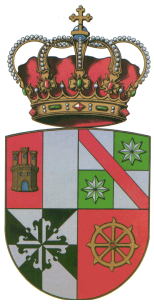
Universidad de Castilla-La Mancha

Departamento de Sistemas Informáticos



Tesis Doctoral

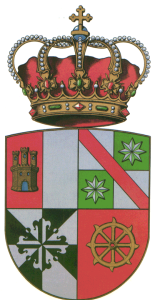
Estructuras Métricas para Búsquedas por Similitud sobre Arquitecturas Heterogéneas Basadas en GPUs

Julio 2013

Roberto Uribe-Paredes

Universidad de Castilla-La Mancha

Departamento de Sistemas Informáticos



Tesis Doctoral

Estructuras Métricas para Búsquedas por Similitud sobre Arquitecturas Heterogéneas Basadas en GPUs

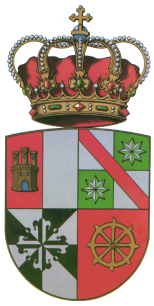
Julio 2013

Programa Oficial de Doctorado en Tecnologías Informáticas Avanzadas.
Disertación para optar al grado de Doctor en Ciencias de la Computación para ser presentada con la aprobación del Departamento de Sistemas Informáticos de la Universidad de Castilla-La Mancha, para su examen y debate público.

Autor : **Roberto Uribe-Paredes**

Directores : **Dr. D. Enrique Arias Antúnez**
Dr. D. Diego Cazorla López

University of Castilla-La Mancha
Computing Systems Department



PhD. Thesis Dissertation

Metric Structures for Similarity Search Supported by GPU-based Heterogeneous Architectures

July 2013

A dissertation for the degree of Doctor of Philosophy in Computer Science to be presented with due permission of the Computing Systems Department, for public examination and debate.

Author : Roberto Uribe-Paredes

Advisors : Dr. D. Enrique Arias Antúnez
Dr. D. Diego Cazorla López

“No es nada” - me dijo el *hombre consciente* -
“es sólo una gota de rocío,... pero,... igual moja”.

ru-p.2013./

A Adolena y Manuel

A Renato, Atenea y Marlies

Agradecimientos

Esta es la última página que escribo de mi Tesis y llegado a este punto, al parecer, se hace dificil mencionar o dar gracias a todos aquellos que han aportado al desarrollo y a la finalización de este trabajo.

Son demasiadas personas a quienes debería agradecer, pero aún a riesgo de no recordar a todos, igualmente agradezco:

A mis hijos Atenea y Renato y a mi esposa Marlies, por soportar, una vez más, aquellas largas ausencias de casa. Por estar conmigo en cada momento de mi vida y haber compartido juntos nuestra gran aventura por Europa. También agradezco a mis padres Adolena y Manuel y a mi hermano Nino por su apoyo incondicional, a mis amigos Claudia, Pancho, Eliana, Rodrigo y Olga por estar permanentemente preocupados de nuestro bienestar.

A mis compañeros del I3A, José Antonio, Maite, Raul, Gabino, Enrique, María, Nuria, Julio, Jesús, Edo y Javier por hacer de mi espacio de trabajo un lugar más agradable (con nuestras tertulias y también con nuestros partidos de paddle).

A Ricardo Barrientos Rojel, por su permanente motivación en aportar con cosas interesantes mi trabajo.

A mis directores de Tesis Enrique Arias y Diego Cazorla, por el apoyo constante y por la paciencia de muchas veces tener que hacer las cosas a mi manera (hasta el último día). A José Luis Sánchez por sus aportes siempre acertados y su permanente disposición a colaborar cada vez que lo necesitaba. A las familias de Kike y Blanca, de Diego, y de Luis y Fernanda, a todos ellos por recibirnos con los brazos abiertos.

Un agradecimiento especial a mi colega Eduardo Peña por su permanente colaboración, tanto en lo profesional como en lo personal.

Quiero agradecer formalmente a la Universidad de Magallanes representada en su rector Dr. Victor Fajardo Morales, al Vicerrector Dr. Juan Oyarzo Pérez y al decano de la Facultad de Ingeniería Dr. Humberto Oyarzo Pérez por las facilidades entregadas para la realización de mis estudios de postgrado y su siempre buena disposición a favorecer mi perfeccionamiento académico.

Finalmente quiero agradecer a la Universidad de Magallanes, a la Universidad de Castilla-La Mancha, al Instituto de Investigación en Informática de Albacete y Fundación Carolina, por el apoyo económico y/o el facilita-

miento de las instalaciones durante la realización de mi estadía en España. Al proyecto SATSIM (Ref: CGL2010-20787-C02-02) dirigido por el Dr. Fernando Cuartero y a la red "Red de computación de altas prestaciones sobre arquitecturas paralelas heterogéneas", CAPAP-H3 (TIN2010-12011-E), ambos proyectos del Ministerio de Ciencia e Innovación de España, por el apoyo en el financiamiento de distintas actividades durante el desarrollo de mi doctorado.

A todos gracias.

Roberto Uribe-Paredes. Albacete, España, Julio 2013.

Resumen

La *búsqueda por similitud* consiste en recuperar todos aquellos objetos dentro de una base de datos que sean parecidos o relevantes a una determinada consulta. Este concepto tiene una amplia gama de aplicaciones en áreas como bases de datos multimedia, reconocimiento de patrones, minería de datos, recuperación de información, etc. La búsqueda por similitud o en proximidad se modela matemáticamente a través de un *espacio métrico*, en el cual un objeto es representado como una caja negra, donde la única información disponible es la *función de distancia* de este objeto a los otros.

En general, el cálculo de la distancia es costoso, por ello el objetivo es reducir la cantidad de evaluaciones de distancia necesarias para resolver la consulta. Para esto existen numerosos métodos, los cuales construyen estructuras llamadas *índices*, pagando un costo en preprocesamiento de los datos a fin de disminuir las evaluaciones de distancia durante la búsqueda.

En la actualidad, la necesidad de procesar grandes volúmenes de datos hace poco factible la utilización de una estructura en aplicaciones reales si ésta no considera su implementación en entornos paralelos o la utilización de sistemas de altas prestaciones. Existe una serie de tecnologías para realizar implementaciones paralelas, siendo una de las más nuevas y menos exploradas, las plataformas basadas en GPU (*Graphics Processing Units*). Estas plataformas resultan ser interesantes debido a su tasa costo/beneficio, siendo de muy bajo coste pero teniendo un enorme potencial para explotar paralelismo a nivel de datos. En general, las GPUs vienen implantadas en sistemas clásicos multicore, por lo que en conjunto la plataforma es llamada heterogénea, al contener dispositivos multicore y many-core (GPUs).

El presente trabajo de Tesis Doctoral desarrolla, analiza y evalúa la implementación de estructuras métricas y sus algoritmos sobre plataformas heterogéneas permitiendo la utilización de todos los recursos disponibles en la plataforma subyacente en la resolución de consultas por rango. Considera adicionalmente diversos aspectos en las diferentes soluciones presentadas, como son las estrategias de planificación para la distribución de consultas, estrategias para la transferencia de datos en función de los distintos niveles de la jerarquía de memoria y análisis del consumo energético de las distintas propuestas.

Índice general

Agradecimientos I

Resumen III

1. Introducción **1**
1.1. Relevancia y Aportes de la Tesis 4
1.2. Organización de la Tesis 6

2. Marco Teórico **9**
2.1. Espacios Métricos 9
2.2. Búsquedas por Similitud 9
2.3. Indexación 11
2.4. Ejemplos de Espacios Métricos 12
2.4.1. Espacios Vectoriales 12
2.4.2. Diccionarios de Palabras 13
2.4.3. Modelo Vectorial para Documentos 14
2.5. Dimensionalidad y Espacios Métricos 15
2.6. Algoritmos de Indexación 16
2.6.1. Modelo General de Indexación 16
2.6.2. Algoritmos Basados en Pivotes 18
2.6.3. Algoritmos Basados en Clustering o Particiones Compactas 20
2.7. Estructuras Métricas y Dinamismo 21
2.8. Estructuras Métricas y Memoria Secundaria 21
2.9. Entornos Paralelos 23
2.9.1. Modelos de Programación 23
2.9.2. Memoria Distribuida 24
2.9.3. Memoria Compartida 25
2.9.4. Clusters Multicores 28
2.9.5. Paralelización de Estructuras Métricas 28
2.9.6. GPUs y Espacios Métricos 30

3. Estructuras Métricas **33**
3.1. *GNAT Evolutivo (Evolutionary GNAT)* 33
3.2. *Lista de Clusters* 36
3.3. *Generic Metric Structure* (*GMS*) 39
3.4. *Spaghettis* 41
3.5. *Sparse Spatial Selection (SSS)* 43
3.6. *LAESA* 45
3.7. Comentarios 45

4. Plataforma Experimental **47**
4.1. Espacios Métricos 47
4.2. Plataforma Computacional 49

5. Implementación Basada en GPU **53**
5.1. Hacia una Implementación sobre GPUs 54
5.1.1. Resultados Preliminares 55
5.1.2. Discusión 61
5.2. GMS Basada en GPU 61
5.2.1. Memoria Compartida (*Shared Memory*) 63
5.2.2. Búsqueda Exhaustiva Basada en GPU 65
5.2.3. Resultados Experimentales y Discusión 66
5.3. Ventajas de una Plataforma basada en Multi-GPU 70
5.4. Conclusiones 72

6. Implementaciones Heterogéneas **75**
6.1. Plataformas Multicore y Many-core 75
6.1.1. Implementación Multicore 76
6.1.2. Implementación GPU 77
6.1.3. Resultados y Discusión Inicial 80
6.2. Implementaciones Heterogéneas 81
6.3. Versión 1: Una Consulta - Un Kernel 83
6.3.1. Tiempos de Ejecución 84
6.3.2. Distribución de los Tiempos de Ejecución y de las Consultas 87
6.3.3. Escalando el Número de Consultas 89
6.3.4. Discusión 95
6.4. Versión 2: *MQ* Consultas - 1 Kernel 97
6.4.1. Tiempo de Ejecución 98
6.4.2. Distribución de los Tiempos de Ejecución y de las Consultas 101
6.4.3. Escalando el Número de Consultas 102
6.4.4. Discusión Respecto de la Versión Orientada a Bloques 106
6.4.5. Escalando la Base de Datos 108
6.5. Conclusiones 109

7. Temas Relacionados **113**
7.1. Jerarquía de Memoria en una Plataforma Heterogénea 113
7.1.1. Transfiriendo Datos desde Memoria Secundaria a Memoria Principal 114
7.1.2. Desde la Memoria Principal hasta la Memoria Global de la GPU 121
7.1.3. Hacia el Solapamiento de Procesos entre la CPU y GPU 122
7.1.4. Discusión 124
7.2. Esquemas de Planificación sobre una Plataforma Heterogénea 125
7.2.1. Análisis de Esquemas de Planificación 125
7.2.2. Planificación **Estático II** 127
7.2.3. Discusión 130
7.3. Consumo Energético 131
7.3.1. Prueba Experimental 1 132
7.3.2. Prueba Experimental 2 134
7.3.3. Discusión 135

8. Conclusiones **137**
8.1. Discusión Final 137
8.2. Aspectos Relevantes y Aportes 139
8.3. Trabajos Futuros 140
8.4. Publicaciones 141

Índice de figuras

1.1. Sistema de recuperación de imágenes basada en contenido. . . 4

2.1. Ejemplos de consultas, por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos en $\mathbb{R}^2$ (fuente figura [26]). 10
2.2. Histogramas de distancias para espacios de baja dimensión (izquierda) y alta dimensión (derecha) (fuente figura [26]). . . 16
2.3. Modelo general para algoritmos de indexación (fuente figuras [26]). Izquierda: construcción del índice. Derecha: consulta sobre el índice. 17
2.4. Taxonomía de los algoritmos existentes para búsquedas por similitud en espacios métricos. 18

3.1. *EGNAT*: Mutación de un *nodo bucket* a un *nodo GNAT*. . . . 34
3.2. *EGNAT*: Partición del espacio, representación de subplanos en un nodo de tipo *GNAT*. 35
3.3. Descarte de subárboles usando rangos. Se descarta D_x ya que $d(q,p) + r < min_d(p, D_x)$. 37
3.4. Construcción de la *Lista de Clusters*. Indicando en (b) las zonas de influencia de 9 *Clusters* construidos según el orden : $C1$, $C2$, $C3$, $C4$, $C5$, $C6$, $C7$, $C8$ y $C9$. 38
3.5. Búsqueda sobre la *Lista de Clusters*. 38
3.6. Estructura métrica genérica y algoritmo de búsqueda. Ejemplo para una consulta q con intervalos $\{(6,10),(5,9),(2,6),(4,8)\}$ a los pivotes. 40
3.7. Representación del *Spaghettis* una vez finalizado el proceso de construcción y mostrando los intervalos obtenidos para distintas búsquedas. 43
3.8. Los objetos encerrados en círculos representan los centros seleccionados como pivotes y M se define como la distancia entre *Obj 1* y *Obj 2*, los más alejados en el espacio. 44

4.1. Arquitectura de la GPU Tesla C1060. 50

5.1. Cálculos de distancia promedio por consulta (columna izquierda) y tiempos totales de ejecución (columna derecha) para el espacio de palabras en español y las estructuras *GMS*, *SSS-Index*, *Spaghettis*, *MSD-Laesa*, *EGNAT* y *Lista de Clusters* (*LC*). 57
5.2. Cálculos de distancia promedio por consulta (izquierda) y tiempos totales de ejecución (derecha) para el espacio de histogramas de colores y las estructuras *GMS*, *SSS-Index*, *Spaghettis*, *MSD-Laesa*, *EGNAT* y *LC*. 58
5.3. Tamaño de las principales estructuras, representativas de las basadas en pivotes (*GMS* y otras) y las basadas en clustering (*EGNAT* y *LC*). 59
5.4. Tiempo de ejecución para GPU de las mejores versiones secuenciales. 68
5.5. Tiempo de ejecución de estructura *GMS* sobre una GPU para las versiones de hasta 32 pivotes y fuerza bruta. Espacio de palabras (izquierda) e histograma de colores (derecha). 69
5.6. Speed-ups para las versiones *GMS* secuenciales sobre las de GPU. Espacio de palabras (izquierda) e histograma de colores (derecha). 70
5.7. Resultados en tiempos absolutos para las versiones *GMS* sobre una plataforma de múltiples GPUs. Espacio de palabras (izquierda) e histograma de colores (derecha). 71
5.8. Speed-ups para las versiones multi-GPUs versus 1 GPU. Espacio de palabras (izquierda) e histograma de colores (derecha). 71
5.9. Speed-ups para las versiones multi-GPUs versus la mejor estructura secuencial, en este caso *Spaghettis*. Espacio de palabras (izquierda) e histograma de colores (derecha). 72

6.1. Dentro de una GPU. Pasos 1, 2 y 3 del algoritmo básico de búsqueda por rango. 79
6.2. Tiempos de ejecución para las implementaciones secuencial, multicore y GPU; para los espacios de palabras (izquierda) e histogramas de colores (derecha). 80
6.3. Diagrama general del sistema heterogéneo. 83
6.4. Tiempos de ejecución para distintos valores del parámetro *chunk* usando planificación dinámica en el espacio de palabras en español. 85
6.5. Tiempos de ejecución para distintos valores del parámetro *chunk* usando planificación dinámica en el espacio de vectores. 85
6.6. Speed-ups para un sistema heterogéneo, en el contexto de un sistema multicore y una plataforma con una GPU. 86
6.7. Añadiendo GPUs a un esquema heterogéneo, cambiando CPU-cores por GPUs. Caso de estudio diccionario en español. 87

6.8. Añadiendo GPUs a un esquema heterogéneo, cambiando CPU-cores por GPUs. Caso de estudio histogramas de colores. 87
6.9. Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 1 GPU. Caso diccionario en español. . 90
6.10. Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 1 GPU. Caso histogramas de colores. . 91
6.11. Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso diccionario en español. 92
6.12. Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso histogramas de colores. 93
6.13. Escalabilidad del sistema. Tiempo de ejecución versus tamaño del bloque escalando el número de consultas desde 10 mil a 40 mil para los rangos 1 y 4 en un sistema con 8 cores y 4 GPUs (espacio de palabras en español con log de 40 mil consultas). 94
6.14. Escalabilidad del sistema. Tiempo de ejecución versus tamaño del bloque escalando el número de consultas desde 100 mil a 900 mil para los rangos que recuperan $0,01\,\%$ y $1\,\%$ en un sistema con 8 cores y 4 GPUs (espacio de histogramas de colores escalado). 94
6.15. Añadiendo GPUs a la segunda versión del esquema heterogéneo. Caso de estudio espacio de palabras en español. . . . 99
6.16. Comparación de resultados para las versiones 1 y 2. Caso espacio de palabras en español. 100
6.17. Detalles del comportamiento de las distintas versiones para bloques de tamaño reducido, usando los valores reales del tamaño de bloque: 1 - 1q, 8 - 0,1 %, 86 - 1 %, 172 - 2 %, 215 - 2,5 %. Caso espacio de palabras en español. 100
6.18. Añadiendo GPUs a la segunda versión del esquema heterogéneo. Caso de estudio histogramas de colores. 101
6.19. Comparación de resultados para las versiones 1 y 2. Caso histogramas de colores. 101
6.20. Detalles del comportamiento de las distintas versiones para bloques de tamaño reducido, usando los valores reales del tamaño de bloque: 1 - 1q, 11 - 0,1 %, 112 - 1 %, 225 - 2 %, 281 - 2,5 %. Caso histogramas de colores. 102
6.21. Versión 2: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso diccionario en español. 103

6.22. Versión 2: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso histogramas de colores. 104
6.23. Versión 2: Escalabilidad del sistema. Tiempos de ejecución versus tamaño del bloque cuando el número de consultas varía de 10 mil a 40 mil considerando los rangos 1 y 4 en un sistema con 8 cores y 4 GPUs para el espacio de palabras en español. 105
6.24. Versión 2: Escalabilidad del sistema. Tiempos de ejecución versus tamaño del bloque cuando el número de consultas varía de 100 mil a 900 mil considerando los rangos que recuperan el 0,01 % y 1 % de los datos, en un sistema con 8 cores y 4 GPUs para el espacio de vectores. 105
6.25. Versión 2: Escalando en número de consultas de 10 mil a 40 mil para el espacio de palabras en español y rangos 1 y 4 en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa $chunk = 1$) y versión 2 con $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$. 106
6.26. Versión 2: Escalando en número de consultas de 100 mil a 900 mil para el espacio de vectores y porcentajes de recuperación de 0,01 % y 1 % en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa $chunk = 1$) y versión 2 con $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$. 107
6.27. Versión 2: Escalando la Base de datos 1 millón de objetos para el histograma de colores (izquierda) y la nueva base de datos de COPHIR (derecha) en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa $chunk = 1$) y versión 2 con $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$. 109

7.1. Sistema heterogéneo completo (con una GPU) y jerarquía de memoria asociada. 116
7.2. Evaluaciones de distancia (promedio por consulta) y tiempo total de ejecución para las estructuras de datos *GMS*, *EGNAT* y *LC* para el espacio palabras (izquierda) y el espacio de histogramas de colores (derecha). 119
7.3. Tiempos de ejecución para los histogramas de colores considerando 0,01 % (arriba) y 1 % (abajo) de recuperación de la información de la base de datos para 1.144 (derecha), 5.455 (centro) y 10.756 (izquierda) páginas cargadas en memoria principal. 121
7.4. Tiempos de ejecución para el diccionadio en español considerando rango=1 (arriba) y rango=4 (abajo). Espacio de almacenamiento en RAM equivalente a 200 (izquierda), 628 (centro) y 1.840 (derecha) páginas. 122

7.5. Tiempos de ejecución para el espacio de histogramas de colores, considerando rangos de recuperación del 0,01 % (arriba) y 1 % (abajo). Además, se incluyen los procesos de búsqueda para el *GMS*, *GMS* sobre GPU (*GMSonGPU*) y considerando sólo los tiempos de accesos (*only accesses*. 123
7.6. Tiempos de ejecución para el espacio de palabras en español, considerando rangos de 1 (arriba) y 4 (abajo). Además, se incluyen los procesos de búsqueda para el *GMS*, *GMS* sobre GPU (*GMSonGPU*) y considerando solo los tiempos de accesos (*only accesses*. 124
7.7. Tiempo de ejecución para el caso de diccionario de Español para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**. 128
7.8. Tiempo de ejecución para el caso de histogramas de color para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**. 128
7.9. Tiempo de ejecución para el caso del espacio de rostros para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**. 129
7.10. Speed-ups para el sistema heterogéneo. 131
7.11. Estudio de potencia consumida en watts para el caso de diccionario de Español e histogramas de color para la prueba presentada en 6.3 con plataformas: 1 core, 1 GPU, multicore (8 cores), heterogénea (híbrida de 8 cores con 1 GPU). 132
7.12. Estudio de potencia consumida en watts para el caso de diccionario de Español, histogramas de color y rostros, considerando 8 cores (GPU 0 %), 4 GPUs (GPU 100 %) y la plataforma heterogénea (4 cores + 4 GPUs) al 90 % de consultas asignadas a las GPUs. 135

Índice de tablas

2.1. Estructuras de datos para búsquedas en espacios métricos. . . 19

4.1. Especificaciones técnicas de la GPU Tesla C1060. 50
4.2. Capacidad de cómputo de la GPU Tesla C1060. 51

5.1. Tiempos de ejecución para los mejores métodos en el espacio de palabras en español. (columna: rango; fila: estructura) . . 59
5.2. Tiempo de ejecución para los mejores métodos en el espacio de histogramas de colores (columna: porcentaje de datos recuperados; fila: estructura de datos) 60

6.1. Mejoras de la versión heterogénea versus las versiones secuenciales, multicore y de una GPU, para los rangos mayores y menores (versión heterogénea de 8 cores y 1 GPU). 96
6.2. Número de consultas por segundo para todas la versiones y los dos espacios de prueba (diccionario en español e histogramas de colores), considerando solamente el rango mínimo y máximo. `MQ` equivalente a $0,1\,\%$ de las consultas. 110

7.1. Número de páginas y porcentaje de la estructura de datos métrica para el espacio de histogramas de colores, considerando *EGNAT*, *GMS* y *LC*. 118
7.2. Número de páginas y porcentaje de la estructura de datos métrica para el espacio de palabras en español, considerando *EGNAT*, *GMS* y *LC*. 118
7.3. Consumo de energía para el caso de estudio de diccionario de Español de la figura 7.11(a). 133
7.4. Consumo de energía para el caso de estudio de histograms de color de la figura 7.11(b). 133
7.5. Consumo de energía para el caso de estudio de diccionario de Español de la sección 7.2 (figura 7.12(a)). 134
7.6. Consumo de energía para el caso de estudio de histograms de color de la sección 7.2 (figura 7.12(b)). 136

7.7. Consumo de energía para el caso de estudio del espacio de rostros la sección 7.2 (figura 7.12(c)). 136

Índice de algoritmos

1. *EGNAT*: Búsqueda por rango r para la consulta q en un nodo de tipo *GNAT*. 36
2. *Lista de Cluster*: Búsqueda por rango r para la consulta q. . 39
3. *GMS*: Algoritmo de búsqueda. 41
4. *Spaghettis*: Algoritmo de Construcción. 42
5. *SSS-Index*: Algoritmo de Selección de Pivotes. 44
6. *MSD*: Algoritmo de Selección de Pivotes para *LAESA*. 46
7. CUDA: Kernel generador de distancias. 62
8. CUDA: Algoritmo de búsqueda, obtención de candidatos. . . 63
9. CUDA: Soluciones finales para la consulta q. 63
10. CUDA: Programa principal. 64
11. CUDA: Transferencia de datos a memoria compartida. Ejemplo: consulta q. 65

Capítulo 1

Introducción

La búsqueda es y será uno de los problemas más interesantes en ciencias de la computación, ha concentrado gran parte de los esfuerzos de científicos y técnicos durante las últimas décadas y es componente importante en la mayoría de las aplicaciones. La búsqueda más conocida es aquella donde "dada una consulta o query sobre un conjunto de datos, se recupera el valor que es igual a la consulta". Ésta es conocida como *búsqueda exacta* y se aplica en bases de datos tradicionales sobre datos estructurados.

Una base de datos tradicional representa la información a través de entidades y relaciones (bases de datos relacionales [30] y otros modelos basados en el anterior), donde cada entidad o relación se compone de registros o tuplas que representan al objeto, el cual está caracterizado por atributos o campos alfanuméricos. La búsqueda se realiza comparando la clave de búsqueda contra uno de estos atributos. Sobre estas bases de datos también se pueden realizar búsquedas más sofisticadas como consultas por rango sobre claves numéricas o prefijos sobre claves alfabéticas. Sin embargo, éstas también se basan en el concepto de que dos claves sean o no iguales o que exista un orden sobre las claves. A pesar de que en la actualidad las bases de datos permiten almacenar tipos de datos más complejos, como por ejemplo, imágenes, las búsquedas se realizan de igual manera sobre los atributos de tipo alfanumérico.

Con la rápida evolución de las tecnologías de la información han surgido nuevos depósitos no estructurados de datos tales como texto libre, imagen, sonido y video. Con la aparición de estos nuevos tipos de datos no sólo se hace necesario almacenarlos, sino también realizar búsquedas sobre ellos en términos de los contenidos reales. Estructurar este tipo de datos es difícil ya sea manual o computacionalmente y restringe de antemano los tipos de búsqueda posibles. Más aún, realizar búsquedas exactas sobre estos datos sería poco útil. Por ejemplo, si se consultase por un elemento sobre una base de datos de imágenes, la consulta sólo podría encontrar su copia digital exacta en la base de datos. El verdadero interés reside, por ejemplo, en buscar

sobre una base de datos de fotografías una imagen que contiene un rostro determinando, donde no necesariamente existe una copia exacta de la misma fotografía; identificación de individuos a través de dispositivos biométricos, donde el dato consulta (voz, retina, etc) podría verse afectado por factores externos; encontrar una especie más parecida a otra en una base de datos de cadenas de ADN, etc.

Este tipo de búsqueda se denomina *búsqueda por similitud* o *búsqueda aproximada* o *en proximidad*. El objetivo es recuperar los objetos más *parecidos* o *relevantes* a la consulta; en este sentido es posible que la consulta sea o no parte de la base de datos.

Existe una amplia variedad de aplicaciones prácticas en diversas áreas donde es interesante utilizar búsqueda por similitud. Ejemplos de éstas son:

Consultas por contenido en bases de datos multimedia: En objetos tales como sonido, video e imagen, carece de interés realizar búsquedas exactas, debido a que la probabilidad de que dos objetos sean iguales es ínfima y para ello tendrían que ser copias digitales. Sin embargo, resulta interesante buscar objetos que sean similares entre sí, por ejemplo, la consulta por una huella digital donde ésta contenga información incompleta, es decir, falten algunos puntos que utilizan algunos mecanismos actuales de identificación.

Bioinformática: Los objetos clásicos de estudio en biología molecular son las secuencias de proteínas y cadenas de ADN, las cuales se pueden modelar a través de secuencias de caracteres. Los biólogos están interesados en encontrar, por ejemplo, mutaciones en distintas especies, o partes de una secuencia dada sobre un conjunto de secuencias más largas de proteínas.

Recuperación de información: Donde es necesario recuperar documentos relevantes a una consulta dada. Una forma tradicional de buscar documentos es determinar una medida de relevancia de acuerdo a la cantidad de veces que aparecen las palabras de la consulta en los documentos. Entonces, sería de interés poder recuperar documentos con contenido semántico similar, es decir, párrafos o frases distintas que signifiquen lo mismo.

Compresión y transmisión de audio y video: Durante la transmisión de una señal de audio o video se puede suponer que un cuadro (una imagen estática en un video) está formado por subcuadros. Entonces es posible enviar un primer cuadro completo para luego enviar de los siguientes cuadros sólo los subcuadros que tengan diferencias substanciales respecto de los previamente enviados.

Detección de plagios: Esta es una aplicación general basada en las anteriores, y cuyo objetivo es determinar si existe plagio de un objeto, por

ejemplo, un artículo de otros o una melodía de otras.

La búsqueda por similitud es un concepto unificador para la mayoría de las consultas de interés sobre datos no tradicionales. La similitud se modeliza usando una *función de distancia* o *métrica*, la cual satisface las propiedades de *positividad*, *simetría* y *desigualdad triangular*, y el conjunto de objetos es llamado un *espacio métrico*. Los objetos de la base de datos son tratados como cajas negras donde sólo se conocen las distancias entre objetos. Se entiende que la única información disponible o posible de calcular es la distancia entre un objeto y otro.

Existe dos tipos de consultas en búsqueda por similitud:

- *Consulta por Rango*: el objetivo es recuperar todos los elementos de la base de datos que estén a cierta distancia (radio de tolerancia) de un objeto consulta.

- *k-vecinos más cercanos (kNN)*: se desea recuperar los k elementos más cercanos a una consulta dada.

El segundo tipo de consulta se puede obtener a través del primero.

La forma trivial de responder a estas consultas es realizando una búsqueda exhaustiva sobre la base de datos, es decir, comparar cada uno de los elementos con la consulta y determinar cuál de ellos cumple con la condición, por ejemplo que la distancia sea menor a un radio de tolerancia. En general, la función de distancia se considera computacionalmente costosa de calcular, por lo que durante la búsqueda, el objetivo general es reducir la cantidad de evaluaciones de distancia. Por ejemplo, la distancia entre dos tramas de video tiene un costo elevado comparado con otros tipos de operaciones realizadas durante la búsqueda.

Los algoritmos para búsqueda por similitud en espacios métricos construyen a priori una estructura llamada *índice* (a veces también llamada *estructura métrica*), que permite reducir los cálculos de distancia durante la búsqueda. La construcción de un índice implica un costo adicional de preproceso de la base de datos, sin embargo, esto permite evitar posteriormente evaluaciones de distancia con algunos objetos, para lo que se usa la desigualdad triangular.

Una idea global de una búsqueda por similitud es la representada en la figura 1.1. Esta figura corresponde a un sistema de recuperación de imágenes basadas en contenido (*Content-Based Image Retrieval o CBIR*, se utiliza también como sinónimo *Features-Based Retrieval*)[1], cuya idea nace del estudio de la percepción humana y tiene como objetivo recuperar información basada en su contenido. Los pasos considerados en su construcción y funcionamiento son: extraer características de las imágenes, almacenarlas en

[1]Imágenes obtenidas de [69].

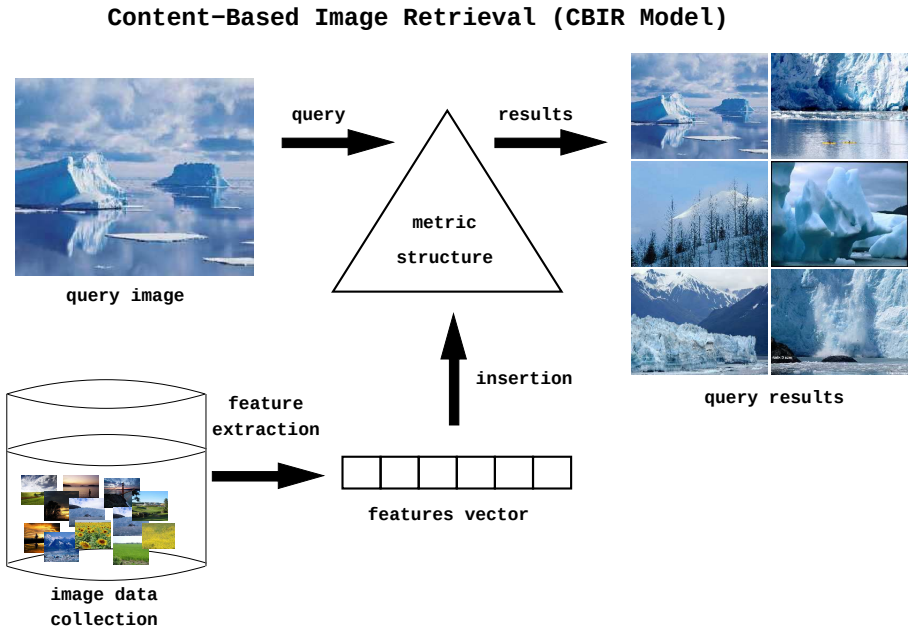


Figura 1.1: Sistema de recuperación de imágenes basada en contenido.

un índice, construir solicitudes de búsqueda. Estos pasos se realizan independientes de la arquitectura del sistema y son los expuestos en la figura 1.1.

Los algoritmos de búsqueda por similitud se dividen en dos grandes áreas: algoritmos basados en pivotes y algoritmos basados en particiones compactas. En [26] se presenta un marco unificador que describe y analiza las distintas soluciones para el problema de búsqueda por similitud, adicionalmente, en [45] se presenta, igualmente, un estudio que incluye soluciones propuestas en la última década.

1.1. Relevancia y Aportes de la Tesis

En la actualidad es impensable la construcción de sistemas de búsqueda reales, por similitud o no, que no implementen sus algoritmos sobre entornos paralelos. A día de hoy existen varios estudios y trabajos sobre estructuras métricas implementadas sobre distintas plataformas paralelas, la mayoría de ellas sobre sistemas de memoria distribuida (del tipo cluster), siendo menos frecuentes aquellas implementadas sobre arquitecturas de memoria compartida (del tipo multicore).

Básicamente, para resolver las cantidades masivas de consultas que pueden llegar a un buscador, las plataformas que soportan todo este esquema (índices, bases de datos, consultas, etc) son arquitecturas de altas prestacio-

nes que contienen cantidades considerables de nodos, todos trabajando en función del proceso común, "la búsqueda". En la actualidad, dichos nodos no poseen necesariamente una arquitectura unificada, de hecho, la aparición de nuevos tipos de dispositivos de procesamiento, o nuevos enfoques de utilización en algunos casos, puede generar una enorme ventaja en la paralelización a nivel de datos.

Hoy en día, existen dispositivos que pueden ejecutar un alto número de *threads* o hilos de procesamiento en paralelo. Ejemplo de estos dispositivos son las GPUs o unidades de procesamiento gráfico (*Graphic Processing Units*). Estas unidades suponen un cambio notable en la computación de altas prestaciones al conjugar un alto nivel de procesamiento y un bajo coste económico. Las GPUs poseen un gran número de núcleos de procesamiento o *cores* junto con una alta velocidad de acceso a los bancos de memoria, lo que las hace idóneas para aplicaciones con necesidades de procesamiento masivo de datos.

Al comenzar este trabajo, existían pocas implementaciones que utilizaban GPUs para procesar búsquedas por similitud, todas ellas atacaban el problema desde la búsqueda exhaustiva y para *kNN*. En la actualidad existen algunos grupos de investigación que desarrollan trabajos sobre estructuras métricas, sin embargo, no se tiene conocimiento de desarrollos en torno a la implementación de versiones heterogéneas, es decir, a la utilización de todos los recursos disponibles en la plataforma subyacente (para una discusión en detalle remitirse a la sección 2.9.6).

Objetivos

La presente Tesis aborda el problema de la búsqueda por similitud desde la visión de la implementación de estructuras métricas sobre plataformas basadas en GPU y en el contexto de implementaciones heterogéneas que aprovechen todos los dispositivos de procesamiento disponibles en la plataforma subyacente presentes en un nodo, sean de tipo *multicore* o *many-core* (GPUs, multi-GPUs).

En el presente trabajo se analiza un conjunto de estructuras, clásicas en su tipo, y con buen rendimiento en búsquedas secuenciales, para determinar cuáles serían las más idóneas para su implementación en este tipo de arquitectura.

El objetivo principal es presentar un conjunto de implementaciones orientadas a la búsqueda eficiente sobre estructuras métricas bajo distintas arquitecturas, de tal manera que todos los dispositivos puedan cooperar, simultáneamente, en la resolución de las consultas ingresadas al sistema.

Se considera que los aportes más relevantes de la presente Tesis son:

- Presentar una serie de algoritmos para distintas estructuras y su implementación sobre plataformas basadas en GPU, multi-GPU y mul-

ticore.

- Lograr mejorar la eficiencia de los algoritmos sobre la nueva arquitectura de los dispositivos gráficos.
- Utilizar todo el potencial de una plataforma heterogénea con sus distintos tipos de dispositivos de procesamiento para solucionar, todos en conjunto, la entrada masiva de consultas al sistema.
- Analizar distintos esquemas de planificación de distribución de consultas sobre un sistema heterogéneo.
- Obtener una visión más profunda de los diferentes aspectos a considerar en la implementación de estructuras métricas sobre plataformas basadas en GPU y plataformas heterogéneas en general, incluyendo aspectos relativos al consumo energético de los distintos dispositivos. Esto permitirá que futuros diseños de estructuras métricas tengan una base considerando los nuevos tipos de problemas y los nuevos tipos de arquitecturas.

1.2. Organización de la Tesis

La organización del presente trabajo de tesis está dividido en 4 partes:

- La primera parte corresponde a una descripción del marco teórico y conceptos básicos sobre el contexto en el que está basada la Tesis, apuntando principalmente a los tópicos de espacios métricos y paralelización de estructuras métricas (capítulo 2), incluyendo una descripción detallada de las estructuras relevantes a la Tesis y utilizadas en una primera etapa para su análisis (capítulo 3). Se incluye también en esta parte una descripción de la plataforma computacional y los espacios métricos de prueba utilizados en la Tesis (capítulo 4).
- La segunda parte corresponde a obtener las primeras aproximaciones para una solución sobre una plataforma basada en GPUs. En el capítulo 5 se obtiene una perspectiva inicial de las estructuras a fin de obtener aquellas más idóneas para su implementación sobre GPUs. En el mismo capítulo se presentan las primeras implementaciones sobre una plataforma basada en GPU y multi-GPU.
- La tercera parte corresponde al desarrollo de una implementación completa sobre una plataforma heterogénea. Presenta dos alternativas de soluciones, analizando empíricamente el comportamiento de ambas soluciones y de los rendimientos de los diferentes dispositivos (capítulo 6).

- La cuarta parte, desarrollada en el capítulo 7, consta de tres secciones donde se analizan las soluciones desde distintas perspectivas. La primera considera los distintos niveles de la jerarquía de memoria y los costos adicionales si se consideran, por ejemplo, los accesos a memoria secundaria. La segunda sección profundiza la situación bajo un nuevo esquema de planificación de distribución de consultas, suponiendo un conocimiento previo respecto de las relaciones entre las velocidades de los diferentes dispositivos de procesamiento. La tercera sección abre una discusión y un análisis en torno al consumo energético de los distintos dispositivos utilizados en la Tesis aquí presentada.

Capítulo 2

Marco Teórico

La similitud se modeliza en muchos casos interesantes a través de un espacio métrico, y la búsqueda de objetos más similares a través de una búsqueda por rango o de vecinos más cercanos. En el presente capítulo se definen formalmente los conceptos referentes a búsquedas en espacios métricos, además se muestra la caracterización de algunos espacios y el estado del arte respecto a temas relevantes actualmente, como son aquellos relativos a dinamismo, memoria secundaria y paralelización de estructuras métricas.

2.1. Espacios Métricos

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

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

Estas propiedades sólo aseguran una definición consistente de la función, pero no pueden usarse para descartar objetos y ahorrar evaluaciones de distancia durante una búsqueda por similitud. Para que d sea realmente una métrica del espacio $\mathbb{X}$ debe satisfacer además, la siguiente propiedad:

- Desigualdad triangular : $d(x, y) + d(y, z) \geq d(x, z)$.

Entonces, el par $(\mathbb{X}, d)$ es llamado *Espacio Métrico*.

2.2. Búsquedas por Similitud

Sea $\mathbb{Y} \subseteq \mathbb{X}$ el conjunto de objetos que componen una base de datos. El concepto de búsqueda o consulta por similitud o en proximidad consiste

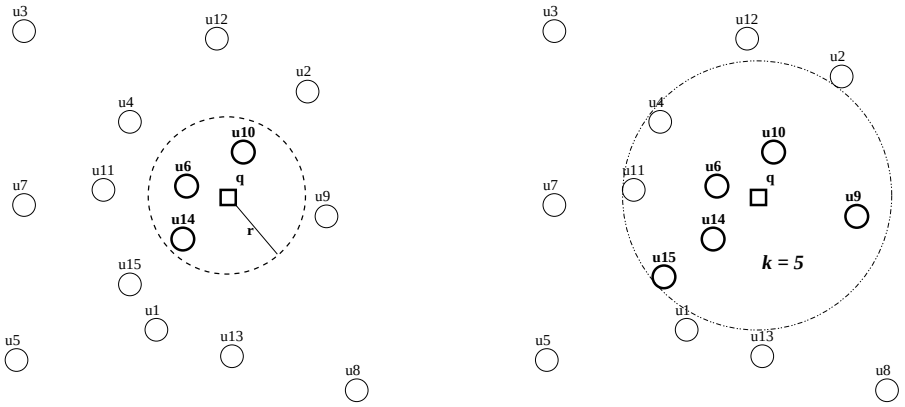


Figura 2.1: Ejemplos de consultas, por rango (izquierda) y por k-vecinos más cercanos (derecha) sobre un conjunto de puntos en $\mathbb{R}^2$ (fuente figura [26]).

en recuperar todos los objetos pertenecientes a $\mathbb{Y}$ que sean parecidos a un elemento de consulta q que pertenece al espacio $\mathbb{X}$.

Las consultas en proximidad sobre espacios métricos son básicamente de dos tipos [26]:

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

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

El tipo básico de consulta es la consulta por rango. En la figura 2.1 se ilustran ambos tipos de consulta. Para mayor claridad, las consultas de la figura 2.1 están realizadas sobre un conjunto de puntos en $\mathbb{R}^2$. 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 muestra 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 (caso de dos o más objetos a la misma distancia de q).

En el presente trabajo, el interés está centrado en la búsqueda por rango, debido a que es la base para resolver el otro tipo de consulta.

2.3. Indexación

Una manera trivial de responder a estos tipos de consulta es examinar exhaustivamente la base de datos $\mathbb{Y}$, lo que toma tiempo $O(n)$. De hecho, ésta sería la única forma de resolver consultas por similitud si no fuese posible construir una estructura que ayude a responder las consultas. Sin embargo, considerando que la función de distancia es *computacionalmente costosa*, no es práctico realizar 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* (estructura de datos métrica) a fin de ahorrar computaciones de distancia en el momento de resolver la búsqueda. Esta estructura permitirá resolver búsquedas posteriores de manera mucho más eficiente.

Un *algoritmo de indexación* (o indexamiento) es un proceso que construye una estructura denominada *índice*, el cual permite resolver búsquedas por similitud evitando, en lo posible, tener que revisar toda la base de datos, ahorrando evaluaciones de distancia al momento de realizar las consultas. Todos los algoritmos de búsqueda 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.

Hay dos casos principales que considerar: cuando el índice y los datos pueden ser mantenidos en memoria principal, y cuando es necesario utilizar memoria secundaria para almacenar los datos y/o el índice. Para los algoritmos de indexación en memoria principal, el objetivo central es reducir el número de cálculos de distancia. Para los algoritmos en memoria secundaria, además de reducir los cálculos de distancia, se debe minimizar también el número de accesos a disco y movimiento de cabezales. En este tema en particular los avances han sido más lentos que los realizados para memoria principal.

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

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

En muchas aplicaciones la evaluación de la distancia es tan costosa que las demás componentes de la fórmula anterior pueden ser despreciadas.

En la actualidad, existen algunos índices diseñados para memoria secundaria que consideran también el costo de la entrada y salida, es decir, la cantidad de accesos y movimientos en memoria secundaria (*read*, *write* y *seek*). Otra característica importante es que la mayoría de las estructuras métricas no posee capacidades dinámicas, es decir, la posibilidad de inserción y eliminación en la estructura posterior a su construcción. Si bien muchas estructuras soportan inserciones posteriores, existe sólo un pequeño conjunto

de estructuras que implementan la eliminación. Éstas, en su mayoría, están diseñadas para memoria principal (ver sección 2.7).

Diseñar estructuras métricas que soporten eliminaciones resulta complejo, y además el problema se agrava por la necesidad de su implementación en memoria secundaria. Todo ello hace aumentar considerablemente el tiempo de respuesta a las consultas.

Una vía para reducir el tiempo está centrada en la paralelización de estructuras. En general, se ha propuesto una cantidad considerable de estructuras métricas para computación secuencial, las cuales pueden ser eficientes frente a búsquedas en espacios multidimensionales y bases de datos de cientos de miles de objetos. Sin embargo, el diseño de estas estructuras está básicamente orientado hacia la optimización de consultas individuales, por lo que no necesariamente mantienen un buen desempeño cuando son llevadas a sistemas paralelos con memoria distribuida o compartida. Existe una serie de trabajos en esta área donde se presentan esquemas de uso tanto de memoria distribuida como compartida, ya sea utilizando plataformas como *clusters* o multiprocesadores. Además, con la constante aparición de nuevo hardware, aparecen en escena plataformas de computación heterogéneas. En este sentido, cabe destacar el éxito que están teniendo las GPUs (*Graphics Processing Units*) las cuales permiten implementar a bajo coste y con excelente rendimiento múltiples aplicaciones.

Uno de los objetivos finales del presente trabajo está relacionado precisamente con la utilización de GPUs en la implementación de estructuras métricas paralelas.

2.4. Ejemplos de Espacios Métricos

A continuación se presentan algunos ejemplos de espacios métricos.

