:               Q_PR.insert(Q_SM[i].getTask(j));
25:               erase(Q_SM[i].getTask(j));
26:            end if
27:         end for
28:      end if
29:   end for
30:   }
31: end while
```

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.

- ***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 [11, 35, 34]. El 80 % de la BD se usó para la construcción del índice y el 20 % restante para el 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 3.2 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*. La *Bulk-Critical* presenta tiempos de ejecución muy altos, debido a que las regiones críticas son muy accedidas, y como es sabido ([8]), 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.

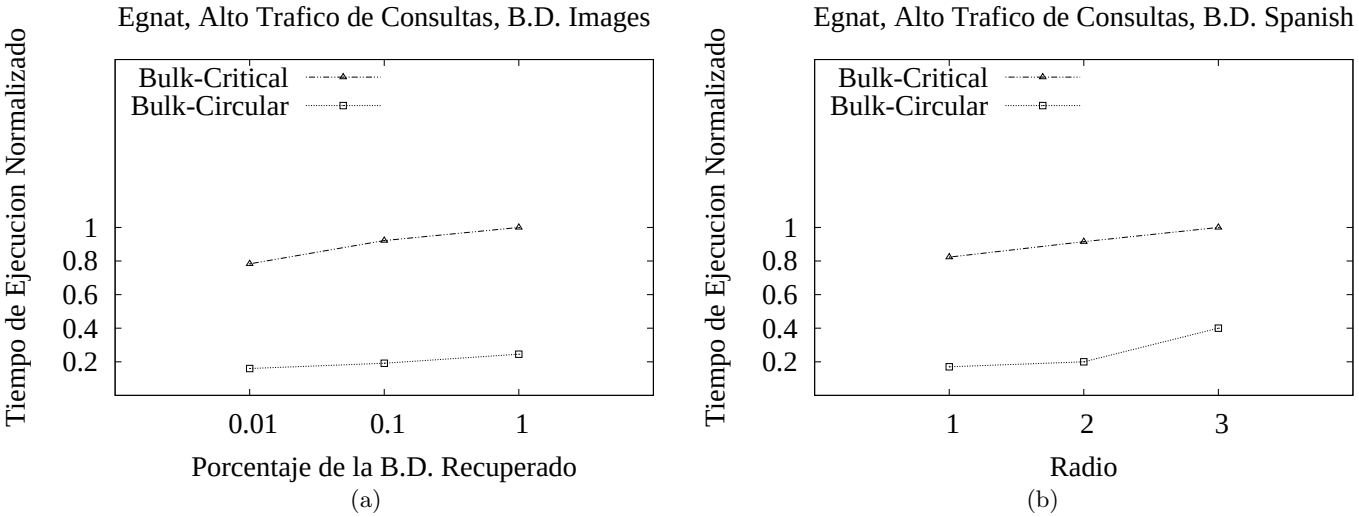


Figura 3.2: Tiempo de Ejecución para la estrategias *Bulk-Circular* y *Bulk-Critical*.

La figura 3.3 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 3.4 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 3.5 muestra un ejemplo de dos distribuciones de consultas con baja frecuencia. 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 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 3.5) 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 3.6 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 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 ayuda al balance de carga.

La figura 3.7 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 mejor speed-up se alcanza con la *Bulk-Circular* con baja frecuencia. Este resultado es respaldado por la figura 3.8, la que muestra la eficiencia $E = (\sum(w_i/max_w))/P$ (usando la LC) de las estrategias *Bulk-Circular* y *Local*, donde w_i es la carga de trabajo del thread i, max_w es la máxima carga de trabajo que realizó algún thread y P es la cantidad de threads. Esto último indica que la ociosidad de los threads es mucho mayor usando la estrategia *Local* con un bajo tráfico de consultas.

3.3.1. 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 uno 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 3.9 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*. La estrategia *Híbrida* es la que muestra el mayor throughput,

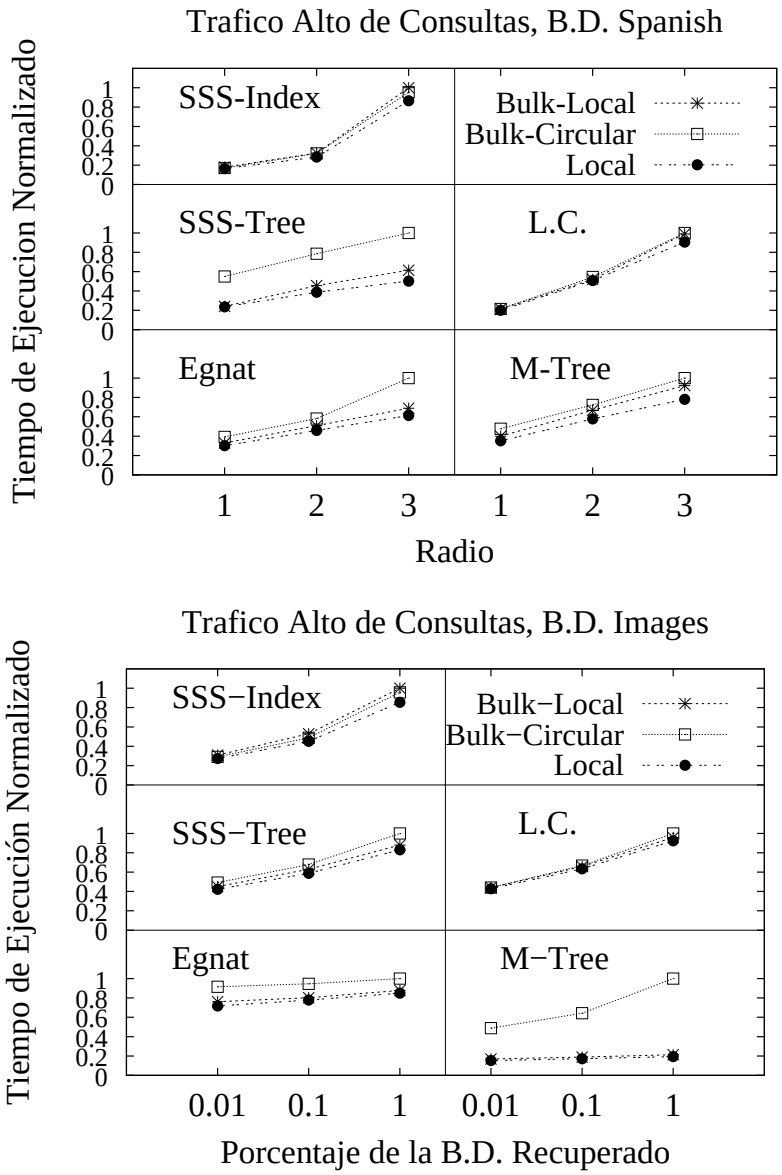


Figura 3.3: Tiempo de Ejecución para un alto tráfico de consultas.

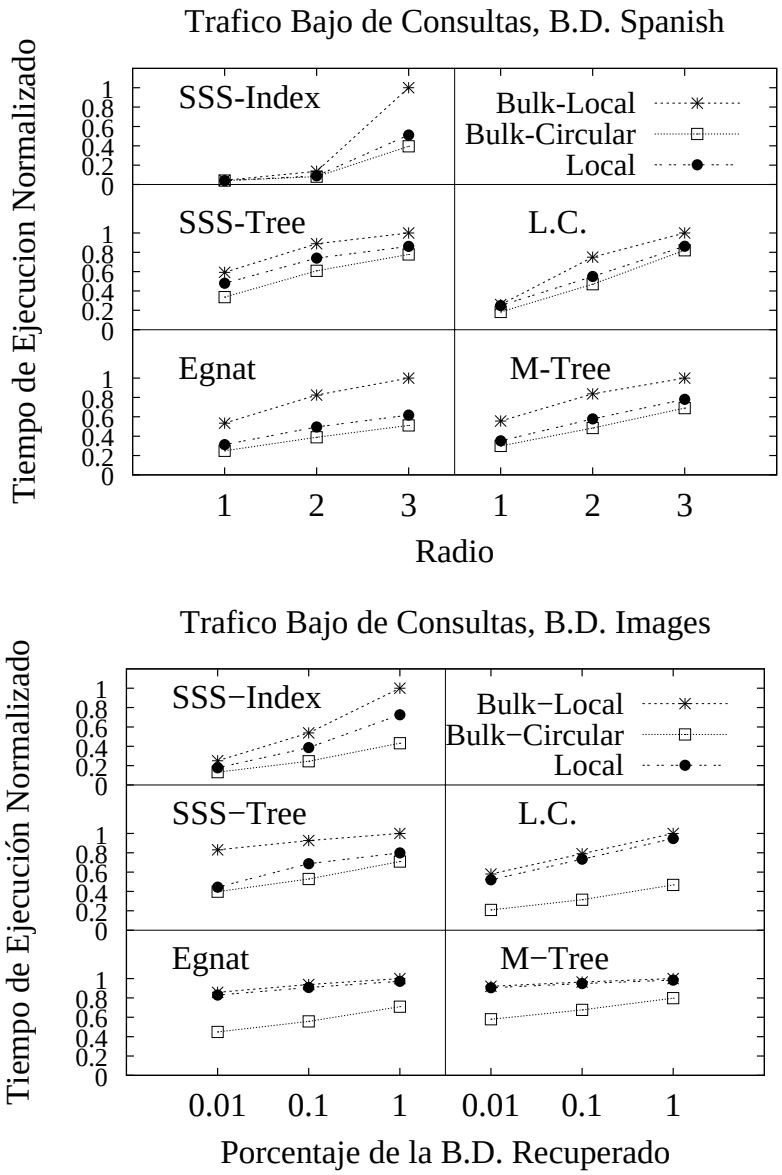


Figura 3.4: Tiempo de Ejecución para un bajo tráfico de consultas.

Seleccion del thread menos cargado

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	q5	q10		q8	q10	q10	q10
8	q9		q10	q11	q9	q9	q9	q9
9	q9	q12	q10	q11	q11	q12		q11
10	q9							
11								
12								

Distribucion circular

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		q11	q12
10	**q9**							
11	**q9**							
12	**q9**							

Figura 3.5: Distribuciones de consultas en un escenario de baja frecuencia.

pues es la que explota las ventajas de ambas estrategias.

3.3.2. 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), y luego cada thread resuelve sus consultas accediendo a su propio índice. Para esto se distribuyó cada elemento de *Spanish* e *Images* 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 BD 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 sincronizaciones, cada thread reporta sus resultados escribiéndolos en una posición distinta de memoria.

