PROCESAMIENTO DE CONSULTAS kNN EN ESPACIOS MÉTRICOS UTILIZANDO GPUs

Ricardo Javier Barrientos Rojel

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Trabajo Fin de Máster en Ingeniería de Computadores
Curso Académico 2010-2011

Directores:

José I. Gómez
Christian Tenllado

Calificación: 9/SOBRESALIENTE

Junio 2011

Autorización de difusión

Ricardo Javier Barrientos Rojel

Junio 2011

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "PROCESAMIENTO DE CONSULTAS kNN EN ESPACIOS MÉTRICOS UTILIZANDO GPUs", realizado durante el curso académico 2010-2011 bajo la dirección de José I. Gómez y Christian Tenllado en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen en castellano

El presente trabajo pretende abordar el análisis, diseño e implementación paralela de algoritmos de búsquedas en espacios métricos. Más concretamente, se analizaron varios índices métricos relevantes en la literatura y se escogieron los que mejor se adaptaban a la tarjetas gráficas actuales (GPUs), plataforma paralela sobre la que se llevó a cabo su implementación.

Dadas, por un lado las altas prestaciones de las modernas GPUs y, por otro, las numerosas restricciones sobre la regularidad del código para conseguir un buen rendimiento, se optó por implementar, con carácter comparativo, una versión exhaustiva de la búsqueda, mejorando propuestas previas. La regularidad del método exhaustivo en los accesos a memoria, clave para explotar correctamente el alto potencial de las GPUs, hace que, en determinados contextos, sea preferible a técnicas algorítmicamente más complejas pero con mayores *asimetrías* en su paralelización.

Los resultados obtenidos permiten afirmar que las GPUs son una plataforma adecuada para la búsqueda en espacios métricos, incluso en contextos de poca carga (es decir, baja frecuencia de llegada de solicitudes). También se realizó un análisis del efecto de la dimensionalidad del espacio en la eficacia de los métodos. Como cabía esperar, a mayor dimensionalidad, más complicado es reducir el número de comparaciones y, por consiguiente, el método exhaustivo se postula como una buena alternativa en términos de rendimiento.

Palabras clave

- kNN
- Espacios métricos
- Bases de datos métricas
- Índices métricos
- GPU
- Búsqueda por similitud

Abstract

This work was devoted to propose and implement algorithms using GPUs to solve queries in metric spaces. We have selected several well known indexes from the literature that solve queries efficiently in metric spaces in a sequential way.

We compared the implementations of those index-based algorithms on GPU against exhaustive search methods. The latter is based in previous work of the area, but we have significantly improved its performance. Performance results are analyzed and discussed to show the paramount importance of bandwidth optimization while coding for GPUs-like parallel platforms.

The influence of the space dimensionality leads to the an interesting finding: on low-dimensional spaces, the indexing methods perform better, while in high-dimensional spaces the exhaustive search method takes advantage. Finally, we can conclude that GPUs are a very good target platform for metric space searching algorithms.

Keywords

- kNN
- Metric spaces
- Metric databases
- Metric indexes
- GPU
- Similarity search

Índice general

Índice I

Agradecimientos II

1. Introducción 1
1.1. Objetivos . . . 2
1.2. Organización de la tesis . . . 3

2. Conocimientos Básicos 4
2.1. Espacios métricos . . . 4
2.1.1. Búsquedas por similitud . . . 4
2.1.2. Indexación . . . 6
2.1.3. Indices métricos . . . 9
2.2. Dimensionalidad . . . 14
2.3. Modelo de programación CUDA . . . 14
2.3.1. Organización y ejecución de threads . . . 16
2.3.2. Reducción de lecturas a memoria . . . 17
2.3.3. Funciones atómicas . . . 17
2.3.4. Ejemplos . . . 18
2.4. Trabajo relacionado . . . 19

3. Estrategias de Distribución y Búsqueda sobre GPU 23
3.1. Búsqueda exhaustiva . . . 24
3.1.1. Solución basada en ordenamiento . . . 24
3.1.2. Reducción basada en Heaps . . . 25
3.2. Búsqueda mediante índices métricos . . . 27
3.2.1. Lista de Clusters (LC) . . . 29
3.2.2. *SSS-Index* . . . 32

4. Resultados experimentales 39
4.1. Experimentos de implementaciones de búsqueda exhaustiva . . . 40
4.2. Resultados experimentales usando indexación . . . 41
4.2.1. Solución de consultas en-línea . . . 46

5. Conclusiones 49
5.1. Contribuciones . . . 50

Bibliografía 51

Agradecimientos

Muy gratamente escribo esta sección para recordar a las personas que contribuyeron en la realizazión de esta tesis.

Por supuesto, mis padres, quienes han sido siempre mi mayor apoyo en mi formación como profesional. De ellos recibí todo el apoyo y confianza necesaria para emprender mis estudios de postgrado en un país tan lejano, y a la vez muy acogedor, como es España.

Gratamente, puedo decir que son varios los profesores que me han ayudado a recorrer el largo camino que significa el ser investigador, y específicamente en la realización de esta tesis. Comenzaré por el profesor Mauricio Marin, por quien siento un profundo aprecio, muchas gracias por todo su tiempo y preocupación en mi persona y formación. Al profesor Francisco Tirado, quien depositó su confianza en mí, ayudándome a la obtención de una beca ministerial que me ha permitido enfocarme en la investigación sin inconvenientes externos. Al profesor Manuel Prieto, de quien siempre he recibido mucha amabilidad, apoyo y guía en mi trabajo. Al profesor Christian Tenllado, quien siempre ha mostrado la mejor disposición a colaborar en mi trabajo de investigación. Al profesor Roberto Uribe, de quien he recibido apoyo y ánimo constante desde el pregrado hasta ahora. Al profesor José I. Gómez, por quien siento un profundo aprecio y también una gran admiración, muchas gracias por estar siempre pendiente de mi trabajo y ayudarme para que éste salga adelante, en usted he encontrado, además de un excelente académico, a un gran amigo.

A todos mis compañeros del grupo ArTeCS, quienes han hecho mucho más grata mi estadía.

También agradezco formalmente al Miniterio de Ciencia e Innovación de España por la beca otorgada a mi persona.

Ricardo J. Barrientos, Junio 2011.

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 tienen 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 y 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 los K resultados más relevantes para una consulta formulada en texto. Esta consulta se puede clasificar de tipo kNN (k Nearest Neighbors), pues retorna los k elementos más cercanos a la consulta dada.

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 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. Se proponen, implementan y evaluan algoritmos de búsqueda utilizando GPUs para resolver consultas de tipo kNN. Se utilizaron distintas bases de datos, y dependiendo de la dimensionalidad intrínseca del espacio resulta conveniente utilizar algoritmos de indexación o de búsqueda exhaustiva.

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.

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 GPU.

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.
- Evaluar el rendimiento de las estrategias desarrolladas sobre procesadores gráficos GPU, utilizando un conjunto de bases de datos de naturaleza distinta respecto de las funciones de distancia entre objetos.
- Dichas estrategias deben ser eficientes en el manejo de operaciones de lectura/escritura, haciendo un buen uso de *shared memory*.

1.2. Organización de la tesis

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

- En el Capítulo 2 se introduce a las áreas de espacios métricos, sistemas distribuidos, el modelo de programación CUDA de NVIDIA, y los trabajos previos relevantes sobre estas áreas.
- En el Capítulo 3 se describen los algoritmos desarrollados sobre GPU (Graphic Processor Unit). Se detallan los métodos usados para implementar los índices seleccionados para este tipo de plataforma. Así como también las dificultades y diferencias con los métodos clásicos en CPU.
- En el Capítulo 4 se muestran los resultados experimentales obtenidos. Se comparan y evalúan los métodos propuestos entre ellos. También ee realiza una comparación contra la correspondiente versión secuencial. Finalmente, se realizan experimentos para evaluar el comportamiento de los algoritmos propuestos bajo un sistema de tiempo real en-línea.
- En el Capítulo 5 se muestran las conclusiones finales del presente trabajo.

Capítulo 2

Conocimientos Básicos

La similitud se modela en muchos casos interesantes a través de un *espacio métrico* [32, 40], y la búsqueda de objetos más similares a través de una búsqueda por rango o de vecinos más cercanos. En el presente capítulo se definen formalmente los conceptos referentes a búsquedas en espacios métricos. También se muestra la arquitectura de la GPU utilizada y su modo de ejecución haciendo énfasis en los puntos importantes a tener en cuenta al momento de programar sobre este tipo de arquitectura.

2.1. Espacios métricos

Un *espacio métrico* es un conjunto X de objetos válidos, con una función de distancia $d : X^2 \rightarrow \mathbb{R}$, 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*.

2.1.1. 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 [12]:

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 2.1 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 grafica el rango necesario para encerrar los 5 puntos. Entonces, se observa que dada una consulta q y una cantidad k (en este ejemplo 5), es posible que existan distintas respuestas.

En [12] se ilustran varios métodos para resolver consultas de tipo kNN, siendo éstos:

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.