2.4.1. Espacios Vectoriales

Los espacios vectoriales son un caso particular de espacio métrico. Un espacio vectorial $\mathbb{R}^k$ es un conjunto de *k-tuplas* de números reales. Existen distintas funciones de distancia para espacios vectoriales. Entre las más comunes están las pertenecientes a la familia L_s o familia de distancias de *Minkowski*, que se define como:

$$L_s((x_1, ..., x_k)(y_1, ..., y_k)) = \left(\sum_{i=1}^{k} |x_i - y_i|^s \right)^{\frac{1}{s}}, \ 1 \leq s \leq \infty$$

Algunos ejemplos de métricas pertenecientes a esta familia de distancias son:

- L_1, conocida como *distancia de Manhattan*:

$$L_1((x_1, ..., x_k)(y_1, ..., y_k)) = \sum_{i=1}^{k} |x_i - y_i|$$

- L_2, conocida como *distancia Euclidiana*:

$$L_2((x_1, ..., x_k)(y_1, ..., y_k)) = \sqrt{\sum_{i=1}^{k} |x_i - y_i|^2}$$

- L_∞, conocida como *distancia del máximo*, que corresponde a tomar el límite cuando s tiende a infinito:

$$L_\infty((x_1, ..., x_k)(y_1, ..., y_k)) = \max_{i=1}^{k} |x_i - y_i|$$

En muchas aplicaciones los espacios métricos son en realidad espacios vectoriales, es decir, que los objetos son puntos *k-dimensionales* y que la similitud se puede interpretar geométricamente. Un espacio vectorial permite más libertad al diseñar algoritmos de búsqueda, porque es posible usar la información de coordenadas y geometría que no está disponible en espacios métricos.

Para espacios vectoriales existen soluciones eficientes. Las más conocidas son los *Kd-trees* [10, 11], *R-trees* [44], *Quadtrees* [74], los *X-trees* [12] y *TV-trees* [50], que soportan consultas por similitud, entre otros. Todas estas técnicas usan ampliamente la información de coordenadas para agrupar y clasificar puntos en el espacio. Sin embargo, éstas son muy sensibles a la dimensión del espacio vectorial. La complejidad de las búsquedas por rango depende exponencialmente de la dimensión del espacio [27].

2.4.2. Diccionarios de Palabras

Para este caso, los objetos del espacio métrico son palabras en algún lenguaje específico. Para medir la distancia entre cadenas de caracteres, una función posible es la conocida como *distancia de edición* o *distancia de Levenshtein*. Esta distancia se define como la cantidad de inserciones, eliminaciones o modificaciones de caracteres que hay que hacer sobre una palabra para convertirla en otra. Esta es una función de distancia discreta y tiene múltiples aplicaciones, como recuperación de texto, procesamiento de señales, comparación de secuencias de ADN, entre otras [60].

2.4.3. Modelo Vectorial para Documentos

En recuperación de información [4] se define un *documento* como una *unidad de recuperación*, la cual puede ser un párrafo, una sección, un capítulo, una página web, un artículo o un libro completo. Los modelos clásicos en recuperación de la información consideran que cada documento está descrito por un conjunto representativo de palabras clave llamadas *términos*, que son palabras cuya semántica ayuda a definir los temas principales del documento.

Uno de estos modelos, el *modelo vectorial*, considera un documento como un vector *t-dimensional*, donde t representa el número total de términos de la colección. Cada coordenada i del vector está asociada a un término del documento, cuyo valor corresponde a un "peso" positivo w_{ij} si es que dicho término pertenece al documento j o 0 en caso contrario. Si $\mathbb{D}$ es el conjunto de documentos y d_j es el *j-ésimo* documento perteneciente a $\mathbb{D}$, entonces

$$d_j = (w_{1j}, w_{2j}, ..., w_{ij})$$

En el modelo vectorial se calcula el *grado de similitud* entre un documento d_j y una consulta q, la cual puede ser vista como un conjunto de términos o como un documento completo, como el grado de similitud entre los vectores $\overrightarrow{d_j}$ y $\overrightarrow{q}$. Esta correlación puede ser cuantificada, por ejemplo, como el coseno del ángulo formado entre ambos vectores:

$$sim(d_j, q) = \frac{\overrightarrow{d_j} \cdot \overrightarrow{q}}{\left|\overrightarrow{d_j}\right| \times |\overrightarrow{q}|} = \frac{\sum_{i=1}^{t} w_{ij} \times w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2 \times \sum_{i=1}^{t} w_{iq}^2}}$$

donde w_{iq} es el peso del *i-ésimo* término en la consulta q.

Los pesos de los términos pueden calcularse de varias formas. Una de las más importantes es utilizando *esquemas tf-idf*, en donde los pesos están dados por:

$$w_{ij} = f_{i,j} \times \log\left(\frac{N}{n_i}\right)$$

donde N es el número total de documentos, n_i es el número de documentos en donde el *i-ésimo* término aparece, y $f_{i,j}$ es la *frecuencia normalizada* del *i-ésimo* término, dada por:

$$f_{i,j} = \frac{freq_{i,j}}{\max\limits_{l=1..t}(freq_{l,j})}$$

donde $freq_{i,j}$ es la frecuencia del *i-ésimo* término en el documento d_j, y $\max_{l=1..t}(freq_{l,j})$ es el máximo valor de la frecuencia sobre todos los términos contenidos en d_j.

Si se considera que los documentos son puntos en un espacio métrico, el problema de búsqueda de documentos similares a una consulta dada se reduce a una búsqueda por similitud en el espacio métrico. Dado que $sim(d_j, q)$ es una medida de similitud y no de distancia, se utiliza el ángulo formado entre los vectores $\overrightarrow{d_j}$ y $\overrightarrow{q}$, $d(d_j, q) = \arccos(sim(d_j, q))$, como función de distancia, la cual se denomina *distancia coseno* [4]. Así definido, se puede decir que el par $(\mathbb{D}, d)$ es un espacio métrico.

2.5. Dimensionalidad y Espacios Métricos

La búsqueda por similitud en espacios métricos es más compleja cuando se realiza sobre los llamados *espacios de alta dimensión*. Muchas técnicas de indexación tradicionales para espacios vectoriales tienen una dependencia exponencial con la dimensión representacional k del espacio, es decir, a medida que la dimensión crece, dichas técnicas se vuelven menos eficientes.

Las técnicas más recientes han podido solucionar esto logrando que su complejidad dependa sólo de la *dimensión intrínseca* del espacio. Esto da a entender que existen espacios cuya dimensión representacional es alta, pero su dimensión intrínseca es baja. Un ejemplo de esto es un plano embebido en $\mathbb{R}^{50}$, donde los tiempos de búsqueda se aproximarán más a los de un espacio de dimensión 2 que a uno de dimensión 50.

La dimensión intrínseca de un espacio no es fácil de definir. Se dice que un espacio es de alta dimensión cuando su histograma de distancias es concentrado, es decir, la *media* es alta y la *varianza* pequeña, lo que indicaría que los objetos están todos más o menos a la misma distancia entre sí. En la figura 2.2 se muestra la diferencia entre histogramas de distancias para espacios de baja y alta dimensión intrínseca.

La dificultad de la búsqueda en espacios de alta dimensión radica en la baja cantidad de objetos que se pueden descartar. Un ejemplo extremo de esto es el siguiente.

Sea $(\mathbb{X}, d)$ un espacio métrico, donde:

- $\forall\, x,\ d(x, x) = 0$
- $\forall\, y \neq x,\ d(x, y) = 1$

Para una búsqueda $(q, r)_d$ en un espacio de estas características, la única información que se tiene al hacer una comparación $d(q, x)$ es si x es parte o no del resultado de la búsqueda. Sin embargo, esta información no sirve para evitar evaluaciones de distancia con los otros objetos del espacio. Es

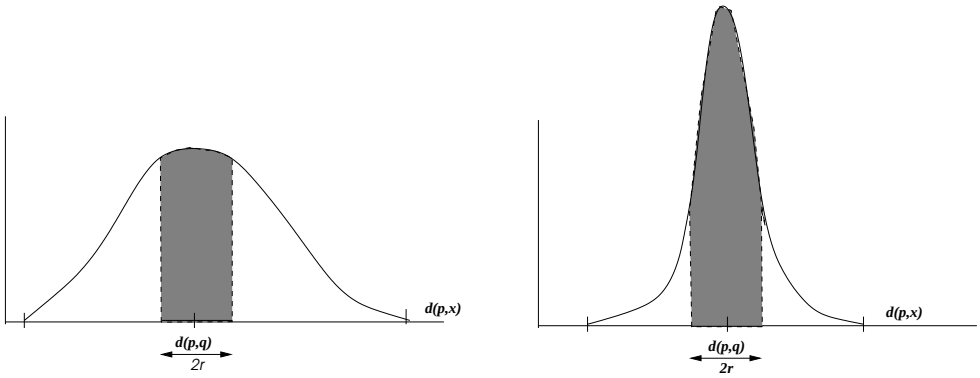


Figura 2.2: Histogramas de distancias para espacios de baja dimensión (izquierda) y alta dimensión (derecha) (fuente figura [26]).

imposible, en este caso, evitar la búsqueda secuencial, independiente de la técnica de indexación utilizada.

Considérese lo siguiente: sea un espacio métrico $(\mathbb{X}, d)$, un elemento $p \in \mathbb{X}$, una búsqueda $(q, r)_d$ y un índice basado en pivotes elegidos aleatoriamente (ver sección 2.6.2). Los histogramas de la figura 2.2 representan dos posibles distribuciones de distancia de un pivote p para los valores de $d(p, q)$. La regla de eliminación permite descartar todos aquellos puntos y tales que $y \notin [d(p, q) - r, d(p, q) + r]$. La figura muestra intuitivamente el porqué la búsqueda se torna más difícil cuando el histograma es más concentrado. Las áreas sombreadas representan los puntos que no podrán descartarse. Es decir, a medida que el histograma se concentra más alrededor de su media disminuye la cantidad de puntos que pueden descartarse usando como dato $d(p, q)$.

En [26] se define la 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. Esta definición es coherente con la noción de dimensión en un espacio vectorial con coordenadas uniformemente distribuidas.

2.6. Algoritmos de Indexación para Espacios Métricos

2.6.1. Modelo General de Indexación

Como se mencionó anteriormente, un algoritmo de indexación es un proceso *off-line* que construye una estructura de datos o índice, la cual permite ahorrar cálculos de distancia al momento de resolver una consulta. Entonces, lo importante es diseñar algoritmos de indexación que sean eficientes para reducir las evaluaciones de distancia durante la búsqueda.

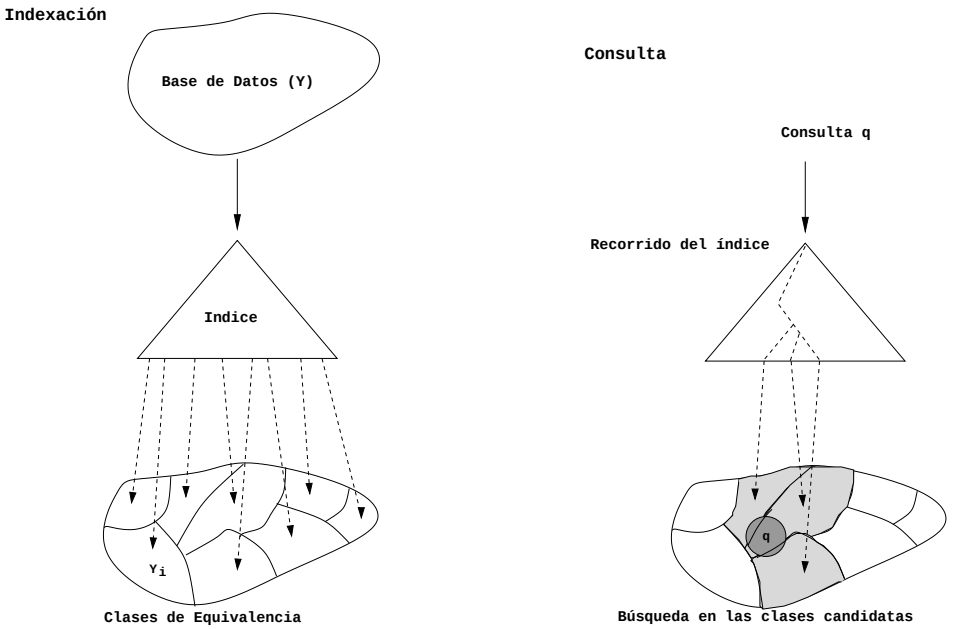


Figura 2.3: Modelo general para algoritmos de indexación (fuente figuras [26]). Izquierda: construcción del índice. Derecha: consulta sobre el índice.

Los métodos para buscar en espacios métricos se basan principalmente en dividir el espacio empleando la distancia a uno o más objetos seleccionados. El no trabajar con las características particulares de cada aplicación tiene la ventaja de ser más general, pues los algoritmos funcionan con cualquier tipo de objeto.

Todos los algoritmos de indexación particionan la base de datos $\mathbb{Y}$ en subconjuntos. El índice permitirá determinar una lista de subconjuntos Y_i candidatos potenciales a contener objetos relevantes a la consulta. A cada subconjunto Y_i podría aplicársele el mismo método de particionamiento, en forma recursiva. Durante la búsqueda, se recorre el índice para obtener los conjuntos relevantes y luego se examina cada uno de ellos (en forma exhaustiva si no se ha continuado recursivamente la indexación). Todas estas estructuras trabajan sobre la base de descartar elementos usando la desigualdad triangular. En la figura 2.3 se muestra este modelo de indexación.

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* [26]. En la figura 2.4 se ilustra una clasificación de los distintos índices de acuerdo a sus características generales; esta clasificación fue presentada en [26] y se le han agregado sólo las estructuras utilizadas en este trabajo que no estaban presentes en el original.

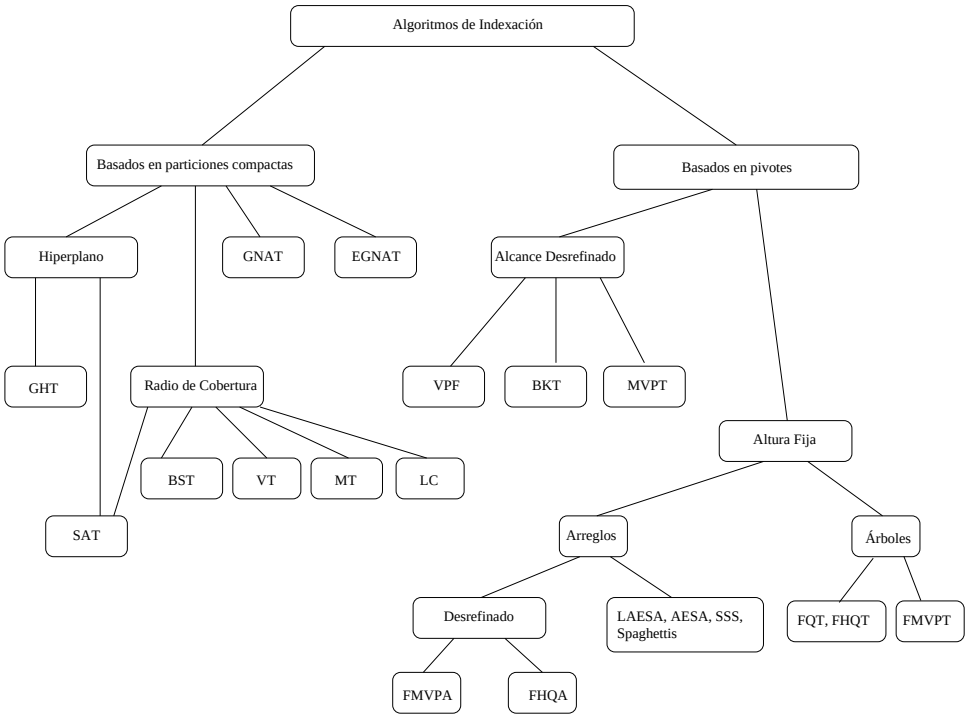


Figura 2.4: Taxonomía de los algoritmos existentes para búsquedas por similitud en espacios métricos.

En la tabla 2.1 se muestra un detalle de las estructuras expuestas en la figura 2.4 con los nombres completos, la referencia a los artículos donde fueron propuestos y las abreviaturas usadas durante el presente trabajo.

2.6.2. Algoritmos Basados en Pivotes

Un *pivote* es un objeto preseleccionado del espacio métrico (puede ser parte de la base de datos o no). Los 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. Las dos partes principales, construcción y búsqueda, son descritas de manera general a continuación (detalles concernientes a estructuras específicas serán descritos en el capítulo 3).

Construcción: Sea $\{p_1, p_2, ..., p_k\} \in \mathbb{X}$ un conjunto de pivotes. Se almacena para cada elemento x de la base de datos $\mathbb{Y}$, su distancia a los k pivotes $(d(x, p_1), ..., d(x, p_k))$.

Búsqueda: 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

Abreviatura	Estructura	Referencia
BKT	Burkhard-Keller tree	[19]
FQT	Fixed Queries tree	[6]
FQHT	Fixed Height FQ-tree	[6, 5, 7]
FQA	Fixed Queries arrays	[24, 25]
Metric Tree	Metric Tree	[82]
VP-tree	Vantage Point tree	[101, 28]
MVP-tree	Multi-Vantage Point tree	[15]
VP-forest	Excluded Middle Vantage Point Forest	[102]
BST	Bisector tree	[48]
GHT	Generalized-Hyperplane tree	[82]
GNAT	Geometric Near-neighbor Access Tree	[16]
EGNAT	Evolutionary GNAT	[88]
VT	Voronoi tree	[34]
M-tree	M-tree	[29]
LC	List of Clusters	[23]
SAT	Spatial aproximation tree	[61, 73]
AESA	Approximating Eliminating Search Algorithm	[98]
LAESA	Linear AESA	[59]
SSS	Sparse Spatial Selection (o SSS-Index)	[70]
Spaghettis	Spaghettis	[24]

Tabla 2.1: Estructuras de datos para búsquedas en espacios métricos.

$|d(q,p_i) - d(x,p_i)| > r$, 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 para encontrar los puntos candidatos, pero no en la cantidad de evaluaciones de distancia. Ejemplos de algoritmos basados en pivotes son: *AESA* [98], *LAESA* [59], *Spaghettis* y sus variantes [24, 64], *FQT* y sus variantes [6] y *FQA* [25]. *LAESA* propone un método de selección denominado *Maximun Sum of Distances* (*MSD*). Otra estructura es *SSS-Index* [70] que es básicamente una tabla con distancias pero define un método de selección de pivotes denominado *Sparse Spatial Selection*.

Las estructuras de tipo árbol utilizan esta técnica en forma indirecta. El árbol se va construyendo tomando el nodo raíz como pivote. Posteriormente el espacio se divide de acuerdo a la distancia de los objetos al pivote. Cada subárbol se construye recursivamente tomando un nuevo pivote de los elementos del subespacio. Las diferencias radican principalmente en la forma

y tamaño de los espacios. La búsqueda realiza un *backtrack* sobre el árbol y utiliza la desigualdad triangular para minimizar los subárboles. Las estructuras que utilizan esta técnica son *BKT* y sus variantes [19, 76], *Metric trees* [82], *TLAESA* [58], *VP-trees* y sus variantes [101, 15, 103].

2.6.3. Algoritmos Basados en Clustering o Particiones Compactas

Los algoritmos basados en clustering dividen el espacio en áreas, donde cada área tiene un *centro*. Se almacena alguna información sobre el área que permita descartarla completamente mediante sólo comparar la consulta con su centro. Los algoritmos de clustering son los mejores para espacios de alta dimensión, que es el problema más difícil en la práctica.

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 con centro 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 asociadas a los 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 al conjunto de objetos que está a distancia máxima r de q.

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 asociada a c_i si $d(q, c_i) - r > rc(c_i)$.

Algunas estructuras combinan estas técnicas, como el caso del *GNAT* [16] y el *EGNAT* [88]. Otras sólo utilizan radio cobertor, como son los *M-trees* [29], la *Lista de Clusters* [23] y *SSSTree* [17], entre otros. Algunas

que utilizan hiperplanos son GHT y sus variantes [82, 67], los Voronoi-trees [34, 66] y el *Metric Index* [68].

En general, existe una cantidad considerable de nuevas estructuras aparecidas en la última década. Una descripcion de muchas de ellas puede ser encontrada en [104] y en [45].

2.7. Estructuras Métricas y Dinamismo

El dinamismo es poco común en las estructuras de datos para espacios métricos [26]. Sin embargo, algunas estructuras permiten inserciones eficientes una vez construidas. La eliminación resulta particularmente complicada, debido a que las estructuras podrían verse seriamente afectadas y habría que reconstruirlas parcial o totalmente, con el costo que conlleva.

En [26] se muestra que las únicas estructuras que soportan completamente inserciones y eliminaciones son las de la familia *FQ* (*FQT*, *FQHT*, *FQA*) dado que no poseen nodos internos, así como *AESA* y *LAESA* ya que son sólo vectores de coordenadas. La mayoría de las estructuras de tipo árbol no soportan eliminaciones, sobre todo en nodos internos. Esto es debido a que de dichos nodos depende la organización de sus subárboles.

El *M-tree* [29] fue diseñado originalmente para memoria secundaria y con capacidades dinámicas y es la estructura referencia en este tipo de algoritmos. Existe también una variante de los *GHTs* que soporta inserciones y eliminaciones [97]. Sin embargo, en años posteriores ha aparecido un conjunto de estructuras que sí consideran el dinamismo, un listado de éstas puede ser encontrado en [45]. Existe una nueva versión del *SAT* [62] que soporta capacidades dinámicas completas, además, el *EGNAT* [63] propone un método genérico denominado *Planos Fantasmas* para realizar eliminaciones sobre árboles métricos.

2.8. Estructuras Métricas y Memoria Secundaria

La implementación en memoria secundaria no es trivial y de hecho la medida de eficiencia ya no es sólo la cantidad de evaluaciones de distancia realizadas, sino también los costos involucrados por accesos a disco. En disco, cuanto más secuenciales sean los accesos para realizar las búsquedas, más eficiente es el proceso, ya que tiene un costo muy elevado el realizar accesos a posiciones no consecutivas dentro del archivo.

La mayor parte de estructuras para búsquedas por similitud en espacios métricos están diseñadas para su implementación en memoria principal. En la actualidad sólo existen algunos algoritmos para espacios métricos generales diseñados para memoria secundaria. Entre ellos están el *M-Tree* [29] y *Slim-Tree* [79], basado en *M-Tree*.

Una de las características principales del *M-Tree* es que es un árbol balanceado. Sin embargo, uno de sus problemas es que está basado en radios cobertores y las áreas se intersectan, lo que provoca traslape o solapamiento (overlap) y por lo tanto menor eficiencia en la búsqueda, ya que las consultas deben recorrer más subárboles. El *Slim-Tree* es una estructura que reduce el grado de solapamiento del *M-Tree*.

Recientemente, en [63] se presenta un estudio completo sobre el *EGNAT*, el cual es una estructura completamente dinámica y optimizada para memoria secundaria. Esta estructura será tratada en detalle en la sección 3.1.

En [80] se propone una técnica denominada *OMNI*, y se plantea utilizar conjuntamente *OMNI* en conjunto con otras estructuras, lo que se denomina *familia OMNI*. Este trabajo selecciona elementos al estilo de las estructuras basadas en pivotes. Dichos elementos se denominan *foci* (focos) y se conserva la distancia de los objetos de la base de datos a los elementos seleccionados. Según sea la estructura utilizada con *OMNI* se le da características específicas orientadas, por ejemplo, a memoria secundaria, para una versión *OMNIR-tree* o dinamismo para una versión *OMNI-Secuencial*, entre otras.

Finalmente, en [18] se desarrollan dos implementaciones sobre memoria secundaria para la versión dinámica del *SAT*, las que se comparan entre ellas y con el *M-Tree*.

Es fundamental encontrar nuevas estructuras y algoritmos que mejoren las búsquedas por similitud en memoria secundaria, dado que hay muy pocas propuestas. Además, este tipo de búsqueda generalmente se aplica a grandes bases de datos, ya sean multimedia, de texto o de otro tipo, lo que implica que todos los diseños pensados como prototipos para memoria RAM van a tener que ser modificados para disco.

Respecto de la eliminación, el problema se agrava dado que no sólo hay que considerar el costo de mover objetos o nodos dentro de una estructura debido a alguna posible reorganización, sino que esto implica una serie de *escrituras* extra, y en general movimientos y lecturas adicionales en disco.

Otras consideraciones en memoria secundaria se refieren al tamaño de la estructura, la cual debe tener una correlación con la cantidad de objetos y el espacio ocupado por cada uno de ellos. Ligado a lo anterior, es importante también mantener una adecuada utilización del espacio dentro de las páginas de disco, de tal manera que estas estén en lo posible llenas.

En resumen, el diseño de una estructura métrica para memoria secundaria y con capacidades dinámicas completas requiere considerar los siguientes aspectos.

Costos de construcción: Junto con las evaluaciones de distancia, se debe considerar también los bloques leídos/escritos en disco y los movimientos de cabezales (seek).

Costos de búsqueda: Se deben considerar los cálculos de distancia, las lecturas y seeks en disco.

Costos de eliminación: Se deben considerar los mismos costos que para el caso de la construcción. Una especial atención requieren las escrituras en disco a consecuencia de posibles reoganizaciones de la estructura por eliminación de elementos claves para el mantenimiento del índice.

Espacio utilizado: El tamaño completo de la estructura en disco, así como también una adecuada distribución del espacio dentro de los bloques de disco (porcentaje de nodos incompletos y promedio de uso del nodo). Se debe tener en cuenta la reutilización de los bloques de disco disponibles producto de la eliminación de todos los datos contenidos en estos.

2.9. Entornos Paralelos

En relación a la paralelización de estructuras métricas, diferentes grupos de investigadores han abordado el tema principalmente desde dos puntos de vista. El primero habla de la implementación de estas estructuras sobre plataformas de memoria distribuida. En este sentido, existe una cantidad importante de trabajos que abordan este problema realizando pruebas sobre clusters de computadores. El segundo punto de vista tiene relación con la paralelización de las estructuras sobre plataformas de memoria compartida multicore, específicamente trabajos relacionados con plataformas de CPUs multicore. Aquí también es importante mencionar nuevas tecnologías que han permitido a los desarrolladores generar aplicaciones sobre nuevos dispositivos, como es el caso de los multicores en las unidades de procesamiento gráfico.

En esta sección se resumen las diferentes plataformas mencionadas y de interés para el trabajo desarrollado en esta tesis.

2.9.1. Modelos de Programación

En torno al desarrollo de la computación paralela, se podría decir que, a bajo nivel (nivel de hardware), se encuentran las tecnologías de memoria compartida y memoria distribuida. En los niveles superiores se encuentran las librerías de programación paralela MPI [43, 77], PVM [39] y OpenMP [22, 32] para sistemas multicore.

Por otro lado, los métodos síncronos y asíncronos utilizados en programas paralelos son considerados dos diferentes escuelas. Intuitivamente se puede pensar que los algoritmos asíncronos son más eficientes debido a que no es necesario realizar periódicamente una sincronización global. Sin embargo, existen casos en los que los algoritmos síncronos presentan un mejor desempeño que los programas asíncronos. Ejemplo de esto es mostrado en [33],

en donde las condiciones de tráfico de consultas que llegan a un buscador determinan qué modo (asíncrono o síncrono) es más adecuado utilizar.

Para el modo asíncrono se utiliza la biblioteca de paso de mensajes MPI y para el modo síncrono se utiliza el modelo BSP [95] de computación paralela (*Bulk Synchronous Parallel*). También, existe la librería de paso de mensajes BSPonMPI [90] que permite ejecutar BSP usando las primitivas de comunicación de MPI. De esta manera se logra comparar experimentalmente ambos modos de computación utilizando la misma biblioteca de comunicación, es decir, MPI.

2.9.2. Memoria Distribuida

MPI

MPI [43, 77] es una librería de comunicación utilizada para desarrollo sobre un cluster de computadores. MPI proporciona rutinas de mensajes que son independientes del protocolo de comunicación. Estas rutinas permiten comunicación tanto colectiva como punto a punto. MPI es uno de los estándares más utilizados en la actualidad.

La mayoría de las implementaciones de MPI consisten en especificar un conjunto de rutinas que pueden ser llamadas desde programas construidos en *Fortran*, *C* y *C++*. Una de las ventajas principales de MPI es que puede ser utilizado sobre casi cualquier arquitectura de memoria distribuida.

Las rutinas de envío de mensajes tienen la forma típica de *send(procesador_destino,buffer_datos,tamaño_buffer)* y pueden ser del tipo bloqueante (el procesador que envía se bloquea hasta que el mensaje es recepcionado por el destinatario), o no bloqueante (el procesador continúa su ejecución normal una vez enviado el mensaje).

La interfaz de MPI está diseñada para proveer funcionalidades de comunicación, sincronización y una topología virtual entre un conjunto de procesadores. Los programas siempre trabajan con procesos que son asignados a diferentes procesadores en tiempo de ejecución a través de un agente (o demonio) que ejecuta los programas MPI[1].

BSP

El modelo BSP de computación paralela fue propuesto en 1990 con el objetivo de permitir que el desarrollo de software sea portable y tenga desempeño eficiente y escalable [96]. BSP propone alcanzar este objetivo mediante la estructuración de la computación en una secuencia de pasos llamados *supersteps* y el empleo de técnicas aleatorias para el encaminamiento (ruteo) de mensajes entre procesadores. El computador paralelo, independiente de su arquitectura, es visto como un conjunto de pares procesadores-memoria,

[1] Subsección obtenida de [33]

los cuales son conectados mediante una red de comunicación cuya topología es transparente al programador.

Los *supersteps* son delimitados mediante la sincronización de procesadores. Los procesadores proceden al siguiente superstep una vez que todos ellos han alcanzado el final del superstep, los cuales son agrupados en bloques para optimizar la eficiencia de la comunicación. Durante un superstep, los procesadores trabajan asíncronamente con datos almacenados en sus memorias locales. Cualquier mensaje enviado por un procesador está disponible para procesamiento en el procesador destino sólo al comienzo del siguiente superstep.

Dada la estructura particular del modelo de computación, el costo de los programas BSP puede ser obtenido utilizando técnicas similares a las empleadas en el análisis de algoritmos secuenciales. En BSP, el costo de cada superstep está dado por la suma del costo en computación (el máximo entre los procesadores), el costo de sincronización entre procesadores, y el costo de comunicación entre procesadores (el máximo enviado/recibido entre procesadores). El costo total del programa BSP es la suma del costo de cada superstep.

Se hace notar que durante un superstep las computaciones realizadas en cada nodo procesador pueden ser realizadas en paralelo utilizando threads de OpenMP los cuales comparten la memoria del nodo y se ejecutan en las CPUs del nodo. A nivel de compilador, por ejemplo INTEL, es sencillo utilizar threads OpenMP en conjunto con MPI y por lo tanto BSPonMPI.

2.9.3. Memoria Compartida

OpenMP

OpenMP (*Open Multi-Processing*) [22, 32] permite realizar programación multi-thread (multi-hilo) sobre memoria compartida utilizando lenguajes como *C*, *C++* y *Fortran*. Consiste en un conjunto de directivas de compilación, rutinas de librerías y variables de entorno. La paralelización de un programa se realiza considerando que un thread principal crea un número específico de threads esclavos (mediante alguna operación *fork*) y las tareas se dividen entre ellos. Idealmente cada thread corre en una CPU de un procesador multicore.

Por defecto, cada thread ejecuta la sección del código paralelo en forma independiente. Es posible utilizar constructores *work-sharing* para dividir las tareas entre los threads de manera que cada uno ejecuta su parte del código. OpenMP permite obtener tanto una paralelización de tareas como la paralelización de datos. Se proporciona al programador un conjunto de primitivas de sincronización tales como *locks* y *critical section*.

El ambiente de ejecución asigna los threads en los procesadores dependiendo del uso de los mismos, la carga de trabajo en el procesador y otros

factores. Pero el programador también tiene la posibilidad de asignar los threads a cada procesador utilizando funciones específicas de OpenMP.

OpenMP tiene la ventaja de tener un alto nivel de abstracción, pues el código resultante difiere poco del secuencial, lo cual es muy útil al momento de depurar. OpenMP se viene desarrollando desde 1997, por lo tanto los compiladores actuales son bastante robustos.

Adicionalmente, existen otras librerías que permiten programación multi-thread como Pthreads [65] y TBB [72].

GPUs

Actualmente las unidades de procesamiento gráfico (GPUs o *Graphics Processing Units*) incluidas en algunas tarjetas gráficas disponen de un alto número de cores (más de 1000 en los últimos productos) con un alto ancho de banda con memoria. Este tipo de dispositivos llegan a multiplicar por 10 el ancho de banda total así como la capacidad de procesamiento respecto de las CPUs [100].

Tradicionalmente, las GPUs han sido utilizadas en aplicaciones interactivas, y son diseñadas para obtener un alto rendimiento de rasterización. Sin embargo, sus características permiten abordar aplicaciones más generales, pudiendo ser éstas aceleradas sobre plataformas basadas en GPU. Esta nueva tendencia recibe el nombre de *General Purpose Computing on GPU* (GPGPU) [1, 51], lo cual significa utilizar las GPUs para aplicaciones para las cuales no fueron originalmente diseñadas. Eso sí, estas aplicaciones deben tener paralelismo intrínseco y un requerimiento computacional intensivo para obtener un buen rendimiento.

Posiblemente, todavía en la actualidad, el mayor problema radique en la programación de este tipo de dispositivos. Actualmente, las empresas fabricantes de GPUs, como NVIDIA o ATI, han propuesto nuevos lenguajes o incluso extensiones para los lenguajes de alto nivel más utilizados. Un ejemplo es la propuesta de NVIDIA con CUDA[2], la cual consiste en una plataforma software que permite utilizar las GPUs para aplicaciones de propósito general, con la capacidad de manejar un gran número de hilos.

En relación a las CPUs, la arquitectura de las GPUs tiene importantes diferencias. Ambas arquitecturas utilizan tecnologías de diseño y fabricación similares, pero se diferencian en el uso que hacen de los recursos. Las CPUs dedican muchos recursos, a la predicción de saltos, ejecución fuera de orden, etc. En cambio las GPUs utilizan estos recursos para tener un mayor número de elementos de procesamiento. Debido a su gran número de elementos de procesamiento y gran ancho de banda con memoria, las GPUs tienen una potencia de cálculo muy superior al de las CPUs.

En lo referente al modelo de ejecución de instrucciones y control, también son diferentes. Así, el modelo *SISD* (*Single Instruction Single Data*), donde una instrucción es aplicada a un dato, es el usado por la CPU. El modelo

de las GPUs es el modelo *SIMD* (*Single Instruction Multiple Data*), donde una única instrucción es aplicada a un conjunto distinto de datos.

La diferencia también se refleja en los objetivos; el objetivo de las CPUs es explotar al máximo el paralelismo del código secuencial a nivel de instrucción. En cambio el objetivo de las GPUs es aumentar la capacidad de procesamiento explotando el paralelismo a nivel de datos para problemas paralelos o que se puedan paralelizar.

Modelo de Programación CUDA de NVIDIA

El modelo de Programación *CUDA* de *NVIDIA* considera a la GPU como un dispositivo capaz de ejecutar un alto número de hilos en paralelo. Este modelo hace una distinción entre el código ejecutado en la CPU (*host*) con su propia DRAM (*host memory*) y el ejecutado en GPU (*device*) sobre su DRAM (*device memory*). CUDA incluye herramientas de desarrollo software *C/C++*, librerías de funciones, y un mecanismo de abstracción que oculta los detalles hardware al desarrollador, todo ello formando una *Application Programming Interface* (*API*) propia.

En CUDA los cálculos son distribuidos en una malla o grid de bloques de hilos. Todos los bloques de hilos tienen el mismo tamaño (número de hilos). Estos hilos ejecutan el código de la GPU, denominado *kernel*. Un *kernel* CUDA ejecuta un código secuencial en un gran número de hilos en paralelo. Los hilos en un bloque (1024 para las últimas generaciones *Fermi* y *Kepler* de NVIDIA) pueden trabajar juntos eficientemente, intercambiando datos localmente en la *memoria compartida* y sincronizando la baja latencia en la ejecución mediante puntos de sincronización llamados *barreras*.

Por el contrario, los hilos ubicados en diferentes bloques pueden compartir datos, solamente accediendo a la *memoria global* de la GPU, que es una memoria de alta latencia.

La dimensiones de la malla y de los bloques de hilos deben ser cuidadosamente elegidos para maximizar el rendimiento de la GPU.

Ejecución de un Programa CUDA sobre una GPU

Una GPU sólo puede ejecutar las funciones declaradas como *kernels* dentro del programa, y sólo puede usar variables alojadas en la *memoria global* de la GPU, lo cual significa que todos los datos necesarios deberán ser movidos a esta memoria previa ejecución del *kernel*. Al ser invocado un *kernel*, éste se ejecutará N veces en paralelo (N hilos distintos).

Los threads (hilos) son organizados en *bloques*, y un bloque se ejecuta completamente en un único multiprocesador. Debido a esto todos los threads de un bloque comparten la misma *memoria compartida* (*shared memory*). Un *kernel* puede ser ejecutado con un máximo de 1024 threads por bloque (*Fermi* y *Kepler*). Los threads de multiprocesadores distintos no pueden ser

sincronizados, sólo lo pueden hacer los que pertenecen al mismo multiprocesador (con la función *__syncthreads()*).

Para soportar la gran cantidad de threads en ejecución se emplea la arquitectura *SIMT* (*Single-Instruction, Multiple-Thread*). La unidad de ejecución no es un thread, sino un *warp*, el cual es un conjunto de 32 threads (y un *half-warp* son los primeros o segundos 16 threads). La forma en que un bloque es dividido en *warps* es siempre la misma, los primeros 32 threads representan el primer *warp*, los 32 threads consecutivos (según el ID del thread) representa 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 (if, switch, do, for, while) pueden disminuir significativamente el rendimiento del programa, pues al haber dos (o más) caminos de ejecución, estos son serializados, aumentando el número de instrucciones y forzando la ejecución del *warp* sólo con los threads correspondientes (deshabilitando el resto).

2.9.4. Clusters Multicores

En la actualidad, la mayoría de los computadores son multicores, por lo que se podría decir que los actuales clusters podrían ser utilizados como híbridos y desarrollar aplicaciones que permitan utilizar tanto la memoria distribuida como la compartida. Aún más, si se incorporan GPUs se podría disponer de una herramienta de muy altas prestaciones.

Hoy existen varias aplicaciones desarrolladas para clusters de GPUs, que van desde la mecánica de fluidos, programación lineal, estudios de proteínas y dinámica molecular, entre muchos otros [36, 91, 92].

La revisión de Noviembre de 2012 del TOP 500 [93], sitúa a dos clusters de GPUs en los primeros 10 puestos, en la segunda y séptima posición respectivamente.

2.9.5. Paralelización de Estructuras Métricas

En la actualidad, existen muchas plataformas y modelos utilizados para la paralelización de estructuras métricas.

Como se ha mencionado al inicio de la sección, existe un número importante de trabajos de investigación orientados hacia la implementación de estructuras métricas sobre memoria distribuida en clusters de PCs. En este contexto, en la literatura se encuentran trabajos que utilizan tanto las librerías de paso de mensajes MPI o PVM, como a su vez los métodos síncronos y asíncronos para la implementación de los programas paralelos.

Algunos de los primeros trabajos se orientaron básicamente al problema de la distribución de los datos en diferentes discos. En [13] se aborda dicho problema, orientado a resolver el tipo de consultas del *vecino más cercano*;

en este trabajo se propone dividir el espacio en cuadrantes, los cuales serán almacenados en los diferentes discos. En [105] y [3] se muestra información de pruebas realizadas sobre el *MTree*; en este caso, los autores dirigieron sus esfuerzos a optimizar la estructura para adecuarla a nodos distribuidos en plataformas de múltiples discos y múltiples procesadores.

Algunos trabajos se han enfocado a la paralelización de diferentes estructuras métricas sobre plataformas de memoria distribuida usando MPI o BSP. En [83, 55] se presentan varios métodos de paralelización de los algoritmos de construcción y búsqueda sobre la estructura *EGNAT*, analizando estrategias de distribución local y global de los datos sobre un cluster de PCs. En [42] se presenta un conjunto de alternativas para procesamiento de consultas en paralelo para estructuras basadas en clustering, específicamente se utilizó la estructura denominada *Lista de Clusters*. En este trabajo se implementaron estrategias locales y globales de distribución del índice dando énfasis a un esquema donde el tráfico de consultas es elevado, adecuando la estructura a dicho contexto. Se considera también en este trabajo el comportamiento del índice cuando los centros de la estructura están también distribuidos local y globalmente [33]. Trabajos similares han sido presentados en [40].

Unos pocos trabajos han sido desarrollados para plataformas distintas a clusters de PCs. Ejemplos de estructuras métricas sobre redes Peer to Peer (P2P) se pueden encontrar en [35, 47, 99]. En estos sistemas existe un conjunto de máquinas denominadas peers que se conectan a otras máquinas más potentes (generalmente servidores) denominadas super-peers. Los peers ponen a disposición del sistema sus objetos mediante la indexación de ellos a nivel de super-peers. La solución de una consulta de usuario se inicia enviando la consulta a uno de los super-peers, luego se utilizan índices almacenados en los super-peers y en cada *peer* para recuperar los objetos más similares a la consulta. En [35] se describe una estrategia llamada *SimPeer*, la cual utiliza la estrategia de indexación denominada *iDistance* propuesta en [46] para realizar búsquedas de consultas por rango sobre *super-peers* y peers. Además, se adapta la estrategia de enrutamiento propuesta en [31] para enrutar las consultas entre los super-peers vecinos. Trabajos similares son desarrollados en [54], donde se presentan estrategias de índices globales, las que son consideradas eficientes para procesamiento de consultas en sistemas P2P.

En términos de memoria compartida, [41] propone una estrategia para organizar el procesamiento de consultas sobre espacios métricos en nodos multicores, para lo cual propone combinar procesamiento de consultas multihilo totalmente asíncronas con masivamente síncronas. El cambio entre los modos se ajusta a una regla que determina el nivel del tráfico de consultas.

2.9.6. GPUs y Espacios Métricos

En la actualidad, los artículos relacionados con el presente trabajo sólo abordan soluciones para las consultas *kNN* sin utilizar estructuras de datos, es decir, en general las GPUs básicamente se utilizan para paralelizar búsquedas exhautivas (fuerza bruta) por lo que no se utilizan estructuras métricas. Ejemplo de esto se puede ver en [49], [38] y [20], aunque este último sólo aborda el caso para $k = 1$.

Todos estos trabajos abordan el problema desde el supuesto de que los elementos de la base de datos poseen una gran dimensión (con una gran dimensión intrínseca), lo cual implica poder descartar muy pocos objetos usando desigualdad triangular, y en donde la indexación es ineficiente frente a la búsqueda secuencial.

En [49] se propone dividir la base de datos de elementos (A) y la de consultas (B) en submatrices de tamaño fijo. La matriz resultante es una matriz donde cada elemento representa la distancia entre un elemento de A y uno de B. Cada submatriz de C es resuelta por un bloque, para lo cual cada bloque carga a *memoria compartida* cada submatriz de A y B para poder escribir las distancias resultantes en la submatriz correspondiente. Cada bloque hará lecturas a *device memory* de acuerdo a las cantidades de filas y columnas de las matrices. Teniendo la matriz de distancias resultante, ésta se ordena usando el *CUDA-based Radix Sort* [75] para posteriormente seleccionar los k primeros elementos como resultado final.

En [38] se implementa el algoritmo de fuerza bruta y se propone que cada thread resuelva la distancia de un elemento de la base de datos contra la consulta, para luego ordenar el arreglo resultante con una variante del *insertion sort*.

En general, en los trabajos anteriores la paralelización se hace en los procesos de construir la matriz de distancias y en el proceso de ordenamiento para obtener los resultados.

En [8] existe una diferencia con lo anterior. Estos trabajos proponen dos estrategias, la primera al estilo de los trabajos anteriores. Sin embargo, la segunda estrategia, llamada *Heap Based Reduction* propone resolver una consulta por cada bloque. Después de haber calculado todas las distancias para una consulta (exhaustivamente), envía en cada lanzamiento de kernel un solo bloque, manteniendo un heap por cada thread del bloque. Cada heap de tamaño k se utiliza para almacenar los k vecinos más cercanos a partir de las distancias entre los elementos de la base de datos y la consulta.

Para finalizar este capítulo, se debe indicar que al momento de la redacción de esta Tesis, aparecen varios grupos, al menos dos, uno en Europa y otro en América que están realizando trabajos que involucran espacios métricos y GPUs. Al respecto, el grupo ARTECS de la Universidad Complutense de Madrid publicó el trabajo [9]. En dicho trabajo se presenta una versión de la estructura *Lista de Cluster* implementada sobre una platafor-

ma basada en GPU. Se sabe que grupos en América están trabajando sobre consultas de tipo *kNN*, en particular *All-kNN*, sin embargo, aún no existen trabajos publicados al respecto.

Capítulo 3

Estructuras Métricas

El presente capítulo expone, en detalle, las estructuras métricas (denominadas también *Índices Métricos* o *Métodos de Acceso Métrico*) que son de relevancia para el presente trabajo de tesis. Dichas estructuras se utilizarán, en capítulos posteriores, para realizar comparaciones y/o implementaciones en/sobre los entornos paralelos mencionados anteriormente.

3.1. *GNAT Evolutivo (Evolutionary GNAT)*

El *GNAT Evolutivo* o *EGNAT* (*Evolutionary Geometric Near-neighbor Access Tree*) presentado en [88, 63] es una estructura de datos completamente dinámica, competitiva para búsquedas por similitud en espacios métricos de alta dimensión y con buen desempeño en memoria secundaria. El *EGNAT* pertenece al grupo de algoritmos basados en particiones compactas y es una optimización en memoria secundaria para el *GNAT* en términos de espacio, accesos a disco y evaluaciones de distancia.

El *EGNAT* es un árbol que posee dos tipos de nodos, un *nodo bucket* y un *nodo GNAT*. Los nodos nacen como bolsas o buckets y cuya única información es sólo la distancia al padre, al estilo de las estructuras basadas en pivotes (un solo pivote). Este mecanismo es el que permite disminuir el espacio requerido en disco para almacenar la estructura y lograr mantener un buen desempeño en términos de evaluaciones de distancia. Si un *nodo bucket* se completa, éste evoluciona a un *nodo GNAT*, reinsertando los objetos de la bolsa al nuevo *nodo GNAT* (ver figura 3.1).