Las figuras 3.10 y 3.11 muestran el tiempo de ejecución y la cantidad de evaluaciones de distancia normalizados al mayor valor observado, 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. Distribuida*), 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 [32] 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.

3.4. Conclusiones

En este capítulo 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-Circular*, *Bulk-Local* y *Bulk-Critical*), siendo este último una implementación del modelo BSP.

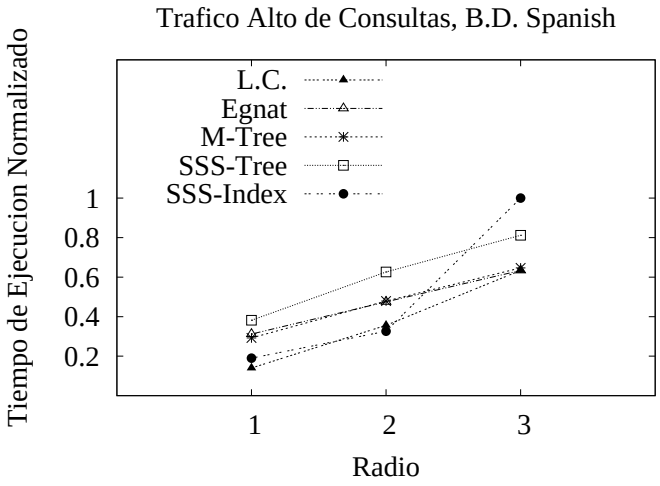
La estrategia *Bulk-Critical* es la que muestra el peor rendimiento, debido principalmente al costo de acceso secuencial a las *regiones críticas*.

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.

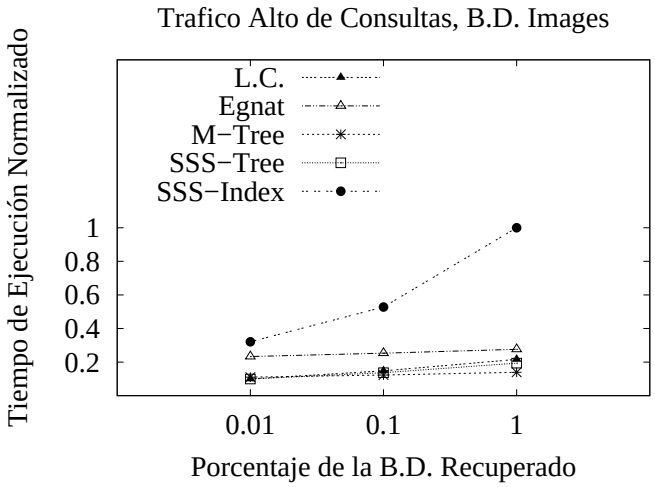
La estrategia *Bulk-Circular* es la de mejor rendimiento en un escenario de bajo tráfico de consultas. En esta estrategia cada thread distribuye los requerimientos de sus consultas entre todos los demás threads, lo que implica que todos los threads participen en la solución de todas las consultas. Esta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

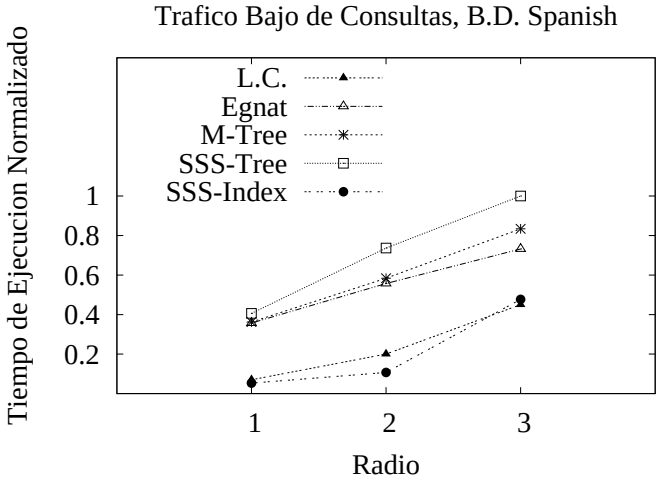
En la Sección 3.3.2 se comparan dos distribuciones de la base de datos. La primera distribuye los elementos de la base de datos entre los threads, y cada uno de éstos construye su propio índice. La segunda mantiene sólo un índice global en memoria principal. Esta última es la que muestra el mejor rendimiento 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.



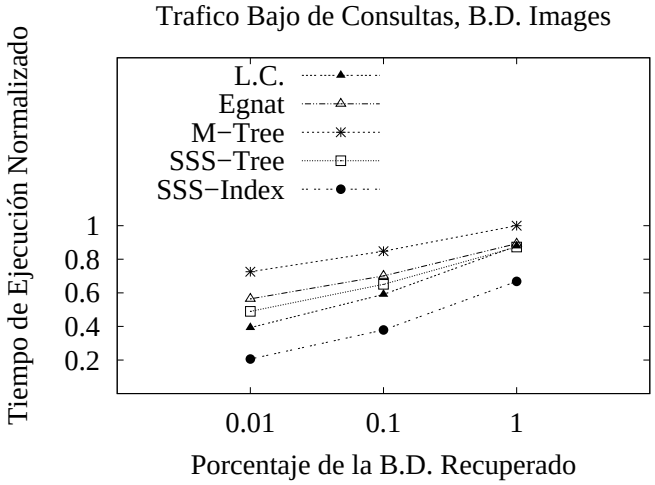
(a) Estrategia *Local*.



(b) Estrategia *Local*.



(c) Estrategia *Bulk-Circular*.



(d) Estrategia *Bulk-Circular*.

Figura 3.6: Comparación de los índices con la estrategia *Local* para alto tráfico de consultas y la *Bulk-Circular* para bajo tráfico.

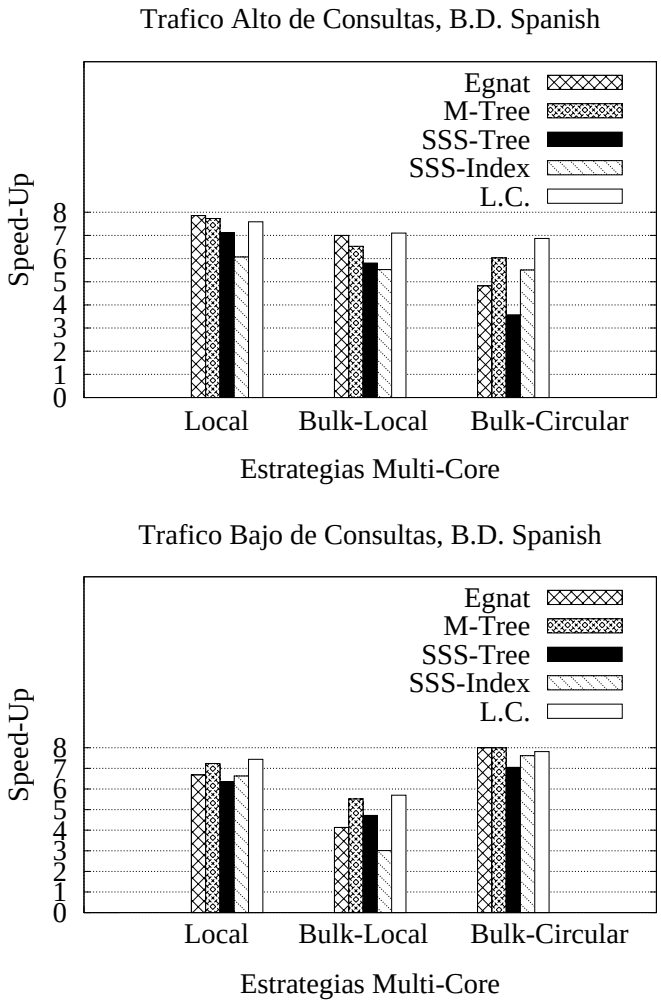


Figura 3.7: Speed-up de los índices sobre la BD Spanish con radio 3.

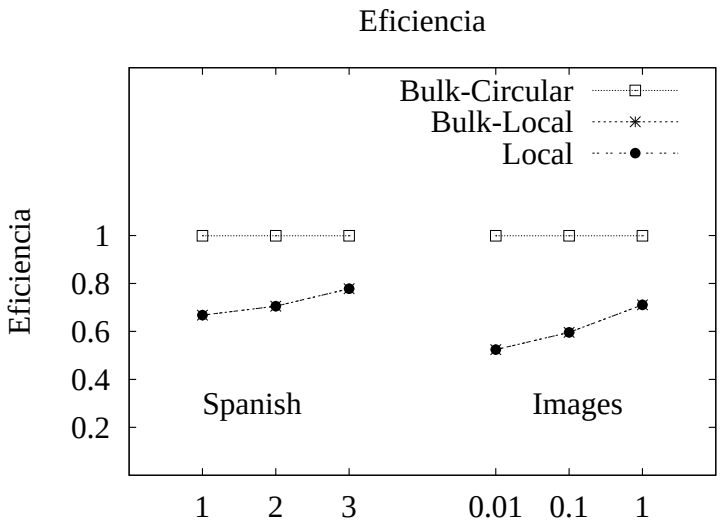


Figura 3.8: Eficiencia de las estrategias multi-core usando la LC con un tráfico bajo de consultas.