Figura 2.1: *Ejemplos de consultas, por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos en* $\mathbb{R}^2$.

2.1.2. Indexación

Una manera trivial de responder a consultas por similitud es examinar exhaustivamente la base de datos Y, lo que implica un tiempo $O(n)$ para una base de datos con n objetos. Considerando que la función de distancia es *computacionalmente costosa* de calcular, en general no se resuelve 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 en el momento de resolver las búsquedas.

Un *algoritmo de indexación* es un proceso que construye una estructura de datos denominada *índice*, que permite realizar búsquedas por similitud evitando, en lo posible, tener que revisar toda la base de datos, ahorrándose evaluaciones de distancia en el momento de realizar las consultas. Todos los algoritmos de búsquedas en espacios métricos utilizan la desigualdad triangular para descartar objetos 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:

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

En muchas aplicaciones la evaluación de la distancia es tan costosa que las demás com-

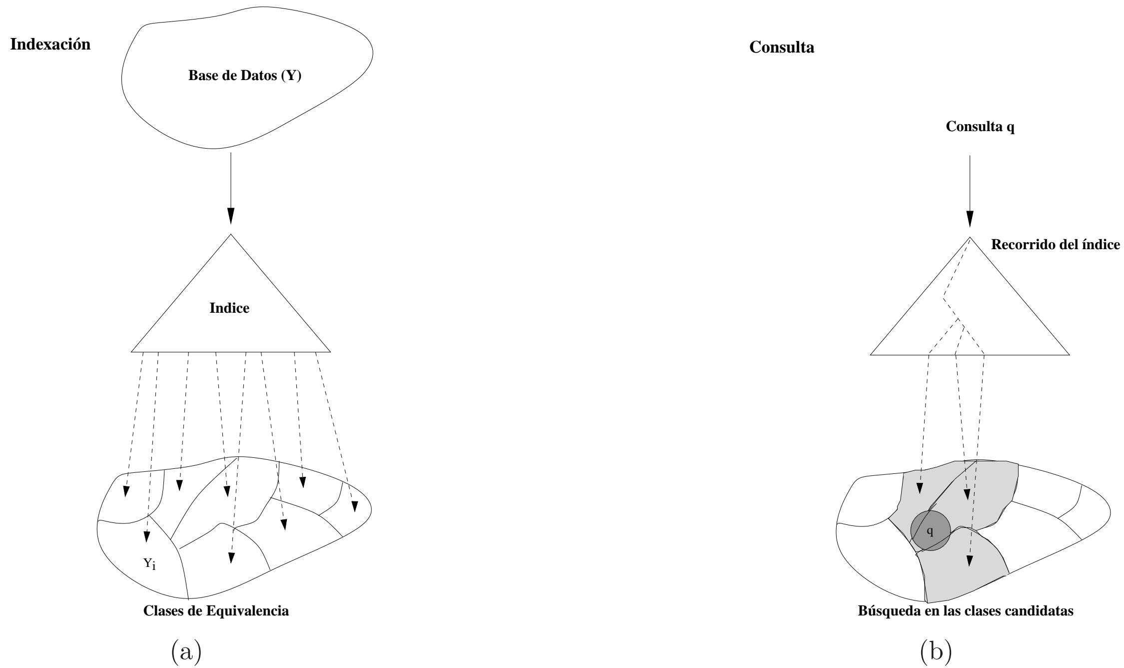


Figura 2.2: *Modelo general para algoritmos de indexación.* **a)***: construcción del índice.* **b)***: consulta sobre el índice.*

ponentes 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 (figura 2.2(a)). 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 (figura 2.2(b)) y luego se examina cada uno de ellos (de 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* [12].

Los algoritmos basados en pivotes realizan una preselección de objetos de la base de datos. Dichos 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. Lo anterior se describe en las dos siguientes etapas:

- 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* [39], *LAESA* [25], *Spaghettis* y sus variantes [8, 28], *FQT* y sus variantes [2] y *FQA* [9].

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 comparando únicamente 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* [27][37]. Otras sólo utilizan radio cobertor, como son los *M-trees* [13] y la *Lista de Clusters* [10]. Algunas que utilizan hiperplanos son GHT y sus variantes [36, 30] y los Voronoi-trees [14, 29].

2.1.3. Indices métricos

Son varios los índices métricos que existen actualmente en la literatura técnica. En el presente trabajo se seleccionaron dos de ellos: *Lista de Clusters* (LC) y *SSS-Index*, debido a que (i) ambos índices han sido ampliamente referenciados y (ii) ambos almacenan su índice en matrices, y por lo mencionado en la Sección 2.3 esto favorece al rendimiento en GPU, pues la localidad espacial de los datos es un factor de gran importancia en este tipo de plataforma. A continuación se describe cada uno de ellos.

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

1: {Sea D la base de datos}
2: {Sea M la distancia máxima posible entre 2 elementos de D}
3: {Sea α una constante real}
4: $PIVOTES \leftarrow \{x_1\}$
5: **for all** $x_i \in D$ **do**
6:     **if** $\forall p \in PIVOTES, d(x_i, p) >= M\alpha$ **then**
7:         $PIVOTES \leftarrow PIVOTES \bigcup \{x_i\}$
8:     **end if**
9: **end for**

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 1. El resultado de éste es un conjunto de pivotes que se encuentran al menos a distancia $M\alpha$ unos de otros (M=distancia máxima entre 2 elementos de la base de datos; α=parámetro real). De esta forma, el índice será una tabla de distancias entre cada pivote y los elementos restantes. El valor óptimo de α observado en [4] se alcanza utilizando valores alrededor de $\alpha = 0{,}4$.

El algoritmo 2 muestra la búsqueda por rango utilizando el SSS-Index, el que consta de los siguientes pasos:

- Se calculan las distancias entre cada pivote y la consulta (línea 6).
- Utilizando la tabla de distancias entre pivotes y elementos más las distancias calculadas en el paso anterior, se intenta descartar por desigualdad triangular cada elemento de la base de datos (línea 11).
- Si no es posible descartar un elemento, entonces se realiza una evaluación de distancia entre la consulta y el elemento no descartado (línea 17).

Lista de Clusters (*LC*)

La *Lista de Clusters* ([10]) 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.