Construcción

La construcción básica del nodo de tipo *GNAT* es como sigue:

1. Se seleccionan k puntos (*centros o splits*), $p_1, \ldots, p_k$ de la base de datos que se va a indexar. En este caso, del *bucket* que está mutando a *GNAT*.

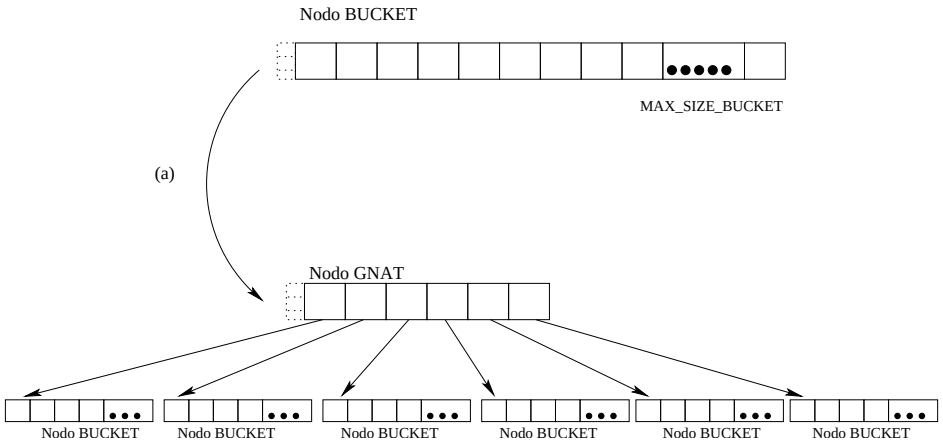


Figura 3.1: *EGNAT*: Mutación de un *nodo bucket* a un *nodo GNAT*.

2. Se asocia cada punto restante del conjunto de datos al split más cercano a él. El conjunto de puntos asociados al split p_i se denota como D_{p_i}.

3. Para cada par de centros (splits) (p_i, p_j), se calcula el rango $range\left(p_i, D_{p_j}\right) = \left[min_d\left(p_i, D_{p_j}\right), max_d\left(p_i, D_{p_j}\right)\right]$, una mínima y máxima distancia $Dist\left(p_i, x\right)$ donde $x \in D_{p_j} \cup \{p_j\}$.

4. El árbol se construye recursivamente para cada elemento en D_{p_i}.

Cada conjunto D_{p_i} va a representar un subárbol cuya raíz es p_i, o lo que es lo mismo, cada D_{p_i} va a corresponder al plano de Voronoi cuyo centro es p_i. En la Figura 3.2 se muestra un ejemplo de construcción del primer nivel de un *GNAT* con *k=4*, también se muestra la tabla de rangos que debe ser almacenada para cada centro p_i. En este ejemplo se insertaron los puntos en orden al valor numérico que tienen.

Búsqueda en el *EGNAT*

En una búsqueda se asume que se desean encontrar todos los puntos con distancia $d \leq r$ al punto q. Durante la búsqueda en un *EGNAT* se determina en primer lugar si el nodo es de tipo *bucket* o *GNAT*.

En el caso de que la búsqueda se realizara sobre un nodo de tipo bucket (tipo bolsa), al mantener la distancia al centro se puede usar la desigualdad triangular para evitar el cálculo directo de la distancia de un objeto consulta sobre los objetos ubicados en los buckets de la siguiente manera:

- Sea q el objeto consulta, p el split que posee un hijo bucket, s_i los elementos ubicados dentro del bucket, r el radio de búsqueda, entonces, si

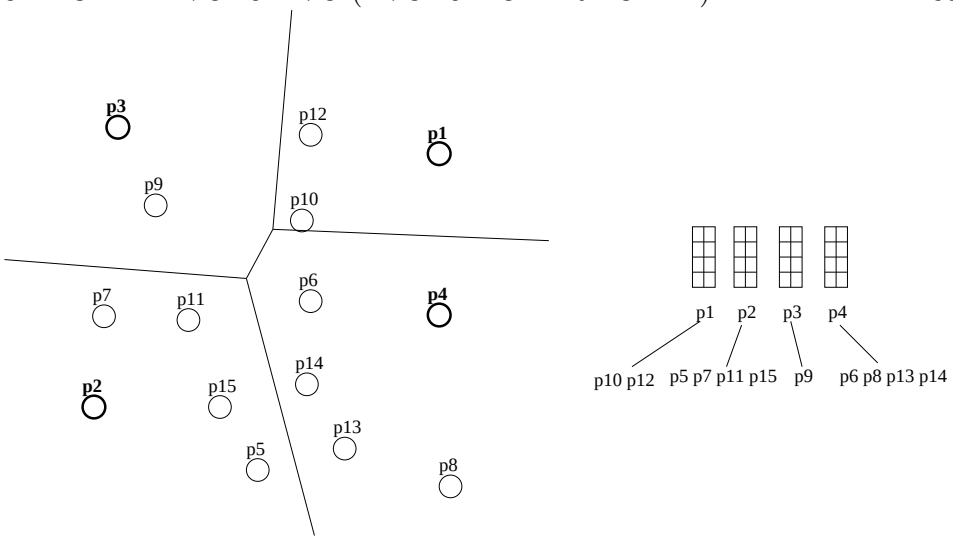


Figura 3.2: *EGNAT*: Partición del espacio, representación de subplanos en un nodo de tipo *GNAT*.

$$Dist(s_i, p) > Dist(q, p) + r \qquad o$$
$$Dist(s_i, p) < Dist(q, p) - r$$

se puede determinar que el objeto s_i no está dentro del rango de búsqueda, de lo contrario hay que necesariamente hacer la comparación directa.

Esto se utiliza también para la búsqueda por rango 0 en los buckets, lo que reduce considerablemente la cantidad de evaluaciones en el momento de eliminar objetos.

Si el nodo es de tipo *GNAT*, la búsqueda se realiza como fue definido originalmente en la estructura *GNAT*.

- Sea P la representación del conjunto de puntos splits del nodo actual el cual posiblemente contiene un vecino cercano a q. Inicialmente P contiene todos los puntos split del nodo actual. La búsqueda se realiza como se muestra en el algoritmo 1.

La razón que permite descartar subárboles durante la búsqueda (la línea 10 del algoritmo) es la siguiente:

Sea un punto $y \in D_x$. Si $d(y, p) < d(q, p) - r$, entonces, por desigualdad triangular, se tiene que $d(q, y) + d(y, p) \geq d(q, p)$, por lo que se deduce que $d(q, y) > r$. Análogamente, si $d(y, p) > d(q, p) + r$, se puede usar desigualdad triangular $d(y, q) + d(q, p) \geq d(y, p)$, para deducir $d(q, y) > r$ (Figura 3.3).

Es interesante notar que para descartar los subárboles durante la búsqueda, el *EGNAT* usa un criterio distinto a los mencionados de hiperplano y

Algoritmo 1 *EGNAT*: Búsqueda por rango r para la consulta q en un nodo de tipo *GNAT*.

busquedarango(nodo P, consulta q, rango r)

1: {Sea R el conjunto de centros a examinar}
2: $R \leftarrow \emptyset$
3: $\{p_0 \in P\}$
4: $d \leftarrow dist(p_0, q)$
5: **if** $d <= r$ **then**
6: se reporta p_0
7: **end if**
8: $range(p_0, q) \leftarrow [d - r, d + r]$
9: **for all** $x \in P$ **do**
10: **if** $range(p_0, q) \cap range(p_0, D_{p_x}) \neq \emptyset$ **then**
11: se agrega x a R
12: **if** $dist(x, q) <= r$ **then**
13: se reporta x
14: **end if**
15: **end if**
16: **end for**
17: **for all** $p_i \in R$ **do**
18: busquedarango(D_{p_i},q,r)
19: **end for**

radio cobertor. El *EGNAT* limita la partición asociada a un centro intersectando anillos alrededor de otros centros, lo que es similar a las técnicas usadas en los algoritmos basados en pivotes. Más detalles de esta estructura se pueden encontrar en [84].

3.2. *Lista de Clusters*

La ***Lista de Clusters*** [23] es una estructura basada en clustering o particiones compactas, la cual es muy similar a una lista enlazada. Fue diseñada para tener un buen desempeño en espacios de altas dimensiones.

En la *Lista de Clusters* se selecciona un centro c perteneciente a la base de datos $\mathbb{Y}$ y un radio r el cual determina la fracción del espacio que abarca la esfera (c, r) definida como el subconjunto de elementos de $\mathbb{Y}$ los cuales están a una distancia no mayor a r del centro c. Luego se define como I a los elementos que están dentro de la esfera de centro c también llamado *Bucket* I, y E definido como el resto de los elementos externos a la esfera de centro c. Este proceso se repite recursivamente. En consecuencia se obtiene una lista compuesta por un centro, un radio y un *Bucket* (c,r,I) denominado *Cluster* (ver Figura 3.4(a)).

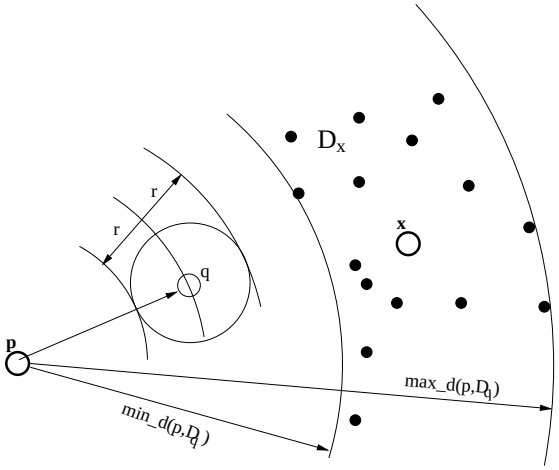


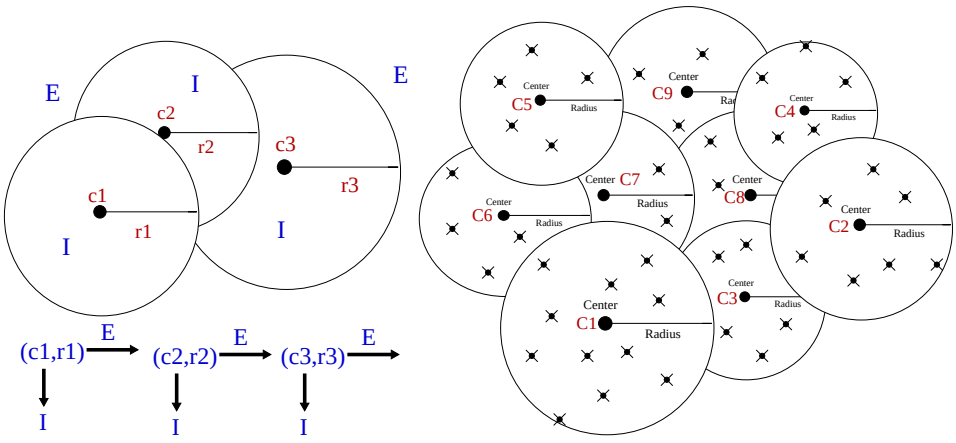
Figura 3.3: Descarte de subárboles usando rangos. Se descarta D_x ya que $d(q,p) + r < min_d(p, D_x)$.

Comparado con otros algoritmos de clustering, la *Lista de Clusters* solamente usa el criterio de radio cobertor y no áreas como en el *Voronoi-Tree*. También es posible ver la *Lista de Clusters* como un caso particular de *Voronoi-tree* o un *M-Tree*, considerando I y E como los subárboles izquierda y derecha de raíz c, con las diferencias de que las estructuras recién mencionadas tratan de construir un árbol balanceado y que además poseen estructuras internas; en cambio la *Lista de Clusters* es extremadamente desbalanceada y no posee estructura interna alguna.

La estructura de datos construida no es simétrica. El primer centro escogido tiene preferencia sobre los centros subsecuentes por lo que se provoca solapamiento entre clusters. La Figura 3.4(b) lo ilustra. Todos los elementos que están dentro del cluster del primer centro ($c1$ en la Figura 3.4(b)) se guardan en su bucket I, a pesar de eso ellos pueden quedar también dentro de los buckets I de centros subsecuentes ($c3$, $c6$, etc. Figura 3.4(b)).

La *Lista de Clusters* provee una característica esencial ausente en otros algoritmos de clustering, en donde la búsqueda necesita entrar en todos los clusters que son intersectados por la esfera de consulta. En esta estructura la búsqueda sobre los clusters restantes puede ser cancelada en cuanto la esfera de consulta esté totalmente contenida en un cluster. En la Figura 3.5 se puede apreciar tres casos de consultas sobre un cluster. En el caso de $q1$ se debe considerar el bucket actual y el resto de los clusters. La bola de consulta para $q2$ está totalmente contenida en el cluster, entonces se realiza la búsqueda sólo en este bucket. Para $q3$ evitamos la búsqueda en el bucket actual.

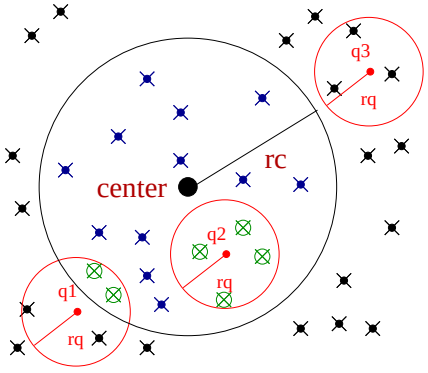
El algoritmo 2 muestra la búsqueda aplicada a una consulta q y un radio



(a) Construcción (b) Distribución general en el espacio

Figura 3.4: Construcción de la *Lista de Clusters*. Indicando en (b) las zonas de influencia de 9 *Clusters* construidos según el orden : $C1$, $C2$, $C3$, $C4$, $C5$, $C6$, $C7$, $C8$ y $C9$.

de búsqueda r sobre la *Lista de Cluster* L.



(a) Tres casos posibles

Figura 3.5: Búsqueda sobre la *Lista de Clusters*.

La estructura posee dos alternativas de construcción, la primera es seleccionar un radio fijo para todos los clusters de la lista y la segunda es seleccionar un tamaño fijo para todos los clusters de la lista, *Cluster de Radio Fijo* y *Cluster de tamaño Fijo* respectivamente.

En las características generales de esta estructura no se especifica como se seleccionan los centros y radios en cada punto del algoritmo de construcción, ya que esto se relaciona con la eficacia y no con la exactitud de la estructura de datos. Una buena selección de centros podría ser más bien

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

busquedarango(lista L, consulta q, rango r)

```
1: if L is empty then
2:    return;
3: end if
4: L = (c, rc, I):E
5: d = dist(c, q)
6: if d <= r then
7:    Agregar c a los Resultados
8: end if
9: if d <= rc + r then
10:   Buscar en I exhaustivamente
11: end if
12: if d > rc − r then
13:   busquedarango(E, q, r)
14: end if
```

costosa, además se debe hacer una selección adecuada del radio para cada centro de cluster. La versión original utiliza el método propuesto para *LAESA* [59] que elige como siguiente centro a aquel que maximice la suma de distancias hacia los centros ya elegidos (ver sección 3.6). En [87, 53, 52] se desarrollan distintas versiones de *LC*, versiones recursivas, con selección de centros por *SSS*, recursivas más *SSS* y otras.

3.3. *Generic Metric Structure* (*GMS*)

Una estructura métrica genérica (*Generic Metric Structure, GMS*) es un índice basado en pivotes y puede considerarse como una tabla de distancias entre los pivotes y los elementos de la base de datos.

Esta estructura no es una idea nueva, aparece en forma implícita dentro de la literatura y en esta tesis se le ha asignado un nombre para ser utilizada en los trabajos realizados. La estructura *GMS* es la forma de reunir algunas antiguas y simples ideas (y la simplificación de otras), pensando en que los procesos de búsqueda finalmente serán llevados a cabo en un dispositivo GPU, y por lo tanto se hace necesario reducir la complejidad de las estructuras de datos y de los algoritmos asociados. En los capítulos desarrollados más adelante, se observará que GMS se adapta mejor al esquema de memoria de una GPU que otras estructuras.

La construcción sería como la indicada en la sección 2.6.2. Un ejemplo de esta estructura puede verse en la figura 3.6.

La búsqueda por rango sobre esta estructura, dada una consulta q y un rango r, es como sigue:

Figura 3.6: Estructura métrica genérica y algoritmo de búsqueda. Ejemplo para una consulta q con intervalos $\{(6,10),(5,9),(2,6),(4,8)\}$ a los pivotes.

1. Para cada consulta q, se calcula la distancia entre q y todos los pivotes $p_1,\ldots,p_k$. Con esto se obtienen k intervalos de la forma $[a_1,b_1],...,[a_k,b_k]$, donde $a_j = d(p_j,q) - r$ y $b_j = d(p_j,q) + r$.

2. Los objetos, representados en la estructura por sus distancias a los pivotes, son candidatos a la consulta q si todas sus distancias están dentro de todos los intervalos.

3. Para cada candidato y_i, se calcula la distancia $d(q,y_i)$, si $d(q,y_i) \leq r$, entonces el objeto y_i es solución a la consulta q.

Detalles de la implementación pueden observarse en el algoritmo 3.

La Figura 3.6 representa una estructura de datos métrica genérica (GMS). Esta estructura es construida usando 4 pivotes para indexar una base de datos de 17 objetos. En este ejemplo la función de distancia es discreta. Para una consulta q con rango $r = 2$ y distancias $d(q,p_j) = \{8,7,4,6\}$ define los intervalos $\{(6,10),\ (5,9),(2,6),(4,8)\}$. Las celdas sombreadas son aquellas que están dentro de los intervalos. Las celdas con trazos de líneas (achuradas) representan a aquellos objetos cuyas distancias están dentro de todos los intervalos y por lo tanto son candidatos. Finalmente, se calcula la distancia directa entre la consulta y los objetos candidatos $(2,13,15)$.

Algoritmo 3 *GMS*: Algoritmo de búsqueda.

rangesearch(query q, range r)

```
1: {Sea 𝕐 ⊆ 𝕏, la base de datos}
2: {Sea P el conjunto de pivotes p_1, ..., p_k ∈ 𝕏}
3: {Sea D la tabla de distancias asociada a q}
4: {Sea S el Índice Métrico}
5: for all p_j ∈ P do
6:    D_j ← dist(q, p_j)
7: end for
8: for all y_i ∈ 𝕐 do
9:    descartado ← false
10:   for all p_j ∈ P do
11:      if D_j − r > S_ij || D_j + r < S_ij then
12:         descartado ← true
13:         break;
14:      end if
15:   end for
16:   if !descartado then
17:      if dist(y_i, q) ≤ r then
18:         agregar y_i al resultado
19:      end if
20:   end if
21: end for
```

3.4. *Spaghettis*

Spaghettis [24] es una estructura basada en pivotes y es una variante de *LAESA* [59]. Propone reducir el tiempo de CPU extra necesario al realizar una consulta utilizando una estructura de datos en donde las distancias a los pivotes están ordenadas por separado, construyendo un arreglo por cada pivote, lo que permite realizar una búsqueda binaria en el rango relevante.

Si para cada pivote se encuentra el conjunto $S_i = \{x : |d(x, p_i) - d(q, p_i)| \leq r\}, i = 1, ..., k$ entonces la lista de candidatos está dada por la intersección de todos estos conjuntos.

Construcción

Durante la construcción de la estructura, se selecciona un conjunto aleatorio de pivotes $p_1, ..., p_k$, los cuales pueden o no pertenecer a la base de datos a indexar. El algoritmo 4 detalla el proceso de construcción. Cada posición en la tabla S_i representa a un objeto de la base de datos que tiene un enlace a su posición en la siguiente tabla, la última tabla enlaza el objeto a su posición en la base de datos.

Algoritmo 4 *Spaghettis*: Algoritmo de Construcción.

1: {Sea $\mathbb{X}$, el espacio métrico}
2: {Sea $\mathbb{Y} \subseteq \mathbb{X}$, la base de datos}
3: {Sea P el conjunto de pivotes $p_1, \ldots, p_k \in \mathbb{X}$}
4: {Sea S_i la tabla de distancias asociada a p_i}
5: {Sea $\cup S_i$ la estructura Spaghettis}
6: **for all** $p_i \in P$ **do**
7: $S_i \leftarrow dist(p_i, \mathbb{Y})$
8: **end for**
9: **for all** S_i **do**
10: Order(S_i)
11: **end for**
12: Cada elemento en S_i almacena su posición en la siguiente tabla (S_{i+1})

Búsqueda

Durante el proceso de búsqueda, dada un consulta q, un rango r la búsqueda por rango sobre un *Spaghettis* tiene básicamente los siguientes pasos:

1. Se calcula la distancia entre q y todos los pivotes $p_1, \ldots, p_k$, para luego obtener k intervalos de la forma $[a_1, b_1], ..., [a_k, b_k]$, donde $a_i = d(p_i, q) - r$ y $b_i = d(p_i, q) + r$.

2. Los objetos que se encuentren en la intersección entre todos los intervalos, se convierten en candidatos a ser respuesta para la consulta q.

3. Para cada objeto candidato y, se calcula la distancia $d(q, y)$ y si es $\leq r$, entonces el objeto y es solución a la consulta.

En la figura 3.7 se representa la estructura *Spaghettis* en su forma original. Ésta está construida usando 4 pivotes para indexar una base de datos de 17 objetos. Se utilizaron los mismos datos usados en la construcción de la estructura GMS de la figura 3.6. Por lo tanto, dada una consulta q con distancia a los pivotes $\{8, 7, 4, 6\}$ y rango de búsqueda $r = 2$, los intervalos generados son: $\{(6, 10), (5, 9), (2, 6), (4, 8)\}$.

En la figura 3.7 se muestran más oscurecidos los intervalos sobre los cuales se realizará la búsqueda. En la misma figura se aprecian con distintos achurados todos los objetos que pertenecen a la intersección de todos los intervalos. Dichos objetos son posibles candidatos a ser solución. Finalmente, para saber si un candidato es respuesta, se debe realizar un cálculo de distancia y determinar si es menor que el rango de búsqueda.

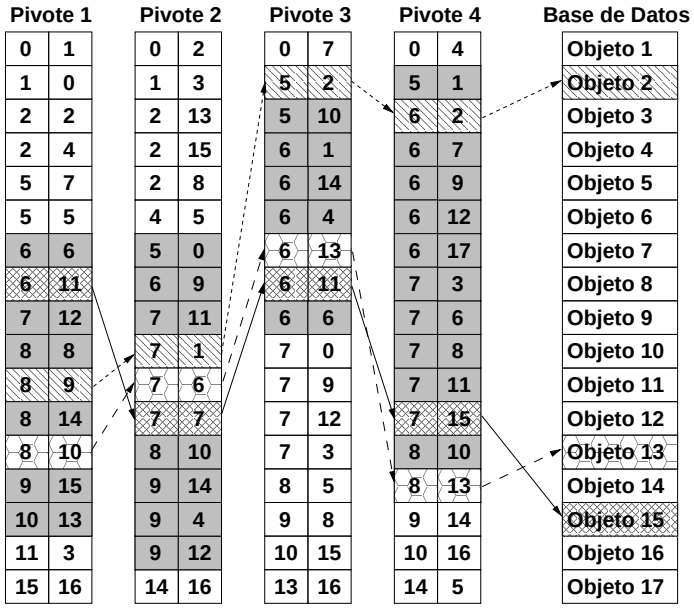


Figura 3.7: Representación del *Spaghettis* una vez finalizado el proceso de construcción y mostrando los intervalos obtenidos para distintas búsquedas.

3.5. *Sparse Spatial Selection (SSS)*

En particular, la elección de pivotes o centros, según sea la estructura, resulta relevante para obtener un mayor rendimiento durante la búsqueda, lo que queda demostrado empíricamente en [21].

Distintas estrategias han sido propuestas como adecuadas para la elección de pivotes. En [59] se propone seleccionar como pivotes aquellos objetos que maximicen la suma de las distancias a los pivotes ya seleccionados. En [16, 101] se siguen heurísticas que tratan de seleccionar pivotes que están lejanos entre sí. En [21] se presenta un criterio de comparación de la eficiencia entre dos conjuntos de pivotes, así también se presentan varias estrategias de selección de conjuntos donde se usa el criterio de eficiencia anterior.

El método denominado *Sparse Spatial Selection* (*SSS*) [70] o también conocido como *SSS-Index*, es una técnica propuesta para la selección de pivotes; fue originalmente implementada sobre una estructura del tipo arreglo y usa la desigualdad triangular para discriminar objetos durante la búsqueda. Dicho método se comportó igual o mejor en términos de eficiencia que los propuestos en [21], con la ventaja adicional de que escoge un conjunto dinámico de pivotes bien distribuidos en el espacio.

Sea $(\mathbb{X}, d)$ un espacio métrico, $\mathbb{Y} \subset \mathbb{X}$ una colección de objetos y M la distancia entre los dos objetos más alejados. En un principio el conjunto de pivotes está formado por el primer objeto de la colección. Luego, para cada

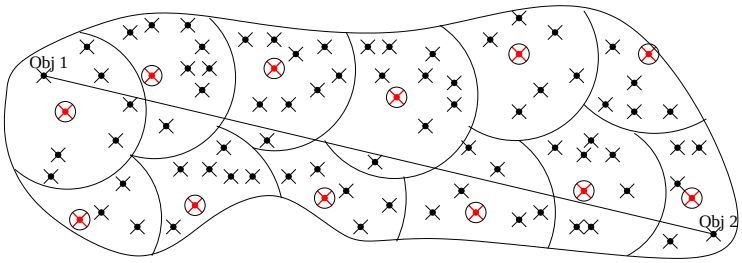


Figura 3.8: Los objetos encerrados en círculos representan los centros seleccionados como pivotes y M se define como la distancia entre *Obj 1* y *Obj 2*, los más alejados en el espacio.

Algoritmo 5 *SSS-Index*: Algoritmo de Selección de Pivotes.

1: {Sea $\mathbb{Y} \subseteq \mathbb{X}$, la base de datos}
2: {Sea M la distancia máxima posible entre 2 elementos de $\mathbb{Y}$}
3: {Sea α una constante real}
4: {Sea P el conjunto de pivotes $p_1, \ldots, p_k \in \mathbb{Y}$}
5: $P \leftarrow \{x_1\}$
6: **for all** $x_i \in \mathbb{Y}$ **do**
7: **if** $\forall p \in P, dist(x_i, p) >= M\alpha$ **then**
8: $P \leftarrow P \bigcup \{x_i\}$
9: **end if**
10: **end for**

elemento de la colección se verifica si está a una distancia mayor o igual a $M\alpha$ de los pivotes seleccionados, si es así, se agrega al conjunto de pivotes, siendo α una constante cuyos valores están cercanos a 0,4 [70]. La Figura 3.8 muestra gráficamente un espacio cualquiera con el conjunto de pivotes (y su radio $M\alpha$) seleccionados con el algoritmo 5.

La construcción es similar a las estructuras *FQA* [25] y *Spaghettis* [24], pero se diferencia en la forma de elegir los pivotes y en la manera de buscar. Básicamente se tiene un arreglo donde la cantidad de filas es la cantidad total de objetos en la Base de Datos y la cantidad de columnas es el número de pivotes.

A la estructura *SSS-Index* puede aplicarse el mismo algoritmo de búsqueda de la estructura *GMS* (algoritmo 3), es decir, se calculan las distancias entre la consulta y los pivotes para posteriormente intentar descartar objetos usando la tabla de distancias almacenada y la desigualdad triangular. Si no es posible descartar un objeto usando todos los pivotes, entonces se debe calcular la distancia directamente entre la consulta y dicho objeto.

3.6. *Linear Approximating Eliminating Search Algorithm* (*LAESA*)

La estructura LAESA [59] (*Linear AESA*) es una mejora propuesta para *AESA* [98]. En general, *LAESA* reduce los costos de almacenamiento y preproceso a cambio de un mínimo aumento en los cálculos de distancia durante la búsqueda. Esta estructura mantiene las mismas características de una estructura basada en pivotes, pero utiliza un método de selección de pivotes conocido como *Suma Máxima de Distancias* (*Maximun Sum of Distances MSD*). La idea de este método es seleccionar un conjunto de pivotes que se encuentren lo más separados entre ellos, logrando así una buena distribución de los pivotes dentro del espacio.

El método para la selección de pivotes se muestra en el algoritmo 6. Se comienza con un pivote base seleccionado arbitrariamente (PB), se selecciona como siguiente pivote base aquél que se encuentre a mayor distancia acumulada con el anterior. El procedimiento continúa calculando la distancia de los sucesivos pivotes base seleccionados al resto de objetos, almacenando las distancias calculadas y seleccionando el siguiente pivote base como aquél para el que la suma de distancias acumuladas al resto de pivotes base seleccionados sea máxima. La condición de finalización se da cuando se ha obtenido el número preestablecido de PB.

LAESA fue diseñado para realizar búsquedas del tipo *kNN*, sin embargo, se puede utilizar para la búsqueda por rango usando el algoritmo 3 de *GMS*. Existen varios trabajos que proponen mejoras sobre la familia de métodos *AESA* como son los expuestos en [57] y [78].

3.7. Comentarios

En términos generales, las estructuras métricas *Spaghettis* [24], *SSS-Index* [70], *LAESA* [59] y otras, pueden ser consideradas como un arreglo bidimensional. Básicamente la diferencia entre éstas es la forma de obtener los pivotes, la forma de almacenar la tabla de distancias y algunos procesos adicionales que permiten mejoras durante la búsqueda. En este sentido, la estructura *GMS* es una simplificación de las estructuras mencionadas.

Como se ha mencionado en el capítulo 2, que se refiere al estado del arte en la presente área, diferentes grupos de investigación se han centrado en torno a estructuras mencionadas y en otras similares en trabajos que abordan temas tales como:

- Aumento de la eficiencia de los procesos de búsqueda en las estructuras,
- Generar nuevos métodos de selección de centros y pivotes, de tal manera que se cubra en forma más adecuada el espacio y por lo tanto se

Algoritmo 6 *MSD*: Algoritmo de Selección de Pivotes para *LAESA*.

1: {Sea $\mathbb{Y} \subseteq \mathbb{X}$, la base de datos}
2: {Sea P el conjunto de pivotes $p_1, \ldots, p_k \in \mathbb{Y}$}
3: {Sea k el número de pivotes }
4: {Sea a un número entero (Pivote Base), $1 \leq a \leq |\mathbb{Y}|$}
5: {Sea S el conjunto de sumas acumuladas, $S_i \in \mathbb{R}$}
6: {Sea MAX valor que almacena suma máxima actual, $MAX \in \mathbb{R}$}
7: **for all** S_i **do**
8: $S_i \leftarrow 0{,}0$
9: **end for**
10: $a \leftarrow random()$
11: $P \leftarrow \{x_a\}$
12: $j \leftarrow 1$
13: **while** $j < k$ **do**
14: $MAX \leftarrow 0{,}0$
15: **for all** $x_i \in \mathbb{Y}$ **do**
16: **if** $x_i \notin P$ **then**
17: $S_i \leftarrow S_i + d(x_i, p_j)$
18: **if** $S_i > MAX$ **then**
19: $MAX \leftarrow S_i$
20: $a \leftarrow i$
21: **end if**
22: **end if**
23: **end for**
24: $P \leftarrow \{x_a\}$
25: $j \leftarrow j + 1$
26: **end while**

reduzcan los cálculos de distancia durante la búsqueda.

- Llevar las estructuras a memoria secundaria y optimizar sus métodos para un mejor desempeño, considerando ahora parámetros como son los accesos a disco entre otros. En la actualidad pocas estructuras métricas están desarrolladas para memoria secundaria, siendo poco viable para las otras estructuras su utilización en aplicaciones reales.

- Paralelización de estructuras métricas. Los trabajos en esta área apuntan tanto a la paralelización de los algoritmos como a la distribución de la base de datos sobre la plataforma disponible.

Capítulo 4

Plataforma Experimental

Este capítulo describe en detalle el ambiente experimental donde se desarrolla la presente Tesis. Se describen los espacios métricos de prueba y la plataforma y arquitectura del hardware utilizada.

4.1. Espacios Métricos

Para los experimentos desarrollados en el presente trabajo se seleccionó una serie de espacios métricos. En una primera etapa se seleccionaron dos espacios considerados representativos para este tipo de problemas.

- El primero corresponde a un diccionario español de 86.061 palabras (denominado también espacio de palabras en español). La distancia utilizada fue la *distancia de edición*, que corresponde al mínimo número de inserciones, eliminaciones o sustituciones necesarias para que una palabra sea igual a otra. Los rangos (o radios) de búsqueda fueron de 1 a 4.

- El segundo espacio, de histogramas de colores, es un conjunto de 112.682 vectores (objetos de dimensión 112) de una base de datos de imágenes. Para este segundo espacio se ha elegido como función de distancia la distancia Euclidiana y se usaron rangos que recuperaban el $0,01\,\%$, $0,1\,\%$ y $1\,\%$ desde el conjunto de datos ($0,051768$; $0,082514$ y $0,131163$, respectivamente). Este espacio es denominado en la Tesis como: "espacio de histogramas de colores" o en su forma genérica "espacio de vectores".

Estas dos bases de datos fueron consideradas como base para la experimentación en la Tesis debido a sus diferencias, por un lado objetos pequeños (palabras) y por otro lado objetos más pesados, es decir, de menor y mayor requerimientos de almacenamiento. Además, representan, el primero, a una función de distancia discreta y el segundo a una función de distancia

continua. Finalmente, el espacio de palabras tiene dimensión intrínseca alta (cercana a 45) versus el espacio de vectores cuya dimensión representacional es alta, pero cuya dimensión intrínseca es baja (2,74). Ambas bases de datos pueden ser encontradas en *The Metric Spaces Library* [37].

Para los experimentos presentados en este trabajo de Tesis ambas bases de datos se dividen en dos conjuntos aleatorios, el primero, de un 90 % de los datos se utiliza para construir la estructura y el 10 % restante se utiliza como consultas. Espacios de menor dimensión no fueron considerados dado que el interés es sobre espacios de dimensión mayor. Los costes de construcción no fueron considerados en los tiempos medidos ya que se considera como etapa de preprocesado de datos.

Para una segunda etapa, otros espacios métricos fueron considerados, entre ellos:

- Rostros: Esta base de datos fue creada de una colección de 8.480 imágenes de rostros obtenidas de *Face Recognition Grand Challenge* [71] (http://www.nist.gov/itl/iad/ig/frgc.cfm). El *Eigen Face Method* [81] fue aplicado para obtener una matriz de proyección, la que puede ser usada para generar vectores de características de cualquier imagen facial. Cada *eigenface* consiste en un vector de 254 componentes. Sobre esta colección se usó una distribución de probabilidad empírica para generar una nueva colección de 120.000 objetos. Los rangos de búsqueda utilizados fueron: $67,831703$; $70,862602$ y $74,811996$ que permiten recuperar el $0,01\,\%$, $0,1\,\%$ y $1\,\%$ de los datos.

Para este espacio de alta dimensión representacional (254) y alta dimensión intrínseca (65,77) se utilizó como medida de similitud la distancia euclidiana y fue dividida en conjuntos de 80 % y 20 % para la construcción de la estructura de datos y para las consultas, respectivamente. Este espacio fue seleccionado para ampliar las pruebas experimentales de la sección 7.2 a un espacio de alta dimension representacional y alta dimensión intrínseca.

A su vez, para pruebas de robustez y escalabilidad de consultas del capítulo 6, los dos espacios de referencia fueron extendidos. Para el caso del espacio de palabras se utilizaron juegos de consultas de entre 10.000 y 40.000 obtenidas de un log de consultas de http://www.todocl.cl. Para el caso del espacio de vectores de dimensión 112, se extendió sintéticamente a un millón de objetos y así formar conjuntos de entre 100.000 y 900.000 consultas sobre una base de datos de 100.000.

Adicionalmente para las pruebas de escalabilidad de la base de datos y consultas del capítulo 6, se utilizaron dos conjuntos de prueba específicamente para un análisis entre distintas soluciones (sección 6.4.5), esto son:

- *Cophir*: La base de datos de *CoPhIR* (*Content-based Photo Image Retrieval*) (http://cophir.isti.cnr.it/) [14], corresponde a metadatos extraidos de imágenes fotográficas de la red social de compartición de

fotografías *Flickr* (http://www.flickr.com). Es una colección de 106 millones de imágenes que contienen cada una 5 descriptores visuales MPEG-7, especificamente: *Scalable Colour*, *Colour Structure*, *Colour Layout*, *Edge Histogram* y *Homogeneous Texture*. Para los experimentos realizados sobre este espacio, se seleccionó un conjunto de 999.987 objetos para construir la estructura y un log de consultas reales de 92.677 objetos. Los objetos son representados en vectores de 64 componentes (dimensión representacional o aparente) y se utilizó la distancia euclidiana con función de distancia.

- Se extendió la base de datos de histogramas de colores, ampliándola 10 veces, conteniendo así un total de 1.014.138 objetos para construir la estructura y 112.682 consultas.

4.2. Plataforma Computacional

Se describe en esta sección el sistema de computación que se ha utilizado para evaluar tanto las implementaciones secuenciales como las diversas versiones paralelas. Se trata de una plataforma híbrida o heterogénea compuesta por dos procesadores convencionales o CPUs y cuatro procesadores gráficos o GPUs. Se usarán indistintamente los términos *Host* y CPU para hacer referencia al primero, y los términos *Device* y GPU para referenciar al segundo.

- **Host**: Está compuesto por dos procesadores Intel Xeon E5530 con 4 cores cada uno, a una frecuencia de 2,4 GHz. El sistema de memoria está formado por caches L1 de 4x32 KB para datos y 4x32 KB para instrucciones, L2 de 4x256 KB y L3 de 8 MB, un total de 48 GB de memoria principal DDR3, con un ancho de banda máximo de 25,6 GB/s.

- **Device**: Está compuesto por cuatro tarjetas gráficas NVIDIA Tesla C1060, las cuales tienen 10 clusters de texturas/procesadores (TPC) cada uno con 30 multiprocesadores (SMs) con 8 cores cada uno. En la figura 4.1 se puede observar la arquitectura de esta GPU y en las tablas 4.1 y 4.2 las características hardware y las capacidades de cómputo, respectivamente.

Adicionalmente, en relación al software y herramientas de desarrollo:

- Herramientas de desarrollo: lenguaje C (gcc 4.3.4), librerías OpenMP y SDK v3.2 de CUDA [2] (http://developer.nvidia.com/).

- Sistema Operativo: GNU-Linux, distribución Ubuntu 10.04.3 LTS, para arquitectura de 64 bits.

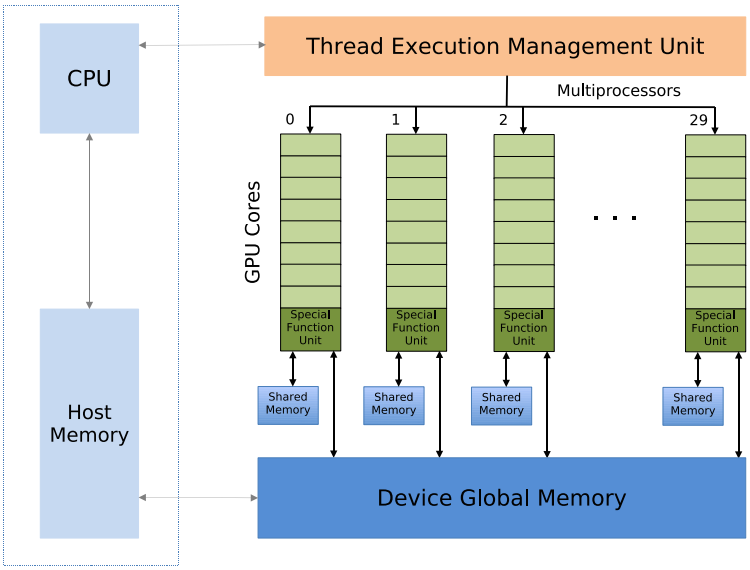


Figura 4.1: Arquitectura de la GPU Tesla C1060.

N. de núcleos	240
Frecuencia de los núcleos	1,296 GHz
Memoria total dedicada	4 GB
Frecuencia de la memoria	800 MHz
Interfaz de memoria	512-bit GDDR3
Ancho de banda de memoria	102 GB/s
Consumo máximo	187,8 W
Interfaz de sistema	PCIE x16 Gen 2
Solución de disipación térmica	Ventilador / Disipador
Entorno de programación	C (CUDA)

Tabla 4.1: Especificaciones técnicas de la GPU Tesla C1060.

Dimensiones máximas x o y de un grid	65535
N. máximo de hilos por bloque	512
Dimensiones máximas x o y de un bloque	512
Dimensión máxima z de un bloque	64
Tamaño de warp	32
N. máximo de bloques residentes por MP	8
N. máximo de warps residentes por MP	32
N. máximo de hilos residentes por MP	1024
N. de registros de 32 bits por MP	16K
Máxima memoria compartida por MP	16KB
N. de bancos de memoria compartida	16
Cantidad de memoria local por hilo	16KB
N. máximo de instrucciones por kernel	2 millones

Tabla 4.2: Capacidad de cómputo de la GPU Tesla C1060.

Capítulo 5

Una Implementación sobre una Plataforma Basada en GPU

El presente capítulo presenta, en una primera sección, las estructuras mencionadas en el capítulo 3 en el contexto de un análisis comparativo entre ellas, de tal manera que permita visualizar el comportamiento de cada una y obtener una perspectiva general. Esto pretende orientar el desarrollo del trabajo hacia una solución adecuada para implementar sobre plataformas basadas en GPU y posteriormente su extensión a plataformas heterogéneas.

En una segunda sección, se desarrolla en detalle una primera versión de la estructura más adecuada obtenida previamente (*GMS*). Se implementa sobre una plataforma basada en GPU y multi-GPUs. Se muestra una visión general del comportamiento de las estructuras sobre un dispositivo como la GPU. Se describen los algoritmos diseñados para la nueva plataforma, incluyendo una versión de búsqueda exhaustiva. Se muestran detalles de como utilizar al máximo algunas características entregadas por este tipo de dispositivo, como es el uso de la memoria compartida. Finalmente, se realiza un análisis comparativo y se presenta una discusión y conclusiones al respecto.

Para esta primera aproximación se han considerado las limitaciones de memoria de la GPU y por ende las restricciones a la cantidad de datos a ubicar en la memoria global del dispositivo, en donde no siempre será posible resolver simultáneamente todas las consultas usando un solo kernel.

5.1. Un estudio Comparativo de Estructuras Métricas: Hacia una Implementación Eficiente sobre GPUs

En la presente sección se ha considerado un conjunto inicial de estructuras métricas para realizar un primer análisis comparativo. Una estructura está basada en clustering y de tipo árbol, otra está basada en clustering, pero de tipo arreglo y el resto están basadas en pivotes y de tipo arreglo. La idea es determinar, experimentalmente, si las diferentes estructuras en términos de costos/beneficios de búsquedas compensan los respectivos costos adicionales que ellas implican. Para esta primera etapa, la elección de las estructuras es principalmente debido a que se podrían considerar como representantes de sus respectivos subconjuntos de estructuras métricas. Para esta primera aproximación se comparan las estructuras: *EGNAT*, *LC*, *SSS-Index*, *MSD-LAESA*, *Spaghettis* y *GMS*. En este sentido la idea es comparar dichas estructuras incluyendo la estructura genérica como alternativa para implementarlas, en un siguiente paso, sobre una plataforma basada en GPUs.

Para los experimentos cada estructura utilizará su método de selección de pivotes correspondiente. Para el caso *GMS* se hará una selección aleatoria de pivotes, también se hará una selección aleatoria de centros para la estructura *EGNAT*. La estructura *LC* utiliza el método de suma máxima de distancias.

Finalmente se debe indicar que se realizó una modificación al método denominado *Spaghettis*, que consistió en realizar el ordenamiento sólo por el primer pivote y no por todo el conjunto dejando así adyacentes las respectivas celdas que representan a un mismo objeto. Para la *LC* se seleccionó la alternativa de clusters de tamaño fijo.

Para este capítulo se preparó un juego de experimentos con el objetivo de tener una perspectiva más clara de algunas características importantes a la hora de implementar una estructura sobre una plataforma del tipo GPU, con las restricciones que ello implica. Los aspectos considerados son:

Tiempo de ejecución. Claramente, el tiempo de ejecución es relevante a la hora de determinar qué algoritmo es mejor en una plataforma secuencial. Existen varios trabajos anteriores que permiten tener una perspectiva amplia del comportamiento de estos algoritmos. Además, muchos realizan análisis comparativos entre las nuevas estructuras propuestas y las anteriores. Sin embargo, en su mayoría, las mediciones corresponden a evaluaciones de distancia, no considerando los tiempos de procesamiento, tanto de operaciones como de I/O, accesos de memoria, etc ([70, 63] y otros ejemplos).

Evaluaciones de distancia. La reducción de cálculos de distancia ha sido el objetivo básico de casi todos los diseños de estructuras y de los algo-

ritmos respectivos. Esto, por supuesto, está en relación al tiempo consumido por el algoritmo durante los procesos de búsqueda. Sin embargo, en la actualidad y con el alto rendimiento de los nuevos dispositivos, no siempre el cálculo de distancia es la operación más costosa, por ejemplo, puede ser también los accesos o cuellos de botella en algunos dispositivos durante la paralelización de algunos algoritmos. Es decir, existen situaciones, en donde operaciones adicionales durante la búsqueda pueden afectar los tiempos independientemente de la reducción de cálculos obtenidos.

En muchas estructuras se puede obtener un aumento del rendimiento en términos de los dos aspectos antes mencionados. Esto, usualmente es factible de hacer (hasta un cierto punto) al aumentar el número de pivotes o centros de una estructura.