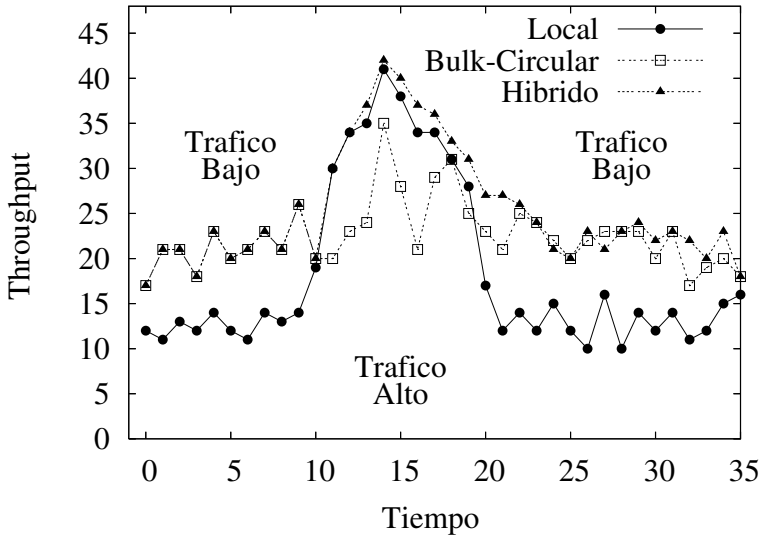
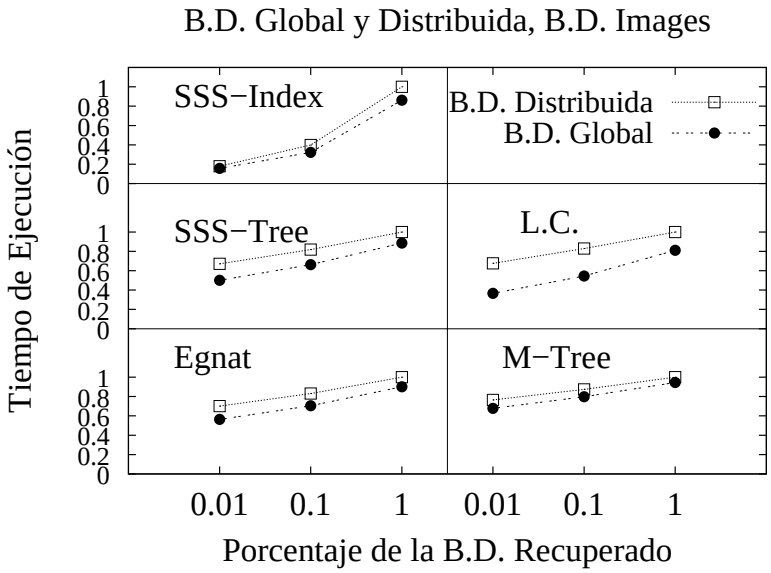
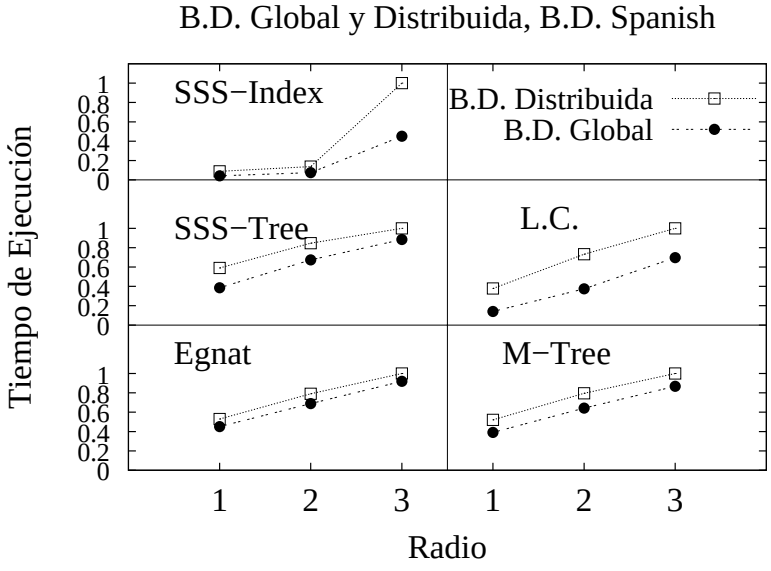


Figura 3.9: Consultas completamente resueltas por unidad de tiempo.

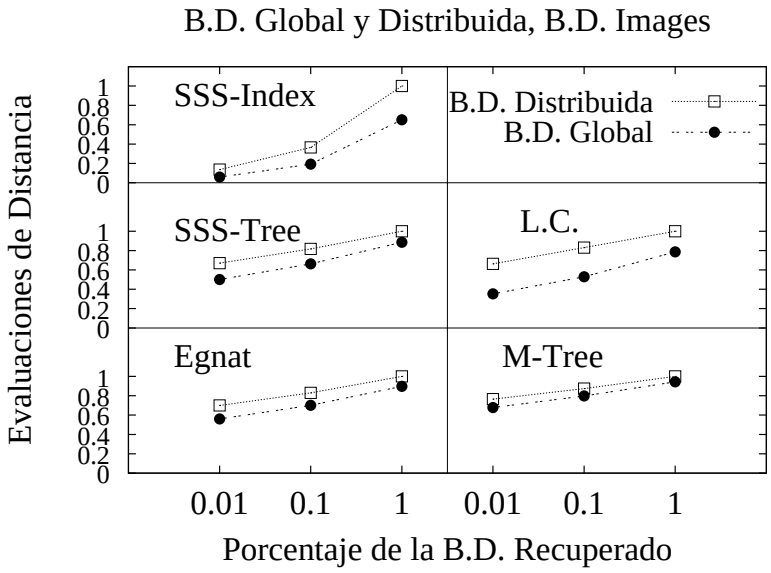


(a)

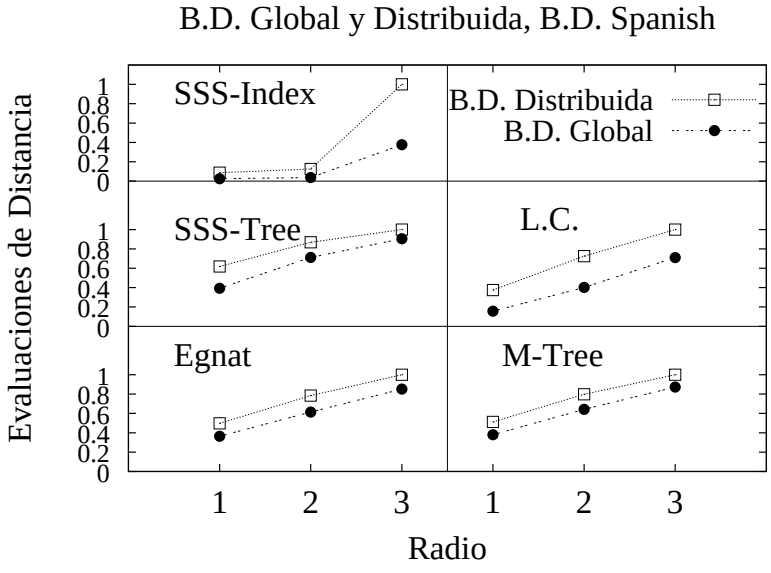


(b)

Figura 3.10: Tiempos de Ejecución normalizados para las bases de datos **a)***Images* y **b)***Spanish*.



(a)



(b)

Figura 3.11: Promedio normalizado de evaluaciones de distancias por consulta para las bases de datos **a)***Images* y **b)***Spanish*.

Capítulo 4

Estrategias de distribución y búsqueda sobre GPU

En este Capítulo se proponen y comparan algoritmos de búsqueda de índices métricos sobre GPU (Graphic Processor Unit) basados en CUDA [1]. 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 2.1.3, 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 mencionados en 2.2.1, consultas por rango y de los k vecinos más cercanos (kNN) de forma separada. Para cada tipo de consulta las estrategias empleadas y los problemas encontrados fueron diferentes.

El trabajo relacionado con esta área utilizando GPUs es escaso (Sección 2.3.2). Existen artículos que proponen una solución de fuerza bruta sobre consultas kNN en [24], [19] y [6], pero este último fue descartado debido a que (i) propone una solución a consultas kNN con $k = 1$, que es un caso particular y de menor complejidad al abordado en este trabajo (kNN con $k > 1$), y (ii) la solución se basa en el manejo de memorias de texturas en GPU, lo que se aleja de los objetivos del presente trabajo.

En todas las implementaciones siguientes (con excepción de las mostradas en 4.2.1.1), 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 (este número está limitado a 512 según las

restricciones mencionadas en la Sección 2.1.3.2). 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*, debido al frecuente acceso a ésta por parte de los threads. *Shared memory* es una memoria de reducido tamaño que se encuentra dentro de cada multiprocesador, y debido a esto es de muy baja latencia. Los datos almacenados en esta memoria son sólo compartidos por todos los threads del mismo bloque (ver Sección 2.1.3.2).

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, pues esto 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 este Capítulo 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 este capítulo es como sigue. En la Sección 4.1 se muestran los algoritmos sobre GPU para resolver consultas por rango usando los métodos *Fuerza Bruta*, *LC* y *SSS-Index*. En la Sección 4.2 se muestran los algoritmos basados en ordenamiento, *LC* y *SSS-Index* sobre GPU para resolver consultas de tipo kNN. En la Sección 4.3 se muestran los resultados experimentales comparando los distintos métodos para resolver consultas por rango y kNN. Finalmente el la Sección 4.4 se dan a conocer las conclusiones del presente capítulo.

4.1. Consultas por Rango

4.1.1. 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.

El algoritmo 11 muestra la búsqueda realizada por el *kernel* (sobre la base de datos de vectores) para la consulta por rango (q, *rango*). Para esto 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. Esto último con la finalidad que threads consecutivos accedan a posiciones contiguas de memoria al leer datos desde *device memory*. Este algoritmo está dividido en etapas delimitadas por la función de sincronización `__syncthreads()`, las que se describen a continuación:

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

Algoritmo 11 Búsqueda por Fuerza Bruta en consultas por rango sobre GPU.

{Cada columna de *BD* es un elemento de la BD}
{Cada fila de *Queries* representa una consulta.}
{Sea $SIZE_{BD}$ el número de elementos de la BD}
{Sea D la dimensión de cada elemento de la BD}

__global__ busquedarango(float **BD*, float ***Queries*, float *rango*)

```
 1: __shared__ float query[D];
 2:
 3: for (i = ID_Thread; i <D; i+=T_Block) do
 4:     query[i] = Queries[ID_Block][i]
 5: end for
 6:
 7: __syncthreads()
 8:
 9: for (j = ID_Thread; j <SIZE_BD; j+=T_Block) do
10:     if distancia(BD, j, query) <= rango then
11:         encontrado()
12:     end if
13: end for
```

__device__ float distancia(float ***M*$_1$, int *col*, float **M*$_2$)

```
dist = 0
for (i = 0; i <D; i++) do
    dist += (M_1[i][col] − M_2[i]) ∗ (M_1[i][col] − M_2[i])
end for
dist = sqrtf(dist)
return dist
```

- En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory* (línea 4).
- 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 (línea 10). En caso que el elemento pertenezca al conjunto respuesta, se invoca la función `encontrado()`.

La función *encontrado()* reporta a un elemento como parte de la respuesta, lo cual dependiendo de las necesidades de la aplicación podría ser implementada de varias formas, pero para fines de este trabajo, esta función sólo aumenta un contador, llevando la cuenta de los elementos encontrados por cada thread.

4.1.2. 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 12. *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.

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 (Sección 2.1.3.3).

El algoritmo 12 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()`. A continuación se describe cada etapa.