Algoritmo 2 *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
```

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 se encuentran a una distancia inferior a r_c de c. Definamos los siguientes subconjuntos de D:

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

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

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

como los elementos externos.

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

El algoritmo de construcción se muestra en el algoritmo 3, el que retorna una lista (c_i, r_i, I_i) (centro, radio, bucket). 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

Algoritmo 3 *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)

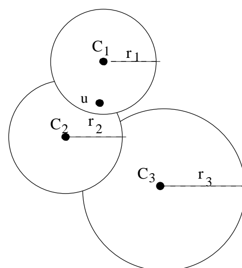


Figura 2.3: Lista de Cluster*: Zonas abarcadas por 3 centros según este orden:* c_1, c_2, c_3.

del primer centro son almacenados sólo en su cluster I, a pesar de que haya intersección con los cluster siguientes (figura 2.3). La selección de centros consiste en dos pasos: (i) el primer elemento de la colección D es elegido como centro, y (ii) los demás centros $c \in D$ serán los elementos que maximicen 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 4 y la figura 2.4 muestra 3 casos posibles de búsqueda dado un cluster (c, r_c). Para la consulta $q1$ (figura 2.4(a)) es necesario buscar sobre el cluster y continuar la búsqueda en el resto de la lista de clusters. Para la consulta $q2$ (figura 2.4(b)) es posible podar la búsqueda debido a que la consulta está completamente contenida dentro del cluster, y para $q3$ (figura 2.4(c)) es posible evitar la búsqueda sobre el cluster por la propiedad de *desigualdad triangular*.

Algoritmo 4 *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** $dist \leq rc + r$ **then**
11:     Buscar en I exhaustivamente
12: **end if**
13: **if** $dist \geq rc - r$ **then**
14:     busquedarango(E, q, r)
15: **end if**

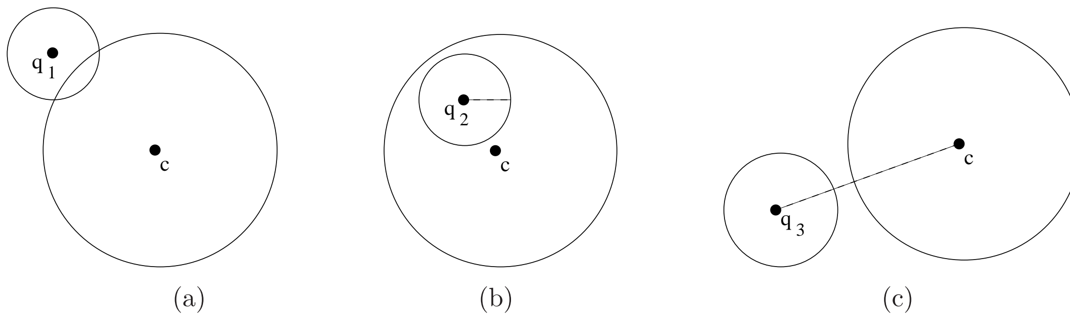


Figura 2.4: Lista de Cluster*: Tres casos de búsqueda.*

2.2. Dimensionalidad

Uno de los mayores obstáculos para el diseño de técnicas de búsqueda eficientes en espacios métricos es la existencia de *espacios de alta dimensión*. Muchas de las técnicas de indexación tradicionales para espacios vectoriales tienen una dependencia exponencial con la dimensión de representación 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 [12] y [11] 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 aproximadamente a la misma distancia entre sí. Por lo tanto, pueden existir espacios cuya dimensión de representación es alta, pero su dimensión intrínseca es baja.

2.3. Modelo de programación CUDA

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

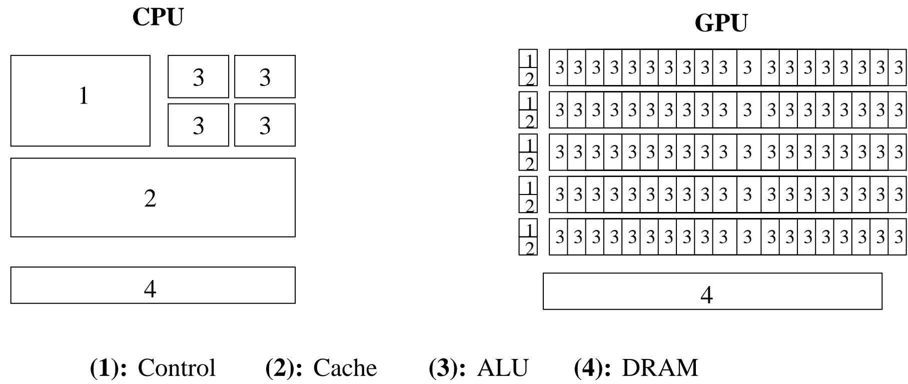


Figura 2.5: *Diferencia esquemática entre una CPU y una GPU.*

del *kernel*.

La figura 2.5 muestra un esquema de la diferencia entre una CPU y una GPU. Esta última está especializada para cómputo intensivo, y por lo tanto más transistores están dedicados para el procesamiento de datos en vez de caching de datos y control de flujo. Esto implica que la GPU es capaz de realizar un mayor número de operaciones aritméticas en punto flotante que una CPU.

El correcto aprovechamiento del sistema de memoria en la GPU es primordial. La figura 2.6 muestra un esquema del hardware de una GPU, la que consta de las siguientes características:

- Una GPU está constituida por una serie de *multiprocesadores*, donde cada uno de éstos cuenta con varios núcleos.
- Cada núcleo cuenta con sus propios registros y todos los núcleos del mismo multiprocesador comparten una *shared memory*
- *Shared memory* es una memoria que se encuentra próxima a los núcleos del multiprocesador, y por lo tanto es de muy baja latencia.
- Cada multiprocesador cuenta con un par de memorias caché compartidas por todos los núcleos, *constant cache* y *texture cache*.

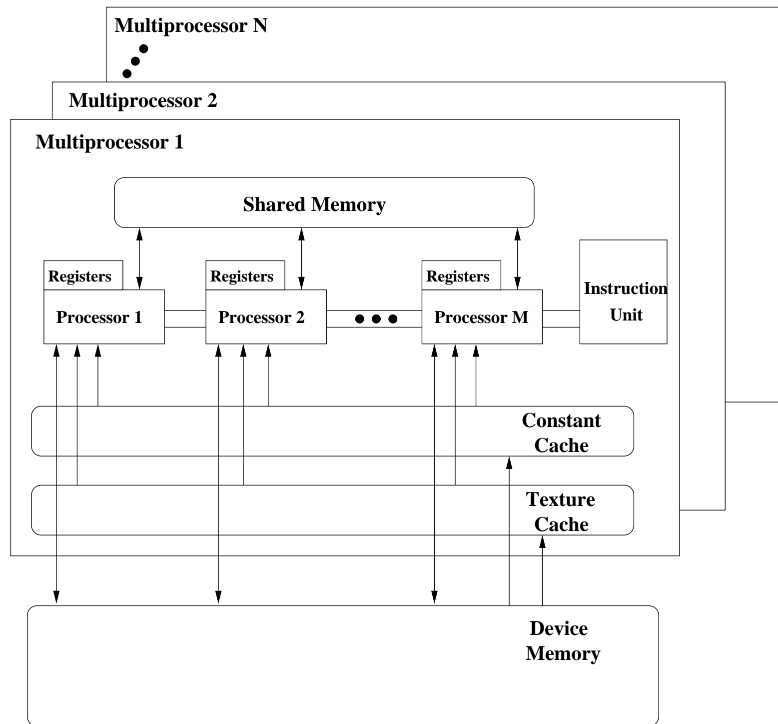


Figura 2.6: *Arquitectura de hardware en una GPU.*

- *Constant cache* es una caché de datos de la *constant memory*, que es una memoria de sólo lectura de reducido tamaño alojada en *device memory*.
- *Texture cache* es una caché de la *texture memory*, que es una memoria optimizada para localidad espacial 2D, y está alojada en *device memory*.
- *Device memory* es una memoria compartida por todos los multiprocesadores, y es de gran tamaño.

2.3.1. Organización y ejecución de threads

Los threads son organizados en *CUDA Blocks*, y cada uno de estos se ejecuta completamente en un único multiprocesador. Esto permite que todos los threads de un mismo CUDA Block compartan la misma *shared memory*.

Un *kernel* puede ser ejecutado por un número limitado de CUDA Blocks, y cada CUDA Block por un número acotado de threads. Sólo los threads que pertenecen al mismo CUDA Block pueden ser sincronizados. Es decir, hilos que pertenezcan a CUDA Blocks distintos sólo pueden ser sincronizados realizando un nuevo lanzamiento de kernel.

Para soportar la gran cantidad de threads en ejecución se emplea una arquitectura SIMT (Single-Instruction, Multiple-Thread). La unidad de ejecución no es un thread, sino un *warp*, que es un conjunto de threads, y un *half-warp* corresponde a la primera o segunda mitad del conjunto de threads de un *warp*. La forma en que un CUDA Block es dividido en *warps* es siempre la misma, el primer conjunto de threads representan el primer *warp*, el segundo conjunto consecutivo de threads el segundo *warp* y así sucesivamente.

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 de salto 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).

2.3.2. 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.

Por lo tanto, si los threads de un *half-warp* acceden a n diferentes segmentos, entonces se ejecutarán n transacciones de memoria. Si es posible reducir el tamaño de la transacción, entonces se lee sólo la mitad de ésta.

2.3.3. Funciones atómicas

Existe un grupo de funciones atómicas que realizan operaciones de tipo *read-modify-write* sobre palabras de 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.

2.3.4. Ejemplos

La figura 2.7 muestra un código que suma 2 vectores utilizando 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 CUDA Block (dado

por la variable *N_BLOQUES*) y 200 threads por cada CUDA Block (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 CUDA Block), T_{Block} (que retorna la cantidad de threads de un CUDA Block) y ID_{Block} (que retorna el identificador del CUDA Block). 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 2.8, donde en el código 2.8(a) se deben realizar instrucciones de sincronización para garantizar el correcto valor del arreglo $result$, mientras que el código 2.8(b) es realizado solamente por los threads de un *warp*, y debido a esto no es necesario utilizar sincronización.

2.4. Trabajo relacionado

Las publicaciones que dan una solución al problema kNN utilizando GPU son [20], [15] y [6]. Todos estos trabajos abordan el caso en que los elementos de la base de datos poseen una gran dimensión (con una gran dimensión intrínseca), lo que implica poder descartar muy pocos elementos (usando desigualdad triangular), y en donde la búsqueda secuencial es competitiva frente a usar indexación.

En [20] 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 2.9). 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 CUDA Block distinto, y para esto cada CUDA Block transfiere a *shared memory* submatrices de A y B para poder

```
#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 = IDThread + ( TBlock *IDBlock );
    /* 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 2.7: *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 2.8: *Ejemplos para ilustrar la sincronización de los threads de un* warp.

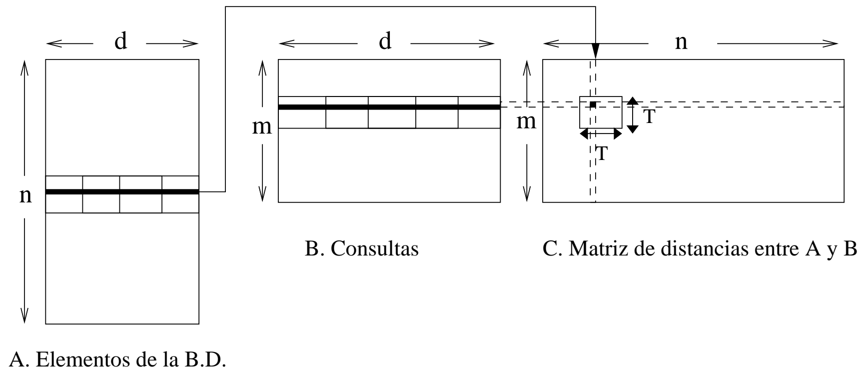


Figura 2.9: *Particionamiento de las matrices de datos y consultas en submatrices.*

escribir las distancias resultantes en la submatriz correspondiente. Esto implica que cada CUDA Block 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 CUDA Blocks necesarios es $(m/T) \times (n/T)$. Luego, teniendo la matriz de distancias resultante, ésta se ordena usando *CUDA-based Radix Sort* ([34]) para posteriormente seleccionar los K primeros elementos como resultado final.