Requerimientos de espacio. Una característica interesante a evaluar es el costo de almacenar una estructura en memoria. El trabajo desarrollado en este capítulo ha considerado solamente el tamaño de los objetos en memoria principal ya que el almacenamiento en memoria secundaria queda fuera del estudio de estos experimentos.

A pesar de que la capacidad de almacenamiento ha aumentado en los dispositivos gráficos, no es suficiente para el problema considerado. El objetivo es que la estructura a implementar no exceda en demasía los costos de almacenamiento. La pregunta a responder es: ¿Cuánto se está dispuesto a sacrificar en términos de espacio versus un aumento de las prestaciones?

5.1.1. Resultados Preliminares

Para todas las estructuras, la cantidad de pivotes seleccionados fueron: $26, 44, 82, 328, 665, 1362$ para el espacio de palabras en español y $30, 35, 44, 57, 74, 119, 155, 244$ para el espacio de histogramas de colores. Estos valores corresponden a los obtenidos inicialmente al variar el parámetro α del método *SSS*. Una vez obtenidas dichas cantidades de pivotes, cada estructura fue construida usando su correspondiente método de selección. Para *GMS*, *Spaghettis*, *LAESA* y *EGNAT* se agregan cantidades de 32 y 500 para el espacio de palabras en español y 32 para el espacio de histogramas de colores (vectores) debido a que arrojaron buenos resultados preliminares. Para cantidades menores a 32 el rendimiento de las estructuras se degrada en demasía, por lo tanto dichos experimentos no son incluidos en las gráficas. En los gráficos tampoco se muestra valores superiores a 328 para el *EGNAT*, debido a que los requerimientos de memoria para construir nodos de este tamaño son excesivoas, superando además ampliamente la capacidad de la memoria principal. Nótese que para el método *SSS* no se puede obtener un valor exacto de pivotes, por ello para esta estructura se inicia en caso del espacio de palabras en español en 44 pivotes y para vectores 35 pivotes. Para el espacio de palabras en español, siendo la distancia discreta, no existe

un valor intermedio entre 328 y 665, por lo que no existe el valor 500 en el *SSS-Index*.

En los gráficos de las figuras 5.1 y 5.2 se muestra en el eje de las abscisas el número de pivotes de cada estructura, o el número de centros en el caso del *EGNAT*. Sin embargo, para el caso de la estructura *LC*, lo que se representa en este eje corresponde al número de elementos que almacena cada cluster de tamaño fijo, es decir, los valores 32, 64 y 128, los cuales se eligieron ya que dieron los resultados más prometedores.

Las figuras 5.1 y 5.2 muestran los resultados preliminares para las versiones secuenciales de los métodos seleccionados, tanto de las evaluaciones de distancia como de los tiempos reales. Con el objetivo de poder visualizar mejor los resultados, los gráficos muestran sólo los intervalos importantes para las estructuras, entre 32 y 500 para palabras, y entre 32 y 119 para los histogramas de colores. En la figura 5.1, y a efectos de claridad, se muestran en gráficas separadas los resultados correspondientes a los rangos 1 y 2 de los de 3 y 4. Obsérvese que para el caso de la estructura *EGNAT* y *LC*, éstas escapan al rango de las gráficas del primer intervalo, tanto para evaluaciones de distancia como para tiempo, es por ello que se incluyeron en la gráfica de mayor cobertura en el eje y.

En general, las estructuras basadas en pivotes son las de mejor rendimiento y los mejores resultados están en torno a los 32 pivotes. En términos de evaluaciones de distancia, el comportamiento de las estructuras depende del número de pivotes (o centros) y del rango. Es así como para los rangos menores se puede observar un punto de inflexión en torno a los 32 y 57 pivotes (y para algunas estructuras en torno a los 44 y 57 pivotes) para el espacio de palabras en español y en torno a los 32 para vectores. A medida que el rango de búsqueda aumenta, un mayor número de pivotes beneficia el proceso de búsqueda ya que es más difícil descartar objetos a grandes rangos con estructuras pequeñas.

Al aumentar el número de pivotes la cantidad de candidatos a solución de una consulta es menor, por lo que los cálculos de distancia disminuyen. Esto debería afectar positivamente a los tiempos de ejecución, sin embargo, dado el alto rendimiento de algunos dispositivos de procesamiento, la función de distancia no resulta computacionalmente tan pesada, y por el contrario la mayor cantidad de pivotes perjudica el rendimiento. En términos de tiempo de ejecución, a medida que crece el número de pivotes existe un cálculo adicional que hay que considerar, debido a comparaciones y desplazamientos dentro de la tabla de distancias. El caso más notorio se encuentra en el espacio de palabras en español, al observar los gráficos de evaluaciones de distancia y tiempos asociados de la figura 5.1 es posible comprobar esto.

Para las estructuras basadas en clustering, el comportamiento es similar, pero al aumentar el número de centros las estructuras usualmente son más estables, como el caso de *EGNAT*. Esto último no ocurre para el caso de *LC*, en donde hay una notoria pérdida de eficiencia tanto en evaluaciones de

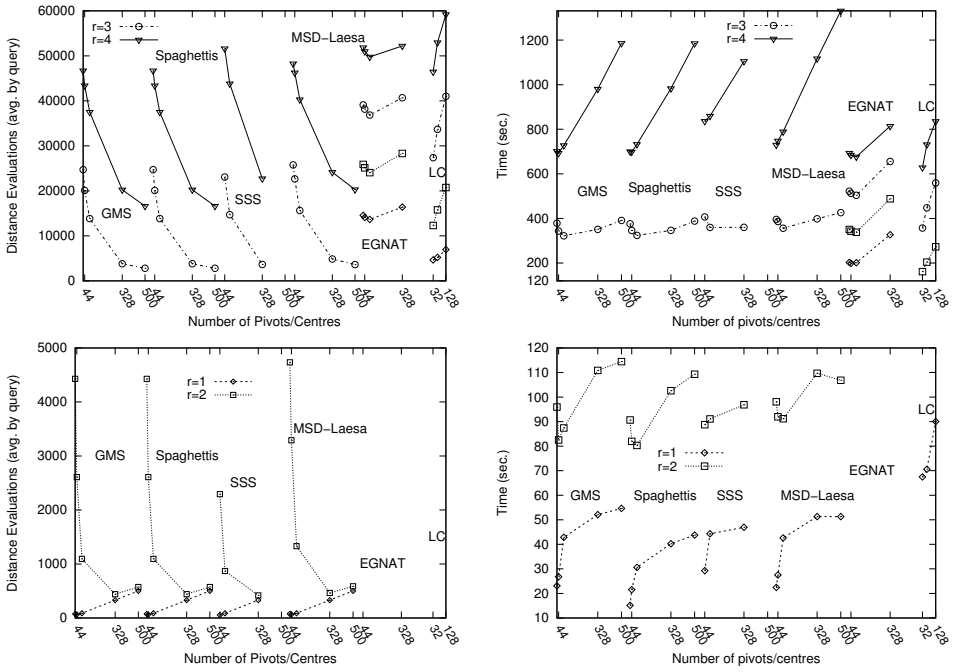


Figura 5.1: Cálculos de distancia promedio por consulta (columna izquierda) y tiempos totales de ejecución (columna derecha) para el espacio de palabras en español y las estructuras *GMS*, *SSS-Index*, *Spaghettis*, *MSD-Laesa*, *EGNAT* y *Lista de Clusters* (*LC*).

distancia como en tiempo de ejecución.

En general, la estructura con mejor rendimiento en rangos bajos es la *Spaghettis* tanto para bajas como para altas cantidades de pivotes. Esto es debido a que la búsqueda binaria realizada para encontrar el rango exacto para el primer pivote optimiza el proceso. La estructura *GMS* es la que le sigue en rendimiento.

Especial atención debe ser considerada para los resultados obtenidos para las estructuras *SSS* y *MSD*. Para estas estructuras se esperaban rendimientos mejores que para la estructura genérica. Sin embargo, los resultados obtenidos no fueron los esperados, siendo la selección aleatoria de pivotes la de mejores prestaciones. Es importante mencionar, en este sentido, que la búsqueda en espacios métricos depende mucho de la forma del espacio, de la dimensión aparente y de la dimensión intrínseca. Es por ello que muchos métodos no se comportan igual en todos los espacios.

A medida que los rangos o radios de búsqueda aumentan, en ambos espacios, se observa que la ventaja inicial del *Spaghettis* (para radios pequeños) se degrada, es decir, en este caso la búsqueda binaria juega en contra, debido a que resulta ser una operación que incrementa el costo, dado que la cantidad de objetos a descartar es mucho menor que para los primeros ran-

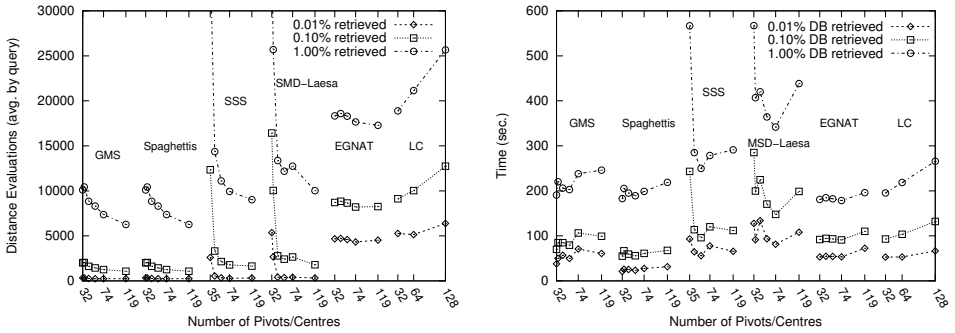


Figura 5.2: Cálculos de distancia promedio por consulta (izquierda) y tiempos totales de ejecución (derecha) para el espacio de histogramas de colores y las estructuras *GMS*, *SSS-Index*, *Spaghettis*, *MSD-Laesa*, *EGNAT* y *LC*.

gos. Para los rangos mayores, las dos estructuras de mejor desempeño son el *EGNAT* y el *GMS*. También se puede observar, que para rangos bajos la *LC* tiene mejores prestaciones que el *EGNAT*, pero no es mejor que las estructuras basadas en pivotes.

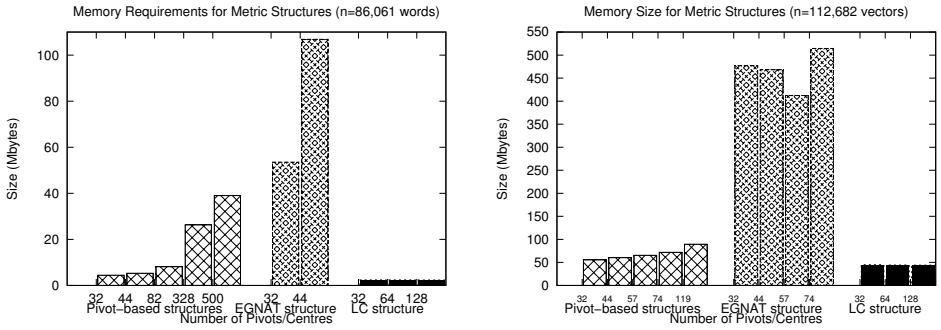
A modo de ejemplo, para las estructuras basadas en pivotes, en el espacio de histogramas de colores y el mayor rango, los objetos descartados son menos del 50 % de la base de datos. Para la base de datos de palabras cuya dimensión intrínseca es alta, la cantidad de objetos descartados va desde los 36,5 % para el primer rango, a sólo el 3 % para el rango mayor (con un solo pivote).

Las tablas 5.1 y 5.2 muestran en detalle, para ambos espacios, las mejores alternativas de las estructuras presentadas. Adicionalmente se agregó una serie de versiones modificadas de la estructura genérica. Para estas versiones se utilizaron los pivotes generados por mezclas de los métodos anteriores. De la base de datos se obtiene primero un subconjunto con *SSS* o aleatorio y sobre éste se aplica *MSD* para fijar el número de pivotes en el de mejor comportamiento 32 o 44 según el espacio. Las tablas mencionadas sólo incluyen las alternativas de mejores resultados, es así como en el caso del espacio de histogramas de colores, usar *SSS* para obtener un primer subconjunto y luego aplicar MSD dio peores resultados que todos los motrados en la tabla 5.2. En general resultados obtenidos para estos últimos casos son satisfatorios pero no mejoran al *Spaghettis* en los radios más pequeños. Sin embargo, a medida que el radio aumenta, estas alternativas se vuelven más competitivas. La ventaja de usar *MSD* sobre un conjunto aleatorio mayor es que permite primero elegir los mejores pivotes para el subconjunto y asegurar que la estructura tendrá un tamaño determinado, por ejemplo para poder ajustar la estructura a nodos que quepan en una página de disco o que se ajusten al espacio disponible en la memoria de una GPU.

Un tema de interés es el espacio requerido por las estructuras. Este as-

Tabla 5.1: Tiempos de ejecución para los mejores métodos en el espacio de palabras en español. (columna: rango; fila: estructura)

Índice	1	2	3	4
Spaghettis 32	15,14	90,66	375,80	699,58
MSD 32 on GMS 665	22,51	89,86	382,35	703,83
MSD 32 on GMS 1362	23,05	90,15	380,10	710,83
MSD-Laesa 32	22,46	98,05	395,62	729,82
GMS 32	23,08	95,93	378,41	700,94
MSD 32 on SSS 665	25,05	92,33	394,45	747,89
MSD 44 on GMS 1362	25,78	76,64	344,74	705,13
MSD 32 on SSS 1362	24,83	96,33	398,35	738,19
SSS-Index 44 ($\alpha = 0{,}55$)	29,25	88,75	406,80	835,76



(a) Resultados generales para el diccionario español.

(b) Resultados generales para los histogramas de colores.

Figura 5.3: Tamaño de las principales estructuras, representativas de las basadas en pivotes (*GMS* y otras) y las basadas en clustering (*EGNAT* y *LC*).

Tabla 5.2: Tiempo de ejecución para los mejores métodos en el espacio de histogramas de colores (columna: porcentaje de datos recuperados; fila: estructura de datos)

Índice	0,01	0,1	1,0
Spaghettis 32	21,00	54,90	182,87
GMS 32	37,77	69,89	190,74
MSD 32 on GMS 119	39,28	71,85	190,26
MSD 32 on GMS 1014	46,55	96,10	246,91
EGNAT 32	53,01	91,86	180,89
SSS-Index 57 ($\alpha = 0{,}6$)	55,77	95,91	249,85
MSD-Laesa 35	91,33	199,75	406,99

pecto puede hacer poco factible la implementación de una estructura sobre un dispositivo como la GPU. La figura 5.3 muestra que la estructura *EGNAT* sobrepasa considerablemente a las estructuras basadas en pivotes. Además, para estas últimas estructuras, como es lógico, el aumento del número de pivotes incrementará notoriamente el tamaño en memoria principal. Para tener una visión más en detalle, se eliminaron los valores más altos del gráfico, en el caso del espacio de histogramas de colores, el *EGNAT* de 119 centros tiene una tamaño cercano a los 2 Gbytes de memoria. Para el espacio de palabras en español, el comportamiento es mucho peor, teniendo el *EGNAT* de 328 centros un tamaño aproximado de 6 Gbytes de memoria (con 84 centros el tamaño de la estructura es sobre los 400 megabytes).

En general, las estructuras basadas en clustering requieren de menos espacio de almacenamiento que las basadas en pivotes, como es el caso de *LC*, sin embargo, esto no ocurre con *EGNAT*. Esto es principalmente debido a que *EGNAT* es una estructura diseñada y optimizada para memoria secundaria. Cada vez que genera un nuevo nodo en el árbol, éste tiene un tamaño un mínimo de una página de disco, que para estos experimentos fue de 4.096 bytes.

Las estructuras de tipo árbol poseen un buen desempeño cuando los radios aumentan y tienen la ventaja, como se puede observar de los gráficos, que son estables y que no varían mucho los costos al aumentar el número de centros o pivotes, por lo que al utilizar un número mínimo de estos se obtienen buenas prestaciones. La desventaja de estas estructuras es que al generar un nuevo nodo del árbol, no existe garantía de que éste se llenará, es por eso que las estructuras crecen enormemente dependiendo de como se distribuyen los objetos dentro de los subárboles. Para la estructura de tipo árbol utilizada en estos experimentos se obtuvo que menos del 20 % de los nodos, en promedio, fueron completados. Respecto de estructuras y memoria secundaria se hablará con más detalle en el capítulo 7.1.

5.1.2. Discusión

De acuerdo a los resultados obtenidos, no es posible seleccionar una única estructura como la mejor, ya que depende de muchos factores, como la distribución del espacio, la dimensión, la selección de pivotes o centros, el rango o radio de búsqueda entre otros.

De las estructuras estudiadas en la sección previa se podría indicar que para el espacio de histogramas de colores, el *Spaghettis* y *EGNAT* han tenido los mejores resultados para bajos y altos rangos, seguidos del *GMS*. Para el caso del espacio de palabras en español, el *Spaghettis* es seguido por combinaciones del *GMS* y otras técnicas (*GMS+MSD*).

Uno de los puntos a considerar para descartar en esta etapa al *EGNAT* es la cantidad de espacio requerido por la estructura para almacenar los datos. Otro punto en contra para esta estructura son las modificaciones necesarias para su implementación sobre un disposivo como la GPU. Sobre una GPU es necesario que las estructuras sean lo suficientemente sencillas para que los hilos (*threads*) puedan dedicarse principalmente a resolver la operación. En este caso, una estructura de tipo árbol (que tendría que ser convertida a un arreglo), que requiere saltos de posiciones dentro de la tabla, restaría rendimiento, ya que un hilo puede detener la ejecución de otros. Se debe recordar que, en el caso de los hilos, lo ideal es que el conjunto de hilos de un *warp* puedan ir moviéndose secuencialmente, además de aprovechar la coalescencia de los datos, la que estaría ausente en este tipo de estructuras.

La simplicidad de la estructura *GMS* y en consecuencia de aquellas de tipo arreglo, permiten obtener mayores beneficios de un dispositivo donde se tiene una gran cantidad de unidades de procesamiento para operaciones sencillas. En este sentido, el caso de la búsqueda binaria de la estructura *Spaghettis* puede provocar, sobre un dispositivo de tipo GPU, más problemas que beneficios. En una GPU, el mayor rendimiento se obtiene cuando un conjunto de hilos procesan datos en posiciones contiguas de memoria, evitando saltos entre distintas posiciones.

Mención aparte es la potencia de cálculo requerida por la función de distancia. Como es obvio, una función más "pesada" infuirá en el comportamiento de una estructura. En este sentido, cuando el costo de la función de distancia es alta, se obtienen beneficios, en términos de tiempo de ejecución, de estructuras con mayor cantidad de pivotes. Todo lo contrario sucede cuando la función de distancia no requiere mucha potencia de cálculo, es decir, un número excesivo de pivotes provoca aumento en el tiempo de ejecución.

5.2. GMS Basada en GPU

Para la estructura *GMS*, el algoritmo de búsqueda se divide en las tres partes de mayor coste computacional. Las tres partes fueron descritas y enumeradas en la sección 3.3 y se resuelven cada una en un kernel distinto. En

Algoritmo 7 CUDA: Kernel generador de distancias.

__global__ KDistances(queries Q, pivots P, distances D)

1: {Let Q be the set of queries}
2: {Let P be the set of pivots}
3: {Let D be the table of distances associated to Q}
4: {Let i be thread Id }
5: **for all** $p_j \in P$ **do**
6: $\quad D_{ij} \leftarrow d(q_i, p_j)$
7: **end for**

primera instancia, el uso de tres kernels es para evitar transferir todos los datos y estructuras de datos a la GPU. Otros trabajos publicados recientemente (menos de 6 meses a la redacción de este informe) implementan versiones asumiendo que toda la información puede ser contenida en la memoria de la GPU, base de datos, estructura, consultas, pivotes y otros datos; lo que dificulta su implementación para aplicaciones reales [8, 9]. Los tres kernels corresponden a:

1. Para optimizar el uso de la GPU, se resuelve en un único kernel la primera parte del algoritmo, es decir, se realiza el cálculo de las distancias entre todas las consultas Q y todos los pivotes P. El proceso consiste en lanzar el kernel con tantos hilos como número de queries ($|Q|$), tal que, cada hilo resuelve de forma independiente las distancias de cada query a todo el conjunto de pivotes. Como resultado, el kernel genera una matriz de tamaño $|Q| \times |P|$ con las distancias correspondientes (D). El algoritmo 7 muestra el seudo código de este kernel.

2. La segunda parte de la paralelización se encarga de determinar si un elemento es o no candidato a una consulta. Para ello se lanza un segundo kernel que ejecuta un hilo por cada elemento (y_i) de la base de datos y determina si dicho dato es o no candidato, es decir, este kernel al finalizar retorna una lista de candidatos para una consulta dada (C). Esta tarea es llevada a cabo en el kernel denominado *KCandidates* (ver algoritmo 8).

3. El último kernel se encarga de determinar qué candidados (obtenidos en el paso 2) son realmente solución a la consulta q. Este kernel recibe como parámetros la lista de candidatos C y la consulta q. El número de hilos lanzado corresponde al número de candidatos para cada consulta y cada hilo resuelve si un candidato es o no solución. Al término del tercer kernel se retorna una lista con las soluciones para la consulta (ver algoritmo 9).

Finalmente, el esquema general para la búsqueda por rango se muestra en el algoritmo 10.

Algoritmo 8 CUDA: Algoritmo de búsqueda, obtención de candidatos.

__global__ KCandidates(range r, GMS S, distances D, candidates C)

1: {Let P be set of pivots $p_1, \ldots, p_2 \in \mathbb{X}$}
2: {Let D be the table of distances associated q}
3: {Let C be list of candidates for q}
4: {Let i be thread Id }
5: {Let `MAX_PIV` be the number of pivots of structure}
6: $discarded \leftarrow false$
7: **for** $j \leftarrow 1$ to `MAX_PIV` **do**
8: **if** $D_j - r > S_{ij} \,||\, D_j + r < S_{ij}$ **then**
9: $discarded \leftarrow true$
10: break;
11: **end if**
12: **end for**
13: **if** !$discarded$ **then**
14: add to C (candidates)
15: **end if**

Algoritmo 9 CUDA: Soluciones finales para la consulta q.

__global__ KSolutions(range r, database $\mathbb{Y}$, candidates C, query q, solutions R)

1: {Let $\mathbb{Y} \subseteq \mathbb{X}$ be the database}
2: {Let C be list of candidates for q}
3: {Let R be list of solutions for q}
4: {Let i be thread Id }
5: **if** $d(c_i, q) \leq r$ **then**
6: add to R (solutions)
7: **end if**

5.2.1. Memoria Compartida (*Shared Memory*)

Internamente la GPU posee su propia jerarquía de memoria. En este sentido, la memoria compartida (*Shared Memory*) viene a ser un segundo nivel dentro de la jerarquía, siendo la memoria global (*Device Memory*) el primer nivel. Como se indicó en la sección 2.9.3, la memoria compartida es más rápida que la memoria global, pero es mucho más pequeña y está limitada a los hilos de un mismo bloque.

Se puede obtener beneficios de la memoria compartida si se utiliza para almacenar datos que sean usados por muchos hilos de un mismo bloque. En el caso de los algoritmos indicados en la sección 5.2 (algoritmos 7, 8 y 9), los tres kernels utilizan datos comunes a todos los hilos:

Algoritmo 10 CUDA: Programa principal.

Main Program

1: KDistances<<<`NUM_QUERIES`/`BLOCK_size`, `BLOCK_size`>>>(Q,P,D)
2: **for all** $q_i \in Q$ **do**
3: KCandidates<<<`MAX_DATA`/`BLOCK_size`, `BLOCK_size`>>>(r,S,D_i,P,C_i)
4: KSolutions<<<`MAX_CANDIDATES`/`BLOCK_size`, `BLOCK_size`>>>(r,$\mathbb{Y}$,C_i,q_i,R)
5: show R
6: **end for**

- En el primer kernel, los hilos calculan las distancias entre todas las consultas y los pivotes, por lo que cada hilo asignado a una consulta requiere acceder a todas las posiciones de memoria donde se ubican los pivotes. Por esta razón, al ser los pivotes datos comunes a todos los hilos, estos se almacenan en memoria compartida.

- Para el segundo kernel, los candidatos son obtenidos al comparar las distancias obtenidas del kernel 1 para una consulta determinada ($D_i \leftarrow d(Q_i, P)$) contra las distancias almacenadas en la estructura *GMS*. Es decir, para obtener los candidatos de Q_i el parámetro enviado al kernel es el conjunto de distancias asociadas a esa consulta i, es decir, D_i. Este conjunto será común a todos los hilos, por lo que este conjunto de distancias es almacenado en memoria compartida.

- En el tercer kernel es la consulta la que se compara directamente con cada objeto candidato de la base de datos, por lo que la consulta es el dato común y se almacena en memoria compartida.

El algoritmo 11 muestra en detalle la transferencia de datos desde la memoria global del dispositivo a la memoria compartida. El algoritmo es un ejemplo genérico que es aplicado a los algoritmos descritos en esta sección, en este caso es la consulta la que es llevada a memoria compartida. Durante la ejecución del kernel, cada hilo de un mismo bloque colabora en llevar parte de la consulta a la memoria compartida (consulta de tamaño `MAX_DIM`). El último bloque se atiende con especial cuidado debido a que no todos lo hilos deben ser activados. En el algoritmo 11 la variable `last_block` representa la cantidad de hilos utilizados del último bloque.

La cantidad de bloques lanzados en el ejemplo corresponde a `MAX_DATA`/`BLOCK_size`, en donde `MAX_DATA` es la cantidad de hilos requeridos y `BLOCK_size` a la cantidad de hilos que puede lanzar un bloque, lo que depende de la arquitectura.

Algoritmo 11 CUDA: Transferencia de datos a memoria compartida. Ejemplo: consulta q.

__global__ ExampleSM(int `MAX_DATA`, query q)

```
1: {Let x be thread Id }
2: {Let MAX_DATA be maximum number of threads required.}
3: {Let BLOCK_size be the block size}
4: {Let MAX_DIM be the object size}
5: {Let shared[MAX_DIM] be a variable defined into shared memory}
6: {Let last_block and sm be a auxiliar variables}
7: {x ← threadIdx.x + (blockIdx.x * BLOCK_size)}
8: if x < MAX_DATA then
9:   if blockIdx.x < (MAX_DATA/BLOCK_size) then
10:    if threadIdx.x < MAX_DIM then
11:      shared[threadIdx.x] = query[threadIdx.x]
12:    end if
13:  else
14:    if blockIdx.x = (MAX_DATA/BLOCK_size) then
15:      last_block = BLOCK_size − (((int)(MAX_DATA/BLOCK_size + 1) *
         BLOCK_size) − MAX_DATA
16:      for (sm ← threadIdx.x;sm < MAX_DIM;sm+ = last_block) do
17:        shared[sm] = query[sm]
18:      end for
19:    end if
20:  end if
21: end if
22: __syncthreads()
23: /*next process*/
```

5.2.2. Búsqueda Exhaustiva Basada en GPU

Para poder realizar una comparación adecuada se implementa una versión de la búsqueda por rango de fuerza bruta o búsqueda exhaustiva.

Dada una base de datos y una consulta, la búsqueda exhaustiva secuencial es un proceso iterativo donde en cada iteración se calcula la distancia entre la consulta y un elemento de la base de datos para determinar si es o no una solución válida. La implementación paralela de este proceso es trivial, y dada las características de las GPUs, una primera idea consiste en lanzar, por cada iteración, un kernel que resuelva una consulta. Cada kernel ejecutará los siguientes pasos:

- El kernel lanzará tantos hilos como elementos haya en la base de datos.
- Cada hilo resolverá en forma directa la distancia entre la consulta y el elemento sobre el que el hilo ha sido lanzado.

- Después del cálculo de la distancia, se comparará ésta con el rango ingresado y marcará si el objeto es o no solución.

- Retornará al programa principal el conjunto de elementos que son solución a la consulta.

Para el caso del algoritmo de búsqueda exhaustiva, al ser el kernel idéntico al tercero de la estructura *GMS*, es la consulta el dato común a todos los hilos, por lo que es ubicada en memoria compartida. La única diferencia entre ambos kernels es que el exhaustivo recorre toda la base de datos y no sólo los candidatos seleccionados.

5.2.3. Resultados Experimentales y Discusión

De la Sección 5.1.1 se obtienen conclusiones que son utilizadas en esta fase del trabajo. De éstas, las de mayor consideración en términos de tiempo de ejecución, son las indicadas a continuación:

- Todas las estructuras y espacios coinciden en que para los menores rangos la mejor cantidad de pivotes es 32 (o cercana a este valor).

- Para el espacio de los histogramas de colores, para los rangos mayores, que recuperan el $0,1\,\%$ y $1\,\%$, en las estructuras *GMS* y *Spaghettis* el mejor valor sigue siendo 32. Sin embargo, para *SSS-Index* y *SMD-Laesa* los valores ideales están en torno a 57 y 74.

- Para obtener un mayor beneficio de la memoria compartida de la GPU, lo ideal es que los datos necesarios por todos los hilos estén todos almacenados en ella. Para el espacio de histogramas de colores, el valor 32 resulta ideal ya que en todos los kernels es posible llevar los conjuntos de datos seleccionados a memoria compartida, no así para valores que superen esta cantidad.

Finalmente, dada las consideraciones anteriores, se eligió como número de pivotes para la estructura genérica la cantidad de 32. La figura 5.4 muestra la comparación entre las estructuras de mejor rendimiento sobre una plataforma con una GPU. Para todas las estructuras se eligieron las mejores versiones, en cantidades de pivotes, para los rangos más bajos en cada espacio. Con el objetivo de tener una visión más amplia e ilustrar mejor el comportamiento de las distintas estructuras, se incluyó en los gráficos de la figura 5.4 una versión sobre GPU de la estructura *LC*, a pesar de que esta estructura no tuvo un buen rendimiento en los espacios de prueba.

La implementación sobre GPU de los algoritmos de búsqueda para las estructuras distintas a *GMS* implicaron sólo algunas leves modificaciones, al ser todas las seleccionadas basadas en pivotes o de tipo arreglo.

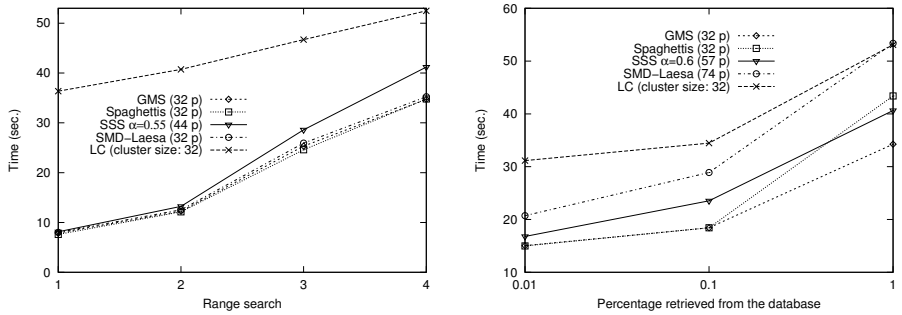
Para la implementación del *Spaghettis* en GPU se utilizó una versión modificada, que sólo está ordenada por el primer pivote. Previo a la llamada al kernel que busca los candidatos, se obtiene con búsqueda binaria el primer objeto dentro de los intervalos generados por la consulta y su posición es pasada al kernel. De esta manera, durante la ejecución del kernel los hilos anteriores a esta posición no son activados. Para el caso de la estructura *Spaghettis* no es adecuado (con la arquitectura actual) implementar una búsqueda binaria sobre la GPU para este tipo de problemas. Los algoritmos de búsqueda convencional tienen ciertas limitaciones en dispositivos como la GPU. El algoritmo de búsqueda binaria al realizar saltos entre posiciones de memoria ralentiza la ejecución de los hilos en una arquitectura del tipo GPU. Este algoritmo tiene características que penalizan el rendimiento en memorias como las presentes en las GPUs, como por ejemplo, patrones de salto poco predecibles, accesos dependientes entre ellos y accesos con poca localidad en los datos.

Para el caso de *LC*, la implementación desarrollada sobre GPU no difiere mucho del algoritmo secuencial presentado en capítulo 3 (algoritmo 2). Esta estructura no posee pivotes y el dato almacenado en memoria compartida es solamente la consulta. La búsqueda se resolvió utilizando un solo kernel. Esta estructura requerirá de un mayor análisis en su diseño sobre GPU para permitir obtener mayor rendimiento sobre este tipo dispositivo. En trabajos como [9] se han presentado versiones para LC que presentan mejores prestaciones.

Se puede observar que las estructuras *GMS* y *Spaghettis* mantienen un comportamiento similar en ambos espacios. Sin embargo, para el espacio de palabras en español y en el rango más pequeño el *Spaghettis* muestra el mejor rendimiento. Igualmente, es importante notar que en ambos espacios esta estructura comienza a generar más costos, en términos de tiempo, para los rangos mayores. Esto último es debido a que el alto rendimiento de la GPU se ve perjudicado por la búsqueda binaria inicial, siendo el caso más notorio el espacio de histogramas de colores.

La diferencia entre las funciones de distancia provoca que la búsqueda tenga comportamientos distintos. La función de distancia euclidiana (para vectores) es más pesada que la función de distancia de edición para palabras debido a las operaciones de punto flotante y la mayor dimensión de los vectores. Esto provoca que en el primer caso se saque mayor beneficio de la estructura de datos, mientras que en el caso de una función con menos requerimientos de potencia computacional se obtienen ventajas mayores al contar con un alto número de unidades de procesamiento.

Las versiones de *SSS-Index* y *MSD-Laesa* no obtuvieron rendimientos adecuados. Además, para el primer kernel, tanto para *SSS-Index* como *MSD-Laesa* en el espacio de histogramas de colores, sus respectivas cantidades de pivotes no entraron completamente en la memoria compartida de la GPU. En estos casos, se llevó a memoria compartida sólo una parte de los pivotes



(a) Diccionario de palabras en español.

(b) Histogramas de colores.

Figura 5.4: Tiempo de ejecución para GPU de las mejores versiones secuenciales.

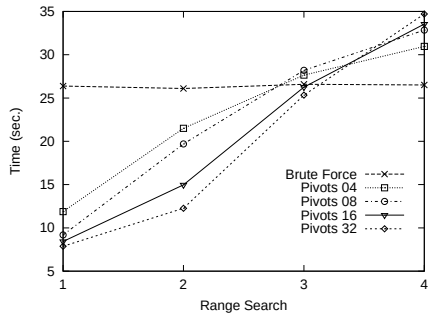
y los otros fueron accedidos desde la memoria global del dispositivo, lo que necesariamente implicó un aumento en el tiempo de procesamiento.

Finalmente, se considera que en términos generales la estructura *GMS* es la que se comporta mejor sobre una plataforma basada en GPU, siendo la estructura que mantuvo un rendimiento más estable en todos los espacios. Además, esta estructura es reprensentativa del conjunto de estructuras basadas en pivotes, por lo que cualquier estructura de este tipo puede utilizar los algoritmos presentados en este capítulo.

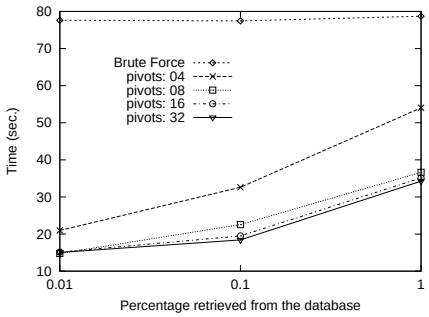
Para tener una visión más amplia del comportamiento de la estructura *GMS* sobre la GPU, la figura 5.5 muestra los resultados para las versiones de 4 a 32 pivotes. Se grafican aquí los tiempos correspondientes al proceso de búsqueda en un estado de régimen permanente del sistema, es decir, los procesos de lectura de archivos no son considerados, sólo la carga de las respectivas consultas y el proceso de búsqueda.

Como se ha comentado previamente, con una cantidad de 32 pivotes (o cercano a 32 en algunos casos) se obtiene un punto de inflexión en la curva, es decir, con valores menores y mayores de pivotes los tiempos aumentan. Se grafica también la versión exhaustiva del algoritmo.

Se puede observar que el comportamiento de la estructura es bastante estable para el espacio de histogramas de colores. Sin embargo, en el espacio de palabras en español el rendimiento de la estructura de 32 pivotes para el rango más alto decrece. Esto se debe a lo indicado previamente, la función de distancia no requiere de mucha potencia de cálculo, por lo que llegado a cierto nivel, aumentar la cantidad de pivotes puede ocacionar mayores demoras al hacer que los hilos recorran o comparen más datos en la estructura. Otro factor importante es que al ser una función discreta, el rango 4 resulta bastante elevado (muchos trabajos utilizan en los experimentos los primeros tres rangos).



(a) Diccionario de palabras en español.



(b) Histogramas de colores.

Figura 5.5: Tiempo de ejecución de estructura *GMS* sobre una GPU para las versiones de hasta 32 pivotes y fuerza bruta. Espacio de palabras (izquierda) e histograma de colores (derecha).

Una observación importante concierne a los resultados obtenidos para la versión en fuerza bruta en el espacio de palabras en español (figura 5.5(a)). Aquí queda en evidencia lo mencionado respecto a la función de distancia. Este es un caso en donde se obtiene un enorme beneficio del uso de la GPU llegando a un speed-up de alrededor de 48 (caso fuerza bruta secuencial versus fuerza bruta en GPU).

La figura 5.6 muestra los correspondientes speed-ups, en este caso, el comportamiento de cada estructura en relación a su versión secuencial. Llama la atención que el menor speed-up conseguido corresponde a la estructura con 32 pivotes, siendo ésta la más rápida. Para entender este resultado, se debe analizar la figura de la siguiente manera: cada speed-up se obtuvo al sacar la razón entre la versión secuencial de la estructura con un número de pivotes determinado versus la misma versión, con el mismo número de pivotes, pero en GPU. Entonces, esto es debido a que la versión secuencial para, por ejemplo, 4 pivotes es de muy bajo rendimiento versus la versión secuencial de 32 pivotes. Por lo tanto el speed-up para la estructura 32 pivotes (secuencial/gpu) no mejora mucho respecto de su versión equivalente secuencial. El rendimiento de la versión de 4 pivotes sobre GPU mejora notoriamente versus su versión equivalente en secuencial.

Se puede notar también que hay diferencias considerables entre los speed-ups para rangos bajos y cantidad de pivotes pequeña versus la estructura con mayor cantidad de pivotes. El rendimiento ofrecido por la utilización de un gran número de hilos de la GPU, dado el alto grado de paralelilzación a nivel de datos, contrariamente aumenta cuando el rendimiento de la estructura decrece. Existen dos casos a considerar cuando ocurre esto, el primero al aumentar el rango, y el segundo cuando se disminuye considerablemente el número de pivotes. Es decir, en ambos casos la tendencia en costos de tiempo es hacia una búsqueda exhaustiva.

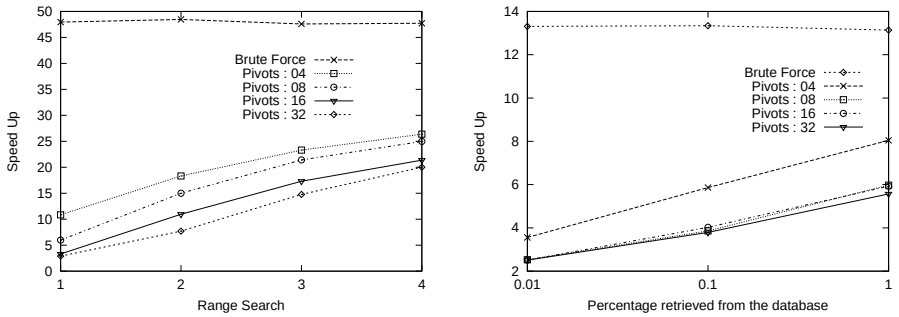


Figura 5.6: Speed-ups para las versiones *GMS* secuenciales sobre las de GPU. Espacio de palabras (izquierda) e histograma de colores (derecha).

5.3. Ventajas de una Plataforma basada en Multi-GPU

Inicialmente se tienen dos alternativas para una implementación sobre una plataforma multi-GPU. La primera implica distribuir la estructura métrica y la base de datos de tal manera que los núcleos o cores procesen sólo parte de cada una. Para este caso las consultas son replicadas sobre todos los cores. Sin embargo, esta alternativa requiere necesariamente una operación de reducción al finalizar todos los cores para reunir todas las soluciones de una consulta. Esta solución es más adecuada para casos donde la estructura no entra completamente en la memoria de la GPU.

La segunda alternativa evita el coste de esperar y reunir las respuestas generadas por cada core. En este caso lo que se distribuye son las consultas y se replica la estructura métrica y la base de datos. Es decir, cada GPU atiende completamente una consulta. En memoria principal la base de datos es compartida por los cores, pero replicada sobre cada GPU.

La implementación de la solución de la sección anterior sobre varias GPUs es la siguiente:

- Considerando que el primer kernel fue diseñado como un proceso previo al ciclo principal de la búsqueda, para su implementación sobre varias GPUs se consideró la distribución del trabajo (ver algoritmo original 7), es decir, se eligió la opción de replicar los pivotes y distribuir las consultas sobre el conjunto de GPUs. En este caso, cada GPU realiza la construcción de una parte de la tabla inicial de distancias $Q \times P$. Al finalizar se reunen las soluciones en la tabla final D.

- El segundo y tercer kernel se replican sobre las GPUs, es decir, en ambos casos se replican los datos necesarios para resolver una consulta en forma independiente. Una vez transferidos los datos iniciales a la memoria global del dispositivo, cada GPU va recibiendo consultas y

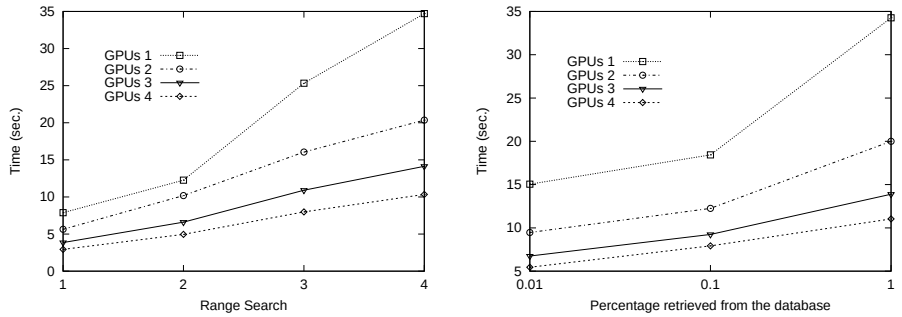


Figura 5.7: Resultados en tiempos absolutos para las versiones *GMS* sobre una plataforma de múltiples GPUs. Espacio de palabras (izquierda) e histograma de colores (derecha).

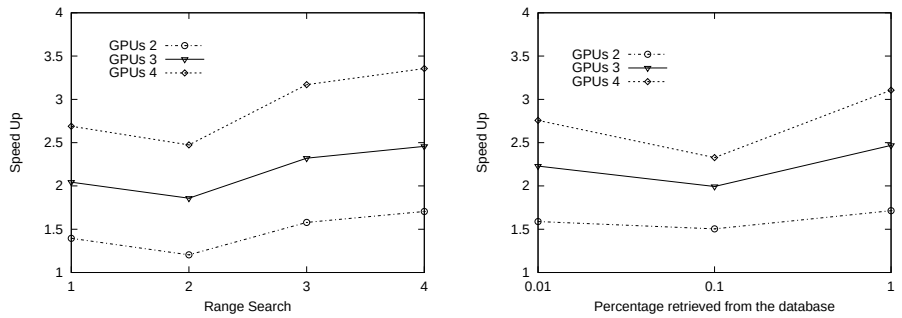


Figura 5.8: Speed-ups para las versiones multi-GPUs versus 1 GPU. Espacio de palabras (izquierda) e histograma de colores (derecha).

entregando las soluciones respectivas, es decir, cada dispositivo actúa como una máquina independiente. En este caso, las GPUs no necesariamente resuelven el mismo número de consultas. En particular, para repartir las consultas sobre las GPUs se utilizó la directiva `#pragma omp parallel for` de *OpenMP*.

La figura 5.7 muestra los resultados para una plataforma con 1 a 4 GPUs para ambos espacios de muestra[1]. La figura 5.8 muestra los speed-ups entre 2, 3 y 4 GPUs versus 1 GPU. La figura 5.9 muestra los speed-ups entre la estructura genérica sobre las distintas cantidades de GPUs versus la mejor versión secuencial, en este caso, el *Spaghettis* de 32.

[1]En esta etapa de la implementación sobre un sistema multi-GPU no se discutirá sobre las implicaciones de los esquemas de planificación en los resultados, se entenderá que la planificación utilizada es la definida por defecto en OpenMP, es decir, estática.

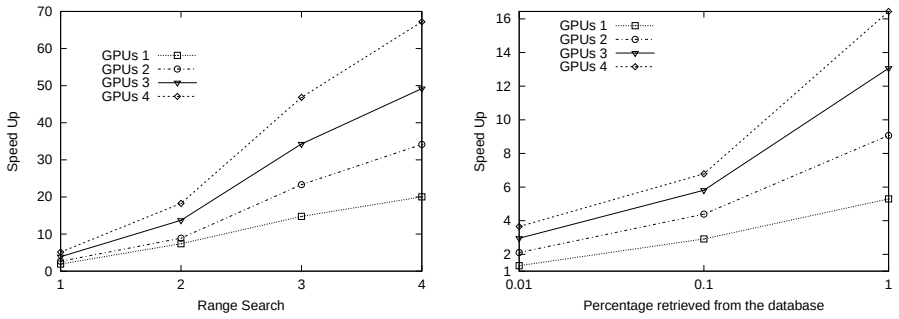


Figura 5.9: Speed-ups para las versiones multi-GPUs versus la mejor estructura secuencial, en este caso *Spaghettis*. Espacio de palabras (izquierda) e histograma de colores (derecha).