- En esta primera etapa cada bloque copia 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 rango de búsqueda (línea 13), éste se agrega al conjunto respuesta. En *shared memory* (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).
- 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).

4.1.3. *SSS-Index*

Este índice se representó por 3 matrices denominadas *PIVOTES*, *DISTANCIAS* y *BD* en el algoritmo 13. *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.

Los elementos de la base de datos se asignan entre los threads siguiendo una distribución Round-Robin, y cada thread se encarga de descartar los elementos que le corresponden utilizando todos los pivotes, y de no ser posible el descarte, el mismo thread realiza la evaluación de distancia entre la consulta y el elemento para verificar si es parte de la respuesta.

El algoritmo 13 muestra el kernel usado por el *SSS-Index* para resolver consultas por rango sobre la base de datos de vectores. Está dividido en etapas delimitadas por la función de sincronización `__syncthreads()`. A continuación se describe cada una de estas etapas.

- En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory* (variable *query*).

Algoritmo 12 *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 *rango*)

```
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 <= rango then
14:         encontrado()
15:     end if
16:     buscar[j] = dist <= RC[j] + rango
17:     if dist < RC[j] - rango 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) <= rango then
27:             encontrado()
28:         end if
29:     end if
30: end for
```

Algoritmo 13 *Kernel* de búsqueda del *SSS-Index* para resolver consultas por rango sobre GPU.

{Cada fila de *Queries* representa una consulta.}
{Sea $SIZE_{BD}$ el número de elementos de la BD}
{Sea $SIZE_{Piv}$ la cantidad de pivotes}
{Sea D la dimensión de cada elemento de la BD}

__global__ busquedarango(float ***PIVOTES*, float ***DISTANCIAS*, float ***BD*, float ***Queries*, float *rango*)

```
 1: __shared__ float query[D]
 2: __shared__ float dist_piv[SIZE_Piv]
 3:
 4: for (i = ID_Thread; i <D; i+=T_Block) do
 5:     query[i] = Queries[ID_Block][i]
 6: end for
 7:
 8: __syncthreads()
 9:
10: {Se obtiene la distancia entre cada pivote y la consulta}
11: for (j = ID_Thread; j <SIZE_Piv; j+=T_Block) do
12:     dist_piv[j] = distancia(PIVOTES, j, query)
13: end for
14:
15: __syncthreads()
16:
17: for (j = ID_Thread; j <SIZE_BD; j+=T_Block) do
18:     descartado = 0
19:     for (i = 0; i < SIZE_Piv; i++) do
20:         if dist_piv[i] < DISTANCIAS[i][j] - rango || dist_piv[i] > DISTANCIAS[i][j] + rango then
21:             descartado = 1
22:             break
23:         end if
24:     end for
25:     if descartado == 0 then
26:         if distancia(BD, j, query) <= rango then
27:             encontrado()
28:         end if
29:     end if
30: end for
```

- En la segunda etapa, cada thread (siguiendo una distribución Round-Robin) obtiene la distancia entre un subconjunto de pivotes y la consulta (línea 11), y la almacena en *shared memory* (línea 12).

- 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 (línea 20) y en caso de no ser posible, el mismo thread realiza la evaluación de distancia entre el elemento y la consulta (línea 26).

En el artículo [4], 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 4.1, el rendimiento óptimo para el *SSS-Index* sobre GPU se consigue con $\alpha = 0{,}6$ (1 pivote). Esta figura muestra tres gráficos correspondientes al tiempo de ejecución, la cantidad de

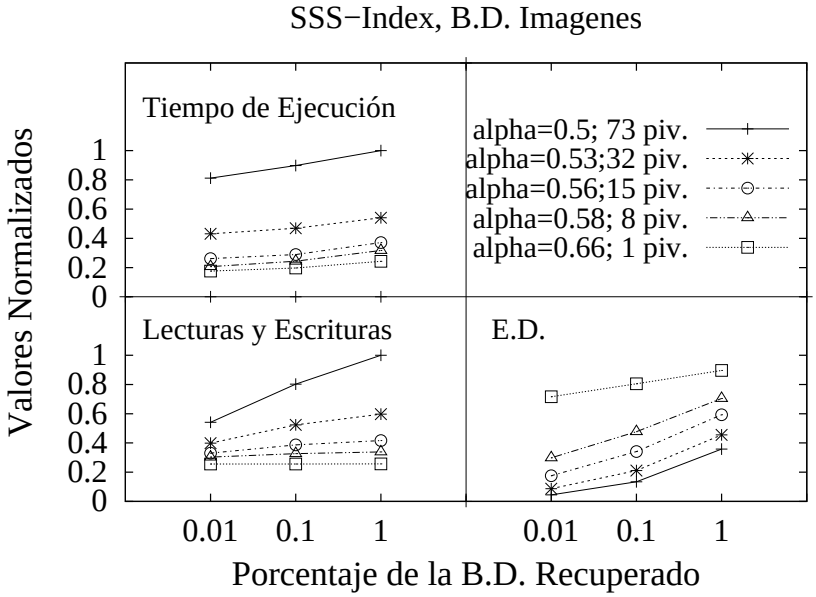


Figura 4.1: Valores normalizados del tiempos de ejecución, cantidad de lecturas/escrituras (de 32, 64 o 128 bytes) a *device memory* y del promedio de evaluaciones de distancia por consulta del *SSS-Index* sobre GPU para la base de datos *Images*.

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.

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 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 *warp* y la irregularidad en los accesos a memoria.

También se implementó y comparó otra estrategia de paralelización sobre este índice. Esta 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 la distancia de la consulta al pivote que le corresponde. Por cada elemento se utiliza una variable en *shared memory* que indica si el elemento ha sido descartado por algún thread. Por las limitaciones de espacio de esta memoria, este proceso se realiza iterativamente sobre un conjunto de E elementos cada vez. En cada iteración, luego de evaluar E elementos, se ejecuta una instrucción de sincronización

Tabla 4.1: Tiempos de ejecución (en segundos) y cantidad de lecturas/escrituras a *device memory*.

Tiempo de Ejecución (segs.)		
% de la B.D. 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)		
% de la B.D. Recuperado	***Distribución-Pivote***	***Distribución-Elemento***
0,01	455551	29699
0,1	461104	29740
1	468319	29806

y los elementos no descartados son distribuidos Round-Robin entre los threads y cada uno de éstos realiza la evaluación de distancia entre la consulta y sus elementos asignados. La tabla 4.1 muestra el tiempo de ejecución real y la cantidad de lecturas/escrituras entre esta estrategia (*Distribución-Pivote*) y la anteriormente descrita que distribuye los elementos entre los threads (*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 rango de búsqueda, también crece la regularidad en la ejecución de instrucciones. La figura 4.2 muestra la cantidad de veces en que al menos un thread de un *warp* diverge para la estrategia *Distribución-Pivote* sobre la base de datos *Images*. Para *Spanish* los resultados fueron similares. Debido al mal rendimiento de esta estrategia, ésta fue descartada y en todos los experimentos posteriores, la versión del *SSS-Index* utilizada es la que utiliza la estrategia *Distribución-Elemento*.

4.2. Consultas de los k vecinos más cercanos (kNN)

A pesar de utilizar los algoritmos de consultas por rango como base para resolver consultas del tipo kNN, se encontraron problemas distintos a los vistos anteriormente derivados del hardware de la GPU. A continuación se presentan los algoritmos de búsqueda de la *LC* y *SSS-Index* para resolver consultas kNN. Ambos índices fueron comparados con una implementación de búsqueda exhaustiva basada en trabajos previos.

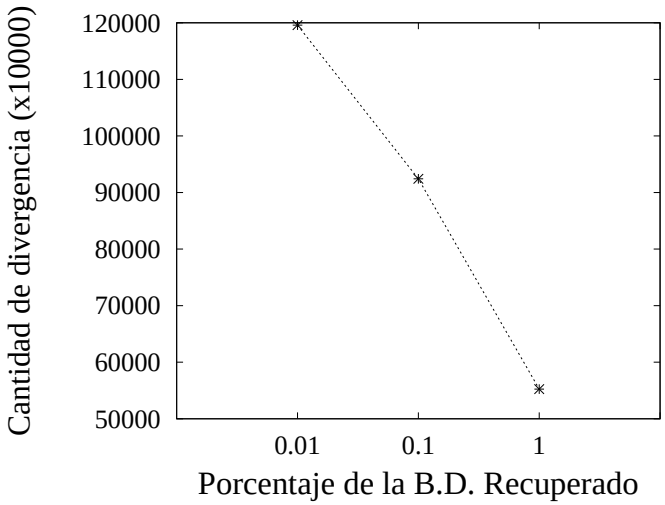


Figura 4.2: Cantidad de veces en que al menos un thread de un *warp* diverge para la estrategia *Distribución-Pivote* sobre la base de datos *Images*.

4.2.1. Búsqueda exhaustiva

4.2.1.1. Solución basada en ordenamiento

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.

Al igual que lo visto en 2.3.2, este método ordena el arreglo de distancias δ. Trabajos previos ([24, 19]) utilizan como método de ordenamiento el *radix sort* [42] e *insertion sort*, pero el método utilizado en este trabajo usa el *GPU-Quicksort* [7], el cual tiene mejor rendimiento que los algoritmos anteriores.

El *GPU-Quicksort* se divide en dos etapas, la primera consiste en dividir el arreglo a ordenar en P particiones, que pueden ser ordenadas independientemente. Para esto se lanza un *kernel* con M bloques de threads, (M=cantidad de particiones actuales) y cada bloque genera dos nuevas particiones con los elementos mayores y menores a un pivote previamente seleccionado. Este proceso es repetido recursivamente hasta alcanzar un umbral que indica que hay una cantidad adecuada de particiones para ejecutar la segunda etapa. Esta consiste en lanzar un *kernel* donde cada bloque se encarga de ordenar una de estas particiones usando el *quicksort* [22], pero implementando la recursión usando una pila en *shared memory*. Cuando la secuencia a ordenar es suficientemente pequeña, se usa el *bitonic sort* [20], el que está implementado de forma nativa en CUDA.

4.2.2. 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 (Sección 4.1.2).

El método comúnmente usado por índices métricos para resolver una consulta de tipo kNN es un método decreciente (según lo mencionado en 2.2.1). Este método fue descartado al momento de utilizar GPU, debido a que todos los threads de un bloque están involucrados en resolver una consulta, y este paralelismo intra-consulta no permite reducir el rango de búsqueda suficientemente rápido, realizando una cantidad de trabajo similar a una búsqueda exhaustiva. Pero por el contrario, el método de radio creciente (Sección 2.2.1) se adapta muy bien a este tipo de plataforma.