En [15] 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.

En [6] se aborda solamente el caso particular de consultas kNN con $k = 1$. Esto es

un caso de menor complejidad al abordado en este trabajo (kNN con $k > 1$), basando la solución en el manejo de memorias de texturas en GPU.

Capítulo 3

Estrategias de Distribución y Búsqueda sobre GPU

En este capítulo se proponen y comparan algoritmos de búsqueda de índices métricos sobre GPU basados en CUDA. Como se mencionó anteriormente los índices seleccionados fueron *LC* y *SSS-Index*, debido a que ambos almacenan su índice en matrices, y por lo mencionado en la Sección 2.3, esto favorece al rendimiento en GPU, pues la localidad espacial de los datos es un factor de gran importancia en este tipo de plataforma.

Una novedad de nuestra propuesta es que explotamos simultáneamente dos niveles de paralelismo: (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.

En todas las implementaciones siguientes (con excepción de las mostradas en 3.1.1), cada CUDA Block 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 comunicarse y sincronizarse. La sincronización es importante porque, entre otras cosa, hace posible la comunicación entre hilos.

Cada *kernel* es lanzado con Q CUDA Blocks (Q=número de consultas a resolver) maximizando el número de threads. Si Q sobrepasa el máximo permitido de CUDA Blocks,

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*.

A continuación se describen los algoritmos de búsqueda exhaustiva, que están basados en los trabajos previos antes mencionados. Posteriormente se describen los métodos de búsqueda implementados utilizando los índices *LC* y *SSS-Index* sobre GPU.

3.1. Búsqueda exhaustiva

En esta Sección se proponen dos estrategias de búsqueda exhaustiva para abordar consultas de tipo kNN. La primera sigue la misma línea que los trabajos previos en esta área ([20] y [15]), es decir, se obtiene la distancia de cada elemento de la base de datos contra la consulta y luego se ordenan estas distancias resultantes, para posteriormente elegir los primeros K elementos como el resultado final. La segunda estrategia resuelve una consulta por cada CUDA Block, y cada CUDA Block utiliza un conjunto de *heaps* [19] para encontrar los K vecinos más cercanos a partir de las distancias entre los elementos de la base de datos y la consulta.

Ambas estrategias reciben como parámetro de entrada el array de distancias δ, donde $\delta[i]$ es la distancia entre el elemento i-ésimo elemento de la base de datos y la consulta. Para obtener este array, se ejecuta un *kernel* con todos los CUDA Blocks necesarios para que cada thread realice la evaluación de distancia entre un elemento de la base de datos y la consulta.

3.1.1. Solución basada en ordenamiento

Siguiendo la misma idea de trabajos previos (Sección 2.4), 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 array, se ejecuta un *kernel* previamente con todos los CUDA Blocks necesarios para que cada thread realice la evaluación de distancia entre un

elemento de la base de datos y la consulta.

Trabajos previos ([20, 15]) utilizan como método de ordenamiento el *radix sort* [33] e *insertion sort*, pero el método utilizado en este trabajo usa el *GPU-Quicksort* [7], que presenta mejor rendimiento que los algoritmos anteriores ([7]).

El *GPU-Quicksort* se divide en dos etapas, la primera consiste en dividir el array a ordenar en P particiones, que pueden ser ordenadas independientemente. Para esto se lanza un *kernel* con M CUDA Blocks, (M=cantidad de particiones actuales) y cada CUDA Block 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 CUDA Block se encarga de ordenar una de estas particiones usando el *quicksort* [18], pero implementando la recursión usando una pila en *shared memory*. Cuando la secuencia a ordenar es suficientemente pequeña, se usa el *bitonic sort* [17], implementado de forma nativa en CUDA.

3.1.2. Reducción basada en Heaps

En esta estrategia, cada CUDA Block se encarga de resolver una consulta por completo. Cada thread del CUDA Block i recorre de forma circular los elementos del array δ de la i-ésima consulta, manteniendo en su heap los K elementos más cercanos a la consulta. Para esto es necesario mantener en *device memory* un heap de tamaño K por cada thread. Este conjunto de heaps es almacenado en una matriz de $K \times T_{Block}$, donde T_{Block} es la cantidad de threads por CUDA Block. Cada columna de esta matriz representa un heap, de forma que la i-ésima columna tendrá los elementos correspondientes al heap del i-ésimo thread.

Una vez que un CUDA Block tiene T_{Block} heaps, se realiza una etapa de reducción. Esta etapa consiste en que el primer *warp* de cada CUDA Block recorre de forma circular los elementos de los heaps anteriores manteniendo los K elementos más cercanos a la consulta en su heap almacenado en *shared memory* esta vez. Y finalmente, en un tercera etapa el

primer thread del CUDA Block es el encargado de recorrer (de forma circular) los elementos de los heaps almacenados en *shared memory* para obtener los K resultados finales, que también son almacenados en un heap en *shared memory*.

El algoritmo de reducción de los elementos de heaps almacenados en *device memory* por parte de los thread de un *warp* se muestra en la función `reduccion_warp()` del algoritmo 5. 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 que se saca un elemento de la raíz (`pop()`) se inserta un nuevo elemento (`push(x)`), se definió la función `popush(x)`, que optimiza la acción conjunta de 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 5 muestra la búsqueda efectuada por el *kernel*, que recibe como parámetro el array de distancias δ. Este algoritmo se divide en 3 etapas delimitadas por la función de sincronización `__syncthreads()` e ilustradas por la figura 3.1. La descripción de cada etapa es la siguiente:

- En la primera etapa, se distribuyen los elementos de δ entre los threads siguiendo una distribución circular (línea 8). Cada thread almacena sus elementos en su heap almacenado en *device memory* (línea 11). Una vez que el heap tiene K elementos, se insertará un elemento sólo si éste es menor que la raíz del heap (la raíz mantiene la mayor distancia de los elementos del heap a la consulta). Al finalizar esta etapa, existen T_{Block} heaps de tamaño K (T_{Block}=cantidad de threads del CUDA Block).

- En la segunda etapa, los heaps de la etapa anterior son asignados a los threads del primer *warp* del CUDA Block según una distribución circular. Cada thread almacena los elementos (si corresponde) en su heap almacenado en *shared memory* (línea 17).

- Finalmente en la tercera etapa, el primer thread del CUDA Block recorre los elementos de los $SIZE_{Warp}$ heaps anteriores ($SIZE_{Warp}$=cantidad de threads en un *warp*) y los

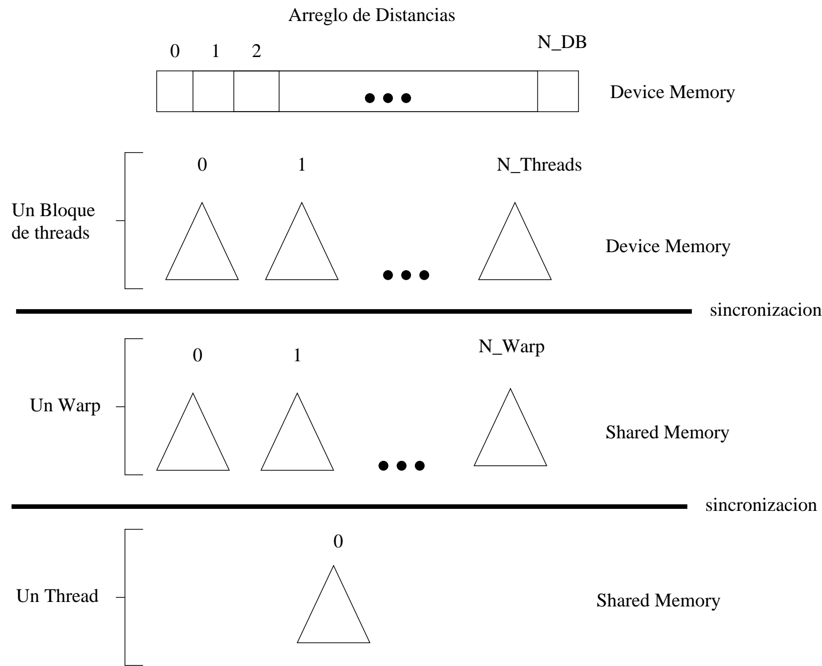


Figura 3.1: *Ilustración de las etapas del algoritmo basado en heaps para la resolución de consultas kNN.*

almacena (si corresponde) en su heap alojado en *shared memory* (línea 23). Al finalizar esta etapa, este heap es el que posee los K elementos respuesta.