5.4. Conclusiones

Como era de esperar, las mejoras obtenidas con el uso de una GPU en el proceso de búsqueda son notables, llegando a obtener speed-ups cercanos a 48 para el espacio de palabras en español y $13,5$ para el espacio de histogramas de colores en los experimentos de búsqueda exhaustiva. Estos valores cambian al considerar sólo *GMS* con la cantidad de pivotes de mejor comportamiento, es decir, 32. En este caso las mejoras van desde los $2,9$ a 20 para el caso del espacio de palabras en español, y de $2,5$ a $5,5$ para el espacio de vectores de histogramas de colores.

Al realizar la comparación entre *GMS* en GPU y la mejor estructura en secuencial, es decir *Spaghettis*, los speed-ups obtenidos varian entre $1,9$ y 20 para el espacio de palabras en español y entre $1,3$ y $5,3$ para el espacio de vectores de histogramas de colores.

Las mejoras obtenidas con múltiples GPUs, considerando el caso de 4 GPUs versus 1 GPU y para los rangos menores y mayores, van desde $2,8$ a $3,1$ para el espacio de histogramas de colores. Para el espacio de palabras en español estos valores fluctúan entre $2,7$ y $3,4$. Como era de esperar, al observar los resultados de la figura 5.8 se puede ver que al ir aumentando la cantidad de GPUs, el tiempo va disminuyendo. Sin embargo, en ninguno de los dos casos de estudio se logró alcanzar un speed-up lineal, es decir, con 4 GPUs no se llegó a un speed-up de 4.

Como se ha mencionado en capítulos previos, en general, las estructuras tienen un mejor comportamiento cuando el rango de búsqueda es menor, debido a que pueden descartar una mayor cantidad de objetos. Al aumentar el rango, la búsqueda tiende a un comportamiento similar al de una búsqueda exhaustiva. Es decir, al aumentar la potencia computacional se obtienen mejores beneficios sobre casos en donde la estructura pierde rendimiento. En otras palabras, las estructuras poseen un buen rendimiento a bajos rangos.

Si a esto se le agrega mayor potencia computacional, como es de esperar, existe una mejora en los tiempos de búsqueda. Sin embargo, esta mejora es mucho mayor al aumentar los rangos, en donde el proceso de búsqueda se torna más "pesado".

Al realizar las comparaciones entre las versiones multi-GPU versus la mejor versión secuencial (*Spaghettis*) las mejoras van desde $3,7$ a $16,5$ para el espacio de histogramas de colores, y desde $5,1$ a $67,2$ para el espacio de palabras en español.

Capítulo 6

Búsqueda por Rango sobre Plataformas Heterogéneas

Existen variadas formas de acelerar los procesos de búsqueda mediante la paralelización de sus algoritmos. Actualmente la mayoría de las plataformas tienden a ser heterogéneas, es decir, poseen conjuntos de dispositivos de procesamiento donde la relación de procesamiento entre estos no es uno a uno. Los núcleos de procesadores, dentro de los distintos dispositivos, no poseen arquitecturas similares, por lo que algunos pueden estar diseñados para realización de operaciones complejas, y otros para operaciones sobre conjuntos masivos de datos. Un ejemplo de este tipo de plataforma es la seleccionada para este capítulo que es del tipo multicore/many-core, específicamente, multicore/multiGPU.

El capítulo anterior presentó una alternativa para ser implementada sobre uno o más dispositivos del tipo GPU, sin considerar los otros elementos de procesamiento disponibles en la plataforma. En el presente capítulo se presenta una serie de implementaciones heterogéneas completas para la estructura *GMS*, en donde tanto los cores de la CPU como los de las GPUs trabajan simultáneamente para resolver la búsqueda, aprovechando al máximo la arquitectura subyacente. Asimismo se describen los diferentes algoritmos propuestos para la plataforma heterogénea seleccionada y se discuten detalles específicos relacionados con las diferentes versiones y los dispositivos específicos sobre los que son implementados.

6.1. Plataformas Multicore y Many-core

Considerando los algoritmos presentados en los capítulos anteriores, en esta sección se describen de manera general los algoritmos y las diferencias entre las distintas implementaciones según el tipo de dispositivo sobre el que es implementado, es decir, núcleos de la CPU y/o GPUs. Se han considerado tres casos: una CPU y una GPU, varios CPU-cores (varias CPUs o varios

cores en un CPU), y multiples CPU-cores con multiples GPUs.

Algunas consideraciones generales comunes para todas las implementaciones heterogéneas:

Preprocesamiento y régimen permanente. Como en los casos previamente estudiados (capítulo 5) la selección aleatoria de pivotes y la construcción de la estructura se realiza en una etapa previa y estos costos no son considerados en el proceso de búsqueda. Se considera que un sistema está en régimen permanente en el momento en que ya está realizando definitivamente la labor para la cual fue implementada, en este caso, la búsqueda por rango. Los procesos como: la carga de archivos desde disco a memoria principal (*GMS*, base de datos y pivotes) al iniciar el programa, la solicitud de memoria para algunas variables y estructuras auxiliares (que se usarán en todo el programa), la transferencia de la estructura *GMS*, la base de datos y los pivotes de la memoria principal a la memoria global de la GPU, se realizan una única vez y no son considerados como costos de búsqueda.

Datos. Las mismas estructuras de datos son usadas en todas las implementaciones, es decir, la base de datos, las consultas, el índice métrico (*GMS*) y los pivotes. Según el tipo de implementación es posible la utilización de estructuras auxiliares para almacenar resultados intermedios. También, dependiendo del nivel de la jerarquía de memoria, algunas estructuras son utilizadas de manera diferente. En la memoria principal del sistema existe una sola copia de los datos y/o estructuras, es decir, no existen copias diferentes para distintos dispositivos.

Procesamiento. Todas las operaciones computacionales esenciales son consideradas en todas las implementaciones. Sin embargo, se consideran para algunas implementaciones dentro de los costos algunas operaciones adicionales. Ejemplo de estos son los procesos de transferencia de datos entre la memoria principal del servidor y la memoria global del dispositivo GPU, además, las transferencias realizadas entre la memoria global de la GPU y la memoria compartida de la misma. En este caso, considerando el sistema ya en régimen permanente, se tomó en cuenta el costo de la transferencia de las consultas, individualmente o en bloque dependiendo de la versión.

Algunos detalles particulares de las implementaciones son incluidas a continuación.

6.1.1. Implementación Multicore

Con el objetivo de utilizar la máxima capacidad entregada por la plataforma, en este caso un procesador dual core con 4 núcleos o cores cada uno,

se lleva a cabo una evaluación inicial del rendimiento de esta plataforma y se comparan los resultados entre sólo multicore versus GPU y versus un sistema híbrido multicore más una GPU.

Básicamente la implementación del algoritmo de búsqueda sobre una plataforma multicore implica la ejecución de una copia local en cada core del algoritmo secuencial. Para ello, se reutiliza el algoritmo presentado en la sección 3.3 (algoritmo 3), es decir, las tres partes de mayor costo computacional del algoritmo. Principalmente la idea es distribuir todas las consultas sobre los cores disponibles en el sistema, 8 en este caso. Todos los cores acceden a la estructura, la base de datos y los pivotes de manera compartida, todos ubicados en la memoria principal del servidor. Se utiliza las facilidades entregadas por OpenMP y sus directivas pragmas para distribuir un ciclo `for` sobre el conjunto de cores. En particular se usa la directiva `#pragma omp parallel for` de OpenMP y se define qué variables serán de uso compartido y qué variables de uso privado, para los cores.

Cada iteración del ciclo resuelve completamente una consulta, por lo que la distribución de las iteraciones sobre los cores es a cargo de la directiva `#pragma`.

6.1.2. Implementación GPU

Durante el desarrollo de la presente Tesis se implementaron una serie de alternativas para la estructura *GMS* sobre una y varias GPUs. La primera, presentada en la sección 5.2 fue orientada hacia una utilización razonable de los recursos disponibles en una GPU y para ello se consideraron 3 kernels que evitaban llevar todos los datos a la memoria de la GPU. Versiones posteriores fueron implementadas con 2 kernels, reuniendo los procesos 2 y 3 del algoritmo original y dejando fuera el cálculo de las distancias iniciales entre la consulta y los pivotes (todas las consultas, todos los pivotes). De hecho una implementación completamente heterogénea fue realizada utilizando la versión de 2 kernels. Ésta y otras versiones fueron publicadas por el autor en [89, 85, 86].

Para disminuir la cantidad de datos en la memoria global del dispositivo GPU, se puede dividir el proceso en varios kernels (como se menciona en el parrafo anterior) o es posible ejecutar el mismo kernel varias veces si con una sola ejecución no se procesa el conjunto total de datos. En la presente sección la implementación realizada sobre GPUs sólo utiliza un kernel para resolver una consulta.

En este caso, las consultas son resueltas una a una con una llamada a un kernel, que incluye el processo completo para resolver una consulta. En un sistema multi-GPU las consultas son enviadas a cada GPU como una unidad independiente, siendo resuelta completamente por la GPU asignada.

Previamente a la llamada del kernel, se ubican los datos necesarios en la memoria global de la GPU: *GMS*, base de datos, pivotes. También son

creadas en los dispositivos algunas estructuras necesarias para almacenar resultados intermedios.

La figura 6.1 muestra gráficamente el algoritmo general ejecutado en el kernel, dentro de una GPU. Cada paso del algoritmo básico (algoritmo 3) descrito en la sección 3.3 es indicado. Las acciones principales realizadas sobre la GPU y por el propio kernel corresponden a:

- La consulta q es transferida a la memoria global de la GPU usando el mecanismo de paso de parámetros de CUDA, e inmediatamente se transfiere la consulta a la memoria compartida del dispositivo[1].
- Para resolver la consulta q se lanzan tantos hilos como elementos tiene la base de datos. De estos hilos, en una primera etapa un conjunto menor de hilos equivalente al número de pivotes (conjunto P) realizan los cálculos entre la consulta y los pivotes p_i, las distancias $d(q, p_i)$ calculadas son llevadas a la memoria compartida de cada multiprocesador[2] (paso 1 del algoritmo). Estas distancias serán utilizadas por todos los hilos comparándolos con los elementos de la base de datos.
- En el siguiente paso, todos los hilos determinarán qué elementos de la base de datos son candidatos a la consulta q (paso 2 del algoritmo). Cada hilo comprueba una fila de la estructura *GMS* (equivalente a un elemento).
- Finalmente, cada hilo que encuentra un candidato realiza el cálculo directo entre el elemento de la base de datos y la consulta q (paso 3 del algoritmo).

Posterior a la ejecución del kernel, los resultados son transferidos a la memoria principal del servidor.

En la figura 6.1 se puede observar qué datos o conjuntos de datos están dentro de la memoria compartida y cuáles dentro de la memoria global de la GPU. También se representa los distintos accesos que hacen los hilos a datos o estructuras ubicadas en dichos bancos de memoria.

Latencia en los Accesos a Memoria Global

Cuando cientos o miles de hilos están ejecutándose simultáneamente, existe una enorme presión sobre el sistema de memoria, el cual puede incrementar de manera significativa el promedio de la latencia de memoria, y como consecuencia disminuir el rendimiento general. Sobre la GPU es posible realizar algunas acciones para disminuir esta latencia. Por ejemplo,

[1] Nótese la diferencia con la figura 6.1, en donde se lleva a memoria global un conjunto de consultas (Q). La idea de esta figura es mostrar una descripcion general para las diferentes alternativas presentadas en este capítulo.

[2] Cada multiprocesador de una GPU tiene su propia memoria compartida.

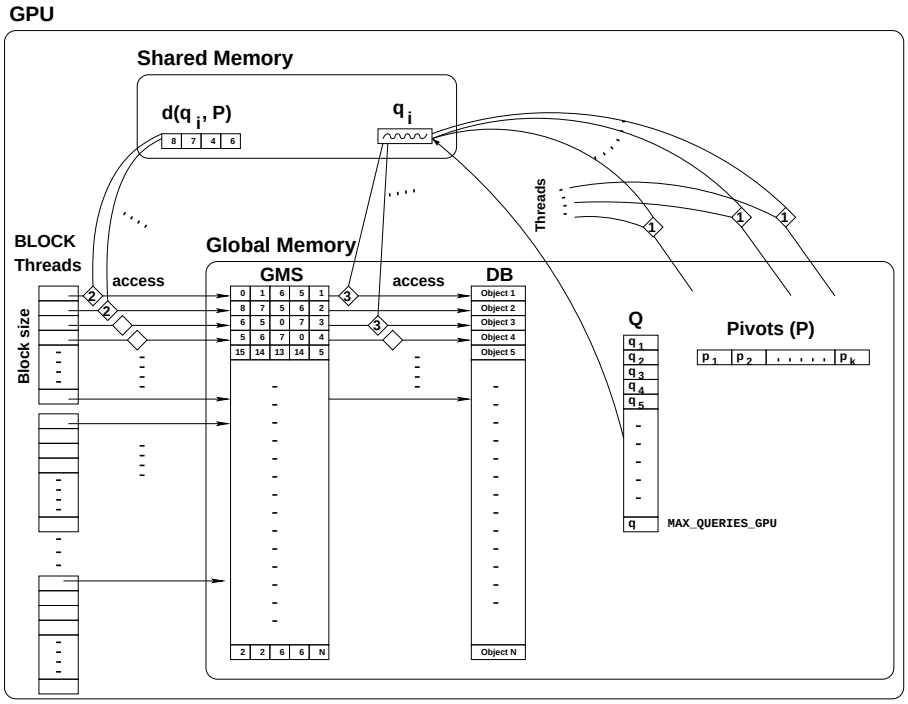


Figura 6.1: Dentro de una GPU. Pasos 1, 2 y 3 del algoritmo básico de búsqueda por rango.

explotar la baja latencia de la memoria compartida o promover los accesos coalescentes de los hilos sobre los datos en memoria global[3].

En el kernel presentado, como primera medida, se enviaron a memoria compartida los datos más utilizados por los hilos. Como segunda medida, la estructura de datos *GMS* fue ligeramente transformada para permitir accesos coalescentes desde los hilos de un *warp*. Si se considera que cada hilo necesita acceder a una fila distinta (y completa) de la estructura, entonces lo conveniente es que cada primer elemento de cada fila esté en una posición contigua de memoria. Entonces, al aplicar la operación de matriz transpuesta sobre la estructura *GMS* se puede lograr que los hilos de un mismo *warp* encuentren los datos en posiciones consecutivas de memoria. El proceso de transformar *GMS* a su transpuesta se realiza posteriormente a la carga de archivos y previo al proceso principal de búsqueda (o podría haber sido parte del preproceso). En un sistema bajo régimen permanente, este proceso no se considera dentro de los costos de búsqueda.

[3]Los accesos a memoria global por los hilos de un *warp* (realmente de medio *warp* para dispositivos con capacidad de computo de 1.x) son coalescentes, es decir, cada hilo se ejecuta sobre una posición de memoria contigua al hilo anterior. Si esto no es así, existe una pérdida de tiempo al tener que moverse el hilo a otra posición distinta de memoria.

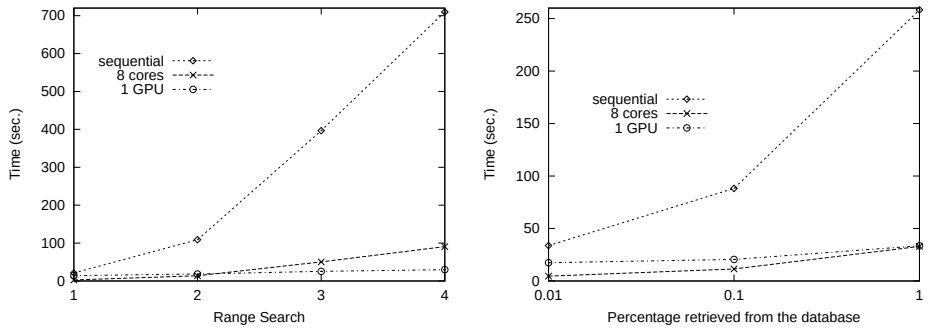


Figura 6.2: Tiempos de ejecución para las implementaciones secuencial, multicore y GPU; para los espacios de palabras (izquierda) e histogramas de colores (derecha).

6.1.3. Resultados y Discusión Inicial

Con el objetivo de obtener una visión más amplia sobre el uso de una plataforma multicore y una basada en GPUs, la figura 6.2 muestra los tiempos de ejecución absolutos para las implementaciones *GMS*: secuencial,4 GPU multicore (con 8 cores) y una GPU usando memoria compartida.

Como era de esperar, mostrado anteriormente para el caso de la GPU, ambas versiones paralelas mejoran el rendimiento considerablemente. Lo interesante en los gráficos de la figura 6.2 son las diferencias en los comportamientos según los rangos y los espacios.

Para el espacio de histogramas de colores, el comportamiento de las distintas versiones muestra algunas diferencias al ir incrementándose el rango. En este caso, la versión de 8 cores tiene mejor rendimiento que la de una GPU, sin embargo para el rango mayor, los rendimientos de las plataformas se igualan. Esto indica que la necesidad de mayor potencia computacional de la búsqueda con rangos mayores favorece el uso de la GPU. Lo mismo ocurre para el espacio de palabras en español y los rangos mayores, es decir, el rendimiento de una GPU es mejor que el del conjunto de cores.

También se puede observar, en el caso de rango 1 para el espacio de palabras en español, como el rendimiento de los 8 cores es mayor que la GPU. Sin embargo, a partir del rango 2 y en adelante el rendimiento de la GPU es mucho mayor. Las diferencias en el comportamiento entre los distintos espacios radican, entre otras razones, por el tipo de función de distancia, discreta o continua, lo que implica que los porcentajes de recuperación con los distintos radios utilizados son notoriamente diferentes (con $r = 1$ en el espacio de palabras en español se recupera mucho menos que un 0,01 %); la dimensión representacional o aparente versus la dimensión intrínseca, entre otros.

Para ejemplificar lo anterior, la cantidad de objetos que se pueden des-

cartar durante la búsqueda, por ejemplo, con 16 pivotes y $r = 1$ en el espacio de palabras en español, puede llegar a ser de hasta el 99,9 % de los elementos de la base de datos, lo que implica que el uso de la estructura es significativamente eficiente. Sin embargo, para un rango $r = 4$ la búsqueda se degrada considerablemente y sólo se pueden descartar cerca de un 39,7 % de los objetos, por lo que en este caso resulta relevante tener una mayor cantidad de unidades de procesamiento (potencia computacional).

Si se observa el caso de estudio del espacio de histogramas de colores, para 16 pivotes y el menor rango de recuperación (0, 01 %), la cantidad de objetos que se pueden descartar es alrededor de 97 % y para el rango de mayor porcentaje de recuperación (1 %) es 90,1 %. Es decir, el comportamiento de la estructura en este segundo espacio es más estable. Con esto se reafirma la idea de que uno de los motivos de la diferencia en el comportamiento es la naturaleza de la función de distancia.

Finalmente, se puede decir que en general, al ir aumentado los rangos o radios de búsqueda, y no poder descartar muchos objetos, el número de hilos disponibles para resolver una consulta resulta relevante, por lo tanto, para estos rangos el rendimiento de una GPU es mucho mayor al alcanzado con los 8 cores de la plataforma. También es interesante recalcar que, a rangos altos, junto con descartar menos datos, durante el recorrido de las filas en la estructura (la comparación de distancias), la tendencia es a utilizar la mayoría o todos los pivotes, lo que aumenta aún más el tiempo de procesamiento.

6.2. Implementaciones Heterogéneas para Búsqueda por Similitud

En nuestros días, dispositivos del tipo GPU/MultiGPU son ampliamente usados para obtener tiempos de ejecución razonables a costos económicos muy bajos. Las GPU/MultiGPU son administradas por una CPU/core/multicore. Estos sistemas son llamados heterogéneos debido a que diferentes arquitecturas son utilizadas bajo una sola plataforma. Sin embargo, en la mayoría de las aplicaciones (o muchas de ellas) las unidades de procesamiento CPU/core/multicore están desabilitadas mientras la aplicación es ejecutada sobre los dispositivos GPU/MultiGPU. Como concecuencia de esto, los mayores rangos (teóricos) de rendimiento de la arquitectura subyacente no pueden ser alcanzados, y por supuesto, el término "plataforma heterogénea" viene a ser más un término convencional que un término real.

En la presente sección se describe de manera general algunas consideraciones para la implementación en un sistema heterogéneo. En las secciones 6.3 y 6.4 se desarrollarán en detalle dos implementaciones heterogéneas considerando dos escenarios diferentes. La primera continúa la línea anterior

y considera que las consultas serán procesadas en forma individual por un único kernel. Bajo este escenario, más objetos de la base de datos pueden ser ubicados en la memoria global del dispositivo. Evidentemente, la consecuencia de esto es el costo adicional en tiempo que implica la transferencia de los datos por cada llamada a un kernel. El segundo escenario corresponde a una implementación orientada a bloques de consultas, en donde se realiza un sola llamada a un kernel para un conjunto de consultas (bloque de consultas). Es decir, un kernel puede resolver un número específico de consultas, lo que redundará en un menor número de transferencias de datos. Sin embargo, en este caso, se reduce la capacidad de almacenamiento de objetos en la memoria global de la GPU, y por otro lado, es necesario mantener estructuras auxiliares de mayor tamaño para almacenar resultados intermedios, reduciendo aún más la capacidad de almacenamiento. Esto último es debido a que en sistemas reales se debe realizar alguna operación sobre los resultados, por lo que las soluciones a cada consulta deben ser devueltos al sistema principal.

En general, una consulta es completamente resuelta por el dispositivo al que es asignada, sea un CPU-core o una GPU. El esquema utilizado distribuye las consultas sobre todos los cores, incluyendo los cores que administran las GPUs.

Los cores comparten las estructuras de datos desde la memoria principal del servidor (*host*), sin embargo, esta información es replicada sobre cada dispositivo GPU. Una región paralela en OpenMP se inicializa utilizando la directiva `#pragma`, en este caso, para distribuir sobre los cores las diferentes iteraciones se utilizó la directiva `#pragma omp parallel for`. Cuatro esquemas de planificación pueden ser considerados para esta directiva:

1. `static`: En este caso, previo a la ejecución del bucle, a todos los hilos se les asigna una cantidad de iteraciones contiguas. Si N es el número de iteraciones y T el número de hilos, entonces por defecto, cada hilo atiende a N/T iteraciones. Es posible modificar dicho valor modificando el parámetro *chunk*. La asignación de los diferentes trozos es *round robin*.

2. `dynamic`: Esta directiva es ideal cuando las iteraciones no suponen la misma carga de trabajo. OpenMP decide qué iteraciones son asignadas a los hilos. La idea es que al hilo que termine su bloque de iteraciones se le asigne el siguiente disponible. El parámetro *chunk* es el que define el número de iteraciones contiguas asignadas a cada hilo, por defecto es 1.

3. `guided`: Es una planificación dinámica con bloques de tamaño que van decreciendo exponencialmente hasta llegar al valor inicial especificado en el parámetro *chunk*.

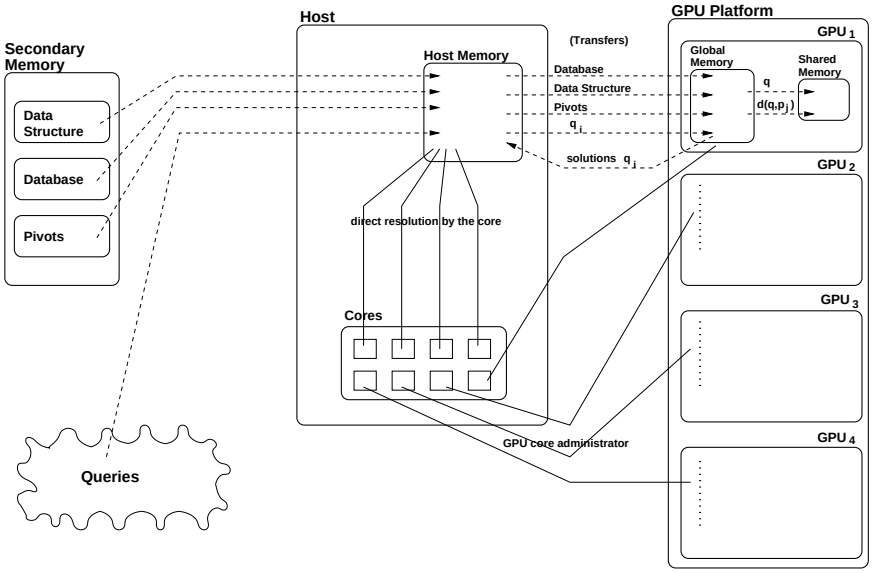


Figura 6.3: Diagrama general del sistema heterogéneo.

4. `runtime`: La planificación queda supeditada a una variable de entorno denominada `OMP_SCHEDULE`, definida en un paso previo a la ejecución.

En general, cuando las iteraciones tienden a tener diferentes tiempos de ejecución, la planificación del tipo `dynamic` resulta la más conveniente. Este es, efectivamente, el utilizado en esta versión heterogénea de *GMS*.

La figura 6.3 muestra un esquema general de las implementaciones, contextualizando algunos elementos esenciales como: los cores que administran GPUs, la jerarquía de memoria y las transferencias de datos, entre otros. En el diagrama se muestran 8 cores y 4 GPUs. Se entiende que cada elemento GPU, dentro del esquema de la figura 6.3, es del tipo mostrado en la figura 6.1.

6.3. Versión 1: Una Consulta - Un Kernel

En esta sección se introduce una implementación considerando los comentarios de la sección previa, en relación al esquema de planificación elegido. En esta implementación cada core o núcleo de la CPU resolverá una consulta a la vez. Igualmente, la GPU resolverá mediante un kernel una sola consulta (bloque de tamaño 1).

Este escenario es adecuado para disminuir la necesidad de memoria disponible en el dispositivo gráfico, es decir, al solo tener una consulta en la GPU es posible, entre otras cosas, aumentar el número de objetos que la base de datos y estructura pueden contener.

Por otro lado, esta opción es la continuación natural a los algoritmos presentados en las secciones 5.2 y 6.1.2, y permite obtener una primera aproximación respecto del comportamiento de la estructura *GMS* sobre un esquema heterogéneo aplicando distintos valores al parámetro *chunk* en la planificación. En esta implementación existe una mayor cantidad de transferencias de datos entre la memoria principal del servidor y la memoria global de la GPU.

6.3.1. Tiempos de Ejecución

Las figuras 6.4 y 6.5 presentan los primeros resultados para la implementación heterogénea. En este caso los gráficos muestran los resultados de una implementación multicore versus una implementación heterogénea usando una planificación del tipo `dynamic`. En los gráficos, el eje X representa el porcentaje de iteraciones asignadas a cada core, equivalente al número de consultas que serán resueltas una a una. Los porcentajes van desde $2,5\,\%$ a $12,5\,\%$, además de un conjunto de valores de bajos porcentajes: $0,1\,\%$, $1\,\%$, $2\,\%$, añadiéndose a estos el tiempo para una iteración ($1q$). Para una mejor visualización de los resultados, cada rango de búsqueda ha sido separado en gráficos individuales. Además, se utilizan leyendas del tipo YZ, en donde la Y representa al número de cores utilizados y Z el número de GPUs, es decir, 80 significa 8 cores y 0 GPUs, mientras 81, 8 cores y 1 GPU. En este último caso uno de los 8 cores administra la GPU.

En las figuras 6.4 y 6.5 se puede observar que en todos los casos el sistema híbrido (heterogéneo) tiene mejor rendimiento, en términos de tiempo de ejecución. Una excepción es el caso de $12,5\,\%$ en donde este valor es equivalente a un esquema estático, es decir, 1/8 del total de iteraciones fue asignado a cada dispositivo, por lo que si existe un dispositivo más rápido que otro y que está obligado a resolver el mismo número de consultas, el más rápido siempre terminará esperando al más lento. Es decir, no es adecuado dividir a partes iguales en un sistema heterogéneo.

Algunas conclusiones se pueden obtener de los gráficos de las figuras 6.4 y 6.5. Primero, que en definitiva la mejor opción se da cuando la cantidad de iteraciones asignadas a cada core es baja, de hecho la de mejor resultado es cuando el valor es una iteración ($1q$) para cada core. Esto es básicamente debido a que cada consulta es resuelta en forma independiente, por ello, por ejemplo, la GPU va tomando consultas a medida que va resolviéndolas, es decir, no tiene que esperar en ningún momento a que termine algun otro dispositivo de procesamiento. Por lo mismo, a medida que el tamaño del bloque de iteraciones asignadas se incrementa, los costos en la búsqueda aumentan. Incluso, existen valores que desajustan en mayor medida el sistema, es por ello que se observan puntos máximos en las curvas, incluso para la versión multicore, donde todos los cores son iguales. Si se observa la figura 6.4, curva multicore (80) y el porcentaje 10, se puede explicar este comportamiento

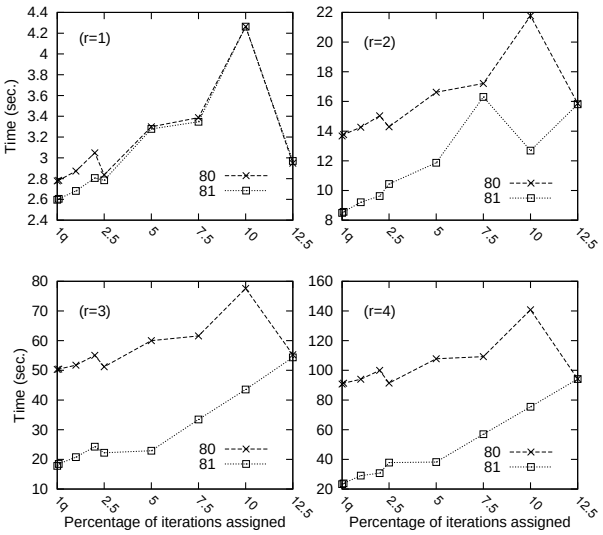


Figura 6.4: Tiempos de ejecución para distintos valores del parámetro *chunk* usando planificación dinámica en el espacio de palabras en español.

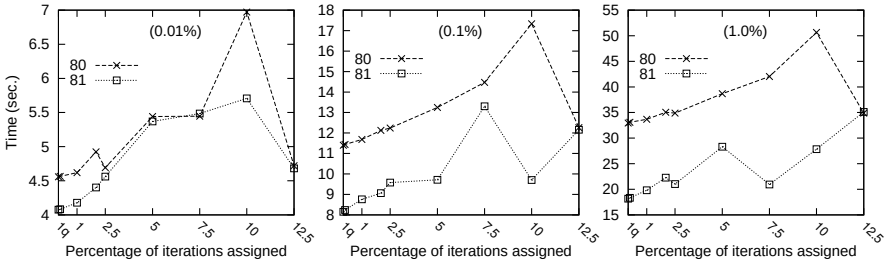
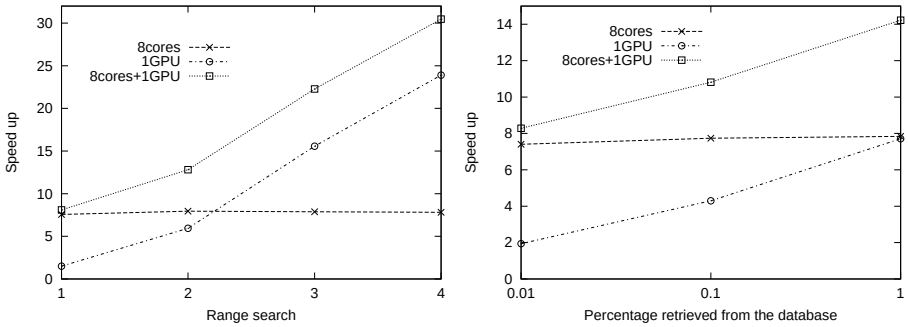


Figura 6.5: Tiempos de ejecución para distintos valores del parámetro *chunk* usando planificación dinámica en el espacio de vectores.

dado que las iteraciones (consultas) fueron divididas en 10 partes, al tener 8 cores o núcleos, se consumen 8 partes en una primera vuelta, y para la segunda sólo quedan dos cores encargados de las restantes partes, habiendo 6 cores que quedan disponibles sin realizar actividad. No ocurre igual para la curva de la implementación heterogénea (81) ya que muy probablemente la GPU alcanzó a consumir esas dos últimas partes, haciendo el equivalente a la ejecución de los 10 bloques en sólo una vuelta. Es importante recordar que las consultas no se resuelven en el mismo tiempo, por lo que hay variaciones en los tiempos de búsqueda en dispositivos de iguales características.

La ventaja de usar todos los recursos disponibles (caso 81) se puede ver en la figura 6.6, en donde la versión heterogénea muestra un mayor rendimiento.

En particular, se puede observar, en los gráficos de speed-ups, que el



(a) Spanish dictionary. (b) Color histograms.

Figura 6.6: Speed-ups para un sistema heterogéneo, en el contexto de un sistema multicore y una plataforma con una GPU.

rendimiento del sistema heterogéneo es acorde al rendimiento de una GPU. Además, existe una diferencia constante, a favor, provocada por los beneficios de utilizar adicionalmente los 7 cores restantes.

Las figuras 6.7 y 6.8 muestran los resultados para un sistema heterogéneo utilizando de 1 a 4 GPUs. Como era de esperar, el comportamiento de 84 es el mejor en "casi" todos los casos, especialmente para bajos números de consultas.

Al reemplazar CPU-cores por GPUs, como era de esperar, los tiempos bajan, aunque el comportamiento es similar, es decir, a medida que se aumenta el tamaño del bloque el rendimiento decrece. Al usar más GPUs es menor la variabilidad del sistema, sobre todo en los rangos mayores. Es posible que en algunos casos, usar 4 GPUs en el sistema heterogéneo no sea mejor que usar menos, todo esto depende exclusivamente del porcentaje en que fueron divididas las iteraciones, en donde siempre es posible que el último bloque o conjunto de bloques sea asignado al dispositivo más lento, debido a que los más rápidos aún están procesando, provocando con ello que el rendimiento global esté subordinado a la ejecución de estos últimos bloques y de los dispositivos a los que fueron asignados.

Como se ha mencionado, de los gráficos se ha podido establecer que, en esta versión del esquema heterogéneo, lo más adecuado es que la planificación dinámica tenga tamaño de *chunk* igual a 1. Esto se explica también teniendo en cuenta que, si bién es cierto, el tamaño del bloque (*chunk*) utilizado por la planificación dinámica a través de la directiva `#pragma#` de OpenMP puede ser superior a 1, en realidad los dispositivos sólo van resolviendo una consulta a la vez.

Un aspecto a considerar, específicamente para el caso de las GPUs, es que el tiempo de transferencia utilizado por estos dispositivos no dependerá del parámetro *chunk*. En la siguiente sección se analizará en detalle la distribución de los tiempos de ejecución, el número de consultas asignadas

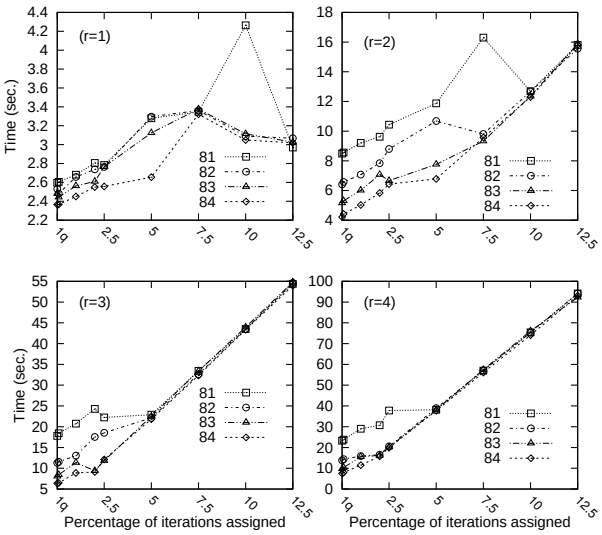


Figura 6.7: Añadiendo GPUs a un esquema heterogéneo, cambiando CPU-cores por GPUs. Caso de estudio diccionario en español.

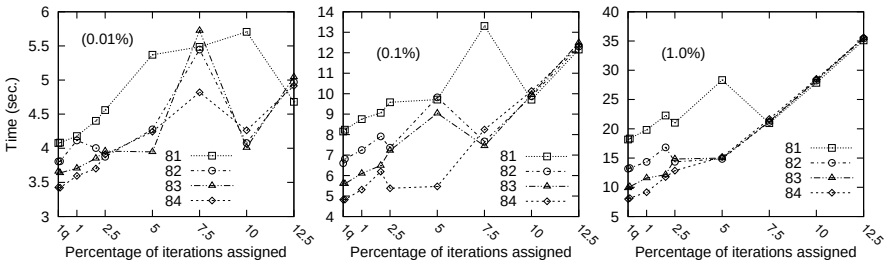


Figura 6.8: Añadiendo GPUs a un esquema heterogéneo, cambiando CPU-cores por GPUs. Caso de estudio histogramas de colores.

a cada dispositivo y la relación entre la cantidad de consultas asignadas y el tiempo de ejecución.

6.3.2. Distribución de los Tiempos de Ejecución y de las Consultas

Primeramente, se mostrará el comportamiento del sistema heterogéneo con sólo una GPU (81), de esta manera se tendrá una visión más amplia al tener un punto de referencia para los datos obtenidos posteriormente con el sistema completo, de 8 cores y 4 GPUs. En estos gráficos se reemplazaron los porcentajes por sus valores reales equivalentes, es decir, en el caso del espacio de palabras corresponde a: 1 ($1q$), 8 ($0,1\,\%$), 86 ($1\,\%$), 172 ($2\,\%$), 215 ($2,5\,\%$), 430 ($5\,\%$), 645 ($7,5\,\%$), 860 ($10\,\%$) y 1.075 ($12,5\,\%$); mientras

para el caso del espacio de vectores son: 1, 11, 112, 225, 281, 563, 845, 1.126 y 1.408.

Las figuras 6.9 y 6.10 muestran detalles de la implementación para ambos casos de estudio. En la columna de la izquierda se muestra la información respecto de las cantidades de consultas resueltas por cada tipo de dispositivo. La columna de la derecha muestra, por un lado, los tiempos promedios para los cores (*CPU time process*), y por otro lado, la distribución de los tiempos de ejecución realizado por un core que administra una GPU, es decir, relacionados con la GPU, por ello se hablará de tiempo de GPU (*GPU time process*). Esencialmente interesa entender en qué se invierte el tiempo en el caso de la GPU. En los gráficos para la GPU el tiempo es dividido en tres partes principales:

Proceso de transferencia: Representa el tiempo gastado en todas las transferencias entre la memoria principal del sistema y la memoria global de la GPU. Incluye la transferencia de consultas hacia la GPU y la recuperación de los resultados por cada consulta desde la GPU a la memoria principal.

Proceso de kernel: Representa el proceso realizado dentro de la GPU para encontrar las soluciones a cada consulta.

Otros procesos: Representa otras operaciones adicionales previas o posteriores a las transferencias y a la ejecución del kernel, es decir, procesos llevados a cabo por el core que administra la GPU. Como ejemplo se puede mencionar: gestión de la GPU, solicitudes de memoria al sistema principal o a la GPU (`malloc` o `cudaMalloc`), identificación de las soluciones (en un sistema real, se realiza un proceso con cada solución, en este caso, sólo se cuenta para determinar que el número esperado de soluciones es correcta), etc.

En ambas columnas, para el caso de los cores, los valores mostrados corresponden al promedio de los 7 núcleos restantes.

A medida que aumenta la carga computacional, es decir, aumentan los radios o rangos de búsqueda, se puede observar que el sistema heterogéneo obtiene mayores beneficios del uso de la GPU. En estos casos es necesario evaluar una mayor cantidad de objetos, entonces, la GPU al poseer una gran cantidad de hilos es capaz de completar más búsquedas que los cores. Por ejemplo, para el espacio de palabras en español la cantidad de consultas procesadas por una GPU es $1,6$ veces mayor que un core para rango $r = 1$, sin embargo, para $r = 4$ es $24,1$ veces superior a un core (al promedio por core). En términos absolutos, para el rango superior, mientras una GPU resuelve 6.672 consultas, 7 cores resuelven en conjunto 1.934 consultas. Para el espacio de vectores el comportamiento es más estable y los valores van desde $2,1$ a $7,8$ veces más las consultas que resuelve una GPU versus un core,

para los radios menores y mayores respectivamente. En términos absolutos, mientras una GPU resuelve 5.931 consultas, 7 cores resuelven en conjunto 5.337 consultas (762, 43 consultas en promedio, por core).

Con respecto a los tiempos de ejecución de la GPU, se puede observar que los tiempos de transferencia y otros procesos, son relativamente constantes por cada consulta debido a que son independientes del rango de búsqueda y del tamaño del bloque asignado a cada procesador. Se debe recordar que las transferencias son realizadas por cada consulta, aunque el parámetro *chunk* sea mayor a 1.

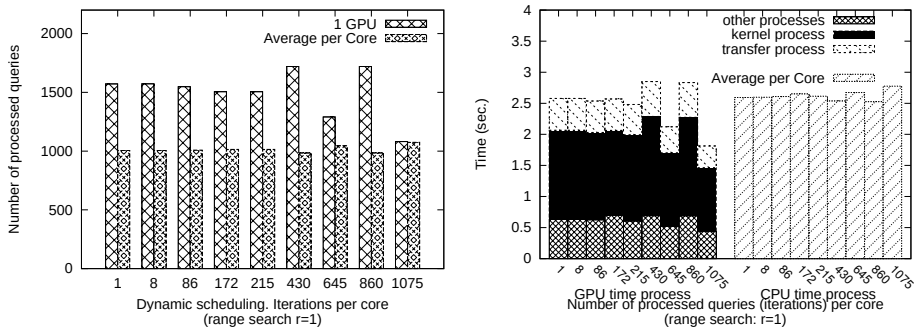
Como es obvio, si se analizan los gráficos de número de consultas y tiempos asociados, por ejemplo el caso de la figura 6.9(a), se reafirma la idea de que al aumentar el tamaño del bloque, es decir, obligar a los dispositivos a resolver las mismas cantidades de consultas, los tiempos individuales de los más lentos aumentan. Al aumentar el rango esto es más notorio; en el caso 6.9(d) se puede observar que mientras para bloques pequeños los tiempos de la GPU son similares, para bloques más grandes disminuyen ya que no alcanza a procesar la cantidad equivalente de consultas (de los rangos menores), así también se observa que los cores van continuamente aumentando el tiempo de procesamiento.

En este punto también es importante recalcar que OpenMP envía las iteraciones a cada core y dentro de cada iteración es el core que administra la GPU el que envía una a una las consultas y ejecuta el kernel. Se distribuyen sobre los cores las iteraciones, que en este caso (versión 1) equivalen a la resolución de una consulta.

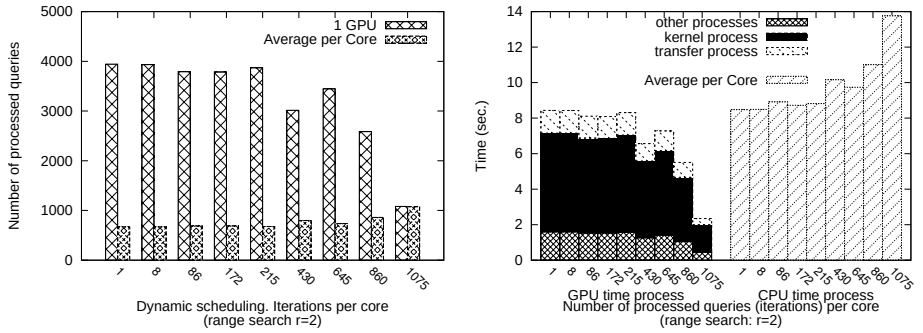
Al considerar el sistema completo, es decir, con 8 cores y 4 GPUs, el comportamiento es similar al comentado previamente, lo que puede ser observado en las figuras 6.11 y 6.12. En los gráficos de estas figuras los valores corresponden a los promedios de cores y GPUs. En estas figuras queda de manifiesto la potencia del sistema heterogéneo al aumentar el número de GPUs. Por ejemplo, para $r = 4$ en el espacio de palabras en español, las 4 GPUs resuelven el 96 % de las consultas y sólo dejan un pequeño remanente a los cores. Para el espacio de histogramas de colores, este porcentaje corresponde a un 88, 6 %. Todos los demás aspectos comentados para el sistema heterogéneo con sólo una GPU son válidos para el sistema con 4 GPUs.

6.3.3. Escalando el Número de Consultas

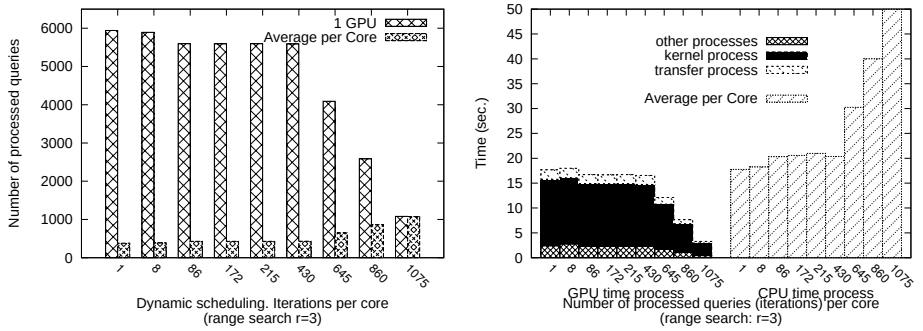
Para estudiar la robustez del sistema, es decir, si se comporta como se espera al incorporar mayor cantidad de datos a procesar, se ha escalado el número de consultas para ambos espacios. Para el espacio de palabras en español se utilizará un conjunto adicional de consultas que van desde las 10 mil a las 40 mil. Para el espacio de histogramas de colores, se amplía a 900 mil consultas. La información detallada de los nuevos conjuntos de datos se encuentra en la sección 4.1.