La figura 4.3 muestra el tiempo de ejecución de la LC usando un método de *rango decreciente* y *rango creciente* sobre la base de datos *Images*. El método de rango decreciente inicializa el rango a $rango = \infty$, 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 rango se realiza usando la función `atomicMin()`. Por otro lado, el método de rango creciente, establece un rango inicial ($rango = r_{ini}$) y se realiza una búsqueda iterativa, y en cada iteración el rango se aumenta ($rango{+}{=}\Delta$). Se utilizó el 1 % de los elementos de la base de datos como elementos de entrenamiento para establecer los valores r_{ini} y Δ de forma previa a la búsqueda.

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

Con respecto al método de rango 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 $rango = r_{ini}$ 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 reutilizaron evaluaciones de distancia realizadas en una iteración anterior,

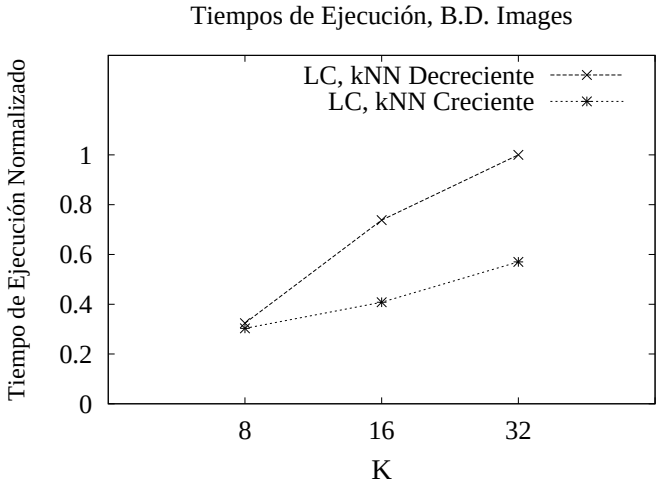


Figura 4.3: Tiempo de ejecución normalizado sobre la base de datos *Images* de un método de rango creciente y decreciente para resolver consulta de tipo kNN usando la LC sobre GPU.

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 [23] 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, en donde el primer *warp* del bloque se encarga de recorrer los elementos de los heaps anteriores, y de esta forma cada thread del *warp* obtiene un heap con sus K elementos más cercanos a la consulta, pero esta vez los heaps se almacenan en *shared memory*. Esta memoria es de tamaño reducido, y el espacio necesario para almacenar los heaps de un *warp* depende del tamaño de K. Finalmente, el primer thread del bloque es el encargado de recorrer los elementos de los heaps almacenados en *shared memory* y almacenar éstos en un nuevo heap, también almacenado en esta memoria, siendo estos K elementos el resultado final. 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 [24, 19, 6].

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.

El algoritmo de reducción de los elementos de heaps almacenados en *device memory* por parte de los thread de un *warp* es mostrado en la función `reduccion_warp()` del algoritmo 14. En este mismo algoritmo se muestra una segunda función (`insercion_heap()`) encargada de insertar un elemento dentro de un heap. Esta inserción se produce sólo si el elemento a insertar es menor que la raíz del heap. Debido a que siempre al sacar un elemento de la raíz (`pop()`) se inserta un nuevo elemento (`push(x)`), se definió la función `popush(x)`, que simplifica ambas funciones, es

decir, intercambia el elemento raíz por el nuevo elemento y hace descender a éste a la posición en el heap que le corresponde.

El algoritmo 15 muestra el pseudo-código del *kernel* usado por la LC para resolver consultas de tipo kNN. Se usa el método de rango creciente y es usado el mismo algoritmo para ambos lanzamientos de *kernels* necesarios para resolver las consultas. Las etapas del algoritmo están delimitadas por la función de sincronización `syncthreads()`. A continuación se describe cada una de estas etapas.

- En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory* (línea 8). Previamente se establece el rango inicial, usando la función `rango_inicial()`, la que toma en cuenta a qué lanzamiento de *kernel* corresponde.

- En la segunda etapa, cada thread (siguiendo una distribución Round-Robin) obtiene la distancia entre un centro de cluster y la consulta (línea 14), y la almacena en *shared memory* (variable $distC$).

- 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 (línea 22) es comparado contra la query (línea 23). Y si el elemento está dentro del rango actual de búsqueda, entonces éste se inserta en el heap del thread almacenado en *device memory* (línea 25). Luego, los centros son distribuidos (Round-Robin) entre los threads y cada centro que se encuentre dentro del rango actual de búsqueda (línea 33) 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* (línea 42).

- 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* (línea 47). La función `evalua_condicion()` establece el valor de la variable *condicion* dependiendo si se han encontrado K resultados y si es el primer o segundo lanzamiento de *kernel*.

4.2.3. *SSS-Index*

Para resolver consultas de tipo kNN, este índice se representó con las mismas estructuras de datos usadas para resolver consultas por rango (Sección 4.1.3).

Al igual que en la LC (y por las mismas razones), se utilizó el método de rango creciente y las consultas son resueltas en dos lanzamientos de *kernels*. También, cada thread utiliza un heap y se realiza una reducción de éstos como en la LC.

Algoritmo 14 Funciones auxiliares usadas por los algoritmos de búsqueda de consultas kNN en los índices LC y *SSS-Index* sobre GPU.

{Sea ID_{Thread} el identificador de un hilo dentro de un bloque.}
{Sea T_{Block} la cantidad de threads por bloque.}
{Sea ID_{Block} el identificador de un bloque.}
{Sea $tid = ID_{Thread} + (T_{Block} * ID_{Block})$ el identificador único del thread entre todos los threads de todos los bloques.}

```
reduccion_warp(Heaps DH, Heaps SH)
  if tid < SIZE_Warp then
    for (j = tid; j < T_Block*(ID_Block+1); j += SIZE_Warp) do
      for (i = 0; i < K; i++) do
        x = DH_j.pop()
        insercion_heap(SH, x)
      end for
    end for
  end if

insercion_heap(Heaps H, Elem x)
  if H_tid.size()< K then
    H_tid.push(x)
  else
    if x.dist < H_tid.top() then
      H_tid.popush(x)
    end if
  end if
```

La figura 4.4 muestra el rendimiento del *SSS-Index* sobre la base de datos *Images* utilizando distintos valores de α. Al igual que lo ocurrido con el algoritmo de búsqueda por rango, el mejor rendimiento se obtiene con 1 pivote (α = 0,66). Las razones de este comportamiento son las mismas que las descritas en la Sección 4.1.3.

El algoritmo 16 muestra el *kernel* usado por el *SSS-Index* para resolver consultas de tipo kNN. Está dividido en etapas delimitadas por la función de sincronización `__syncthreads()`, las que se describen a continuación.

- En la primera etapa los threads colaboran para copiar la consulta que le corresponde resolver al bloque de threads a *shared memory* (línea 5).
- 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* (línea 11).
- 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 (línea 19) y en caso de no ser posible, el mismo thread realiza la evaluación de distancia entre el elemento y la consulta. Y cada thread, si corresponde, almacena los elementos en su heap en *device memory* (línea 27).
- En la cuarta etapa, el primer *warp* accede a los elementos de los heaps de la etapa anterior.

Algoritmo 15 *Kernel* de búsqueda de la LC para resolver consultas de tipo kNN sobre GPU.

{Sea DH_i el heap del thread i almacenado en *device memory*}
{Sea SH es el conjunto de heaps del primer *warp* almacenado en *shared memory*}
KNN_LC(float ***CENTROS*, float **RC*, float ***CLUSTERS*, float ***Queries*)

1: __shared__ float *query*[D]
2: __shared__ float *distC*[$SIZE_{Centros}$]
3: __shared__ float *rango*=rango_inicial()
4: __shared__ int *minC*=$SIZE_{Centros}$
5: __shared__ int *condicion* = 1
6: *tid* = theadIdx.x
7: **for** ($i = ID_{Thread}$; $i <$*D*; i+=T_{Block}) **do**
8: *query*[i] = *Queries*[ID_{Block}][i]
9: **end for**
10:
11: __syncthreads()
12: {Se obtiene la distancia de todos los centros a la consulta.}
13: **for** ($i = ID_{Thread}$; $i < SIZE_{Centros}$; i+=T_{Block}) **do**
14: *distC*[i] = distancia(*CENTROS*, i, *query*)
15: **end for**
16:
17: __syncthreads()
18: **while** *condicion* **do**
19: **for** ($i = ID_{Thread}$; $i < SIZE_{Clusters}$ && (i/B_{Size})$\leq$ *minC*; i+=T_{Block}) **do**
20: *indC* = i/B_{Size}
21: *rc* = *RC*[*indC*]
22: **if** *distC*[*indC*] $\leq$ *rc* + *rango* **then**
23: **if** ($x.dist$ = distancia(*CLUSTERS*, i, *query*))$<$ *rango* **then**
24: $x.ind$=i
25: insercion_heap(DH_{tid}, x)
26: **end if**
27: **end if**
28: **if** *distC*[*indC*]$<$ *rc* - *rango* **then**
29: atomicMin(*minC*, *indC*)
30: **end if**
31: **end for**
32: {Si algún centro es parte de la respuesta se agrega a los resultados.}
33: **for** ($i = ID_{Thread}$; $i < SIZE_{Centros}$; i+=T_{Block}) **do**
34: **if** ($x.dist$=*distC*[i])$\leq$ *rango* **then**
35: $x.ind$=i
36: insercion_heap(DH_{tid}, x)
37: **end if**
38: **end for**
39:
40: __syncthreads()
41: {Un *warp* almacena los elementos de los heaps anteriores en $SIZE_{Warp}$ heaps en *shared memory*}
42: reduccion_warp(DH, SH)
43:
44: __syncthreads()
45: {Un thread recorre exhaustivamente SH y selecciona los primeros K elementos}
46: **if** *tid* == 0 **then**
47: get_first_K(SH)
48: *cantidad_resultados* = get_first_K(SH)
49: *condicion* = evalua_condicion(*cantidad_resultados*, *query*)
50: **end if**
51: __syncthreads()
52: **end while**

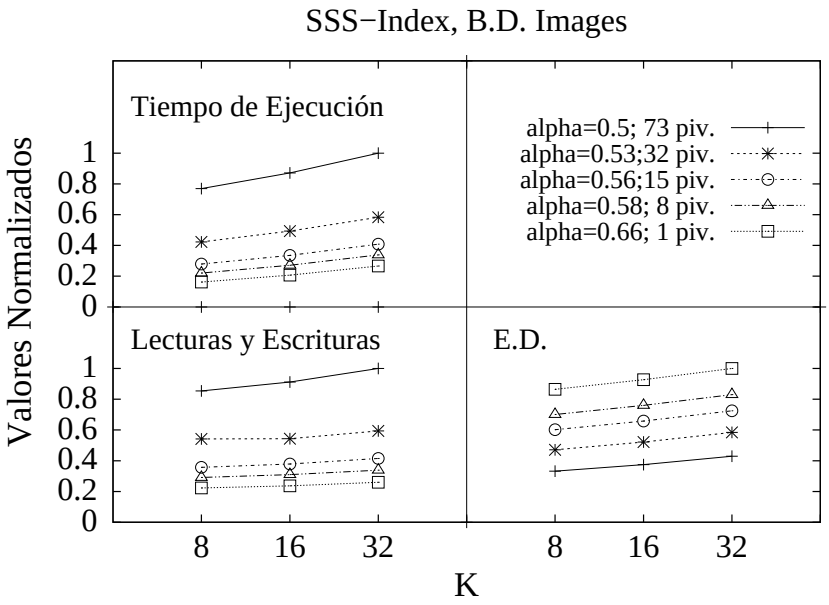


Figura 4.4: Valores normalizados del tiempos de ejecución, cantidad de lecturas/escrituras (de 32, 64 o 128 bytes) a *device memory* y del promedio de evaluaciones de distancia por consulta del *SSS-Index* sobre GPU para la base de datos *Images*.