Es importante resaltar que los heaps pertenecientes a threads del mismo *warp* tienen sus elementos raíz en posiciones consecutivas de memoria. De este modo, la consulta si un nuevo elemento debe incluirse en el heap se realiza favoreciendo el acceso contiguo a memoria. Sólo cuando hay divergencias, empiezan los problemas. Aún así, varios hilos del *warp* llamarán de forma simultanea a `popush` con lo que irán recorriendo cada heap de forma sincronizada, sin perjudicar demasiado el fusionado de operaciones de lectura/escritura.

3.2. Búsqueda mediante índices métricos

A continuación se describen los métodos de búsqueda propuestos utilizando los índices *LC* y *SSS-Index*. Como se mencionó anteriormente, ambos índices utilizan matrices para almacenar su información, siendo esto muy conveniente para favorecer el acceso contiguo a

Algoritmo 5 Kernel de la reducción basada en heaps.

Heap_Reduction(δ)

1: {Sea δ el array de distancias entre los elementos de la BD y la consulta}
2: {Sea DH_i el heap del thread i almacenado en *device memory*}
3: {Sea SH_i el heap del thread i almacenado en *shared memory*}
4: {Sea $SIZE_{Warp}$ el tamaño de un *warp*}
5: {Sea *tid* el identificador del thread dentro del CUDA Block}
6:
7: {Cada thread almacena los elementos en un heap}
8: **for** ($i = tid$; $i < \delta$.size(); i += T_{Block}) **do**
9:     $x.dist$=$\delta[i]$
10:     $x.ind$=i
11:     insercion_heap(DH, x)
12: **end for**
13:
14: ___syncthreads()
15:
16: {Un *warp* almacena los elementos de los heaps anteriores en $SIZE_{Warp}$ heaps en *shared memory*}
17: reduccion_warp(DH, SH);
18:
19: ___syncthreads()
20:
21: {Un thread recorre exhaustivamente SH y selecciona los primeros K elementos}
22: **if** $tid == 0$ **then**
23:     get_first_K(SH)
24: **end if**

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

memoria por parte de threads del mismo *warp*.

3.2.1. 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 6. $CENTROS$ es una matriz de $D \times SIZE_{Centros}$ (D=dimensión de los elementos, $SIZE_{Centros}$=cantidad de centros de clusters), donde cada columna representa un centro de cluster. RC es un array de tamaño $SIZE_{Centros}$ 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.

Como se mencionó anteriormente, los datos se disponen pro columnas para favorecer el acceso contiguo a memoria (Sección 2.3.2).

El método comúnmente usado por índices métricos para resolver una consulta de tipo kNN es un método decreciente (según lo mencionado en 2.1.1). Este método fue descartado al momento de utilizar GPU, debido a que todos los threads de un CUDA Block 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 2.1.1) se adapta muy bien a este tipo de plataforma.

El método comúnmente usado por índices métricos para resolver una consulta de tipo kNN es un método de *rango decreciente*, sin embargo, como se observa a continuación un método *de rango creciente* es una mejor alternativa.

La figura 3.2 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 circunstancias: (i) tras 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()`, debido a que hay varios threads haciendo comparaciones en paralelo y cada uno puede creer haber encontrado un valor mínimo. 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. Esto se debe principalmente a que los threads de un CUDA Block 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 CUDA Block), lo que no permite reducir el rango suficientemente rápido como lo haría el 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 CUDA Block resuelve una consulta completamente), la carga de trabajo de cada CUDA Block 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 CUDA Blocks realizarán la misma cantidad de trabajo en el primer *kernel*, mientras que en el segundo sólo habrá una diferencia con aquellos CUDA Blocks que resuelvan consultas que requieran $I \geq 3$ (estas consultas representan un porcentaje menor), y al ser menor la cantidad de CUDA Blocks 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

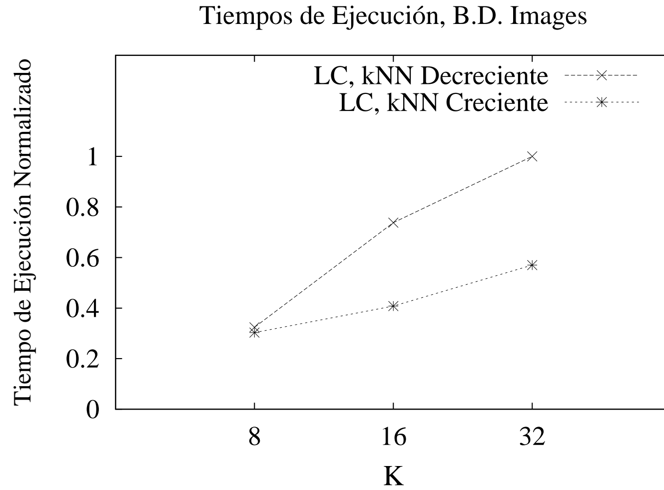


Figura 3.2: *Tiempo de ejecución normalizado sobre la base de datos* Images *de un método de rango creciente y decreciente para resolver consulta de tipo kNN usando la* LC *sobre GPU.*

evaluaciones de distancia realizadas en una iteración anterior, debido a que esto incrementa la divergencia de threads del mismo *warp* y también la irregularidad en el patrón de acceso a memoria, empeorando el rendimiento.

Al igual que la implementación basada en heaps, también se utilizó un heap por cada thread para mantener los K elementos más cercanos a la consulta hasta el momento. Posteriormente, se utiliza el mismo método de reducción a heaps en *shared memory* por parte de los threads del primer *warp* del CUDA Block. Finalmente, el primer thread del CUDA Block es el encargado de recorrer los heaps almacenados en *shared memory* para entregar los K elementos resultado.

El algoritmo 6 muestra el pseudo-código del *kernel* usado por la *LC* para resolver consultas de tipo kNN usando un CUDA Block. 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 a resolver en *sha-*

red memory (línea 8). Previamente se establece el rango inicial, usando la función `rango_inicial()`, que toma en cuenta a qué lanzamiento de *kernel* corresponde.

- En la segunda etapa, cada thread (siguiendo una distribución circular) 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 los thread siguiendo una distribución circular. Cada elemento que pertenece a un cluster que no puede ser descartado usando desigualdad triangular (línea 22) es comparado contra la consulta (línea 23). Y si el elemento está dentro del rango actual de búsqueda, se inserta en el heap del thread almacenado en *device memory* (línea 25). Luego, los centros son distribuidos (circularmente) 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 CUDA Block 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 41).

- Finalmente en la última, etapa el primer thread de cada CUDA Block se encarga de recorrer y almacenar los elementos de los $SIZE_{Warp}$ heaps de la etapa anterior en *shared memory* (línea 45). 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*.

3.2.2. *SSS-Index*

Este índice se representó por 3 matrices denominadas *PIVOTES*, *DISTANCIAS* y *BD* en el algoritmo 7. *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 $SIZE_{piv} \times SIZE_{BD}$

Algoritmo 6 *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 *Query)
 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] = Query[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)≤ minC; i+=T_Block) do
20:       indC = i/B_Size
21:       rc = RC[indC]
22:       if distC[indC] ≤ 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])≤ rango then
35:          x.ind=i
36:          insercion_heap(DH_tid, x)
37:       end if
38:    end for
39:    ___syncthreads()
40:    {Un warp almacena los elementos de los heaps anteriores en SIZE_Warp heaps en shared memory}
41:    reduccion_warp(DH, SH)
42:    ___syncthreads()
43:    {Un thread recorre exhaustivamente SH y selecciona los primeros K elementos}
44:    if tid == 0 then
45:       get_first_K(SH)
46:       cantidad_resultados = get_first_K(SH)
47:       condicion = evalua_condicion(cantidad_resultados, query)
48:    end if
49:    ___syncthreads()
50: end while
```

($SIZE_{BD}$=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.

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. Como en los métodos anteriores, cada CUDA Block se encarga de resolver una consulta completamente.

Los elementos de la base de datos se asignan entre los threads siguiendo una distribución circular, 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 7 muestra el *kernel* usado por el *SSS-Index* para resolver consultas de tipo kNN usando un CUDA Block. 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 a resolver en *shared memory* (línea 5).
- En la segunda etapa, cada thread (siguiendo una distribución circular) obtiene la distancia entre un pivote y la consulta, y la almacena en *shared memory* (línea 11).
- En la tercera etapa, cada elemento de la matriz *DISTANCIAS* es asignado a un thread siguiendo una distribución circular. 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. 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 CUDA Block 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*.

El número de pivotes que se escogen para indexar la base de datos tiene un gran impacto en el rendimiento. La cantidad de pivotes depende del parámetro $\alpha \in \mathbb{R}$ $(0 < \alpha \leq 1)$ descrito en la Sección 2.1.3. 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 3.3, 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 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. Todos los valores fueron normalizados al mayor valor observado en el experimento.

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

Esto significa que realizar menos evaluaciones de distancia no compensa el costo causado por la divergencia de los *warps* y la irregularidad en los accesos a memoria.

Las particularidades de la tarjeta gráfica hacen que otras estrategias de paralelización, viables en entornos más convencionales, no sean viables en la GPU, como es el caso de la siguiente estrategia. Esta consistió en distribuir circularmente los pivotes (en lugar de

Algoritmo 7 *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 *Query, 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] = Query[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
```