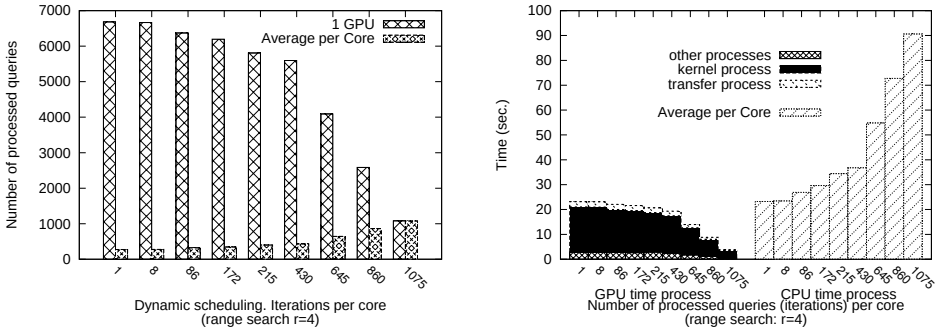
(a) $r = 1$



(b) $r = 2$

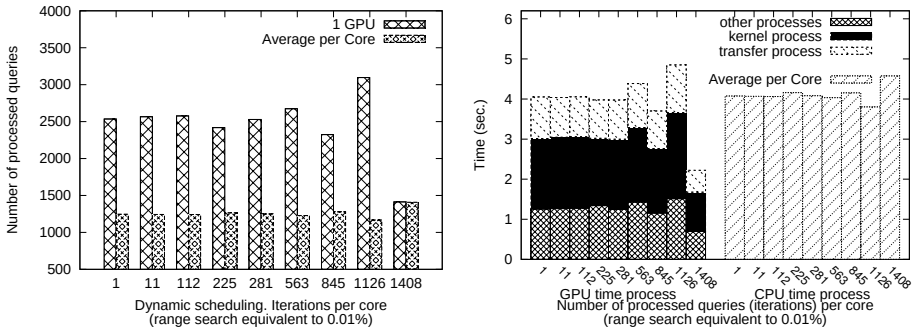


(c) $r = 3$

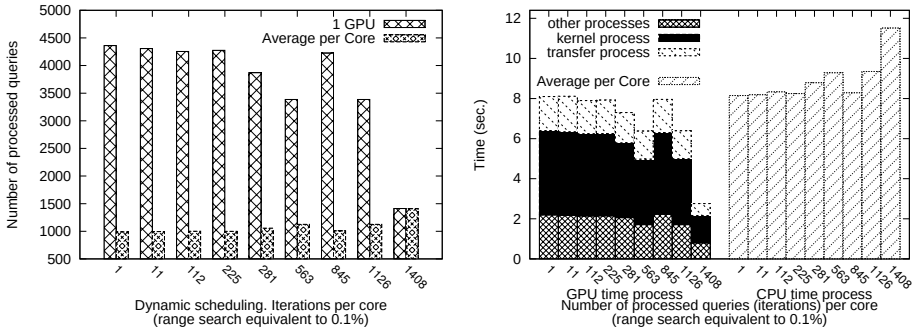


(d) $r = 4$

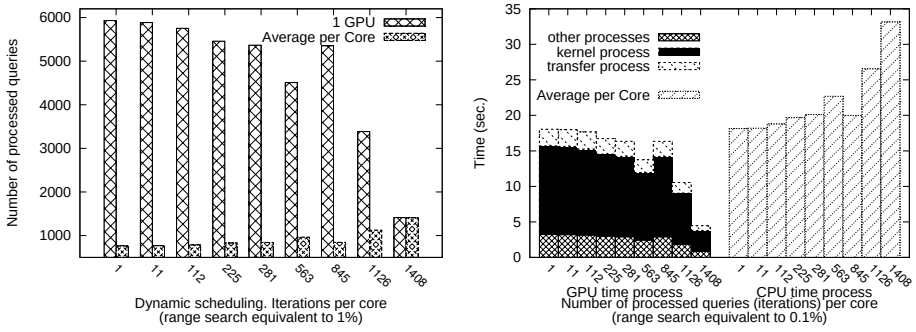
Figura 6.9: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 1 GPU. Caso diccionario en español.



(a) $0,01\,\%$

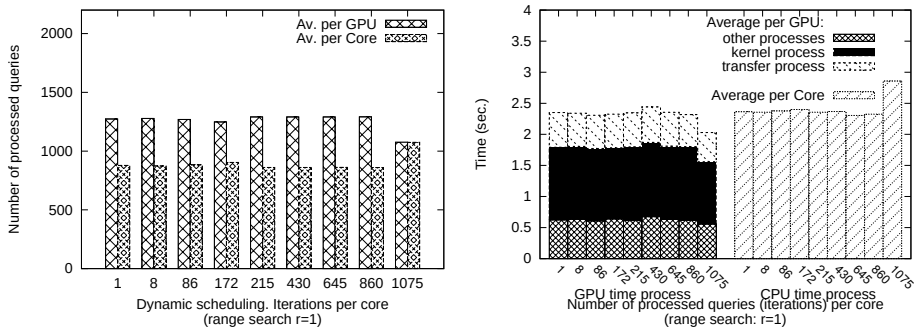


(b) $0,1\,\%$

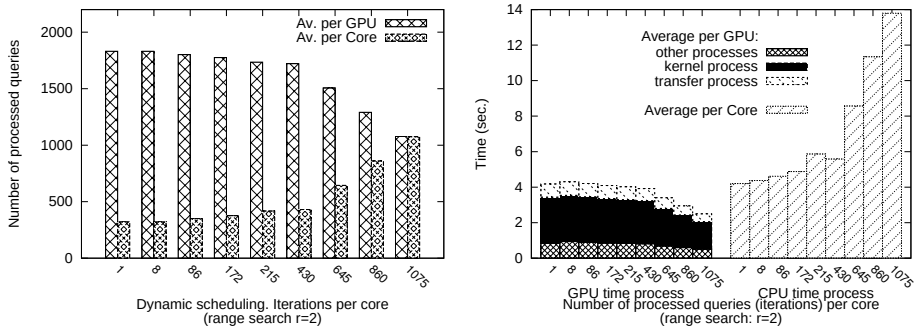


(c) $1,0\,\%$

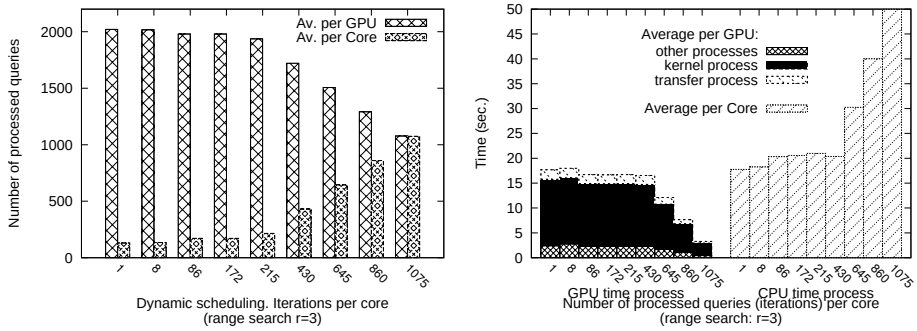
Figura 6.10: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 1 GPU. Caso histogramas de colores.



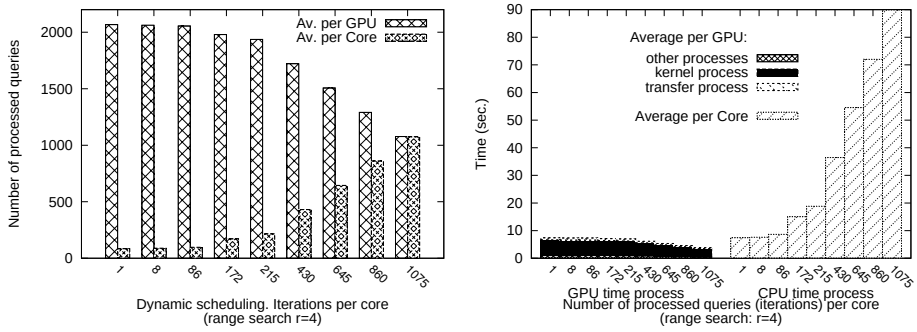
(a) $r = 1$



(b) $r = 2$

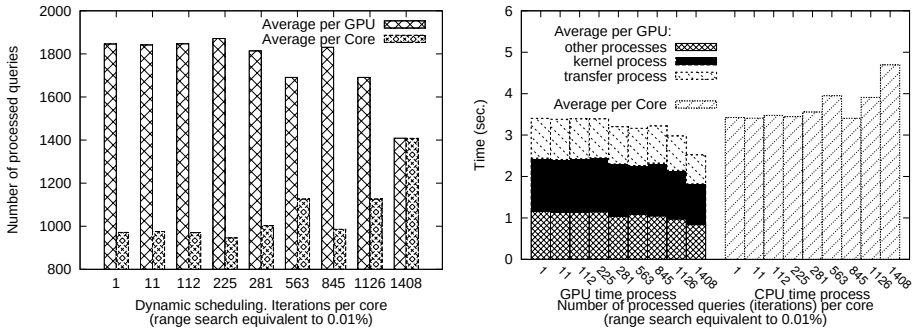


(c) $r = 3$

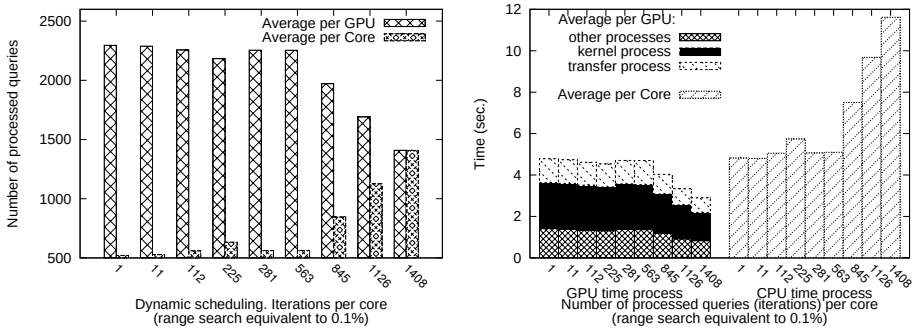


(d) $r = 4$

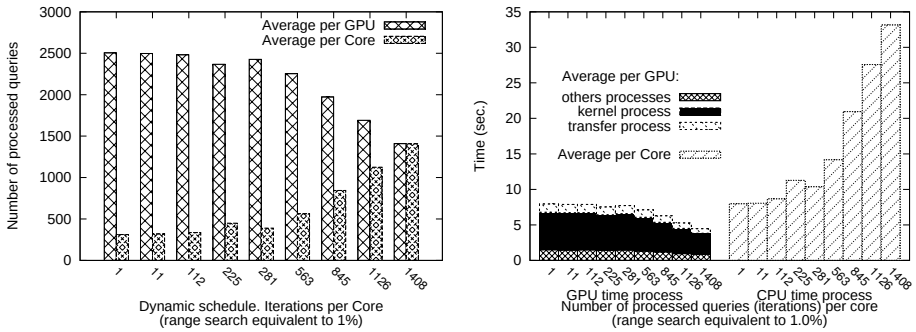
Figura 6.11: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso diccionario en español.



(a) $0,01\,\%$



(b) $0,1\,\%$



(c) $1,0\,\%$

Figura 6.12: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso histogramas de colores.

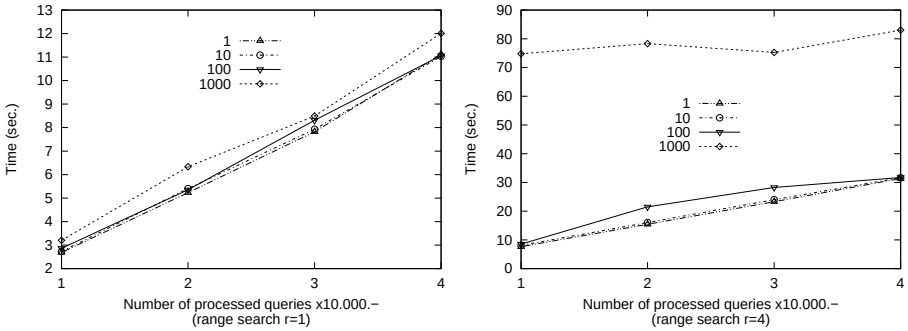


Figura 6.13: Escalabilidad del sistema. Tiempo de ejecución versus tamaño del bloque escalando el número de consultas desde 10 mil a 40 mil para los rangos 1 y 4 en un sistema con 8 cores y 4 GPUs (espacio de palabras en español con log de 40 mil consultas).

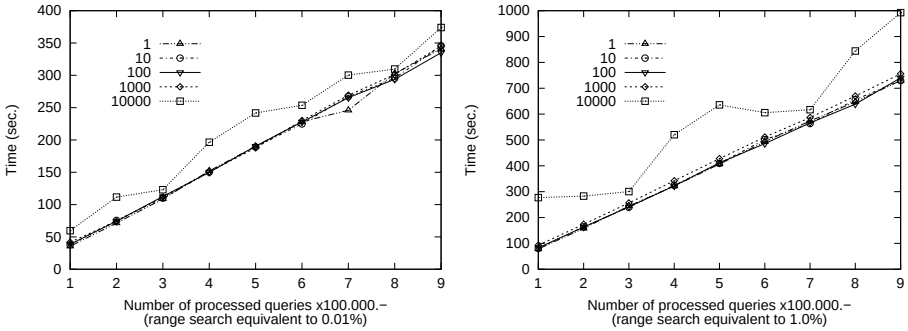


Figura 6.14: Escalabilidad del sistema. Tiempo de ejecución versus tamaño del bloque escalando el número de consultas desde 100 mil a 900 mil para los rangos que recuperan $0,01\,\%$ y $1\,\%$ en un sistema con 8 cores y 4 GPUs (espacio de histogramas de colores escalado).

A priori, si la base de datos se mantiene constante, los tiempos de ejecución se incrementan linealmente al aumentar el número de consultas también linealmente. Es decir, el sistema global es escalable.

La figura 6.13 muestra los resultados sobre el espacio de palabras en español al ir incrementando el número de consultas de 10 mil a 40 mil considerando el sistema completo (8 cores y 4 GPUs) y el menor y mayor rango de búsqueda (1 y 4 respectivamente). La figura 6.14 muestra los resultados para el espacio de vectores, donde el incremento del número de consultas va desde los 100 mil a 900 mil considerando el sistema completo y los radios equivalentes al menor y mayor porcentaje de recuperación ($0,01\,\%$ y $1\,\%$ respectivamente). Las leyendas 1, 10, 100 y 1.000 en los gráficos representan el valor del parámetro *chunk* (tamaño del bloque), para el espacio de vectores se agregó 10.000.

Como se ha mencionado, se puede observar que el costo en tiempo es lineal al aumentar el número de consultas, mostrando la escalabilidad del sistema. En particular, para el espacio de vectores cuando el tamaño del bloque es 1 se aprecia una ligera mejora sobre los otros valores, sin embargo, también es posible observar que para valores 10, 100 e incluso 1.000 los resultados son buenos. De todas maneras, era de esperar estos resultados, es decir, que el rendimiento para $1q$ sea mejor que las otras alternativas, y que además, al aumentar el número de consultas total a resolver, es posible aumentar el número de iteraciones en un bloque sin bajar el rendimiento. Igualmente ocurre para el espacio de palabras en español, en este caso, la cantidad de consultas es muy inferior, pero se ve una mejora para los tamaños de bloque de 10 y 100, aunque notoriamente sigue siendo mejor cuando se procesa una consulta cada vez.

Un comportamiento que llama la atención es lo ocurrido en el espacio de palabras en español con tamaño del *chunk* igual a 1.000. Como se ha comentado antes, 1.000 consultas sobre 10 mil es un porcentaje considerable por lo que el comportamiento esperado es malo. Se entiende que para el caso con rango 4, al necesitar mayor potencia de cálculo, los costos son elevados, por lo que aunque las GPUs procesen varios bloques, un bloque de este tipo sobre un core tiene un alto costo. En cambio, al reducir a rango 1, y considerando los resultados de las secciones previas, vemos que un core es eficiente a rangos bajos usando esta estructura métrica. Retomando el caso $r = 4$, los valores reales en estos experimentos fueron que las GPUs resolvieron 6 mil de un total de 10 mil consultas, y sobre el conjunto de 40 mil, las GPUs resolvieron 36 mil, es decir, en ambos casos, los cores solamente alcanzaron a resolver un bloque de consultas, por lo que se deduce que las GPUs quedaron esperando a que estos dispositivos terminaran, afectando considerablemente el rendimiento global del sistema.

6.3.4. Discusión

A lo largo de esta sección se ha presentado un sistema completamente heterogéneo que utiliza todas las unidades de procesamiento provistas por la plataforma, utilizando tanto CPU-cores como las GPUs disponibles. La implementación heterogénea mejora los tiempos de ejecución de la implementación que utiliza sólo multicore o GPU. En un primer test utilizando una configuración básica de 8 cores y 1 GPU y el rango mayor ($r = 4$), el sistema es cerca de 30 veces más rápido que las implementaciones secuenciales, $3,9$ veces más rápido que la implementación multicore y hasta 1,3 veces que la implementación en GPU para el espacio de palabras en español. Respecto del espacio de histogramas de colores, las mejoras van de un $1,8$ versus multicore; $1,8$ versus un sistema con una GPU; $10,4$ al compararla con la versión secuencial de *GMS* y de 10 contra la mejor versión secuencial, *Spaghettis*.

Un detalle de las mejoras conseguidas se reflejan en la tabla 6.1. Cada

Tabla 6.1: Mejoras de la versión heterogénea versus las versiones secuenciales, multicore y de una GPU, para los rangos mayores y menores (versión heterogénea de 8 cores y 1 GPU).

Espacios / Implementaciones	Diccionario Español		Histogramas de Colores	
	1	4	0.1	1.0
sequencial (*Spaghettis* 032)	5,79	29,84	4,9	9,99
sequencial (*GMS* 032)	8,1	30,47	9,1	10,44
multicore (8 cores)	1,07	3,9	1,12	1,81
1 GPU	5,38	1,27	4,27	1,84

celda representa la mejora o speed-up obtenido de la versión heterogénea de 8 cores y 1 GPU, según el espacio, rango (menor y mayor) versus la plataforma donde fue implementado el algoritmo.

Con el sistema utilizando todos los dispositivos, es decir con 8 cores y 4 GPUs, las mejoras sobre el primer test (81) van desde, $1,1$ a $3,12$ para el menor y mayor rango respectivamente, en el espacio de palabras en español. Para el histograma de colores, desde $1,2$ a $2,27$ para el menor y mayor rango de recuperación. Es decir, versus la mejor versión secuencial se pueden lograr speed-ups que van desde los $6,36$ a $93,01$ para el espacio de palabras en español y desde los $5,82$ hasta los $22,71$ para el espacio de histogramas de colores.

Respecto de la distribución de consultas, sobre problemas con mayor carga computacional, el uso de GPUs beneficia enormemente el rendimiento global, sobre todo cuando los bloques son pequeños. Además los CPU-cores igualmente pueden apoyar el procesamiento de datos sobre pequeños conjuntos sin generar *cuellos de botella*.

En cuanto a la distribución de los costos en tiempos de ejecución, la transferencia de datos implica un porcentaje considerable en el uso de la GPU, un poco menos de un tercio en el rango menor, aunque mucho menos en los rangos mayores. Al aumentar el rango de búsqueda aumenta también el uso de la GPU en el procesamiento y disminuyen los otros costos (en porcentaje). Por supuesto que esto lleva a pensar en mecanismos que permitan reducir los costos en transferencias (los "otros procesos" no pueden ser reducidos ya que representan operaciones llevadas a cabo por el núcleo administrador de la GPU). La clave a responder es por qué el mejor comportamiento se da cuando el bloque es de sólo una consulta. Como se ha comentado previamente, el aumento del tamaño del bloque no libera a la GPU de las transferencias individuales de las consultas (para esta versión), por lo que la distribución de las consultas afecta en términos de cantidad de consultas que está obligado un dispositivo a procesar, donde el peor caso es si esa cantidad es demasiado grande para un dispositvo muy lento. También se puede mencionar que internamente OpenMP interpreta el proceso de un conjunto de consultas como si fueran una a una. OpenMP envía una consul-

ta a la vez a la GPU, pagándose los costos de dicha transferencia individual. Esta operación puede ser mejorada si internamente se obliga a OpenMP a enviar el conjunto completo de consultas en una sola transferencia a la GPU.

Finalmente, como se esperaba el sistema es escalable en términos de número de consultas.

6.4. Versión 2: *MQ* Consultas - Un Kernel

De acuerdo a los comentarios finales de la última sección (denominada versión 1), en la presente sección se introduce una implementación "orientada a bloques". Es decir, el objetivo será reducir el número de transferencias y disminuir el número de llamadas al kernel y por lo tanto disminuir el tiempo de procesamiento de consultas por cuenta de la GPU. En esta nueva versión, denominada versión 2, cada kernel resolverá realmente el bloque de consultas asignado.

Para llevar a cabo esta implementación, se deben realizar algunos cambios sobre el código.

Primeramente, se modifica el código encargado de las llamadas al kernel porque, como se mencionó de la sección 6.3.2, es necesario "engañar" a OpenMP indicándole que enviará una iteración a un core, pero esa iteración corresponderá a un bloque de consultas, y de esta manera enviar el bloque completo para que sea resuelto por un kernel.

Para lo anterior, el ciclo variará no de consulta en consulta, sino, de bloque en bloque. Supongamos que se tienen las siguientes variables: `NUM_QUERIES`, número de consultas; `MQ`, máximo número de consultas por bloque (equivalente al parámetro *chunk* de la versión anterior); y `NUM_BLOCKS`, el número de total bloques de consultas. Entontes, el ciclo principal iterará en función del número de bloques: $\texttt{NUM_BLOCKS} = \frac{\texttt{NUM_QUERIES}}{\texttt{MQ}}$.

Nótese que antes de la llamada al kernel, debe ser enviado el bloque completo de consultas a la memoria global de la GPU. En esta versión se reducen `MQ` transferencias a una sola, aunque de tamaño `MQ` veces mayor. Igualmente la cantidad de llamadas al kernel es reducida en la misma proporción, y cada kernel resuelve `MQ` consultas por llamada.

En cuanto a los cambios realizados en relación al kernel propuesto en la sección 6.1.2, se tienen los siguientes:

1. Se ubican en memoria global de la GPU las estructuras necesarias para almacenar las soluciones de todas las consultas de un bloque de `MQ` consultas, siendo `MQ` el tamaño de un bloque.

2. Después de esto, durante el proceso de solución, `MQ` consultas son transferidas a la memoria global de la GPU por parte del core administrador de la GPU.

3. Realmente el kernel es el mismo presentado en la sección 6.1.2, pero en este caso el kernel es el que itera `MQ` veces resolviendo completamente el bloque de consultas. En cada iteración, dentro del kernel, se va ubicando en la memoria compartida del dispositivo la consulta que está resolviéndose.

4. Finalmente, las soluciones para las `MQ` consultas de cada bloque son transferidas desde la GPU a la memoria principal del sistema.

En esta versión, se evitan tranferencias y llamadas a kernels, sin embargo, los cores que no poseen GPUs igualmente resuelven el bloque de consultas asignadas. Por lo tanto, al igual que la versión 1, bloques muy grandes provocarán cuellos de botella.

Un caso especial resulta ser cuando el último bloque de consultas es menor a `MQ`. En este caso, dicho conjunto de consultas es resuelto consulta a consulta utilizando el mecanismo de la versión 1.

6.4.1. Tiempo de Ejecución

Para poder comparar la versión 2 con la versión anterior, se procede a realizar un estudio similar al realizado en la sección 6.3. Los gráficos presentados en esta sección poseen la misma forma y nomenclatura utilizada en la versión anterior, es decir, los gráficos fueron separados por rangos y en las leyendas se representan con dos dígitos el número de cores y GPUs que se utilizan. En el eje X aparece el tamaño del bloque, en porcentaje relativo al total de consultas ingresadas al sistema (excepto $1q$ que significa 1 consulta). Para mayor detalle en algunos gráficos fue utilizado el valor real correspondiente. En el eje Y es representado el tiempo.

En las figuras 6.15, 6.16 y 6.17 se muestran los resultados en términos de tiempos de ejecución para el diccionario en español.

En la figura 6.15 se puede observar que al aumentar la complejidad del problema (rangos mayores), el tamaño del bloque influye negativamente sobre el sistema global, a pesar del aumento del número de GPUs. Para esta versión es también válida la explicación respecto del mismo fenómeno en la versión 1, es decir, al aumentar el tamaño del bloque, la influencia de los dispositivos más lentos es mucho más notoria, produciéndose cuellos de botella. Aún así, la mala influencia no es tan notoria como en la versión 1. Sin embargo, a medida que el número de consultas por bloque aumenta, los comportamientos de ambas versiones tienden a ser muy similares. Cuando los tamaños de los bloques son bajos, las GPUs pueden procesar más dinámicamente los bloques, es decir, procesar un mayor número de consultas, aumentando el rendimiento del sistema (la figura 6.21 muestra información relativa a las cantidades de consulta procesadas)

La figura 6.16 muestra los resultados presentados en la sección 6.3 (versión 1: v1 en la leyenda del gráfico) con respecto a la versión 2 (v2 en la

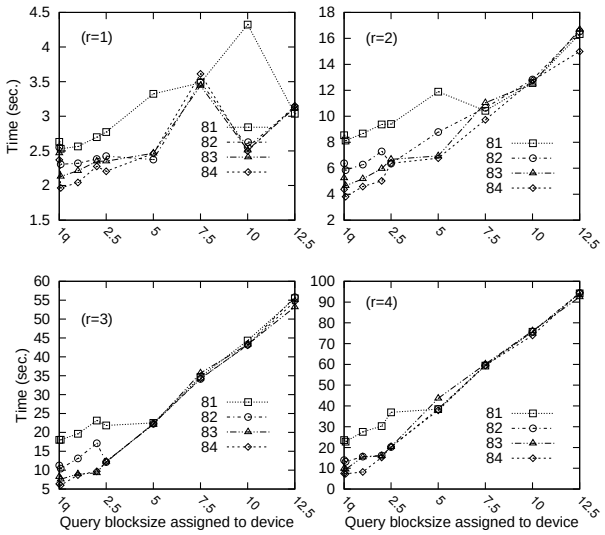


Figura 6.15: Añadiendo GPUs a la segunda versión del esquema heterogéneo. Caso de estudio espacio de palabras en español.

leyenda). La versión 2 muestra mejoras respecto de la 1, pero más significativamente para los tamaños de bloque más pequeños. Haciendo un acercamiento a los gráficos podemos observar que para la versión 2, el mejor rendimiento es cuando el número de consultas por bloque es $0,1\,\%$ del total, es decir, 8 para el caso del espacio de palabras en español (ver figura 6.17). En este gráfico se muestra en el eje X el valor real de los tamaños de bloque. Las mejoras de la versión 2 sobre la versión 1, de los mejores casos en ambas implementaciones, son del orden de $1,2$ veces, lo que equivale a una reducción en el tiempo de ejecución de un $17\,\%$, decreciendo en el peor caso hasta un $1,5\,\%$.

Las figuras 6.18, 6.19 y 6.20 representan la misma información para el caso de estudio de los histogramas de colores. Al igual que para el espacio de palabras en español, el incremento de GPUs en el sistema es relevante para los bloques con tamaño más pequeño. Al igual que en el caso de palabras, el mejor rendimiento se obtuvo con un tamaño equivalente al $0,1\,\%$, es decir 11 consultas. La versión 2 logró una mejora de $1,33$ veces la versión 1, lo que es equivalente a una reducción en el tiempo de ejecución cercana al $25\,\%$ en el mejor caso. En el peor caso, la reducción del tiempo de ejecución fue cercana al $11\,\%$

En ambos espacios, las ventajas obtenidas en la versión 2 eran esperadas al tener menor cantidad de llamadas a los kernels y menos transferencias de datos. Sin embargo, al aumentar los rangos de búsqueda las mejoras de los tiempos de ejecución decrecen debido a que los costos de procesamiento de los kernels se hacen más pesados lo que amortiza en menor medida lo

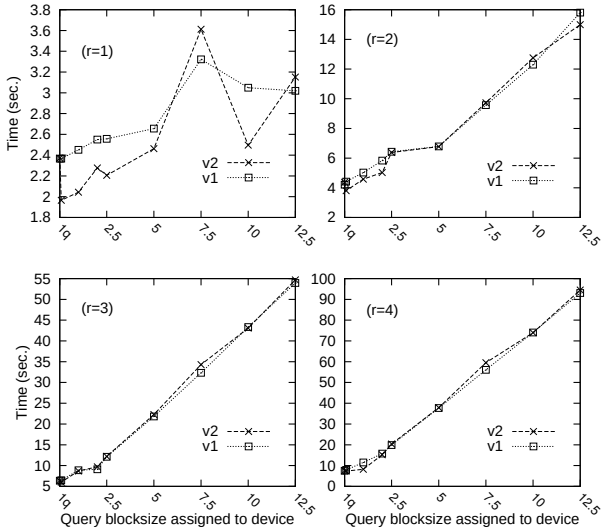


Figura 6.16: Comparación de resultados para las versiones 1 y 2. Caso espacio de palabras en español.

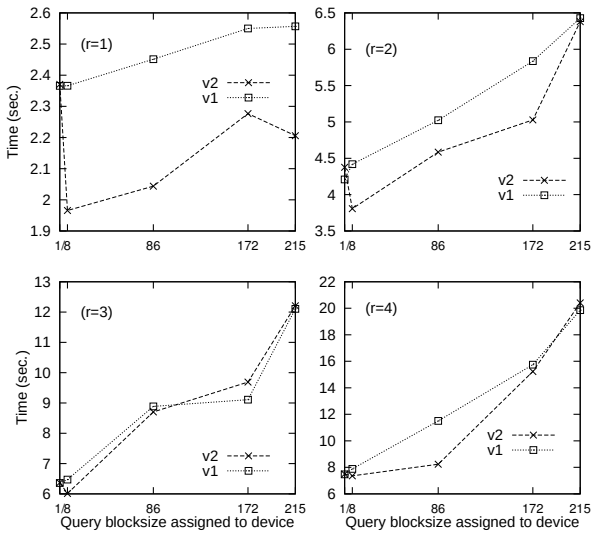


Figura 6.17: Detalles del comportamiento de las distintas versiones para bloques de tamaño reducido, usando los valores reales del tamaño de bloque: 1 - 1q, 8 - 0,1 %, 86 - 1 %, 172 - 2 %, 215 - 2,5 %. Caso espacio de palabras en español.

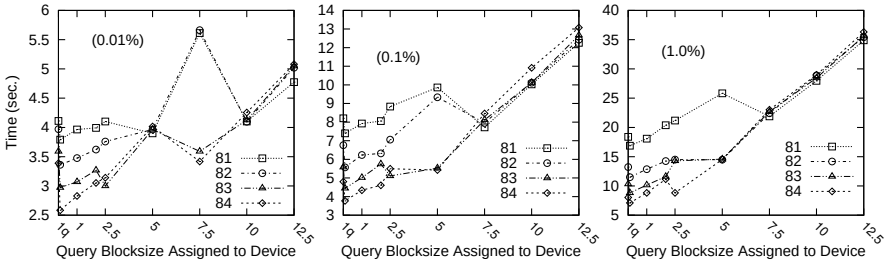


Figura 6.18: Añadiendo GPUs a la segunda versión del esquema heterogéneo. Caso de estudio histogramas de colores.

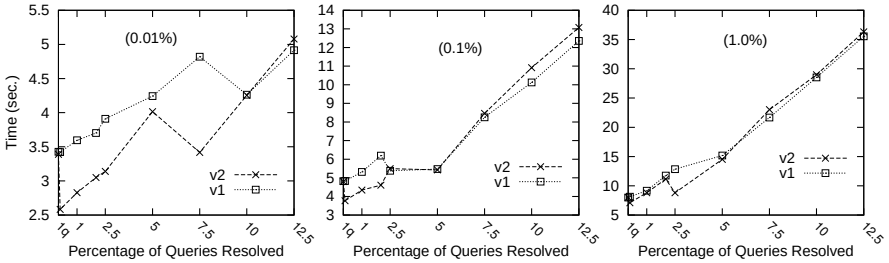


Figura 6.19: Comparación de resultados para las versiones 1 y 2. Caso histogramas de colores.

ahorrado en transferencias y llamadas a kernels.

6.4.2. Distribución de los Tiempos de Ejecución y de las Consultas

Las figuras 6.21 y 6.22 muestran la distribución de las consultas entre los dispositivos (columna izquierda) y la distribución de los tiempos de ejecución entre los tres procesos más importantes en la GPU (columna derecha) para un sistema con 8 cores y 4 GPUs.

De estas figuras se pueden obtener datos relevantes, como por ejemplo, que existe una reducción importante en la transferencia de datos entre la memoria principal y la GPU, lo que implica un aumento del uso del kernel. Por ejemplo, para el caso de los histogramas de colores, y para la versión 1, la transferencia implicaba un $28,3\,\%$ del total del tiempo utilizado por el core administrador de la GPU, mientras que en la versión 2 sólo corresponde a un $16,9\,\%$ para el rango que recupera el $0,01\,\%$ de los datos. Sin embargo, para el rango equivalente al $1\,\%$ va de un $14,8\,\%$ a un $7,6\,\%$, es decir, una reducción cercana al $50\,\%$. Los efectos sobre el kernel son, que mientras en la versión 1 el kernel utilizaba entre un $37,7\,\%$ y un $66,4\,\%$ del tiempo total de la GPU (del core que administra la GPU) para el menor y mayor rango respectivamente, en la versión 2 va de un $49,2\,\%$ a un $75,7\,\%$, aumentando

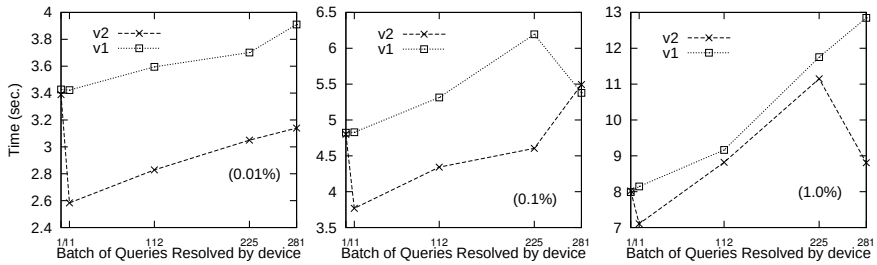


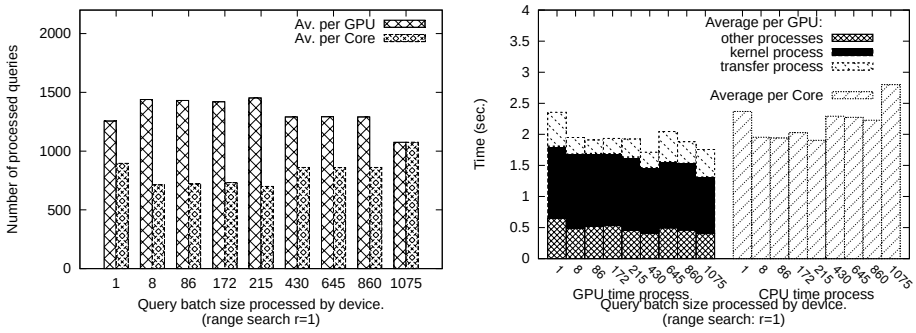
Figura 6.20: Detalles del comportamiento de las distintas versiones para bloques de tamaño reducido, usando los valores reales del tamaño de bloque: 1 - 1q, 11 - 0,1 %, 112 - 1 %, 225 - 2 %, 281 - 2,5 %. Caso histogramas de colores.

el uso de la GPU de manera significativa. Lo anterior provoca que el kernel llegue a procesar casi un 13 % más de consultas para el rango más bajo, mientras sólo llega a un $1,5$ % en el rango mayor.

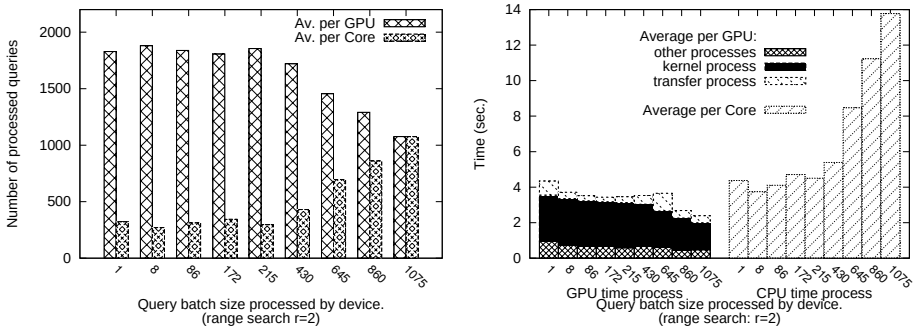
Para el espacio de palabras en español el comportamiento es idéntico, reduciendo la versión 2, con bloques de tamaño $0,1$ %, los tiempos en todos los rangos de búsqueda. Para el rango menor ($r = 1$) la cantidad de consultas adicionales procesadas fue de 13 %, aunque para el mayor rango no alcanza a llegar a un $0,05$ %. Como se explicó anteriormente, al aumentar los rangos, el comportamiento tiende a ser similar en ambas versiones. Con respecto a los tiempos de transferencia, para este espacio hubo una reducción sobre el 50 % en el rango mayor y cerca de 46 % para el rango menor. En los gráficos de tiempo de la figura 6.22 se puede ver claramente que la alternativa de 11 consultas por bloque resulta ser la mejor; si se observa el gráfico para $0,01$ % de recuperación se aprecia que este tamaño de bloque tiene el menor tiempo tanto en GPU como en los cores. En el caso de bloques de mayor tamaño el tiempo de la GPU disminuye, pero esto es debido a su poca utilización, ya que se verifica que los cores han aumentado su carga de trabajo y por ende el tiempo (ver figura 6.21(a), columna derecha, *CPU time process*).

6.4.3. Escalando el Número de Consultas

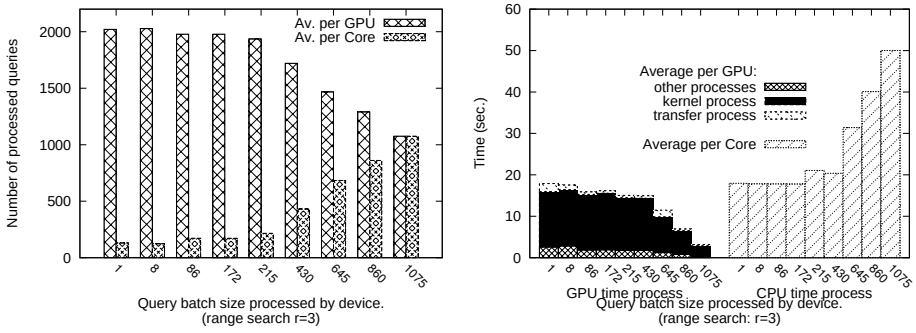
Al poner el sistema bajo *stress*, es decir, aumentar el número de consultas, el mejor comportamiento sigue indicando que no es 1 el valor ideal para el tamaño del bloque, de hecho, este valor tiende a estar entre 10 y 100, en ambos espacios, e incluso en algunos experimentos del espacio de vectores el valor de 1.000 también obtuvo un buen rendimiento y en algunos casos superiores a los tamaños 10 y 100. Esto último se debe básicamente a la gran cantidad de consultas que debe soportar el sistema y a que, por ejemplo, un tamaño de 100 no provoca aún un cuello de botella en los dispositivos más lentos. Para el espacio de vectores un bloque de tamaño 10 mil no puede



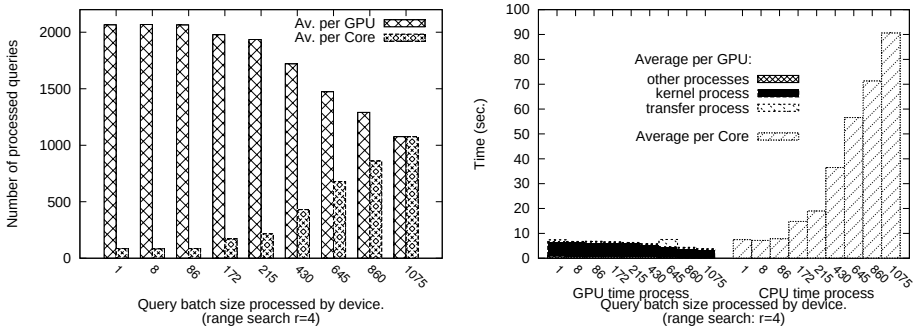
(a) $r = 1$



(b) $r = 2$

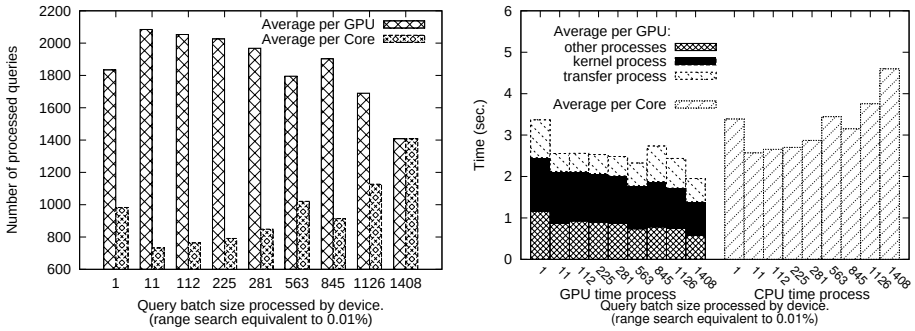


(c) $r = 3$

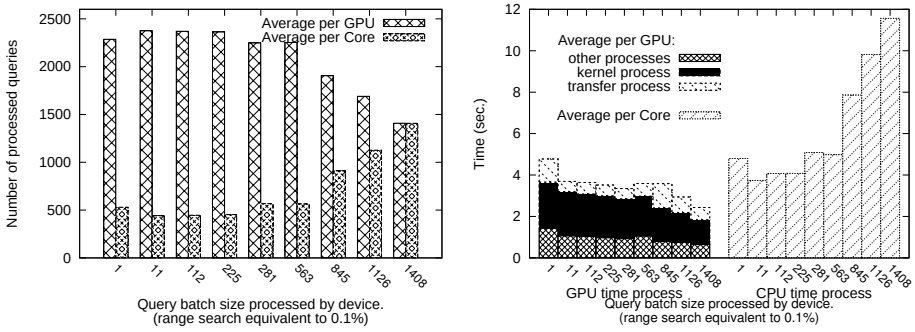


(d) $r = 4$

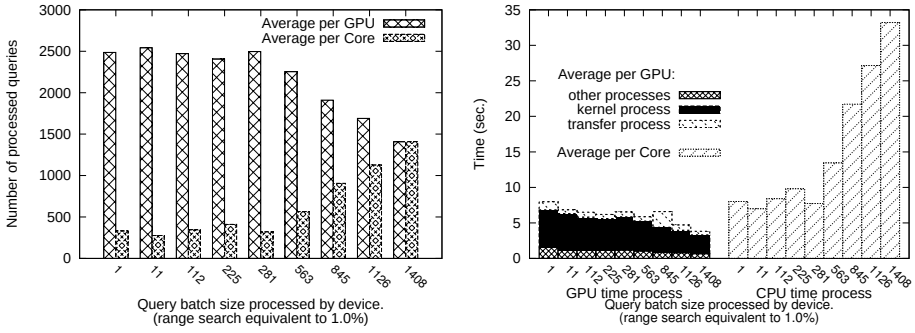
Figura 6.21: Versión 2: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso diccionario en español.



(a) $0,01\,\%$



(b) $0,1\,\%$



(c) $1,0\,\%$

Figura 6.22: Versión 2: Distribución de consultas (columna izquierda) y tiempos de ejecución (columna derecha) sobre un sistema heterogéneo considerando 8 cores y 4 GPUs. Caso histogramas de colores.

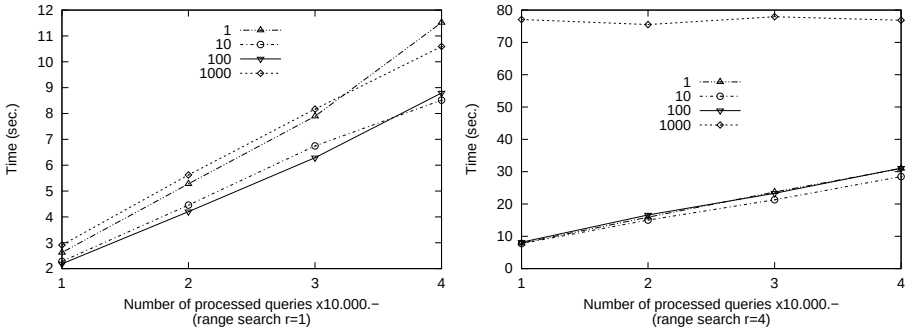


Figura 6.23: Versión 2: Escalabilidad del sistema. Tiempos de ejecución versus tamaño del bloque cuando el número de consultas varía de 10 mil a 40 mil considerando los rangos 1 y 4 en un sistema con 8 cores y 4 GPUs para el espacio de palabras en español.