Cada thread del *warp* almacena los elementos en su heap almacenado en *shared memory* (línea 34).

- 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* (línea 39). La función `evalua_condicion()` establece el valor de la variable compartida *condicion* dependiendo si se han encontrado K resultados y si es el primer o segundo lanzamiento de *kernel*.

4.3. Resultados experimentales sobre GPU

La GPU utilizada fue una NVIDIA Tesla T10, 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 3.1.

Los gráficos de lectura/escritura mostrados en este Capítulo representan la suma total de estas operaciones. Recordemos que la lectura (o escritura) mínima permitida es de 32 bytes (Sección 2.1.3.2), 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 (Sección 2.1.3.2).

Algoritmo 16 *Kernel* de búsqueda del *SSS-Index* para resolver consultas de tipo kNN sobre GPU.

{Sea $SIZE_{BD}$ el número de elementos de la BD}
{Sea DH_i el heap del thread i almacenado en *device memory*}
{Sea SH es el conjunto de heaps del primer *warp* almacenado en *shared memory*}
{Sea $SIZE_{Warp}$ el tamaño de un *warp*}
{Sea D la dimensión de los elementos de la BD}

KNN_SSS-Index(float ***PIVOTES*, float ***DISTANCIAS*, float ***BD*, float ***Queries*, float *rango*)

```
1: __shared__ float query[D]
2: __shared__ float dist_piv[SIZE_piv]
3: __shared__ int condicion = 1
4: for (i = ID_Thread; i < D; i+=T_Block) do
5:     query[i] = Queries[ID_Block][i]
6: end for
7:
8: __syncthreads()
9: {Se obtiene la distancia de la consulta a todos los pivotes}
10: for (i = ID_Thread; i < SIZE_piv; i+=T_Block) do
11:     dist_piv[i] = distancia(PIVOTES, i, query)
12: end for
13:
14: __syncthreads()
15: while condicion do
16:     for (j = ID_Thread; j < SIZE_BD; j+=T_Block) do
17:         descartado = 0
18:         for (i=0; i < SIZE_piv; i++) do
19:             if dist_piv[i] < DISTANCIAS[i][j] - rango || dist_piv[i] > DISTANCIAS[i][j] + rango then
20:                 descartado = 1
21:                 break
22:             end if
23:         end for
24:         if descartado == 0 then
25:             if (x.dist=distancia(BD, j, query)) <= rango then
26:                 x.ind=i
27:                 insercion_heap(DH, x)
28:             end if
29:         end if
30:     end for
31:
32:     __syncthreads()
33:     {Un warp almacena los elementos de los heaps anteriores en SIZE_Warp heaps en shared memory}
34:     reduccion_warp(DH, SH)
35:
36:     __syncthreads()
37:     {Un thread recorre exhaustivamente SH y selecciona los primeros K elementos}
38:     if tid == 0 then
39:         get_first_K(SH)
40:         cantidad_resultados = get_first_K(SH)
41:         condicion = evalua_condicion(cantidad_resultados, query)
42:     end if
43:     __syncthreads()
44: end while
```

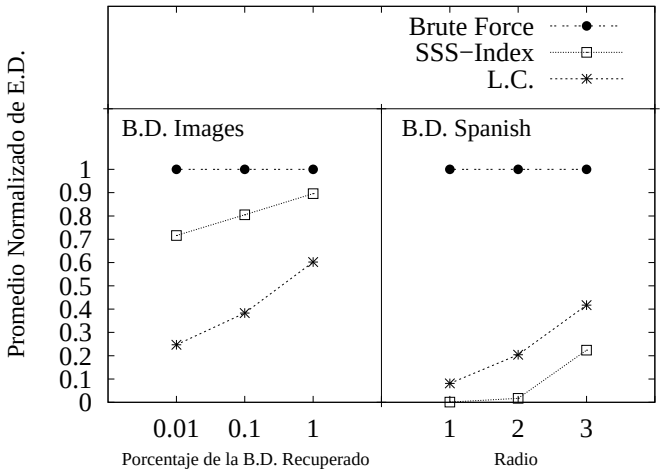
4.3.1. Experimentos sobre consultas por rango

En esta sección 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 Sección 4.1.3, la que utiliza $\alpha = 0{,}66$ (versión que genera 1 pivote) y para la base de datos *Spanish* $\alpha = 0{,}5$ (64 pivotes).

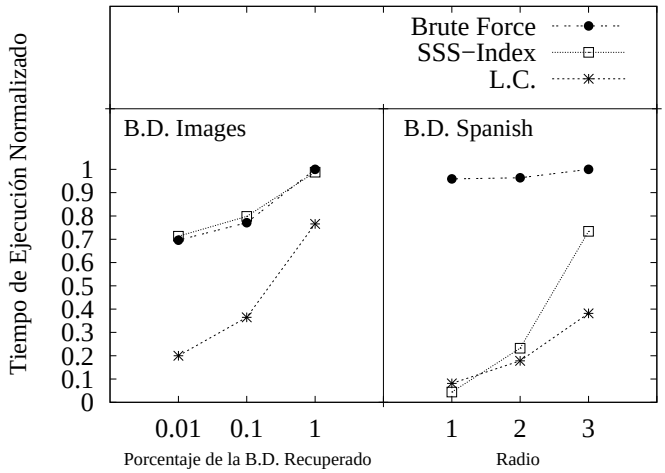
La figura 4.5 muestra tres gráficos correspondientes al (i) tiempo de ejecución, (ii) cantidad de lecturas/escrituras a *device memory* y (iii) el promedio de evaluaciones de distancia por consulta de los algoritmos *Fuerza Bruta*, *LC* y *SSS-Index*. Todos los valores fueron normalizados al mayor valor observado en el experimento.

Con respecto al número de evaluaciones de distancia (figura 4.5(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 4.5(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 4.5(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 Sección 2.1.3, 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.

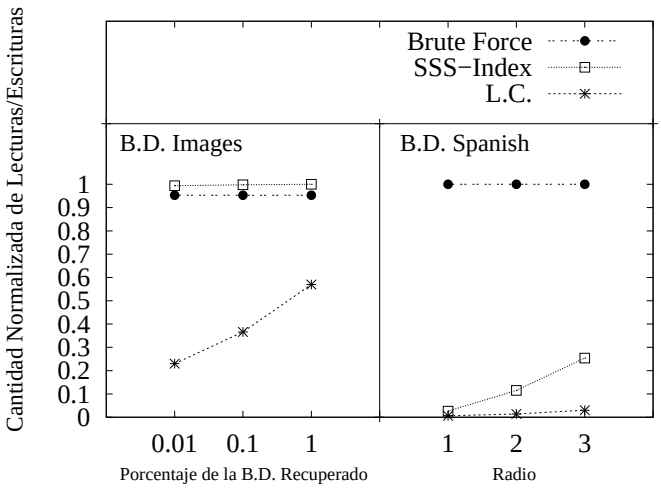
La única excepción a lo mencionado anteriormente se presenta con la base de datos *Spanish* con $radio = 1$, en donde a pesar de hacer un número mayor de lecturas/escrituras, el *SSS-Index* aventaja en tiempo a la *LC*. Esto es debido a que la cantidad de lecturas/escrituras para esta base de datos con $radio = 1$ 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 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.



(a) Evaluaciones de Distancia



(b) Tiempos de Ejecución



(c) Lectura/Escrituras a Memoria

Figura 4.5: 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.

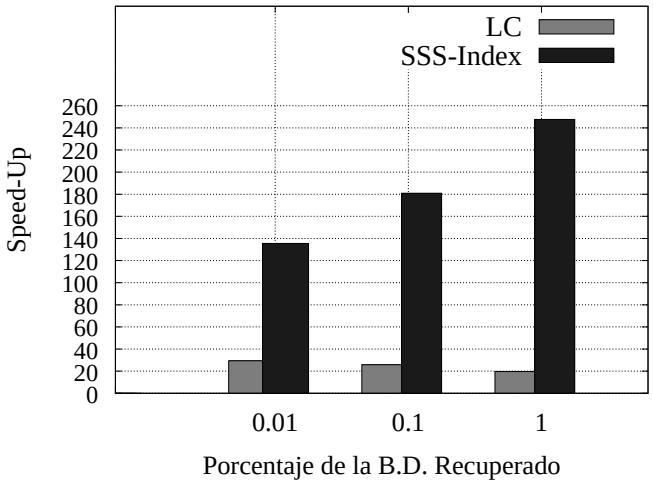


Figura 4.6: Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión secuencial sobre la base de datos *Images* resolviendo consultas por rango.

La figura 4.6 muestra el speed-up de la versión en GPU de la *LC* y *SSS-Index* para consultas por rango sobre su correspondiente implementación secuencial optimizada. Los resultados muestran para el *SSS-Index* hasta un 248x de speed-up recuperando el 1 % de la base de datos, y para la *LC* hasta un 29x de speed-up, lo que parece difícil de alcanzar en un servidor multi-core de tamaño medio. El comportamiento del *SSS-Index* en cuanto al gran speed-up alcanzado y el aumento de éste mientras mayor es el porcentaje de recuperación (a diferencia de la *LC*) se explica en parte porque la versión del *SSS-Index* utiliza sólo 1 pivote, y éste es elegido aleatoriamente. Por lo tanto, la versión de este índice no utiliza el método SSS (algoritmo 3) para seleccionar el pivote. Debido a esto no es equivocado decir que esta versión no se corresponde con el *SSS-Index* como se presentó originalmente en [4].