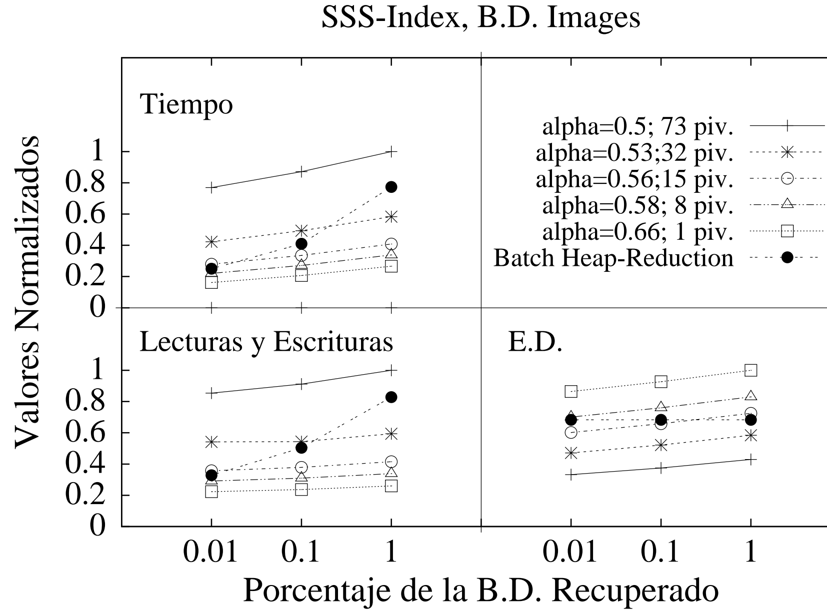


Figura 3.3: *Valores normalizados del tiempos de ejecución, cantidad de lecturas/escrituras (de 32, 64 o 128 bytes) a* device memory *y del promedio de evaluaciones de distancia por consulta del* SSS-Index *sobre GPU para la base de datos* Images*.*

los elementos de la base de datos) 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, una vez que se evalúan E elementos, se ejecuta una instrucción de sincronización y los elementos no descartados son distribuidos circularmente entre los threads y cada uno de éstos realiza la evaluación de distancia entre la consulta y sus elementos asignados. La tabla 3.1 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*.

Cuadro 3.1: *Tiempos de ejecución (en segundos) y cantidad de lecturas/escrituras a* device memory.

Tiempo de Ejecución (segs.)		
K	***Distribución-Pivote***	***Distribución-Elemento***
8	131,0	3,2
16	144,5	4,3
32	160,8	6,0
Cantidad de Lecturas/Escrituras (x10000)		
K	***Distribución-Pivote***	***Distribución-Elemento***
8	697574	27031
16	743240	29627
32	822808	33129

Los resultados muestran un mal rendimiento para la estrategia *Distribución-Pivote*, alcanzando un tiempo de ejecución de hasta 40.9 veces más que *Distribución-Elemento* con $K = 8$. Esto se explica debido a la gran cantidad de lecturas/escrituras realizadas, debido a la gran irregularidad en las instrucciones de accesos a *device memory*. Esta irregularidad se explica porque algunos threads (del mismo *warp*) pueden descartar ciertos elementos, mientras que otros no, y mientras mayor sea la cantidad de elementos examinados, mayor será esta divergencia en la secuencia de instrucciones de estos threads. Para la base de datos *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*.

Capítulo 4

Resultados experimentales

La GPU utilizada fue una NVIDIA GeForce GTX 280 (*Compute Capability 1.3*), con 30 multiprocesadores, 8 núcleos por multiprocesador y 16KB de *shared memory*. El tamaño de *device memory* es de 4GB. Los experimentos sobre las versiones secuenciales fueron ejecutados sobre una máquina con las características descritas en la tabla 4.1.

Los gráficos de lectura/escritura mostrados en este capítulo representan la suma total de estas operaciones. Las operaciones de lectura (o escritura) a *device memory* son de 32 bytes, 64 o 128 bytes, pero estas dos últimas son operaciones de 32 bytes fusionadas. Estos gráficos fueron agregados debido a que las operaciones de lecturas/escrituras en GPU poseen una gran latencia (Sección 2.3.1).

Se medirá el rendimiento de cada implementación y se mostrará la estrecha relación entre accesos a memoria y rendimiento.

El presentar experimentos con valores normalizados, es con la finalidad de apreciar mejor las diferencias entre los diferentes métodos propuestos.

Este capítulo está organizado como sigue. En la Sección 4.1 se muestran los experimentos comparando los métodos de búsqueda exhaustiva. Luego, en la Sección 4.2 se comparan los métodos de indexación y el método de búsqueda exhaustiva que presentó mejor rendimiento.

Cuadro 4.1: *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

4.1. Experimentos de implementaciones de búsqueda exhaustiva

En esta sección se comparan los métodos de búsqueda exhaustiva descritas en 3.1. Se utilizó una base de datos de alta dimensión, y los valores de K utilizados fueron $K = 8$, 16 y 32, pues estos son valores usados en trabajos previos [20, 15, 6]. K es el parámetro que indica la cantidad de resultados que se deben recuperar en la consulta kNN.

El método basado en ordenamiento, descrito en 3.1.1, fue denominado *Ordering Reduction*. El método basado en heaps, descrito en 3.1.2, fue denominado *Heap-Reduction*. El mismo método basado en heaps, pero resolviendo un conjunto de Q consultas en cada lanzamiento de *kernel* se denominó *Batch Heap-Reduction*, y cada CUDA Block resuelve una consulta distinta. El valor de Q está limitado por el espacio necesario en *device memory* para almacenar los heaps de cada thread y el arreglo de distancia δ de cada consulta a resolver. La base de datos utilizada se denominó *Faces*, que se describe a continuación:

- ***Faces*** **:** Esta base de datos fue creada a partir de una colección de 8480 vectores que representan imágenes de rostros obtenidas de *Face Recognition Grand Challenge* [31]. Se aplicó el método GaborPCA para obtener una representación *Eigen Faces* [35] de dimensión menor de dichas imágenes. Estos vectores se usaron como una distribución de probabilidades para generar vectores aleatorios hasta completar 120000 imágenes

de rostros de dimensión 254. Se usó la *distancia euclidiana* para medir la similitud entre los objetos. El 80 % de la base de datos se usó para la construcción del índice y el 20 % restante para el archivo de consultas.

La figura 4.1 muestra el tiempo de ejecución normalizado para los tres métodos. Este experimento se realizó sobre la base de datos *Faces* variando el número de elementos en la base de datos.

El método *Batch Heap-Reduction* es el que obtiene el mejor tiempo de ejecución, pero mientras mayor es la cantidad de datos, éste tiende a igualarse con *Heap-Reduction*. Esto es debido a que mientras mayor sea el tamaño de la base de datos, mayor es la cantidad de espacio necesitado por *Batch Heap-Reduction* en *device memory* para almacenar los arreglos de distancias y los heaps necesarios. Y por lo tanto, Q decrece mientras mayor sea la cantidad de elementos.

La figura 4.2 muestra el tiempo real de ejecución para los tres métodos. Este gráfico muestra que no hay una diferencia para *Ordering Reduction* al variar K, pues el único momento en donde se utiliza este parámetro es cuando se escogen los primeros K elementos del arreglo de distancias ya ordenado como resultado final. Este gráfico también muestra la diferencia significativa de 36.85 % (en promedio) usando distintos K en *Heap-Reduction*, en cambio *Batch Heap-Reduction* presenta un 4.9 % en promedio de diferencia al variar los valores de K. Esto es debido principalmente a que *Batch Heap-Reduction* consigue ocultar mejor las latencias por accesos discontinuos a memoria. Esto es respaldado por el hecho que cada *kernel* es lanzado con un conjunto de Q CUDA Blocks, mientras que en *Heap-Reduction* cada *kernel* se ejecuta con sólo 1 CUDA Block, teniendo menos probabilidad de ocultar latencias con instrucciones de threads distintos.