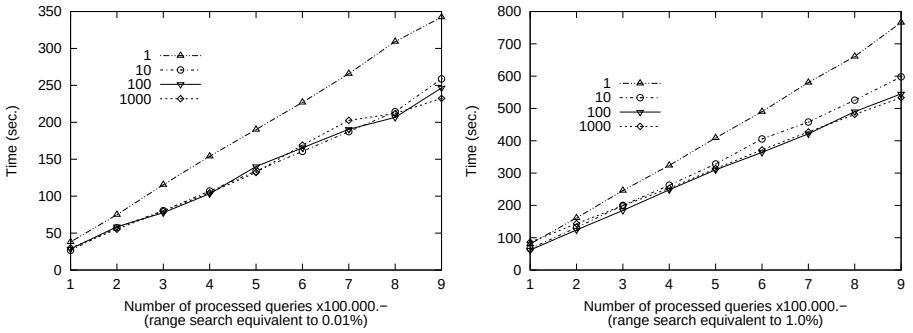


Figura 6.24: Versión 2: Escalabilidad del sistema. Tiempos de ejecución versus tamaño del bloque cuando el número de consultas varía de 100 mil a 900 mil considerando los rangos que recuperan el 0,01 % y 1 % de los datos, en un sistema con 8 cores y 4 GPUs para el espacio de vectores.

ser usado debido a que la disponibilidad de memoria en el disposivo no es suficiente para almacenar todas las estructuras auxiliares. Las figuras 6.23 y 6.24 muestran los resultados comentados.

Las figuras 6.25 y 6.26 muestran resultados comparativos para el espacio de palabras en español y el espacio de vectores usando las mejores alternativas de la versión 1, presentada en la sección 6.3, y para la versión 2. Para esta última versión se incluyó los tamaños $\texttt{MQ} = 100$ y $\texttt{MQ} = 10$.

Como se ha mencionado, se aprecia una mejora en los tiempos de ejecución de la versión orientada a bloques. Para el espacio de histogramas de colores esta mejora se refleja en una reducción de los tiempos de ejecución para el radio 0,01 % (respecto de la versión 1) que van desde los 25 % y 20 % para 100 mil consultas y $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$ respectivamente, hasta 25 % y 28 % para 900 mil consultas y $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$ respectivamente. En este

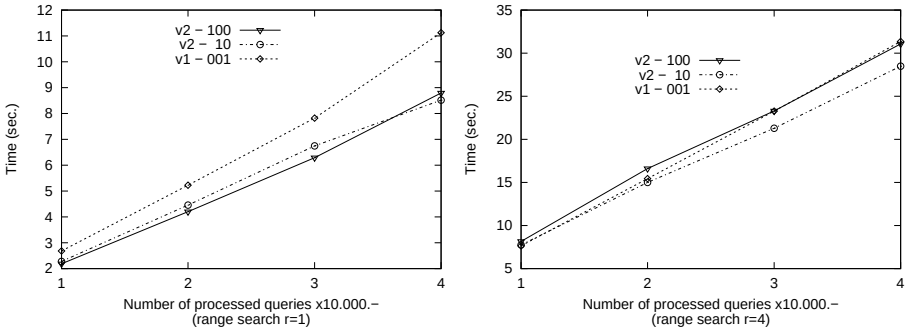


Figura 6.25: Versión 2: Escalando en número de consultas de 10 mil a 40 mil para el espacio de palabras en español y rangos 1 y 4 en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa *chunk* = 1) y versión 2 con `MQ` = 10 y `MQ` = 100.

espacio se observa que la mejor alternativa para rangos altos es `MQ` = 100; para el rango 1 % la reducción de tiempo varía entre un 21 % y 25 %, en el mejor de los casos para 100 mil consultas y 900 mil respectivamente. Sin embargo, para el rango menor también es competitiva, aunque existen varios puntos donde la versión con tamaño de bloque 10 es mejor que la de 100.

Para el espacio de palabras en español se debe hacer un análisis más profundo. Se incluyen las dos mejores alternativas `MQ` = 10 y `MQ` = 100. Por ejemplo, para bloques de tamaño 100 y rango de búsqueda 1, los rangos de mejora van desde 19 % (10 mil consultas) hasta 21 % (40 mil consultas). Sin embargo, un bloque de tamaño 100 puede no ser el más apropiado para rango 4. Para este rango los mejores resultados fueron obtenidos con bloques de tamaño 10. Lo anterior quiere decir que 100 aún puede ser un valor muy grande para este número de consultas (consultas totales), y su mejor comportamiento a $r = 1$ es debido que a tan bajo rango las consultas son rápidamente resueltas. Así, un bloque de ese tamaño igualmente puede ser descartado de manera rápida por el dispositivo más lento, lo que efectivamente no ocurre para $r = 4$.

6.4.4. Discusión Respecto de la Versión Orientada a Bloques

Como se esperaba, la implementación orientada a bloques mejora el rendimiento respecto de la versión de una consulta. En la versión 2 los tiempos de transferencias se reducen, en el mejor caso, hasta en un 50 % respecto de la versión 1. Los tiempos de ejecución son mejorados entre un 25 % y 27 % cuando es escalado en número de consultas.

Hay que considerar, sin embargo, que la versión 2 del sistema heterogéneo requiere de mucha más memoria en el dispositivo gráfico. Para esta versión se requiere una cantidad de memoria global proporcional al número de consul-

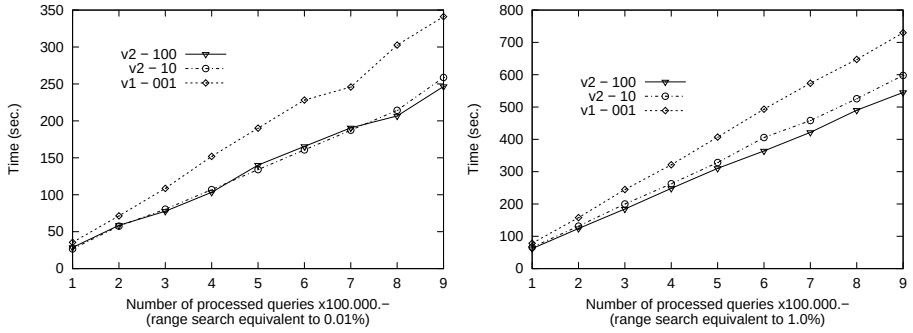


Figura 6.26: Versión 2: Escalando en número de consultas de 100 mil a 900 mil para el espacio de vectores y porcentajes de recuperación de 0,01 % y 1 % en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa *chunk* = 1) y versión 2 con `MQ` = 10 y `MQ` = 100.

tas de un bloque; este espacio de memoria es para almacenar los resultados relativos a cada consulta. En particular, se requiere una matriz de dimensiones `MQ` × `MAX_DATA`, siendo `MQ` el tamaño del bloque y `MAX_DATA` el número total de objetos de la base de datos (`MAX_DATA` = $|\mathbb{Y}|$, ver definiciones de estas variables en la sección 5.2). En la versión 1, solamente se necesita un arreglo de un tamaño máximo de `MAX_DATA`. Para la versión 2 la cantidad de consultas por bloque está supeditada al espacio disponible en la GPU, sin embargo, igualmente hay que tener en cuenta que los mejores resultados fueron obtenidos para bajos valores de tamaños de bloque.

Con respecto a las versiones secuenciales, los speed-ups obtenidos van entre el 14,28 a 26,68 para el menor y mayor rango en el *GMS* secuencial y 7,72 a 25,53 para el *Spaghettis* secuencial en el espacio de vectores. Para el espacio de palabras en español, los speed-ups conseguidos varían entre los 10,7 y 96,44 para el menor y mayor rango respecto del *GMS* secuencial y entre un 7,65 y 94,45 para la versión secuencial del *spaghettis*.

Finalmente se hace mención del tratamiento al último bloque. El tamaño de este bloque puede no ser exactamente `MQ`, lo que provoca que no es resuelto con el mecanismo anterior (versión 2), sino utilizando el esquema propuesto en la versión 1. En este caso, cada consulta del último bloque es distribuida sobre el conjunto de dispositivos y es resuelta en forma individual. Esto provoca una reducción en el rendimiento general del sistema debido a una sincronización previa a la resolución de las consultas del último bloque, a la distribución de las consultas sobre todos los cores y a su resolución. Lo anterior fue diseñado sobre la base de que se desconoce el número de consultas que ingresan al sistema.

6.4.5. Escalando la Base de Datos

No es trivial el problema de la transferencia de datos entre la memoria principal del sistema y hacia o desde la memoria global del dispositivo gráfico. Al definir a priori el tamaño del bloque de consultas se puede calcular un valor óptimo de acuerdo a las propiedades de la tarjeta gráfica en cuestión. En la versión 2 del sistema, la memoria necesaria para almacenar las soluciones obtenidas en la GPU es solicitada previa a las búsquedas, por lo que esto es hecho una sola vez y la transferencia hacia la GPU es solamente de un bloque de consultas por cada kernel, es decir `MQ` elementos ($\texttt{MQ} \times \texttt{MAX_DIM}$ valores flotantes o enteros). Sin embargo, la transferencia final realizada desde la GPU a la memoria principal es $\texttt{MQ} \times \texttt{MAX_DATA}$ valores de tipo entero o entero largo (`int` o `long int`) para guardar los identificadores de los objetos que son soluciones de las respectivas consultas del bloque. Si consideramos la transferencias para la base de datos original de histogramas de colores y con $\texttt{MQ} = 10$, tenemos alrededor de 4.480 bytes ($0,0043$ Mbytes) de subida (a la GPU) y 3,86 Mbytes de bajadas (desde la GPU). En este caso, interesa entonces la transferencia de datos desde GPU a la memoria principal, por lo que para $\texttt{MQ} = 100$, la transferencia de las soluciones desde GPU a memoria principal es de 38,68 Mbytes de datos.

Para ampliar la perspectiva del comportamiento del sistema, en su versión 2, se muestra en la figura 6.27 los resultados para un nuevo conjunto de experimentos para dos nuevas bases de datos similares en tamaño. Para mantener una relación con los datos anteriores, la primera base de datos es el espacio de histogramas de colores amplificado 10 veces de manera sintética, con una base de datos de 1.014.138 objetos y 112.682 consultas. La segunda base de datos utilizada es una base de datos real de imágenes denominada COPHIR, de dimensión 64, con 999.987 objetos y un log de consultas reales ingresadas al sistema de 92.677. La descripción detallada de los nuevos conjuntos de datos se encuentra en la sección 4.1.

Utilizando los mismos tamaños de bloques, se tiene, por ejemplo, que para el primer espacio se requieren sucesivas transferencias de alrededor de $386,86$ Mbytes, con $\texttt{MQ} = 100$. La figura 6.27 muestra ambos espacios, en la columna izquierda la base de datos de COPHIR y en la columna derecha el espacio de histogramas de colores. Para ambos espacios el comportamiento es similar, considerando que los tamaños son parecidos.

Considerando los resultados obtenidos en la sección anterior para 100 mil consultas, en donde $\texttt{MQ} = 100$ arrojó buenos resultados (mejores que $\texttt{MQ} = 10$ en el rango mayor), podemos observar que nuevamente en estos experimentos el valor para $\texttt{MQ} = 10$ resulta ser el mejor, tanto a bajos rangos como en los más altos.

Dos detalles se observan en el gráfico para COPHIR, el primero es que con $\texttt{MQ} = 100$ el rendimiento baja demasiado, a nivel que la versión 1 del sistema supera a la versión 2. El segundo detalle a considerar es que para

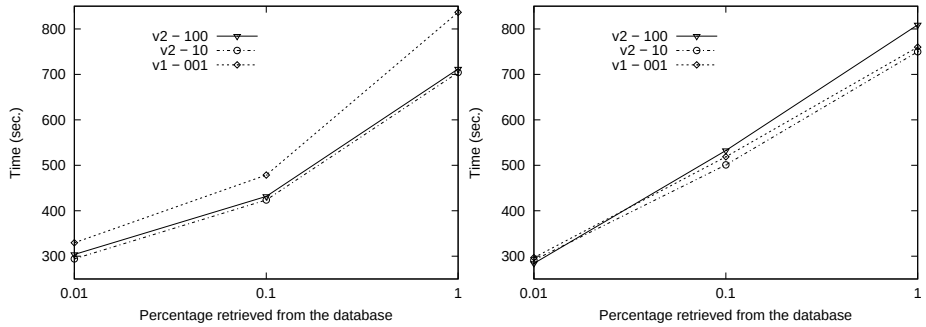


Figura 6.27: Versión 2: Escalando la Base de datos 1 millón de objetos para el histograma de colores (izquierda) y la nueva base de datos de COPHIR (derecha) en un sistema con 8 cores y 4 GPUs. Comparando resultados para la versión 1 (alternativa $chunk = 1$) y versión 2 con $\texttt{MQ} = 10$ y $\texttt{MQ} = 100$.

el rango 0,01 en COPHIR, el tamaño de bloque 100 es el mejor. Esto se ha comentado en varias oportunidades a lo largo de este informe, y es principalmente que a bajos rangos (o en el caso del más bajo) la estructura se comporta muy bien, en este caso la demora provocada por una transferencia de tamaño superior es solapada por la rapidez de la respuesta con 100 elementos por bloque.

6.5. Conclusiones

En el presente capítulo se han presentado dos implementaciones heterogéneas para búsqueda por similitud en espacios métricos, usando la estructura genérica denominada en este trabajo como *GMS*. La primera versión tiene en cuenta que cada consulta es resuelta en forma independiente, y aquellas asignadas a una GPU, por sólo un kernel (1 consulta - 1 kernel). La segunda versión corresponde a una implementación orientada a bloques, la que transfiere un bloque completo (*chunk*) que contiene `MQ` consultas, las que son resueltas en un único kernel (`MQ` consultas - 1 kernel).

En relación a la memoria utilizada, la memoria requerida por la versión 2 es muy superior a la versión 1, dejando menos espacio para almacenar objetos de la base de datos o elementos en la estructura. La memoria adicional requerida por la versión 2 es proporcional a la cantidad de consultas de cada bloque y es necesaria para almacenar los resultados de cada una de las consultas.

Los tiempos de transferencia y las llamadas al kernel han sido reducidas por la versión 2, con respecto a la versión 1. En este caso resulta menos costoso transferir `MQ` consultas en un "paquete" que transferir `MQ` "paquetes" de una consulta cada uno (además de `MQ` llamadas a un kernel). En general, en el mejor caso, existe una reducción en tiempo cercana al 50 % en la

Tabla 6.2: Número de consultas por segundo para todas la versiones y los dos espacios de prueba (diccionario en español e histogramas de colores), considerando solamente el rango mínimo y máximo. `MQ` equivalente a 0,1 % de las consultas.

Espacios / Algoritmos	Diccionario Español		Histogramas de Colores	
	1	4	0,01	1,0
secuencial (1 core)	409	12	333	44
multicore (8 cores)	3098	95	2472	342
1 GPU (1 q - 1 k)	616	290	648	336
1 GPU (`MQ` q - 1 k)	769	310	873	372
4 GPUs (1 q - 1 k)	2131	1122	2069	1305
4 GPUs (`MQ` q - 1 k)	2922	1216	3086	1493
Sistema Completo v 1	3638	1152	3292	1411
Sistema Completo v 2	4378	1169	4362	1586

transferencia total de datos.

Otra medida de rapidez del algoritmo vendrá de responder a la pregunta: ¿Cuántas consultas son procesadas por segundo?. La respuesta es dada en la tabla 6.2, la que muestra los resultados en términos de cantidad de consultas resueltas en un segundo, relativa a todas la implementaciones y los rangos extremos utilizados en cada espacio (menor y mayor rango).

En la tabla 6.2 se usaron los valores de tamaño de bloque `MQ` de mejores resultados en términos generales. Para ambos espacios este valor coincidió con el 0,1 % (obtenido experimentalmente) del total de consultas que se ingresa al sistema. Para el espacio de palabras en español el tamaño ideal del bloque fue de 8 consultas y para el espacio de vectores de 11. Se debe recordar que para la versión 1 del sistema (v1), el tamaño del *chunck* fue de 1 (en esta versión el tamaño del bloque siempre es 1, independientemente del tamaño del *chunck*).

A simple vista, de la tabla 6.2 se obtienen algunas conclusiones esperadas. En ambos casos el comportamiento de las distintas implementaciones tiende a ser similar, evidentemente la versión 2 supera con creces a la versión 1, resolviendo sobre un 20 % más de consultas para el rango más bajo, en el espacio de palabras en español y casi un 25 % en el espacio de vectores. Sin embargo, al ir aumentado el rango esta ventaja decrece considerablemente, sobre todo en el caso del espacio de palabras en español.

De la tabla 6.2 también se pueden obtener algunas observaciones interesantes. Por un lado, el sistema multicore muestra excelentes resultados para rangos muy bajos de búsqueda, como es el caso de palabras, en donde supera a la versión de 4 GPUs. Para el espacio de vectores su rendimiento igualmente es bueno, pero no logra superar al sistema con 4 GPUs. Para rangos medios y altos la implementación multicore se degrada considerablemente, siendo superada incluso por un sistema de sólo una GPU. Los resultados

mostrados en la tabla reafirman las conclusiones obtenidas en capítulos anteriores, referentes a que las GPUs proporcionan un alto poder de cómputo, y en el caso de las estructuras métricas las mayores prestaciones se obtienen (versus multicore) cuando los rangos de búsqueda son altos.

Si se observa la fila correspondiente a "4 GPUs (`MQ` q - 1 k)" (tabla 6.2) y la columna rango 4 (diccionario español), llama la atención que en este caso 4 GPUs funcionando sin cores adicionales superan al sistema completo. Este efecto indica que aún es posible afinar un poco más el parámetro de tamaño de bloque para que realmente los cores, siendo más lentos, aporten al sistema y no lo retrasen.

Por ejemplo, si se activan solamente las GPUs en un sistema heterogéneo, es decir 4 cores y 4 GPUs, para el espacio de vectores el tamaño de bloque de 0,1 % sigue siendo el más adecuado, mientras que para el espacio de palabras en español este valor oscila entre el 0,1 % y 1 %.

Los speed-ups alcanzados por la versión 2 varían entre aproximadamente 8 para los rangos más bajos en ambos espacios y entre un 26 y 94 para los rangos mayores en el espacio de palabras en español y vectores respectivamente, comparándola con la mejor estructura secuencial (*Spaghettis*). La variabilidad de los speed-ups depende del comportamiento de la estructura. En general, las estructuras a rangos altos tienen un rendimiento degradado por el aumento en los cálculos de distancias (la menor cantidad de objetos descartados durante la búsqueda). Esto provoca que la utilización de una mayor potencia de cálculo entregada por una plataforma paralela contribuya a repartir la carga computacional y con ello a aumentar considerablemente este rendimiento.

Considerando solamente la versión 2, algunos datos relevantes obtenidos indican que la cantidad de consultas resueltas por el conjunto de GPUs es sobre 2 veces más que el conjunto de cores y casi 25 veces más cuando el rango utilizado es el mayor, es decir, entre un 67 % y un 96 % del total de consultas que entran al sistema son resueltas por las GPUs, estos datos para el espacio de palabras en español. Para el espacio de vectores entre un 74 % y un 90 % del total de consultas son resueltas por las GPUs. Lo anterior prueba que las GPUs han sido mejor explotadas a rangos mayores.

Queda de manifiesto que el sistema es escable al obtener un rendimiento lineal al aumentar el número de consultas que resuelve el sistema. El comportamiento de ambas versiones heterogéneas no varía significativamente. Para la versión 2 es posible obtener algún beneficio adicional si se estudia más a fondo el parámetro del tamaño del bloque. En los experimentos mostrados en la sección previa, se muestra que los valores de `MQ` = 100 son competitivos con los de `MQ` = 10, sobre todo en el espacio de vectores, en donde la cantidad de consultas van desde los 100 mil hasta las 900 mil.

Las conclusiones obtenidas al escalar la base de datos y las consultas, para vectores, indican lo relevante de considerar los tamaños en función de las transferencias, las cuales pueden provocar una disminución en los

rendimientos del sistema. Es decir, los tamaños de bloque no solamente se definen en función de la velocidad con la que los dispositivos más lentos pueden procesar un bloque de consultas, sino también, en que el bloque no entorpezca el buen rendimiento producto de transferencias demasiado grandes.

Capítulo 7

Ampliando la Perspectiva sobre Temas Relacionados

Existen algunos temas adicionales, muchos de los cuales no siempre son considerados lo suficiente en el diseño o la implementación de estructuras métricas. El no considerar dichos temas puede significar que la implementación en sistemas reales de estas estructuras no tengan el comportamiento o el rendimiento esperado. En esta capítulo se discuten algunas materias consideradas relevantes y afines a la presente Tesis.

Para evitar distraer la atención del lector respecto de las implementaciones heterogéneas y estructuras métricas implementadas sobre plataformas basadas en GPU como materia principal, estos temas adicionales, pero afines y relevantes, son desarrollados en el presente capítulo. Estos temas son: jerarquía de memoria y accesos a memoria secundaria, esquemas de planificación de distribución de consultas alternativos y finalmente, consumo eléctrico y/o consumo energético de las implementaciones sobre distintos dispositivos.

7.1. Estructuras Métricas sobre la Jerarquía de Memoria en una Plataforma Heterogénea

Un tópico de especial interés es el manejo de los datos en los distintos niveles en la jerarquía de memoria en una plataforma cualquiera. En este sentido, la mayoría de los trabajos sobre estructuras métricas abordan sólo un nivel de memoria, la RAM. Sin embargo, existe una serie de trabajos que consideran la utilización de la memoria secundaria y el análisis desde la perspectiva de optimización del espacio utilizado, de la mejora en los accesos, tanto de lecturas (*reads*) y escrituras (*writes*) en disco, como de movimientos de cabezal realizado durante las búsquedas (*seeks*). A su vez, algunos trabajos desarrollados sobre plataformas basadas en GPU diferencian básicamente dos niveles de memoria dentro del dispositivo: *Memoria*

Global y *Memoria Compartida*. Si bien, muchos de estos últimos trabajos consideran los costos de tranferencia entre memoria principal y memoria del dispositivo, la mayoría asume que toda la información puede ser contenida dentro de la memoria del dispositivo.

En esta sección se presenta un análisis experimental de una implementación adaptada a memoria secundaria de la estructura *GMS*, enfocada hacia su posterior implementación sobre una GPU. La sección está dividida en tres partes: la primera considera la implementación sobre memoria secundaria y un análisis de los costos de transferencia. La segunda parte corresponde a la implementación sobre GPU y a un analisis de los costos que incluyen las transferencias de los distintos trozos de la estructura y base de datos, así como de la cantidad de consultas que pueden ser resueltas de una sola vez por el dispositivo. La tercera parte es una discusión de como puede ser reducido el tiempo de ejecución al solapar la ejecución de procesos de distintos dispositivos.

Las comparaciones fueron realizadas contra la estructura *EGNAT* [88, 63], la cual fue específicamente diseñada para memoria secundaria. La otra estructura utilizada es la *Lista de Clusters* o *LC* [23], la cual es frecuentemente utilizada y es una estructura de referencia en muchos trabajos del área y para la cual se implementó una versión optimizada para memoria secundaria y una implementación sobre GPU.

Las tres estructuras representan ejemplos clásicos de diferentes tipos de estructuras. Adicionalmente, *EGNAT* resulta una estructura de tamaño medio comparada con *GMS*, que resulta ser la estructura que utiliza más espacio de las tres. La tercera, *LC*, se implementó en su versión original, de tal manera de reducir al máximo el espacio de almacenamiento, resultando ser la de menor tamaño y por lo tanto la que menos accesos a disco realiza.

Finalmente, a parte de las características indicadas para cada estructura, se entiende que el uso de las dos estructuras adicionales es para tener un marco de análisis comparativo considerando que resultados previos daban que *GMS* era una estructura idónea para implementar sobre GPU. De esta manera se obtiene una base sólida que demuestra que *GMS* tiene buen comportamiento a distintos niveles en la jerarquía de memoria de una plataforma heterogénea.

7.1.1. Transfiriendo Datos desde Memoria Secundaria a Memoria Principal

En general, para la implementación en memoria secundaria, todas la estructuras se dividen en nodos del tamaño de una página de disco, las que se van cargando a la memoria principal cuando sean requeridas. Para las implementaciones de esta sección se utilizó como tamaño de una página de disco el valor 4.096 bytes, que es la que se indica en la arquitectura utilizada.

La versiones utilizadas para todas las estructuras fueron las detalladas

en el capítulo 3. *LC* fue construida utilizando clusters de tamaño fijo; para los experimentos aquí presentados la capacidad máxima del cluster fue de 32 objetos. Para la estructura *GMS* se ha utilizado la versión de 32 pivotes.

Las estructuras *GMS* y *LC* son de tipo arreglo, por lo que se implementó de manera sencilla un método para construir las estructuras en base a nodos de igual o menor tamaño que una página de disco. En este aspecto, las implementaciones de *GMS* y *LC* sobre memoria secundaria tienen características similares. Durante el proceso de contrucción sobre ambas estructuras se realiza lo siguiente:

- Se ejecuta el algoritmo original para obtener los elementos de cada fila de la estructura.
- Se completa el número de filas soportadas por una página de disco (Cluster en caso de *LC* y número de objetos en caso de *GMS*).
- Se enlaza cada página de disco obtenida con la siguiente.

Además, para hacer más fácil la sintonización de los experimentos, se definieron algunas variables adicionales:

1. `MAX_CLUSTERS_PAGE` y `MAX_OBJECT_PAGE`: Estas dos variables definen el máximo número de clusters por página para el caso *LC* o máximo número de objetos por página (cada objeto es representado por una fila con las distancias a los pivotes), para el caso de *GMS*.
2. `MAX_PAGES_RAM` y `MAX_QUERIES_RAM`: Para todas las estructuras se utilizan estos dos parámetros en la ejecución de los experimentos, los cuales definen la cantidad de páginas soportadas en memoria principal y la máxima cantidad de consultas que serán cargadas en RAM.

Para realizar la comparación entre las diferentes estructuras sobre memoria secundaria, se ha considerado que el objeto real no sería almacenado dentro de la estructura, sino en una estructura adicional (o archivo) que sólo contendría los objetos. Esto permite poder realizar una serie de análisis y comparaciones respecto del espacio utilizado por las estructuras abstrayéndose del tamaño de los objetos. A consecuencia de lo anterior, se modificó el *EGNAT* para adaptarlo al esquema y poder realizar las comparaciones bajo los mismos parámetros.

La figura 7.1 muestra el esquema general de un sistema heterogéneo, distinguiéndose 4 niveles de memoria, desde la memoria secundaria a la que se accede por páginas de disco, la memoria principal del sistema y dos niveles básicos de memoria del dispositivo gráfico. En los experimentos presentados en esta sección no se analizaron la influencia de la memoria cache proporcionada por la plataforma ni de los registros de la GPU.

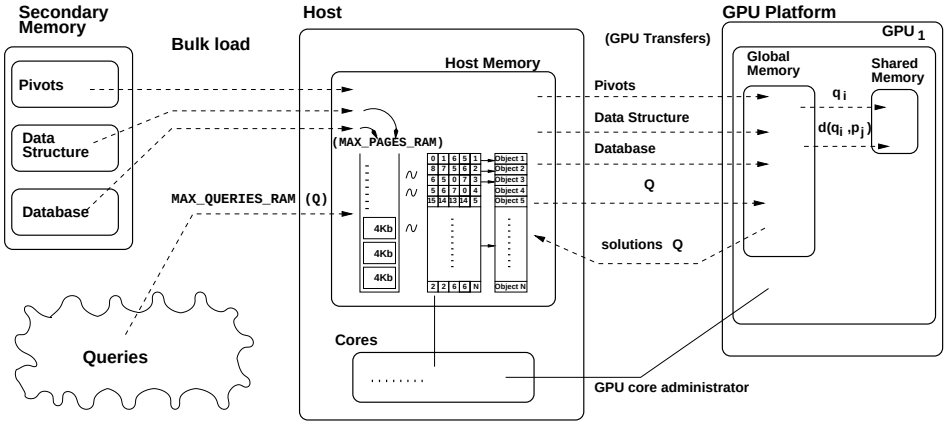


Figura 7.1: Sistema heterogéneo completo (con una GPU) y jerarquía de memoria asociada.

Básicamente, el proceso de búsqueda es el reflejado en la figura 7.1. En todas las estructuras primero se realiza la carga en memoria principal de las primeras `MAX_PAGES_RAM` páginas o nodos y sobre este subconjunto de la estructura (y base de datos) se realiza la búsqueda de las primeras `MAX_QUERIES_RAM` consultas.

Existe una diferencia respecto de los algoritmos, entre *EGNAT* versus *LC* y *GMS*. *EGNAT* está diseñado para resolver una consulta a la vez, por lo que no influye la cantidad de consultas que pueden ser leídas inicialmente[1]. En el caso de las otras dos estructuras, éstas pueden resolver todas las consultas leídas sobre el subconjunto de datos cargados. La diferencia radica en que en el primero, al ser un árbol, no todas las consultas recuperan los mismos nodos, no así en el caso de las otras dos estructuras.

Análisis Preliminar

Los experimentos fueron realizados sobre las dos bases de datos de prueba principales, el espacio de palabras en español y los histogramas de colores. Esto resulta idóneo ya que poseen caraterísticas singulares que las hace representantes de diferentes conjuntos y tipos de datos (características descritas en 4.1). Además, desde el punto de vista de los experimentos en memoria secundaria, una característica no profundizada en capítulos anteriores es el peso del objeto. Mientras una palabra tiene un tamaño de 28 bytes, un histograma de colores tiene un tamaño de 448 bytes en esta implementación, lo que agrega un componente más al análisis.

Durante el proceso de búsqueda, las páginas son transferidas desde memoria secundaria a memoria principal una a una hasta completar el máximo

[1]Muchos trabajos se han orientado a la optimización de consultas individuales sin considerar el procesamiento de consultas por lotes.

soportado (MAX_PAGES_RAM). Las tablas 7.1 y 7.2 muestran el detalle de la información relativa al espacio necesario para almacenar los datos en cada estructura. El espacio es dividido en las páginas requeridas por el índice y las páginas requeridas para almacenar los objetos contenidos en dichos trozos de índice.

Cada fila en las tablas mencionadas muestra la información relativa a cada estructura para una capacidad máxima del sistema, en términos de número de páginas que se pueden alojar en memoria principal (MAX_PAGES_RAM). La primera columna corresponde a la capacidad del sistema, es decir, el valor para MAX_PAGES_RAM. Posteriormente hay tres columnas por cada estructura o índice. La primera indica el número de páginas de la estructura que puede ser llevada a RAM. La segunda corresponde al número de páginas equivalentes en donde están almacenados los objetos indicados en la columna uno de la estructura (indicado como Índice). La tercera columna corresponde al porcentaje que esas páginas significan dentro del total de datos de la estructura. Cada estructura tiene entre paréntesis el total de páginas usadas por el índice sin contar la páginas utilizadas por los objetos.

A modo de ejemplo, en la tabla 7.1 que refleja los datos de los histogramas de colores, obsérvese la fila 3. En este caso el sistema puede contener en RAM un máximo de 10.756 páginas de disco. Para la estructura *EGNAT*, de ese total de páginas, 1.976 corresponden al índice y 8.780 corresponden a las páginas que almacenan los objetos reales (enlazados en las 1.976 páginas del índice). Lo anterior corresponde a un 79,31 % del total de páginas de la estructura *EGNAT*. El total de páginas de la estructura *EGNAT* son 13.561, de las cuales 2.469 son del índice y 11.092 son páginas que contienen los objetos. Las cantidades de páginas seleccionadas correspondieron básicamente a un criterio de seleccionar cantidades baja, media y alta, intentando que en al menos dos valores no se llegara al 100 %.

En las tablas 7.1 y 7.2 se puede observar que *LC* es la estructura que requiere menos espacio para almacenar el total de datos, y *GMS* es la que necesita la mayor cantidad de espacio de almacenamiento. En la literatura del área se indica que las estructuras basadas en pivotes requieren mayor espacio de almacenamiento respecto de las estructuras basadas en clustering o particiones compactas. La diferencia de tamaño entre *EGNAT* y *LC* se debe a que el primero guarda mucha información relativa a las particiones del espacio comparado con la Lista de Clusters, que guarda sólo el radio cobertor y alguna información adicional general respecto del número de elementos del cluster.

Un dato adicional no reflejado en las tablas es que, mientras que para *GMS* sólo es posible almacenar 29 objetos por página (es decir, los identificadores de 29 objetos), para el *EGNAT* es posible almacenar 338 objetos en el *bucket* (nodo de tipo bolsa) y 20 objetos en un nodo de tipo *GNAT*. Sin embargo, *LC* resulta ser un caso extremo ya que es posible almacenar hasta 14 clusters de 32 objetos cada uno más el centro del cluster, es decir 462

Páginas	*EGNAT* (2469)			*GMS* (3498)			*LC* (220)		
	Índice	Objetos	%	Índice	Objetos	%	Índice	Objetos	%
1.144	198	945	8,42	274	870	7,83	22	1.112	10,02
5.455	998	4.457	40,22	1.306	3.843	37,34	105	5.306	47,83
10.756	1.976	8.780	79,31	2.578	8.178	73,71	208	10.511	94,75
13.561	2.469	11.092	100	3.250	10.309	92,93	220	11.117	100
14.594	-	-	-	3.498	11.096	100	-	-	-

Tabla 7.1: Número de páginas y porcentaje de la estructura de datos métrica para el espacio de histogramas de colores, considerando *EGNAT*, *GMS* y *LC*.

Páginas	*EGNAT* (1.792)			*GMS* (2.671)			*LC* (168)		
	Índice	Objetos	%	Índice	Objetos	%	Índice	Objetos	%
200	151	48	8,53	166	33	6,21	48	152	28,62
628	492	136	27,05	524	104	19,62	151	477	90,02
1840	1.433	407	79,24	1.535	305	57,47	168	531	100
2.322	1.792	530	100	1.937	384	72,52	-	-	-
3.201	-	-	-	2.671	530	100	-	-	-

Tabla 7.2: Número de páginas y porcentaje de la estructura de datos métrica para el espacio de palabras en español, considerando *EGNAT*, *GMS* y *LC*.

identificadores de objetos. Una diferencia radical entre las estructuras que pueden contener más objetos es que mientras en *EGNAT*, que es un árbol, no siempre sus nodos están completos, de hecho el porcentaje de utilización del nodo es relativamente bajo (ver detalles de porcentajes de utilización en [84]), en cambio, *LC* usa totalmente cada cluster, salvo el último.

Al generar las estructuras conteniendo sólo los identificadores de los objetos, podemos hacer algunas observaciones adicionales al comportamiento de dichas estructuras al comparar ambas tablas. Por ejemplo, en el caso de *LC* una página contiene la misma cantidad de objetos[2] independientemente del tipo y del tamaño de estos, es decir, para el espacio de histogramas de colores, con 168 páginas se necesitarían 8.490 páginas para almacenar los objetos respectivos. Esto último da una idea de la cantidad adicional de lecturas provocadas sólo por el tamaño del objeto.

Para el caso del *EGNAT* esto no garantiza la misma cantidad de páginas para una misma cantidad de objetos, debido a que los nodos pueden variar ya que los objetos se distribuyen espacialmente dentro del árbol.

La figura 7.2 muestra los resultados generales sobre las tres estructuras considerando las lecturas en disco de manera secuencial por páginas. Esta figura es para tener una visión inicial del sistema, considerando que toda la estructura y base de datos cabe en memoria principal, por lo que los costos de lectura al transferir los datos desde memoria secundaria a principal fueron pagados una sola vez al inicio.

[2]Siempre es posible alguna variación provocada por la diferencia de tipos, por ejemplo en el caso de las distancias (discretas o continuas).

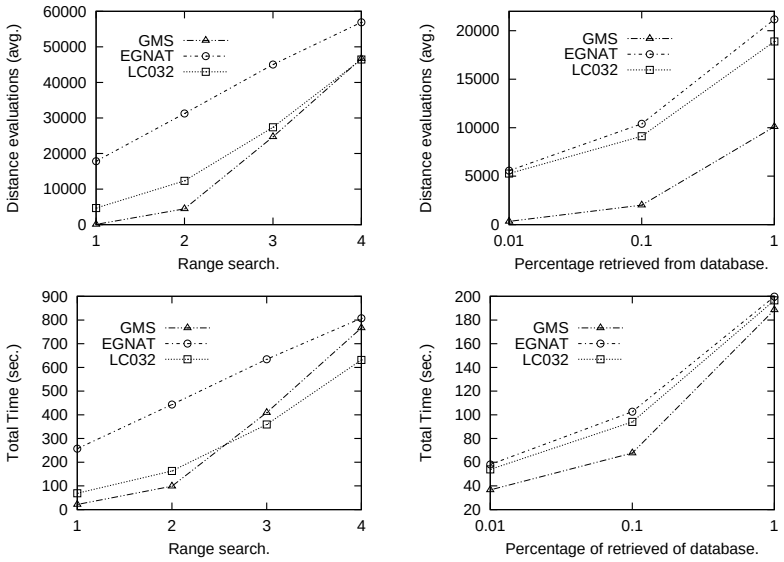


Figura 7.2: Evaluaciones de distancia (promedio por consulta) y tiempo total de ejecución para las estructuras de datos *GMS*, *EGNAT* y *LC* para el espacio palabras (izquierda) y el espacio de histogramas de colores (derecha).

Se observa que la estructura *GMS* tiene mejor comportamiento que las otras dos estructuras, el único caso que no es así es para el espacio de palabras en español y en los rangos más altos.

Los resultados mostrados aquí son coherentes con los mostrados en los gráficos de la sección 5.1.1, sin embargo, hay que considerar dos elementos adicionales en los experimentos de la presente sección. El primero, es que los costos aumentan notoriamente al considerar las lecturas a disco en bloques de 4.096 bytes, es decir, por páginas. El segundo elemento está relacionado con el *EGNAT*, la modificación mencionada previamente, de eliminar los objetos reales de la estructura y usar otra adicional para almacenarlos, hace que el *bucket* (nodo de tipo bolsa) crezca en 30 veces más (más objetos en el mismo espacio), aproximadamente, y que el nodo pueda contener algunos objetos más. Esto último variará los resultados en términos de tiempo, evaluaciones de distancia y sobre todo, reducirá el tamaño de la estructura (comparados con lo obtenido en la sección 5.1.1).

Análisis de Resultados

Las figuras 7.3 y 7.4 muestran los tiempos de ejecución necesarios para resolver todo el conjunto de consultas ingresado al sistema. En el eje Y se identifica el número de consultas que pueden ser cargadas en RAM, en porcentaje relativo al total de consultas ingresadas al sistema, por lo que los porcentajes corresponden a 86, 860 y 8.606 consultas para el espacio

de palabras en español (1 %, 10 % y 100 % respectivamente) y a 112, 1.126 y 11.268 consultas para el espacio de histogramas de colores (1 %, 10 % y 100 %). El eje X muestra el tiempo de ejecución.

Las leyendas dentro de los gráficos indican qué porcentaje de la estructura ha podido ser alojada en RAM, según la capacidad máxima del sistema para ese experimento (ver también tablas 7.1 y 7.2).

De los gráficos es posible extraer algunas observaciones y conclusiones:

- Para una mejor representación de los resultados, porcentajes de consultas en RAM menores a 1 % no son graficadas. Para casos menores a 1 % el comportamiento de las estructuras *GMS* y *LC* se tornan demasiado deficientes, no así para *EGNAT* que se mantiene constante en relación a la cantidad de consultas que llegan al servidor, y sólo varía su rendimiento en términos del número de páginas alojadas en RAM. En relación a las otras dos estructuras, *EGNAT* tiene un rendimiento muy elevado para estos bajos porcentajes de consultas.

- Para *GMS* y *LC* es más eficiente procesar porciones grandes de consultas, ya que disminuye la cantidad de veces que debe cargar en memoria los trozos de la estructura y la base de datos. Por ejemplo, para un tamaño de bloque de consultas en RAM de 1 % se debería cargar en RAM cada trozo de la base de datos y estructura (independiente del tamaño del trozo) 100 veces. Si la cantidad de consultas en RAM es el 100 %, quiere decir que cada trozo de la estructura sólo debera ser cargada en RAM una vez.

- Para el caso de los histogramas de colores, cuando el porcentaje de consultas en RAM es sobre 10 % los mejores resultados son obtenidos con *GMS*, aunque *LC* mantiene resultados cercanos a *GMS*. Como se ha dicho, *LC* guarda mucho menos información respecto de los objetos y del espacio en la estructura, lo que obliga a discriminar menos objetos durante la búsqueda, arrojando como resultado un rendimiento menor al de *GMS* (ver adicionalmente tablas 7.1 y 7.2).

- Para el espacio de palabras en español el comportamiento es levemente diferente. Primero, debido al reducido tamaño de la estructura *LC*, que utiliza sólo el 22 % del espacio que utiliza *GMS*, y que además, que dicho tamaño permite que para bajos porcentajes de nodos en RAM lo mínimo que carga *LC* es sobre un cuarto de la estructura. Aún con las ventajas de *LC* sobre *GMS* a rangos bajos de búsqueda, *GMS* resulta mucho mejor que *LC*. Para rangos mayores, donde la búsqueda se torna más "pesada" y considerando las ventajes mencionadas, *LC* tiene un rendimiento superior a *GMS* (ver figura 7.2).

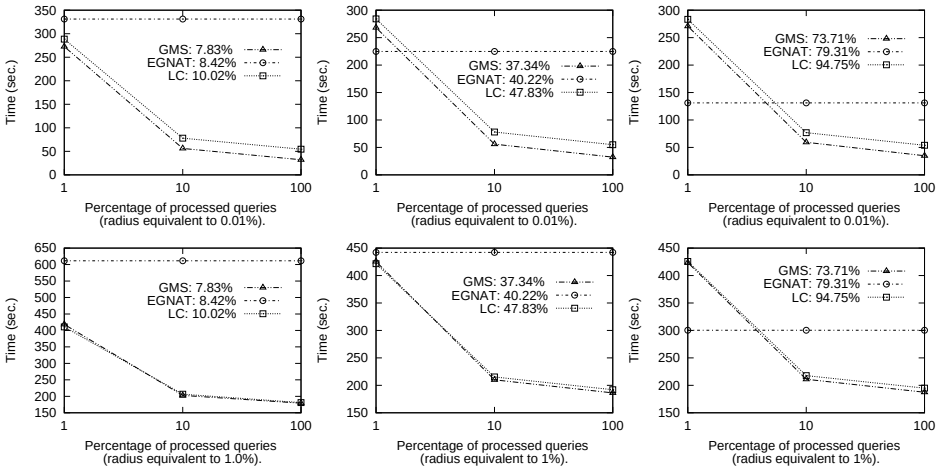


Figura 7.3: Tiempos de ejecución para los histogramas de colores considerando 0,01 % (arriba) y 1 % (abajo) de recuperación de la información de la base de datos para 1.144 (derecha), 5.455 (centro) y 10.756 (izquierda) páginas cargadas en memoria principal.

7.1.2. Desde la Memoria Principal hasta la Memoria Global de la GPU

El paso siguiente esperado es llevar los procesos de búsqueda a un dispositivo de procesamiento masivo de datos como la GPU. En los experimentos de esta subsección se abstraerá del uso de los demás elementos de procesamiento de la plataforma (cores) y se dará énfasis a los resultados que incluyen los accesos a memoria secundaria y a una GPU. Como ya se ha visto en el capítulo anterior, se sabe que podemos obtener mayores ventajas al usar todos los dispositivos de la plataforma.

La implementación llevada a GPU es la de la estructura *GMS* propuesta en la sección 6.4, es decir, la versión *GMS* sobre GPU del sistema heterogéneo versión 2, considerando la solución de varias consultas en un solo kernel. Incluye esta versión el uso de la memoria compartida del dispositivo, así como también el acceso coalescente a los datos de la estructura. También en esta implementación se consideró que solamente se lleva a memoria una porción de la estructura y de la base de datos. Por lo tanto, se puede parametrizar el porcentaje de la base de datos y de la estructura así como el número consultas que se quieren llevar a la memoria global del dispositivo gráfico.

En las figuras 7.5 y 7.6 se observa que el proceso de búsqueda llevada a cabo sobre la GPU se reduce de manera significativa, especialmente cuando la carga de trabajo es elevada (rangos mayores de búsqueda). Para estos casos, la mejora obtenida sobre el espacio de histogramas de colores va entre

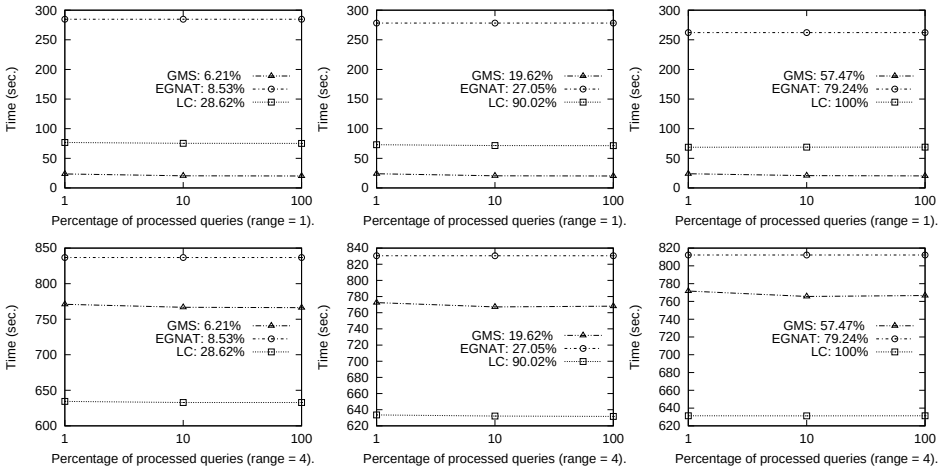


Figura 7.4: Tiempos de ejecución para el diccionadio en español considerando rango=1 (arriba) y rango=4 (abajo). Espacio de almacenamiento en RAM equivalente a 200 (izquierda), 628 (centro) y 1.840 (derecha) páginas.

un 30 % y un 75 % mientras que para el espacio de palabras en español, esta mejora es de alrededor de un 89 %.

Las leyendas de las figuras 7.5 y 7.6 corresponden a: *GMS*, *GMS* sobre GPU (*GMSonGPU*) y el tiempo cuando se realizan todos los procesos salvo la búsqueda (*only accesses*).