También se realizó el mismo experimento anterior, pero utilizando las versiones OpenMP de los índices presentadas en el Capítulo 3. La figura 4.7 muestra este experimento, en donde las versiones sobre GPU alcanzan hasta un 76.1x y 7.9x sobre el *SSS-Index* y *LC* respectivamente.

4.3.2. Experimentos sobre consultas de tipo kNN

En esta sección se presentan los experimentos comparativos entre el algoritmo basado en ordenamiento y los índices *LC* y *SSS-Index* para resolver consultas de tipo kNN. La plataforma utilizada fue la misma que en los experimentos de consultas por rango. Las bases de datos de elementos y consultas fueron las mismas que se utilizaron en los experimentos de consultas por rango (Sección 4.3.1).

La figura 4.8 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.

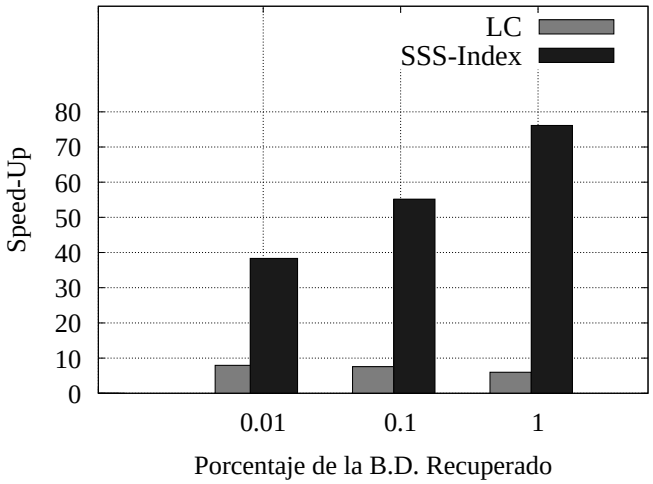


Figura 4.7: Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión en OpenMP sobre la base de datos *Images* resolviendo consultas por rango.

Tabla 4.2: Tiempos reales de ejecución (en segundos) entre el algoritmo de ordenamiento y la *LC*.

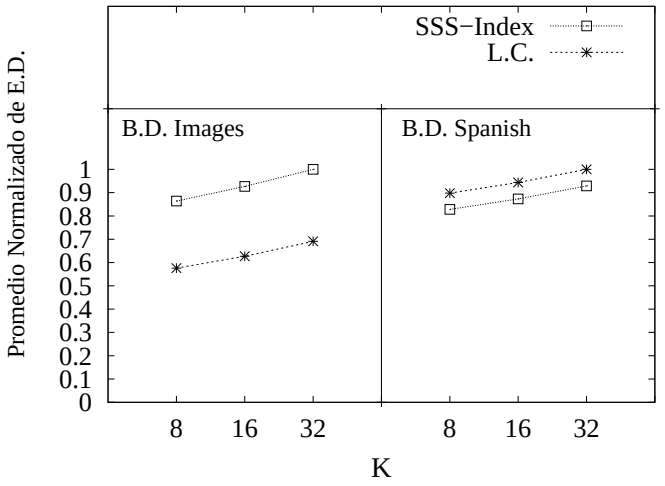
K	**Algoritmo de Ordenamiento**	**Lista de Clusters**
8	154,8	3,19
16	154,8	4,3
32	154,8	6,0

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, a modo de ejemplo, la GPU utilizada posee un throughput de 8 instrucciones aritméticas por ciclo de reloj en cada multiprocesador (Sección 2.1.3.2). Por lo anterior, el índice *LC* es el que presenta mejor rendimiento (considerando el tiempo de ejecución) para las bases de datos *Images* y *Spanish*, pues también es el que consigue menor cantidad de lecturas/escrituras a memoria.

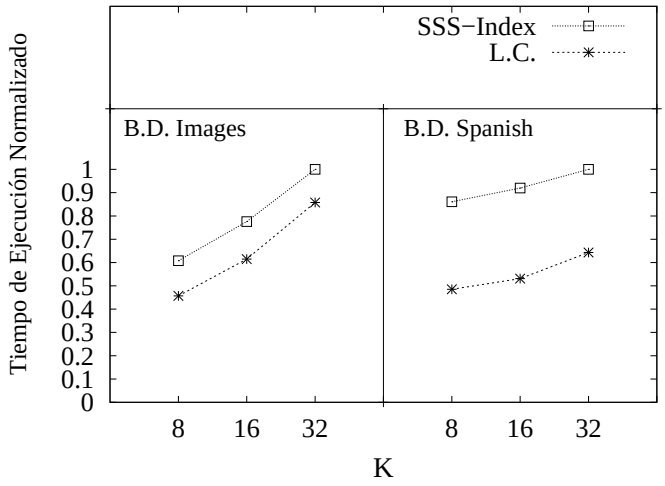
Cabe destacar que las instrucciones de escritura sólo se realizan al acceder a los heaps almacenados en *device memory*.

La figura 4.9 muestra el speed-up de los índices *LC* y *SSS-Index* sobre su correspondiente versión secuencial optimizada. Se alcanza un speed-up de hasta 17.4x en la *LC* y 144.2x para el *SSS-Index*.

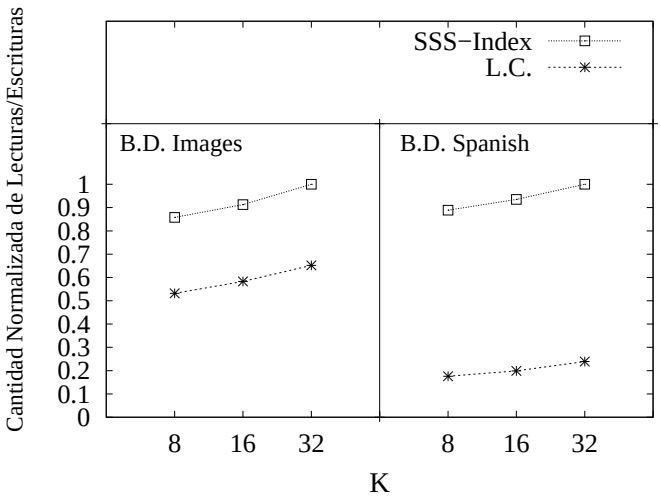
La tabla 4.2 muestra los tiempos de ejecución entre la *LC* y el algoritmo de ordenamiento (Sección 4.2.1.1). Siendo este último ampliamente superado por la *LC*. Al variar el valor de K en el algoritmo de ordenamiento, éste no sufre variación, pues este parámetro sólo interviene al momento de seleccionar los primeros K elementos del arreglo de distancias ordenado.



(a) Evaluaciones de Distancia



(b) Tiempos de Ejecución



(c) Lectura/Escrituras a Memoria

Figura 4.8: 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.

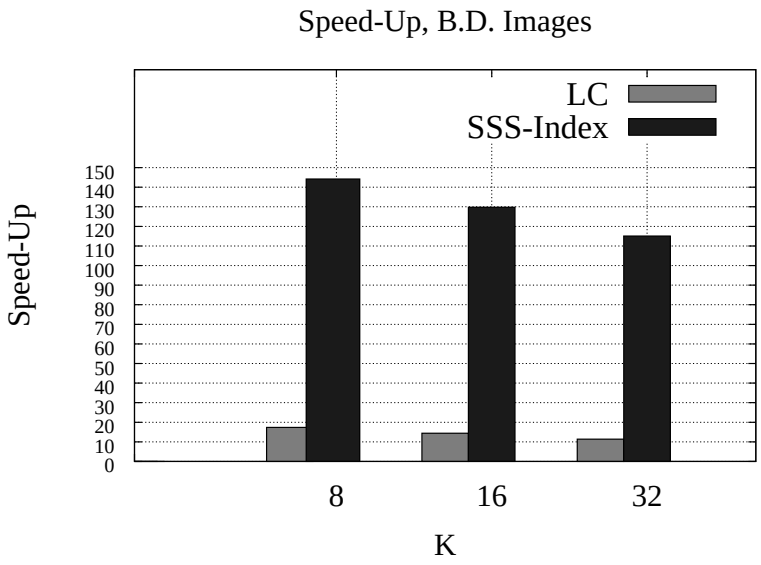


Figura 4.9: Speed-Up de las versiones en GPU de la *LC* y *SSS-Index* sobre su correspondiente versión secuencial optimizada sobre la base de datos *Images* resolviendo consultas de tipo kNN.

4.3.3. 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 [32].

Para evaluar si los índices sobre GPU serían eficiente sobre este contexto, se ejecutó un experimento de productividad en función del número de consultas disponibles en paralelo para resolver consultas por rango. La figura 4.10(a) muestra los resultados para la *LC*. El eje x indica cuantas consultas son lanzadas en paralelo (empezando desde 5 consultas al mismo tiempo), mientras que el eje y muestra la productividad del sistema medido en el número de consultas procesadas por segundo. No es 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 conjunto de 200 se alcanza casi la máxima productividad.

La figura 4.10(b) muestra el speed-up para la *LC* sobre su correspondiente implementación OpenMP. La barra etiquetada como *Batch Ilimitado* corresponde al speed-up de la *LC* en la figura 4.7. Las barras etiquetadas como *Batch=30* y *Batch=100* corresponden con el escenario donde tan pronto como se tienen 30 (o 100) consultas en el sistema, se lanza un *kernel* para resolverlas. En este experimento (se usaron 23831 consultas), esta estrategia implica 795 (239 en el caso de *Batch=100*) invocaciones de *kernels*. Aún si la productividad no está en el punto más alto, las versiones en GPU siempre superan las versiones en OpenMP. Cabe destacar que en este experimento la implementación OpenMP asume el mejor caso, es decir, se conocen todas las consultas desde el comienzo.

Un speed-up más alto que 5x (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.