4.2. Resultados experimentales usando indexación

En esta sección se presentan los experimentos comparativos entre los índices *LC* y *SSS-Index* con el método *Batch Heap-Reduction*, que fue el que mejor rendimiento mostró en la

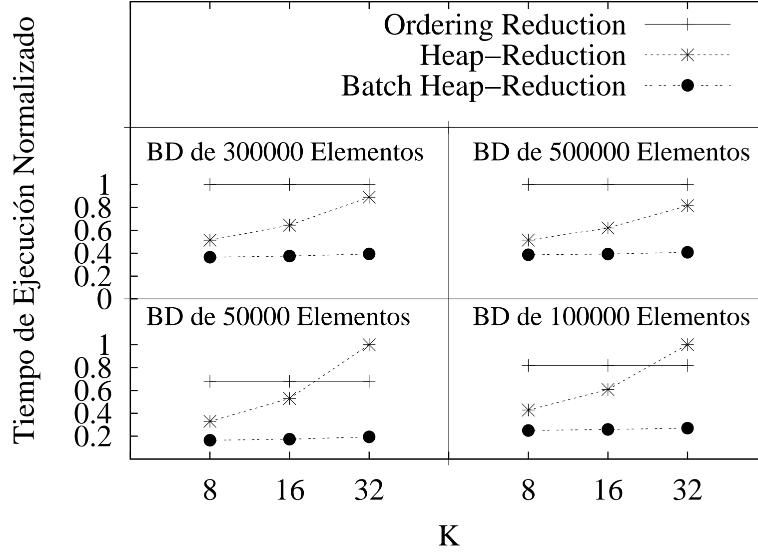


Figura 4.1: *Tiempo de ejecución normalizado de las implementaciones de* Fuerza Bruta *usando distintos K y distintos tamaños de BD.*

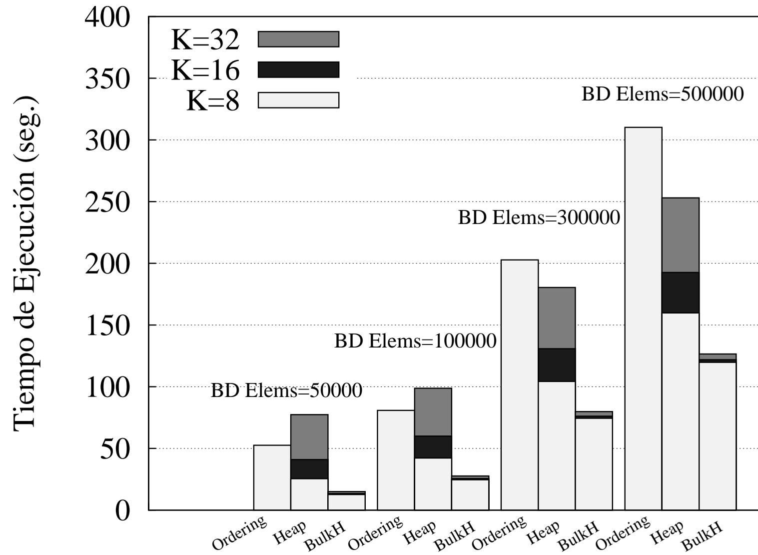


Figura 4.2: *Tiempo real de ejecución de las implementaciones de* Fuerza Bruta *usando distintos K y distintos tamaños de BD.*

Sección 4.1.

Además de la base de datos *Faces*, se incluyó a los experimentos una base de datos de vectores de baja dimensión y una base de datos de palabras. A continuación se describen ambas.

- ***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 rangos de búsqueda 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 [10, 27, 26]. El 80 % de la BD se usó para la construcción del índice y el 20 % restante para el archivo de consultas.

- ***Spanish*** **:** Diccionario español con 51589 palabras. Se usó la *distancia de edición* (o *distancia de Levenshtein*) [21] con radio 1, 2 y 3, pues éstos son radios usados en trabajos previos [27, 22, 26]. Esta distancia entrega la cantidad mínima de inserciones, eliminaciones o reemplazos para que una palabra sea igual a otra. Las consultas para esta base de datos fue un archivo de 40000 consultas, obtenidas desde la Web Chilena en el dominio todo.cl.

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*, para la base de datos *Images* se seleccionó $B_{Size} = 64$, y para *Spanish* $B_{Size} = 32$, donde B_{Size} es la cantidad de elementos en un cluster. En el caso del *SSS-Index*, para la base de datos *Images* se seleccionó la versión mostrada en la Sección 3.2.2, 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 4.3 muestra tres gráficos correspondientes a: (i) el promedio de evaluaciones de

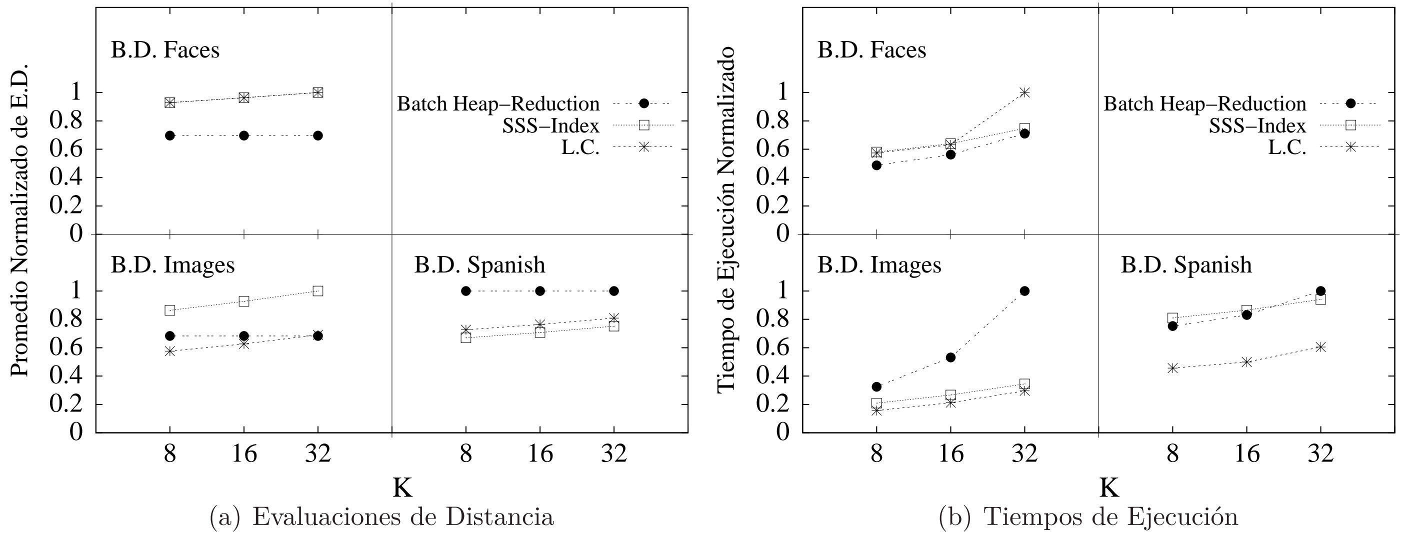
distancia por consulta, (ii) el tiempo de ejecución, y (iii) la cantidad de lecturas/escrituras a *device memory*. Todos los valores están normalizados al mayor valor observado en el experimento.

Con respecto al número de evaluaciones de distancia (figura 4.3(a)), en la base de datos *Spanish* el comportamiento es el esperado, es decir, los métodos de indexado logran reducir la cantidad de evaluaciones de distancia. Pero en la base de datos *Images* el método *Batch Heap-Reduction* presenta un mejor rendimiento que el *SSS-Index* por utilizar este último un método de rango creciente con sólo 1 pivote (Sección 3.2.2), y en *Faces* el método *Batch Heap-Reduction* supera a ambos índices. Esto último es debido a que en la base de datos *Faces*, que es de alta dimensionalidad, los índices métricos no son capaces de reducir el número de evaluaciones de distancia [12], y al usar un método de rango creciente algunas evaluaciones de distancia se realizan más de una vez, lo que agrega un *overhead* reflejado en este gráfico.

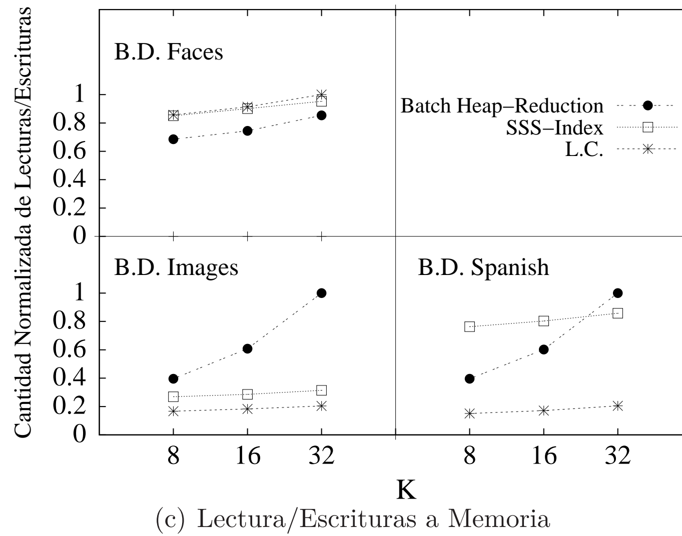
Sin embargo, al observar el comportamiento de los métodos en el gráfico de evaluaciones de distancia, éste no coincide con el gráfico de tiempos de ejecución (figura 4.3(b)) como sí ocurre en trabajos previos ([16, 38, 23]). Esto se explica con el gráfico de lecturas/escrituras (figura 4.3(c)), pues el uso eficiente del ancho de banda determina en gran medida el rendimiento final. Cabe destacar que las instrucciones de escritura sólo se realizan al acceder a los heaps almacenados en *device memory*.

El alineamiento de acceso a memoria por parte de threads consecutivos sobre este tipo de plataforma es un factor muy relevante en el rendimiento. Como se mencionó en la Sección 2.3 cuando un *warp* realiza accesos no consecutivos en memoria, el hardware no es capaz de fusionar dichos accesos pudiendo llegar a realizar 32 accesos a memoria en vez de un acceso fusionado.

El método *Batch Heap-Reduction* realiza menos evaluaciones de distancia que el *SSS-Index*, pero ejecuta más operaciones de lectura/escritura (sobre la base de datos *Images*). Esto se debe a la gran cantidad de operaciones de escritura del método *Batch Heap-Reduction*,



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



(c) Lectura/Escrituras a Memoria

Figura 4.3: *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 métodos* Batch Heap-Reduction, SSS-Index *y* LC *sobre GPU.*

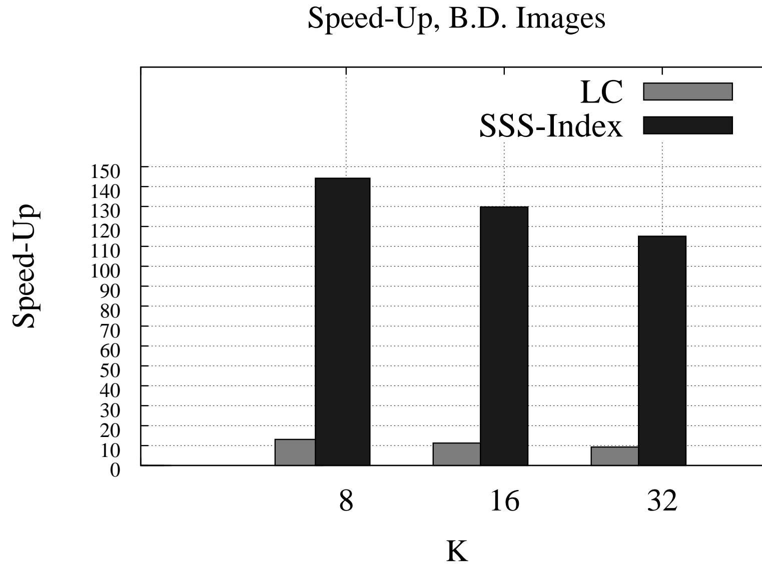


Figura 4.4: *Speed-Up de las versiones en GPU de la* LC *y* SSS-Index *sobre su correspondiente versión secuencial optimizada sobre la base de datos* Images.

pues éste intenta insertar todos los elementos del array de distancias δ en los heaps almacenados en *device memory*. Pero el *SSS-Index* sólo intenta insertar un elemento x en el heap si x no se ha podido descartar, y además x debe estar dentro del rango de búsqueda actual.

La figura 4.4 muestra el speed-up de los índices *LC* y *SSS-Index* sobre su correspondiente versión secuencial optimizada. Se alcanza un speed-up de hasta 13x en la *LC* y 144.2x para el *SSS-Index*.

4.2.1. Solución de consultas en-línea

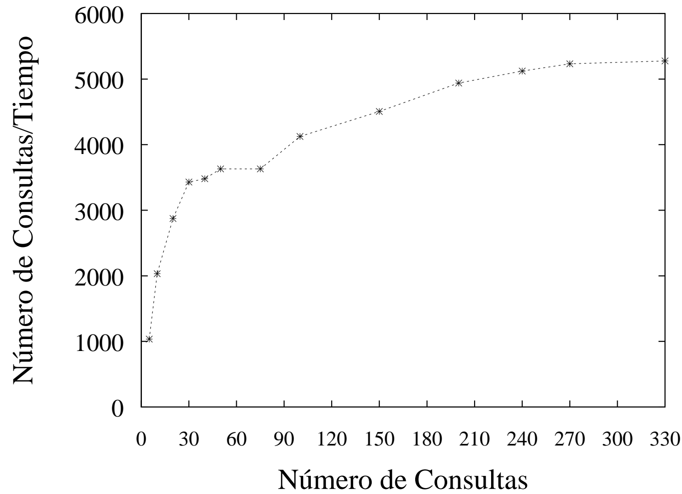
Los experimentos en GPU hasta ahora han sido asumiendo un alto tráfico de consultas, es decir, el conjunto de consultas por resolver es conocido con anticipación. Pero esta asunción puede ser inasequible en sistemas de tiempo real en-línea, como motores de búsqueda web [24].

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

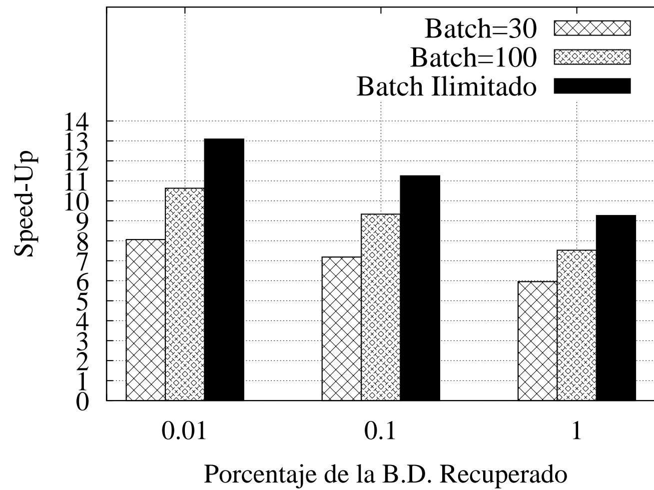
para resolver consultas por rango. La figura 4.5(a) muestra los resultados para la *LC*. El eje x indica cuantas consultas son lanzadas en paralelo (empezando desde 5 consultas al mismo tiempo), mientras que el eje y muestra la productividad del sistema medido en el número de consultas procesadas por 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 4.5(b) muestra el speed-up para la *LC* resolviendo las consultas por lotes, tal como se explica a continuación. La barra etiquetada como *Batch Ilimitado* corresponde al speed-up de la *LC* en la figura 4.4. 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*.

Un speed-up más alto que 7x (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.



(a) Productividad resolviendo distintos números de consultas



(b) Speed-Up sobre la versión secuencial utilizando distinto número de batch.

Figura 4.5: **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 versión secuencial optimizada.*

Capítulo 5

Conclusiones

En el presenta trabajo se han propuesto, implementado y comparado algoritmos usando 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, una característica conveniente al utilizar una tarjeta gráfica. 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.

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.

La selección del método con mejor rendimiento depende de la dimensionalidad del espacio. El índice *LC* muestra un mejor desempeño en cuanto a tiempo de ejecución sobre las bases de datos *Images* y *Spanish* logrando un speed-up de hasta 17.4x sobre su versión secuencial optimizada. Pero sobre la base de datos de alta dimensionalidad *Faces*, el método de búsqueda exhaustiva (denominado *Batch Heap-Reduction*) toma ventaja. Esto se explica por la ineficiencia por parte de los índices al intentar descartar elementos utilizando una base de datos de alta dimensionalidad.

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.

5.1. Contribuciones

El presente trabajo ha dado lugar a las siguientes publicaciones:

- “Heap-Based k-Nearest Neighbor Search on GPUs”, XXI Jornadas de Paralelismo, pages 559-566, Valencia, España, Septiembre 2010.
- “kNN Query Processing in Metric Spaces using GPUs”, In 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), Bordeaux, France, September 2011. (CORE A).

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 fixed-queries 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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

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

[16] V. Gil-Costa, R.J. Barrientos, M. Marin, and C. Bonacic. Scheduling metric-space queries processing on multi-core processors. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010)*, pages 187–194, Pisa, Italy, February 2010. IEEE Computer Society.

[17] 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.

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

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

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

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

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

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

[30] 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.

[31] P.J. Phillips, P.J. Flynn, T. Scruggs, K.W. Bowyer, J. Chang, K. Hoffman, J. Marques, J. Min, and W. Worek. Overview of the face recognition grand challenge. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005. CVPR 2005*, pages 947–954, 2005.

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

[33] 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.

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

[35] M. Turk and A. Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

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

[37] R. Uribe. Manipulaci'on de estructuras m'etricas en memoria secundaria. Master's thesis, Facultad de Ciencias F'isicas y Matem'aticas, Universidad de Chile, Santiago, Chile, Abril 2005.

[38] R. Uribe, G. Navarro, R.J. Barrientos, and M. Marin. An index data structure for searching in metric space databases. In *6th International Conference on Computational Science (ICCS 2006)*, volume 3991 of *Lecture Notes in Computer Science*, pages 611–617, Reading, UK, May 2006. Springer.

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

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