4.4. Conclusiones

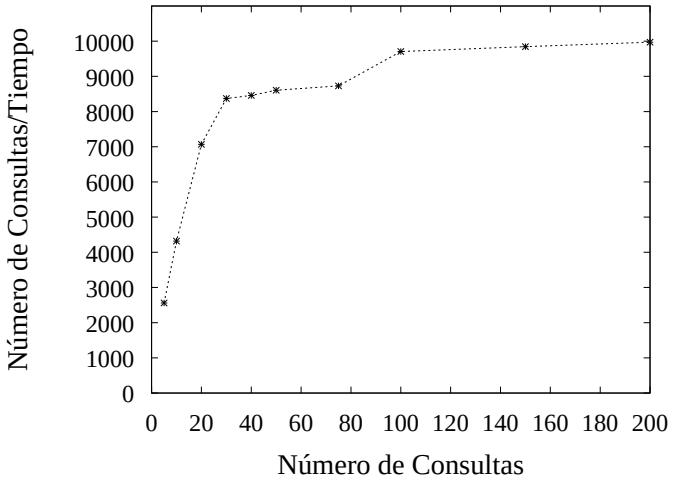
En este capítulo, se han propuestos y comparado algoritmos basados en CUDA sobre GPU para resolver consultas en espacios métricos. Se utilizaron los índices LC y *SSS-Index* debido a que almacenan su índice en matrices, lo que es una característica conveniente al utilizar una GPU. Los parámetros óptimos para ambos índices son muy diferentes, en particular, con el *SSS-Index* el óptimo se alcanza utilizando sólo un pivote, lo que muestra que este índice y su algoritmo de selección de pivotes es ineficiente sobre GPU, pues este único pivote es elegido aleatoriamente.

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, ambos índices utilizan un conjunto de heaps en *device memory* y una reducción de éstos a *shared memory* para obtener los K elementos resultados. También en este tipo de consultas el método de rango creciente presenta un mejor rendimiento que el decreciente.

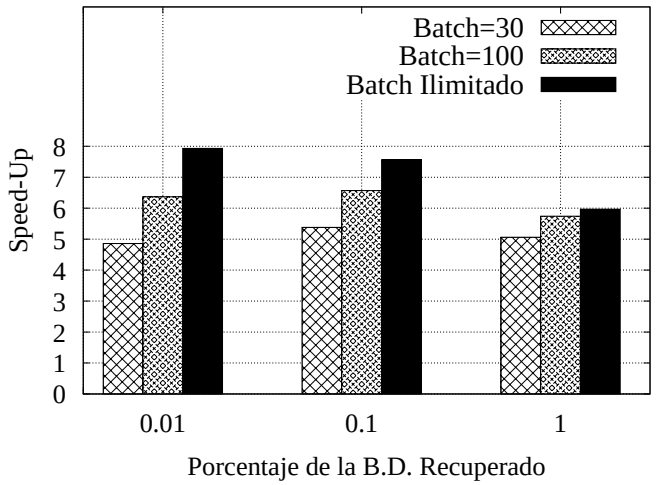
En ambos tipos de consultas, la LC presentó el mejor rendimiento, debido a que presenta un buen alineamiento en los accesos a memoria y regularidad en la ejecución de instrucciones por parte de threads consecutivos. Este índice muestra un speed-up de hasta 29.5x sobre su versión secuencial optimizada, y de 7.9x sobre su versión OpenMP.

Se comparó un algoritmo de ordenamiento basado en papers anteriores con el índice LC, en donde este último obtiene una amplia superioridad en tiempo de ejecución.

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.



(a) Productividad resolviendo distintos números de consultas



(b) Speed-Up sobre la implementación OpenMP utilizando distinto número de batch.

Figura 4.10: **a)** Productividad (número de consultas dividido por su correspondiente tiempo de ejecución) resolviendo distintos números de consultas de la *LC* sobre la base de datos *Images*. **b)** Speed-Up de la *LC* resolviendo un conjunto de consultas (de 30 y 100) al mismo tiempo sobre la correspondiente implementación OpenMP con la base de datos *Images*.

Capítulo 5

Conclusiones

En el presente trabajo se han desarrollado algoritmos sobre 2 plataformas paralelas distintas. La primera de ellas, una plataforma multi-core de 2 CPU, 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 el Capítulo 3 (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 el Capítulo 4 (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 uno de ellos son muy diferentes, en particular, con el *SSS-Index* el óptimo se alcanza utilizando sólo un pivote, lo que muestra que este índice y su algoritmo de selección de pivotes es ineficiente sobre GPU, pues este único pivote es elegido aleatoriamente. En el mismo capítulo, 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, ambos índices utilizan un conjunto de heaps en *device memory* y una reducción de éstos a *shared memory* para obtener los K elementos resultados. También en este tipo de consultas el método de rango creciente presenta un mejor rendimiento que el decreciente.

El índice LC es el que mejor rendimiento muestra sobre ambas plataformas alcanzando un speed-up de 7.8x en la plataforma multi-core y 29.5x en GPU sobre la versión secuencial. La implementación en GPU de este índice alcanza un speed-up de hasta 7.9x sobre la implementación en la plataforma multi-core.

Bajo un 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.

5.1. Trabajos futuros

- En el área de búsqueda en plataforma multi-cores:
  - Diseñar estrategias específicas para algún índice, que explote las características de éste, 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 búsqueda en 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.
  - Diseñar políticas de escritura de resultados para aplicaciones que requieran mayor información de los elementos encontrados.
  - Resolver consultas de mayor dimensionalidad probando si en este tipo de plataforma es eficiente la utilización de índices frente a una implementación de fuerza bruta que tiene la ventaja de poseer mayor uniformidad en la ejecución de instrucciones.

Bibliografía

[1] Cuda: Compute unified device architecture. In *©2007 NVIDIA Corporation*.

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

[3] Sergei Brin. Near neighbor search in large metric spaces. In *the 21st VLDB Conference*, pages 574–584. Morgan Kaufmann Publishers, 1995.

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

[5] N. R. Brisaboa, O. Pedreira, D. Seco, R. Solar, and R. Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *SOFSEM*, volume 4910 of *Lecture Notes in Computer Science*, pages 186–197. Springer, 2008.

[6] Benjamin Bustos, Oliver Deussen, Stefan Hiller, and Daniel Keim. A graphics hardware accelerated algorithm for nearest neighbor search. In *Computational Science (ICCS)*, volume 3994, pages 196–199. Springer, 2006.

[7] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:1.4–1.24, 2009.

[8] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.

[9] 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)*, pages 38–46. IEEE CS Press, 1999.

[10] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[11] 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)*, pages 75–86. IEEE CS Press, 2000.

[12] E. Chávez and G. Navarro. Measuring the dimensionality of general metric spaces. *Department of Computer Science, University of Chile, Tech. Rep. TR/DCC-00-1*, 2000.

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

[14] 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*, pages 426–435, 1997.

[15] V. Gil Costa and M. Marin. Distributed sparse spatial selection indexes. In *PDP*, pages 440–444. IEEE Computer Society, 2008.

[16] V. Gil Costa, M. Marin, and N. Reyes. An empirical evaluation of a distributed clustering-based index for metric space databases. In *ICDE Workshops*, pages 386–393. IEEE Computer Society, 2008.

[17] Veronica Gil Costa, Mauricio Marín, and Nora Reyes. Parallel query processing on distributed clustering indexes. *Journal of Discrete Algorithms*, 7(1):3–17, 2009.

[18] F. Dehne and H. Noltemeier. Voronoi trees and clustering problems. *Informations Systems*, 12(2):171–175, 1987.

[19] Vincent Garcia, Eric Debreuve, and Michel Barlaud. Fast k nearest neighbor search using gpu. *Computer Vision and Pattern Recognition Workshop*, 0:1–6, 2008.

[20] Naga K. Govindaraju, Nikunj Raghuvanshi, Michael Henson, David Tuft, and Dinesh Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[21] J. M. Hellertein, J. F. Naughton, and A. Pfeffer. Generalized search trees for databases systems. In *21st Conference on Very Large Databases (VLDB)*, Zurich, Switzerland, 1995.

[22] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[23] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973.

[24] Quansheng Kuang and Lei Zhao. A practical gpu based knn algorithm. pages 151–155, Huangshan, China, 2009.

[25] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

[26] M. Marin. Range queries on distributed spatial approximation trees. In M. H. Hamza, editor, *Databases and Applications*, pages 140–144. IASTED/ACTA Press, 2005.

[27] M. Marin and V. Gil Costa. (sync—async)$^{+}$ mpi search engines. In Franck Cappello, Thomas Hérault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 117–124. Springer, 2007.

[28] M. Marin, V. Gil Costa, and Carolina Bonacic. A search engine index for multimedia content. In Emilio Luque, Tomàs Margalef, and Domingo Benitez, editors, *Euro-Par*, volume 5168 of *Lecture Notes in Computer Science*, pages 866–875. Springer, 2008.

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

[30] Mauricio Marín, Veronica Gil Costa, and Cecilia Hernández. Dynamic p2p indexing and search based on compact clustering. In *SISAP*, pages 124–131. IEEE Computer Society, 2009.

[31] Mauricio Marin, Flavio Ferrarotti, and Veronica Gil-Costa. Distributing a metric-space search index onto processors. In *Proceedings of the 2010 39th International Conference on Parallel Processing*, ICPP '10, pages 433–442, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Mauricio Marin, Veronica Gil-Costa, Carolina Bonacic, Ricardo Baeza-Yates, and Isaac D. Scherson. Sync/async parallel search for the efficient design and construction of web search engines. *Parallel Computing*, 36(4):153 – 168, 2010.

[33] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[34] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[35] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734 – 747, 2011. Selected Papers from the 2nd International Workshop on Similarity Search and Applications SISAP 2009.

[36] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(9):989–1003, 1997.

[37] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing.* O'Reilly, 1996.

[38] H. Noltemeier. Voronoi trees and applications. In *International WorkShop on Discrete Algorithms and Complexity*, pages 69–74, Fukuoka, Japan, 1989.

[39] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex schemes of geometric object. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203, Springer-Verlag 1992.

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

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

[42] 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*, pages 1–10. IEEE, 2009.

[43] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10, 2009.

[44] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, pages 40:175–179, 1991.

[45] R. Uribe. Manipulación de estructuras métricas en memoria secundaria. Master's thesis, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Santiago, Chile, Abril 2005.

[46] R. Uribe, G. Navarro, R. Barrientos, and M. Marin. An index data structure for searching in metric space databases. In Vassil N. Alexandrov, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science (1)*, volume 3991 of *Lecture Notes in Computer Science*, pages 611–617. Springer, 2006.

[47] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[48] E. Vidal. An algorithm for finding nearest neighbor in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.

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