En esta implementación se lleva a memoria global de la GPU un trozo de la base de datos (`MAX_PAGES_RAM`) y un conjunto de consultas (`MAX_QUERIES_RAM`). Es decir, el tamaño de bloque utilizado en los experimentos (de las figuras 7.5 y 7.6) fue el indicado como la máxima cantidad de consultas en RAM. Con esto se evita separar nuevamente los conjuntos de consultas transferidas desde memoria secundaria a memoria principal y el bloque completo es transferido a la GPU y resuelto, de esta manera se evita realizar particiones adicionales. Esto resulta adecuado considerando que el cuello de botella se encuentra en los accesos a disco y no en la selección del tamaño de bloque, además, considerando que en este caso en particular, los cores individuales no están activados. En otras palabras, el tamaño del dispositivo más pequeño es el que define la cantidad de datos que se llevan a RAM y que posteriormente se procesan en GPU.

7.1.3. Hacia el Solapamiento de Procesos entre la CPU y GPU

En los sistemas reales siempre será necesario considerar los accesos a los distintos niveles de memoria del sistema. La existencia de memorias cuyos accesos son más lentos provoca una reducción en el rendimiento de

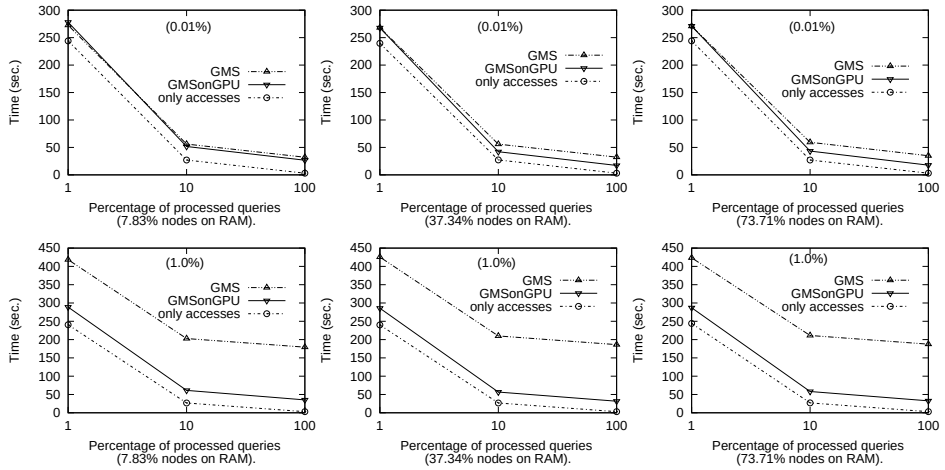


Figura 7.5: Tiempos de ejecución para el espacio de histogramas de colores, considerando rangos de recuperación del 0,01 % (arriba) y 1 % (abajo). Además, se incluyen los procesos de búsqueda para el *GMS*, *GMS* sobre GPU (*GMSonGPU*) y considerando sólo los tiempos de accesos (*only accesses*.

los sistemas y con ello un desaprovechamiento de otros recursos que, por si solos, pueden ser muy rápidos.

En esta sección se ha demostrado que los costos de acceso a memoria secundaria son excesivos. Sin embargo, esto puede ser reducido aprovechando la arquitectura multicore, es decir, se puede acceder a los distintos niveles de la jerarquía de memoria con distintos dispositivos. En este caso, existen algunas restricciones, como por ejemplo que desde GPU no se puede acceder a disco. Sin embargo, esta tarea podría quedar asignada a alguno de los cores o a varios, mientras la GPU puede realizar el proceso de búsqueda. Es decir, es posible solapar el proceso de transferencia de datos desde memoria secundaria con el proceso de búsqueda, y así reducir los tiempos generales del sistema. Los pasos para esto serían:

1. Un CPU-core transfiere datos desde memoria secundaria hacia memoria principal.

2. Un CPU-core transfiere datos desde memoria principal a la memoria global del dispositivo GPU y lanza el código de la GPU (kernel).

3. Ahora, mientras la GPU resuelve el bloque de consultas, un core se encarga de una nueva transferencia de datos hacia memoria principal.

4. Cuando la GPU termina el proceso de búsqueda, el CPU-core encargado de la GPU realiza la transferencia de los resultados desde la GPU a memoria principal. Entonces continúa con el siguiente bloque de consultas si existe disponibilidad de éste.

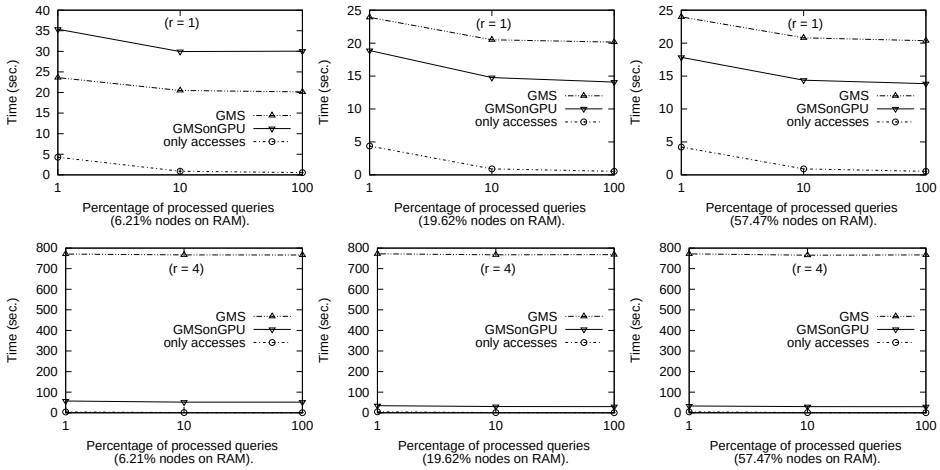


Figura 7.6: Tiempos de ejecución para el espacio de palabras en español, considerando rangos de 1 (arriba) y 4 (abajo). Además, se incluyen los procesos de búsqueda para el *GMS*, *GMS* sobre GPU (*GMSonGPU*) y considerando solo los tiempos de accesos (*only accesses*.

Nótese que la clave es el paso 3, en donde realmente se encuentra solapada la transferencia de datos con la resolución de consultas. Si se observan las figuras 7.5 y 7.6, la línea "*only accesses*" presenta el tiempo gastado en la transferencia de datos. Desde este valor se puede estimar el tiempo de ejecución de una implementación que solape estos procesos. Esta estimación puede ser calculada como un valor próximo a $MAX(only\ accesses, GMSonGPU - only\ accesses)$, esto es, el máximo valor entre el tiempo empleado en la transferencia de datos y el proceso de búsqueda en GPU.

De acuerdo a lo expuesto y a los resultados preliminares presentados en la sección 7.1.1, una implementación que solape estos procesos puede reducir los tiempos entre 18 % y 40 % respecto del espacio de histogramas de colores y alrededor de un 13 % en el caso del espacio de palabras en español.

7.1.4. Discusión

En esta sección, se ha presentado una implementación completa que considera todos los elementos participantes en una aplicación real. Se han desarrollado implementaciones que consideran los costos de accesos a memoria secundaria para las estructuras *GMS* y *LC*, es decir, optimizados para disco. Los resultados experimentales indican que para estas estructuras lo ideal es trabajar con bloques de consultas (llevadas a RAM) de 1 % o más para paliar las pérdidas de rendimiento ocasionado por las lecturas a disco. Con ambas estructuras se logró obtener mejores tiempos que con la estructura

EGNAT, la cual fue diseñada específicamente para memoria secundaria.

Adicionalmente se propone el solapamiento de procesos, dada la independencia de estos, de tal manera que mientras algunos dispositivos se encargan de traer objetos y trozos de la estructura a RAM, otros puedan resolver las consultas.

7.2. Esquemas de Planificación sobre una Plataforma Heterogénea

En el capítulo 6 se presentaron dos implementaciones de sistemas que utilizan toda la plataforma disponible, usualmente una plataforma heterogénea con distintos dispositivos de procesamiento de distintas arquitecturas. En dichas implementaciones no existe mayor información respecto a los diferentes dispositivos y se realiza una serie de experimentos para determinar (o sintonizar) algunos valores adecuados para ciertos parámetros como: Tamaño del bloque de consultas que debe resolver una GPU, tamaño del bloque de consultas considerando las distintas velocidades de los dispositivos de procesamiento, tamaño del *chunk* (o cantidad de bloques de iteraciones asignadas a los distintos cores) en un sistema multicore, entre otros. Sin embargo, si se tiene información previa, por ejemplo, en base a esta misma serie de resultados obtenidos experimentalmente, es posible definir una planificación más adecuada de acuerdo a dichos resultados.

El objetivo de la presente sección no es necesariamente obtener una planificación mejor que la conseguida en la sección 6.4, sino, obtener una visión más profunda del comportamiento de la estructura seleccionada y de los parámetros a considerar para su implementación sobre un sistema heterogéneo, lo que permitirá una base sólida para futuras implementaciones considerando algún otro tipo de dispositivo acelerador.

7.2.1. Análisis de Esquemas de Planificación

Un uso apropiado de los *pragmas* de OpenMP determina el modo en que las consultas se distribuyen en los recursos de procesamiento (cores o GPU) y con ello provocar un gran impacto en las prestaciones.

Algunos posibles esquemas de planificación pueden ser:

1. **Estático I:** Si se utiliza la directiva `#pragma omp parallel for` con planificación **static**, se distribuye la misma cantidad de consultas entre todos los recursos. Claramente, y como se ha explicado en la sección 6.3, esta planificación no es apropiada para una plataforma heterogénea debido al hecho de que las GPUs son dispositivos más rápidos que los cores. Esto significa que los cores podrían representar un cuello de botella de cara a obtener las mejores prestaciones.

2. **Dinámico I:** Si se utiliza la directiva `#pragma omp parallel for` con una política de planificación **dynamic**, ésta corresponde a una distribución a bloques de consultas a un recurso, y una vez un recurso finaliza cogería el siguiente bloque asignable. En este caso, la carga se balancea, pero las prestaciones dependen del tamaño del bloque. Sin embargo, un análisis en profundidad muestra que OpenMP asigna internamente un bloque de consultas pero de una en una. Por tanto, parece que el tamaño del bloque no afecta a las prestaciones, y sin embargo el tiempo perdido en las transferencias puede ser muy alto porque hay tantas transferencias a GPU como consultas. Por otro lado, debemos tener en cuenta que el último bloque puede ser asignado a un core y de nuevo este elemento de proceso representa el cuello de botella del sistema completo (ver secciones 6.3 y 6.4).

3. **Dinámico II:** Se utiliza la directiva `#pragma omp parallel for` con política de planificación **dynamic**, pero en este caso el programador engaña a OpenMP forzando a calcular y transmitir un bloque de consultas completamente. En este caso el tiempo de transferencia se reduce, pero no se logra evitar el efecto del último bloque asignado, es decir, siempre es posible que el último bloque quede a disposición de un dispositivo lento (esquema desarrollado en la versión 2 del sistema heterogéneo, en sección 6.4).

4. **Estático II:** En lugar de utilizar la directiva de OpenMP `#pragma omp parallel for` que crea los hilos y distribuye las consultas manualmente, se utiliza la directiva `#pragma omp parallel`, esto es, sólo se crean los hilos y de manera estática se asigna un porcentaje de consultas a las GPUs y otro porcentaje a los cores. En este caso, sólo se realiza una única transferencia de datos beneficiando las prestaciones. Este esquema estático de asignación manual permite:

   - Primero, manejar los tamaños de bloque sin riesgos de que dicho tamaño sea demasiado grande para un dispositivo muy lento. Esto debido a que este valor sólo es usado por la GPU y no por otro dispositivo.
   - Segundo, no existe un último bloque que contenga menos datos del tamaño definido y que por lo tanto no pueda ser asignado a una GPU. En este caso, siendo la asignación manual no ocurre esto y en el peor de los casos, puede quedar sólo una consulta, la cual puede ser arbitrariamente asignada a cualquier dispositivo. Este punto indica que no es necesario considerar este caso especial, que en la versión 2 del sistema heterogéneo era resuelto utilizando la versión 1 para este último bloque.
   - Tercero, al evitar la existencia de un último bloque, se evita también que pueda ser asignado a un dispositivo más lento. Esto

dado que el número de consultas asignadas a la GPU es mayor que el número de consultas asignadas a los cores, además, de que el valor es especificado para cada tipo de dispositivo.

Este esquema es posible implementarlo si previamente se tiene una relación entre los dispositivos del sistema heteregéneo, por ejemplo, el speed-up. En otras palabras: "si se conocen las velocidades de unos y otros dispositivos o la relación de estas velocidades, siempre es posible deducir un valor inicial de distribución y que además éste sea cercano al óptimo".

Los tres primeros esquemas han sido desarrollados o analizados en el capítulo 6, siendo el más adecuado el **Dinámico II**. Una alternativa a este esquema de planificación es el mencionado como **Estático II**. En la siguiente sección se realiza una discusión sobre los porcentajes de consultas asignadas a GPU y a cores.

7.2.2. Planificación Estático II

En la planificación de tipo **Estático II** se realiza la asignación de un único bloque a cada GPU con el total de consultas asignadas. Para obtener algunas conclusiones, el porcentaje de consultas asignadas a la GPU varía desde 0 % (sólo los cores resuelven las consultas) hasta 100 % (sólo las GPUs resuelven las consultas). Las pruebas se realizarán sobre las bases de datos principales de estudio (espacio de palabras en español e histogramas de colores), pero se agregará adicionalmente un espacio denominado ROSTROS que corresponde a un espacio de alta dimensión, tanto aparente como intrínseca, lo cual amplía la visión del comportamiento del sistema considerando un espacio donde la función de distancia es "pesada", la dimensión aparente es alta (254) y la dimensión intrínseca es sobre 65. Una descripción detallada de este espacio se encuentra en 4.1.

Las Figuras 7.7, 7.8 y 7.9 muestran los resultados obtenidos para los tres casos de estudio. El eje X representa el porcentaje de consultas asignadas a las GPUs, mientras que el eje Y representa el tiempo de ejecución en segundos. En el caso de las GPUs se incluye el tiempo de transferencia de datos.

Se puede observar que:

- Cuando el número de consultas asignadas a las GPUs es menor al 50 %, es más eficiente usar sólo cores. Este resultado significa que el número de consultas asignadas a las GPUs es reducido y por tanto la implementación no saca partido de la potencia computacional de dicho dispositivo y los cores representan el cuello de botella del sistema heterogéneo. Sin embargo, el tiempo de ejecución está lejos de ser el mejor.

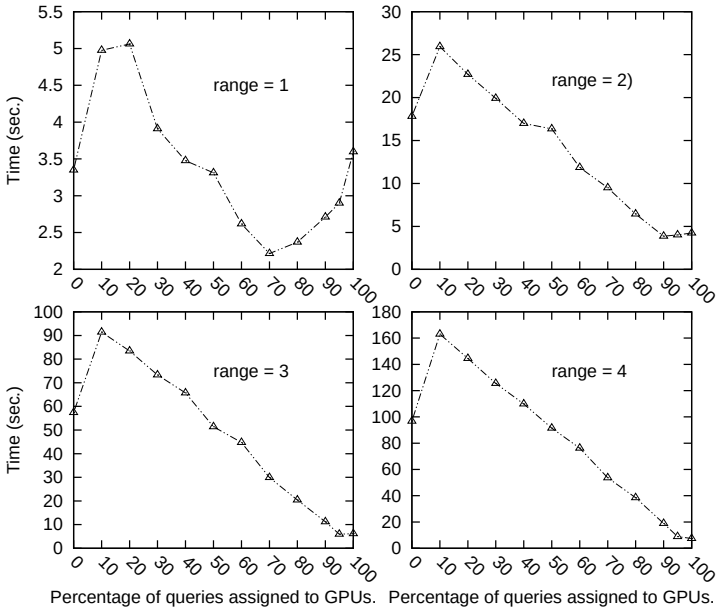


Figura 7.7: Tiempo de ejecución para el caso de diccionario de Español para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**.

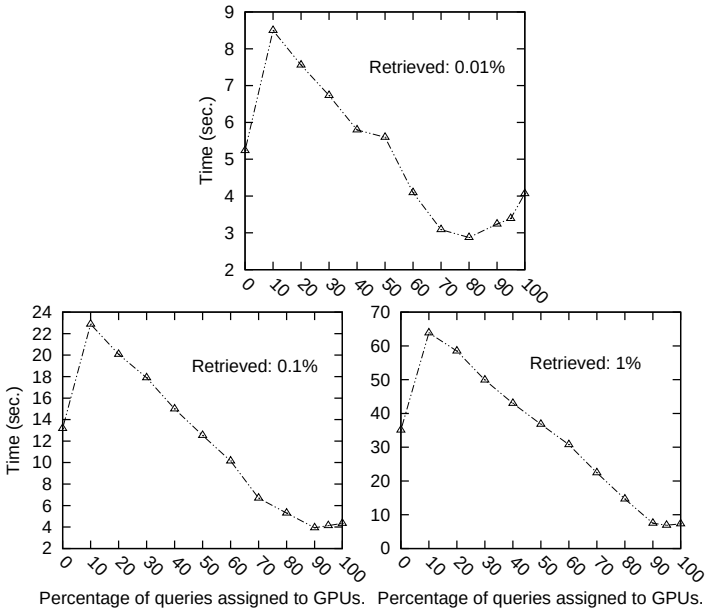


Figura 7.8: Tiempo de ejecución para el caso de histogramas de color para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**.

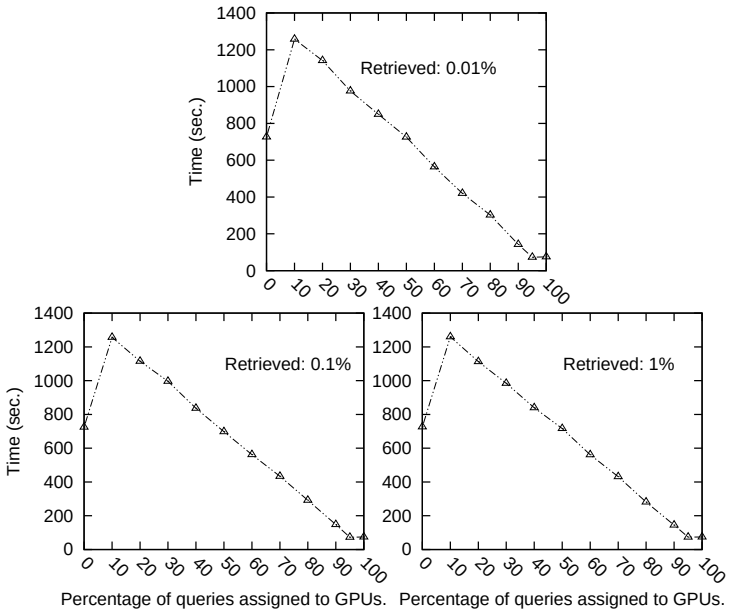


Figura 7.9: Tiempo de ejecución para el caso del espacio de rostros para porcentajes de consultas asignadas a la GPU desde 0 % a 100 %. Planificación **Estático II**.

- El mejor resultado, en términos de tiempo de ejecución, se obtiene cunado el número de consultas asignadas a las GPUs es mayor del 80 %. En este caso, el tiempo de procesamiento empleado por las GPUs y los cores es muy similar, esto es, las GPUs resuelven más consultas que los cores pero empleando aproximadamente el mismo tiempo. En las figuras se puede observar un efecto valle justo en el punto donde se obtienen las mejores prestaciones.

- En los casos de estudio considerados en este trabajo, excepto para el caso de rostros, conforme la complejidad del problema crece (por ejemplo, rango 4 para el diccionario de Español o 1 % para la recuperación de información en el caso de histograma de color) el mejor resultado se obtiene sólo usando GPUs. Esto es debido a que la contribución en el tiempo de ejecución de la utilización de los cores es muy reducida. Sin embargo, si el número de consultas fuese más elevado que el considerado en esta sección, de nuevo aparecería el efecto valle en las figuras 7.7, 7.8 y 7.9.

- En la figura 7.9 podemos apreciar que las prestaciones obtenidas son similares independientemente del porcentaje de recuperación de información, y el efecto valle aparece aproximadamente cerca de un 95 % de consultas asignadas a las GPUs. Este comportamiento se explica

por varios motivos, el primero, que al ser un espacio de alta dimensión las estructuras métricas, sobre todo las basadas en pivotes, bajan mucho el rendimiento, de hecho el rendimiento es muy cercano a un fuerza bruta. Al aumentar el rango o reducir el rango de búsqueda, la tendencia siempre será a tiempos cercanos a la búsqueda exhaustiva. Dicho esto, como se ha comentado en diversas partes de la presente Tesis, cuando mayor partido se saca de las GPUs, es cuando más carga de trabajo existe, por lo que es justificado entonces que las curvas para este espacio muestren una tendencia que favorece el uso de GPUs. Los cores disponibles en los experimentos para el espacio de rostros sólo aportan una mínima parte al rendimiento.

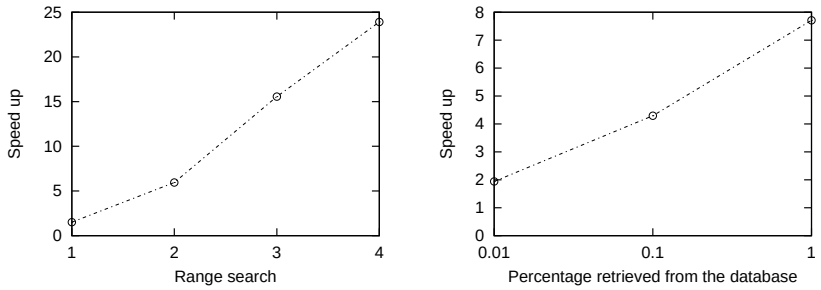
7.2.3. Discusión

Los resultados son coherentes respecto de los presentados en las secciones 6.3 y 6.4. Sin embargo, la idea inicial de la planificación es que dado un speed-up se puede determinar la asignación de recursos. La idea anterior es trivial, pero en esta sección queda demostrado al comparar los resultados de los gráficos de las figuras 7.7, 7.8 y 7.9 con los respectivos speed-ups presentados en la figura 7.10. En esta figura se obtiene la relación entre las velocidades de los dos dispositivos del sistema heterogéneo, core versus GPU (1 core versus 1 GPU). Es decir, entre la ejecución secuencial sobre un core y la ejecución en paralelo sobre una GPU.

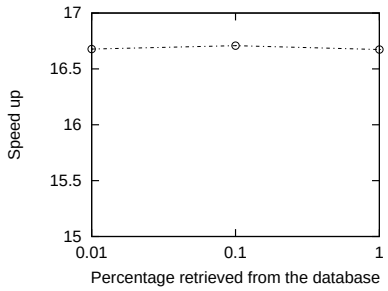
Se puede observar entonces, que para el caso del espacio de palabras en español y rango 1, la ganancia de velocidad es cercana a 2, lo que significa que se debe asignar al menos el doble de consultas a las GPUs que a los cores. De hecho, el mejor porcentaje de asignación de consultas para el rango 1 es alrededor de 70 % para GPUs y el resto para cores. Cuando el rango es 2, la ganancia de velocidad está cercana a 6. Por tanto se debe asignar 6 veces más consultas a las GPUs que a los cores. Para este caso, las mejores prestaciones se obtienen para porcentajes de consultas asignadas a GPUs cercanos a 84 %. De hecho, los mejores resultados se han obtenido para porcentajes de consultas asignadas alrededor de 90 %. Un razonamiento similar se puede hacer para el caso de histogramas de color.

Sin embargo, para el caso de estudio del espacio de rostros, y debido a su peculiar comportamiento, es evidente que la ganancia de velocidad es siempre la misma. Se puede observar en la Figura 7.10(c) que el speed-up obtenido es cercano a 16. Esto significa que debemos asignar 16 veces más consultas a las GPUs que a los cores. Siguiendo un razonamiento similar al planteado en el párrafo anterior, las mejores prestaciones se obtendrían para un porcentaje de consulas asignadas a GPUs cercanas a un 95 %. Precisamente, los resultados que se presentan en la Figura 7.9.

Finalmente se puede decir, que los rendimientos de esta planificación son similares a los obtenidos en la sección 6.4 que corresponden a la planificación



(a) Diccionario Español. (b) Histogramas de color.



(c) Rostros.

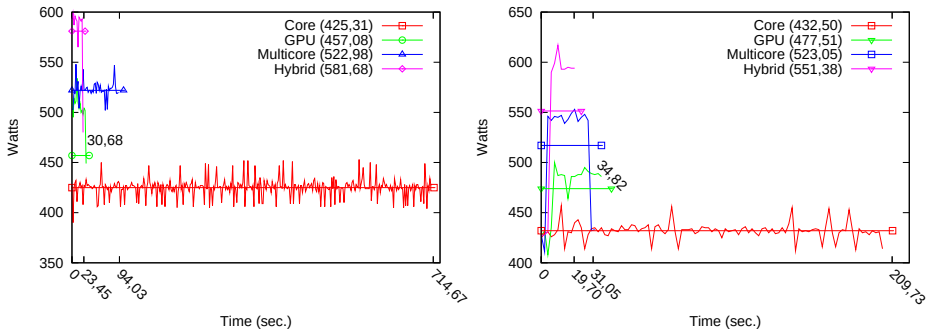
Figura 7.10: Speed-ups para el sistema heterogéneo.

utilizada para la versión 2 del sistema heterogéneo, equivalente a una planificación del tipo **Dinámico II**. La diferencias entre ambos esquemas pueden ser de un 0,79 % y hasta un 2,95 % a favor del esquema **Estático II**, pero sólo para los rangos mayores. Esta leve mejora es debida a la disminución de transferencias entre uno y otro esquema.

7.3. Consumo Energético

Hoy en día hay un gran interés en realizar implementaciones eficientes tanto computacionalmente (menos tiempo de ejecución) como de consumo de energía. De hecho, hay una tendencia global hacia la computación a Exaescala (*Exascale computing*), un nivel de cómputo que será viable gracias al paralelismo y al escalado de la tecnología, pero siempre y cuando se reduzcan los niveles de potencia eléctrica consumida [56].

En el desarrollo de la presente Tesis se ha considerado en las distintas etapas el análisis del consumo energético requerido por las distintas implementaciones propuestas. En esta sección se resume, en base a dos ejemplos presentados en los distintos capítulos, la relación del consumo eléctrico de las distintas arquitecturas de la plataforma seleccionada. Estos resultados se presentan como una medida adicional a las medidas de tiempo de ejecución y ganancia de velocidad presentadas previamente. La potencia consumida



(a) Diccionario Español. (b) Histogramas de colores.

Figura 7.11: Estudio de potencia consumida en watts para el caso de diccionario de Español e histogramas de color para la prueba presentada en 6.3 con plataformas: 1 core, 1 GPU, multicore (8 cores), heterogénea (híbrida de 8 cores con 1 GPU).

se mide en términos de vatios (watts) consumidos por la plataforma completa, y en julios (*joules*) cuando se considera el consumo de energía. Las medidas se han realizado utilizando el analizador de potencia PZ4000 de YOKOGAWA [94].

Se seleccionaron aquellas dos pruebas experimentales más representativas y que incluyeron la mayoría de los dispositivos utilizados en la presente Tesis. De esta manera se pretende generar discusión en base al uso de los distintos recursos provistos por la arquitectura subyacente.

7.3.1. Prueba Experimental 1

El primer experimento incluye el uso de un core, es decir, de una implementación secuencial y permite compararla con una implementación con sólo una GPU, un sistema multicore y un sistema híbrido (heterogéneo). El test fue presentado en el capítulo 6, en la sección 6.3 y corresponde a la primera versión del sistema heterogéneo cuando el sistema incluye sólo una GPU. Los gráficos correspondientes a las tomas de tiempo de estos experimentos están en las figuras 6.4, 6.5 y 6.6.

Para este primer experimento los resultados se muestran en la figura 7.11. Se observa como el introducir una GPU produce un aumento del consumo de energía, pero por otro lado implica una reducción drástica del tiempo de ejecución. En la figura 7.11 se muestran los resultados para los valores extremos, es decir, de $r = 4$ para el espacio de palabras en español y recuperación de 1 % para los histogramas de colores. Cada línea en la gráfica representa a una plataforma y el consumo en watts (potencia) en los intervalos de toma de muestras durante la ejecución del programa. Entre paréntesis, a cada plataforma se le agrega el promedio de la potencia consumida.

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

Tabla 7.3: Consumo de energía para el caso de estudio de diccionario de Español de la figura 7.11(a).

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

Tabla 7.4: Consumo de energía para el caso de estudio de histograms de color de la figura 7.11(b).

Este primer experimento, (en su presentación inicial en la seccion 6.3) tenía como objetivo reducir los tiempos de procesamiento de consulta, por lo que es válido preguntarse si el usar una arquitectura paralela conlleva costos adicionales, en este caso de energía. Las tablas 7.3 y 7.4, relacionan la potencia (*watts*) utilizada por el sistema con el tiempo, es decir, la energía total utilizada por el sistema, medida en *joules*. En particular, la tabla 7.3 presenta los resultados en *joules* para el caso de estudio del diccionario de español, mientras que la tabla 7.4 presenta los resultados para el caso de estudio de histograma de color. Se debe indicar que los *joules* se han calculado como el promedio del consumo de potencia multiplicado por el tiempo invertido por la implementación. Además, se agrega una última columna que muestra el ahorro de consumo entre las diferentes plataformas y la secuencial (1 core).

En general, se observa que al introducir una plataforma paralela, bien los multicores, una GPU o híbrido, el consumo de potencia, evidentemente, es mayor que el secuencial. No obstante, al reducir drásticamente el tiempo de ejecución, los *joules* también se reducen drásticamente. Así pues, para el caso de estudio de diccionario en español, la reducción de consumo energético para el caso de la implementación heterogénea (*hybrid* en el gráfico), es de más del 90 %, mientras que para el caso de los histogramas de colores la reducción es cercana al 88 %. Por tanto se desprende que si bien el consumo energético puntual es mayor al emplear el paralelismo (es evidente que al emplear más recursos se consume más energía), al estar activos durante mucho menos tiempo, el consumo global es radicalmente menor.

Plataforma	Tiempo	Watts	Joules
8 cores	17, 82	494, 82	8.817,69
4 GPUs	4, 23	648, 55	2.743,37
Heterogénea	3, 87	746, 20	2.887,79

Tabla 7.5: Consumo de energía para el caso de estudio de diccionario de Español de la sección 7.2 (figura 7.12(a)).

7.3.2. Prueba Experimental 2

El segundo experimento es precisamente el presentado en la sección anterior (7.2), la cual es una muestra que incluye todos los dispositivos utilizados en todos los experimentos y usando la máxima potencia del sistema heterogéneo.

Para esta prueba experimental se han considerando las siguientes características:

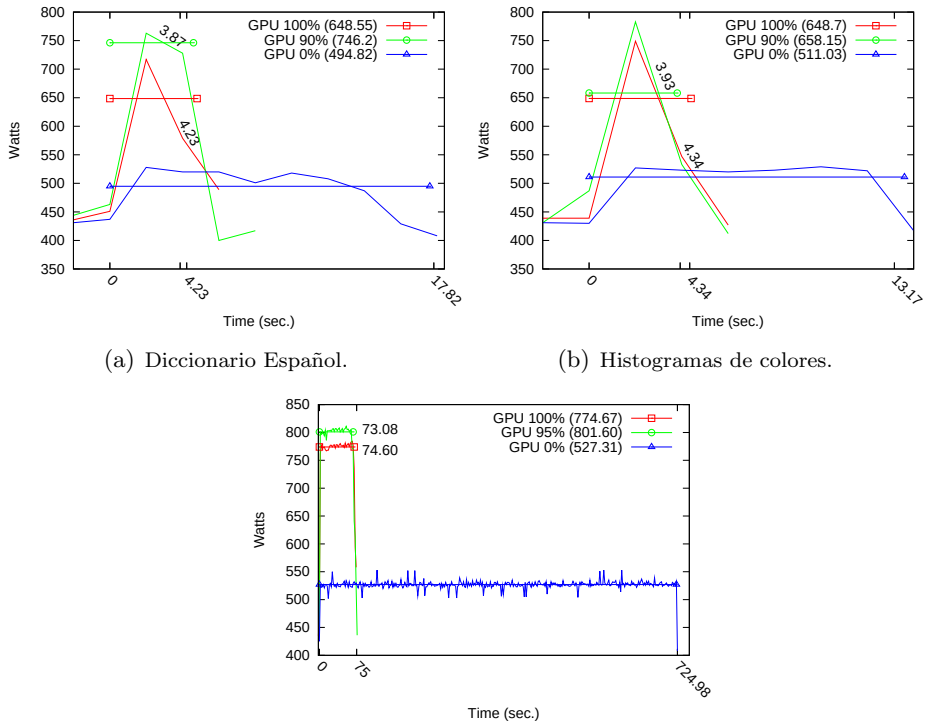
1. Sería deseable conocer la potencia consumida sólo por los cores. En este caso, podemos estudiar el impacto de introducir las GPUs.

2. Además, sería deseable conocer si los cores son o no necesarios, teniendo sólo GPUs.

3. Finalmente, comparar los resultados previos con la plataforma heterogénea. Al realizar este estudio, el número de pruebas se acota ya que hay que buscar unos rangos de valores en las pruebas donde se produzca la anterior casuística. Por tanto, para el caso de diccionario de Español se ha considerado rango 2 que proporciona un mejor resultado cuando las GPUs procesan el 70 % de las consultas. Por el mismo razonamiento, para el caso de histogramas de color se utilizará 0,1 % de información recuperada y para el caso de caras el 0,1 % de información recuperada. Esto también permite ampliar el punto de vista en relación a la prueba experimental 1 de consumo energético (prueba realizada para rangos mayores).

La Figura 7.12 muestra la potencia consumida en *watts* para los tres casos de estudio del experimento mencionado, considerando 8 cores (GPU 0 %), 4 GPUs (GPU 100 %) y la plataforma heterogénea (4 cores + 4 GPUs). En dicha figura se puede observar que introducir una GPU implica un crecimiento en la potencia consumida, pero por otro lado, hay una ligera disminución del tiempo de ejecución.

Las Tablas 7.5, 7.6 y 7.7, relativas a la potencia consumida (*watts*) frente a tiempo de ejecución (segundos), se presentan los *joules* para los tres casos de estudio considerados.



(a) Diccionario Español.

(b) Histogramas de colores.

(c) Espacio de rostros.

Figura 7.12: Estudio de potencia consumida en watts para el caso de diccionario de Español, histogramas de color y rostros, considerando 8 cores (GPU 0 %), 4 GPUs (GPU 100 %) y la plataforma heterogénea (4 cores + 4 GPUs) al 90 % de consultas asignadas a las GPUs.

En estas tablas podemos observar que los *watts* consumidos por las GPUs o la plataforma heterogénea son muy similares, sin embargo necesitan entre 3 y 7 veces menos energía que con respecto al caso de sólo cores. Evidentemente, hay un gran impacto en la potencia consumida cuando se introducen las GPUs, pero también hay que considerar que el problema se resuelve en menos tiempo y que por tanto la energía global consumida se reduce considerablemente.

7.3.3. Discusión

En esta sección se ha presentado un análisis experimental desde una perspectiva del consumo energético. Este análisis está restringido al tipo de procesos desarrollados en la presente Tesis y no tiene por objetivo definir la bondad o perversidad de uno u otro dispositivo en términos del consumo y duración de los procesos. La idea es generar discusión mediante la evaluación de los experimentos aquí mostrados, además, teniendo en cuenta que de

Plataforma	Tiempo	Watts	Joules
8 cores	13, 17	511, 03	6.730,26
4 GPUs	4, 34	648, 70	2.815,36
Heterogénea	3, 93	658, 15	2.586,53

Tabla 7.6: Consumo de energía para el caso de estudio de histograms de color de la sección 7.2 (figura 7.12(b)).

Plataforma	Tiempo	Watts	Joules
8 cores	724, 98	527, 31	382.289,21
4 GPUs	74, 60	774, 07	57.745,62
Heterogénea	73, 08	801, 60	58.580,93

Tabla 7.7: Consumo de energía para el caso de estudio del espacio de rostros la sección 7.2 (figura 7.12(c)).

antemano los dispositvos del tipo GPU son considerados de alto consumo energético.

En esta sección se ha mostrado, efectivamente, que las GPUs tienen un consumo de potencia elevado respecto de los cores o núcleos de procesamiento de las CPUs. Entonces, la pregunta podría haber sido ¿Qué tanto se está dispuesto a sacrificar en función de la reducción del tiempo de procesamiento?, es decir ¿Cuánta energía?. Para el caso específico de la búsqueda por similitud en espacios métricos utilizando estructuras basadas en pivotes, no fue necesario responderla. Los experimentos demostraron claramente que la energía utilizada de los distintos sistemas que utilizan GPUs, si bien eran más elevadas (en término de potencia, *watts*), eran totalmente compensados con una reducción del tiempo de procesamiento, tanto es así, que considerando el proceso completo (finalizado) resultó ser mucho más eficiente energéticamente.

Capítulo 8

Conclusiones

8.1. Discusión Final

Durante las diferentes etapas de este trabajo de Tesis, la mayoría de ellas presentadas en esta memoria, se ha demostrado a través de diferentes implementaciones que, efectivamente, las unidades de procesamiento gráfico son un gran apoyo a las tareas de procesamiento masivo de datos.

En el caso particular de las búsquedas por similitud, los resultados siempre dependerán de un conjunto de factores como son: la complejidad de la función de distancia, la dimensión representacional o aparente, el tipo de estructura utilizada y otros más. Adicionalmente, los factores aumentan al considerar la arquitectura de hardware.

Desde el punto de vista de la arquitectura, las GPUs proveen de una enorme capacidad de cómputo dado su alto grado de paralelismo a nivel de datos. Por otro lado, la utilización de estructuras métricas mejora el rendimiento de la búsqueda tanto en un entorno secuencial como paralelo. Reuniendo estos dos elementos, se obtiene un gran potencial de resolución de búsquedas por similitud.

Tras un estudio inicial sobre diferentes estructuras métricas, expuesto en la sección 5.1, se consideró que una estructura adecuada para su implementación sobre una plataforma basada en GPU era una versión genérica de una estructura basada en pivotes, la cual se denominó *GMS* (*Generic Metric Structure*) para identificarla dentro del conjunto. En los principales casos de prueba utilizados en la Tesis, espacio de palabras en español (diccionario español) y espacio de histogramas de colores (espacio de vectores), se obtuvieron considerables mejoras respecto de la estructura de mejor comportamiento secuencial (para los rangos menores, *Spaghettis*).

Cada una de las versiones presentadas mejora a las anteriores, es así como en las primeras versiones de las implementaciones sobre una GPU de la estructura *GMS* versus *Spaghettis* secuencial, alcanzaron mejoras de 1,9x y 20x para el menor y mayor rango de búsqueda en el caso del espacio de

palabras y entre un 1,3x y 5,3x para el menor y mayor rango de recuperación en el caso del espacio de vectores. Con las últimas versiones sobre una GPU, estos valores llegaron a 24,52x y 6,82x en los rangos mayores para el espacio de palabras y de vectores respectivamente.

Claramente, al incluir más unidades de procesamiento gráfico las mejoras aumentaron llegando a 67,2x y 16,5x para los mismos casos anteriores (mayores rangos en el espacio de palabras y espacio de vectores).

Finalmente se incluyeron las unidades de procesamiento de la CPU, los cores o núcleos, para aportar en los procesos de búsquedas, convirtiendo así al sistema en un sistema realmente heterogéneo que utiliza todos los recursos de la plataforma subyacente. Los speed-ups conseguidos para la mejor alternativa implementada sobre la plataforma heterogénea están entre 10,7x y 96,44x para el menor y mayor rango en el caso del diccionario español y entre 7,72x y 25,53x para el caso de los histogramas de colores.

Los mayores speed-ups fueron obtenidos para los rangos más altos en los diferentes espacios, que es donde más esfuerzo se requiere para resolver una consulta por rango. Es aquí donde más beneficios se obtienen de los dispositivos gráficos.

Durante el desarrollo de las versiones heterogéneas fue relevante considerar los esquemas de planificación más adecuados acorde al tipo de dispositivos utilizados y a la diferencia de velocidades en el procesamiento de consultas y así evitar que aquellos dispositivos más lentos perjudicaran el rendimiento del sistema global.

Además, sin lugar a dudas, una forma de determinar las ganancias obtenidas por uno u otro sistema es verlas en términos absolutos, por ejemplo, en consultas resueltas en una unidad de tiempo. En este aspecto la tabla 6.2 de la sección 6.5 muestra las diferencias entre las distintas versiones en consultas por segundo. En este caso la versión heterogénea (segunda versión) llega a resolver hasta 97 veces más consultas que la mejor versión secuencial en el espacio de palabras y hasta 36 veces más en el espacio de vectores.

Considerando que dado los requerimientos de plataformas como la GPU, no es posible implementar sobre ellas las mejores versiones secuenciales de estructuras métricas. Entonces, sería factible, si se disponen de los recursos de memoria, que un sistema heterogéneo pueda soportar distintas estructuras, por ejemplo, una para cargar sobre las GPUs y otra para utilizarla con los cores de las CPUs, es decir, "lo mejor de los dos mundos".

Al finalizar la Tesis es importante mencionar que las GPUs no sólo representan una excelente alternativa para su utilización en este tipo de problemas, sino que además, representan una alternativa adecuada, también, desde la perspectiva del consumo energético, el cual es un factor no siempre considerado. En este aspecto el uso de GPUs en un sistema de recuperación de información puede llegar a reducir la energía a valores cercanos al 60 % y en los mejores casos hasta un 90 % versus otras plataformas, dependiendo del tipo de búsqueda y de la plataforma contra la que se compara. De este

modo, se compensa el uso de un dispositivo de elevado consumo, como la GPU, frente a otros como es el caso de la CPUs.

8.2. Aspectos Relevantes y Aportes

El objetivo principal del presente trabajo de Tesis era incluir a las GPUs como un componente en el procesamiento de consultas durante las búsquedas por rango en sistemas de recuperación de información y obtener reducciones en los tiempos de procesamiento. Por lo anterior, se considera que el principal aporte de esta Tesis es:

- Presentar una serie de estrategias eficientes para el procesamiento paralelo de consultas por rango sobre plataformas heterogéneas con componentes basados en GPU.

El aporte principal de la Tesis está basado en un conjunto de resultados y aportes interesantes que resuelven diferentes problemas y que abarcan aspectos concretos de la implementación de estructuras métricas sobre plataformas basadas en GPU.

En relación a las estructuras métricas:

- Análisis y evaluación de un conjunto de estructuras para determinar la o las más adecuadas para su implementación sobre una plataforma basada en GPU.
- Evaluación desde el punto de vista de los accesos a memoria secundaria y su repercusión sobre el sistema global heterogéneo y los efectos de particionar todos los componentes del sistema de recuperación antes de subir a la GPU: estructura métrica, base de datos y consultas.
- Consideración de la mayoría de los niveles presentes en la jerarquía de memoria de la plataforma heterogénea, desde la memoria secundaria hasta la memoria global y la memoria compartida de los dispositivos gráficos.

Relativo a GPUs y sistemas heterogéneos:

- Diseño e implementación de algoritmos de búsqueda para distintas estructuras métricas sobre plataformas basadas en GPU y multi-GPUs.
- Utilización de las características propias de la arquitectura para optimizar las búsquedas, como coalescencia y el uso adecuado de los diferentes niveles en la jerarquía de memoria.

- Desarrollo de versiones heterogéneas con distintas visiones sobre la resolución de consultas sobre GPUs.
- Implementación y evaluación de distintas alternativas de estrategias de planificación para distribución de consultas sobre plataformas heterogéneas.

Relativo a temas generales:

- Análisis y evaluación experimental del consumo de energía para los distintos sistemas propuestos durante el desarrollo del trabajo.
- Abrir distintos puntos de discusión que van desde el consumo energético de los dispositivos gráficos y de su utilización en sistemas recuperadores de información a la relevancia de las dimensiones representacionales e intrínsecas de los espacios, así como la complejidad de la función de distancia de cara a la implementación de estructuras sobre plataformas con dispositivos de gran capacidad de procesamiento.

Finalmente y no menos importante, es mencionar que el presente trabajo de Tesis Doctoral permite obtener una visión más profunda y amplia de los diferentes parámetros a considerar en la implementación de algoritmos y estructuras métricas sobre plataformas heterogéneas considerando los distintos componentes que en ella puede existir, como en este caso son CPU multicore y GPUs.

8.3. Trabajos Futuros

Una serie de propuestas quedan para desarrollo futuro, entre ellas.

Relativo a estructuras métricas y GPUs:

- Analizar y evaluar el rendimiento de estructuras de tipo arreglo, como es el caso de la *Lista de Cluster* y otras de rendimiento similar, con diferentes esquemas de implementación sobre la memoria global de una GPU.
- Analizar y evaluar la posibilidad de implementar estructuras métricas basadas en clustering y de tipo árbol sobre plataformas basadas en GPU.
- Analizar en profundidad las implicaciones y los efectos de los espacios de alta dimensión en estructuras implementadas sobre dispositivos basados en GPUs.

En el contexto general de sistemas heterogéneos:

- Desarrollo de algoritmos para inserción y eliminación eficiente sobre estructuras soportadas por plataformas heterogéneas.
- Proponer y analizar nuevas alternativas a los esquemas de distribución de consultas sobre plataformas heterogéneas.
- Generar en base a los esquemas planteados en la Tesis, estática y dinámica, la posibilidad de una sintonización automática en la asignación de los porcentajes de consultas enviadas a uno u otro tipo de dispositivo.

Sobre otros modelos, otros dispositivos y otras aplicaciones:

- Evaluar las diferencias de rendimiento de las estructuras al utilizar otras herramientas y entornos de programación diferentes de CUDA, como OpenCL, OpenACC y otros basados en nuevos modelos de programación como MPI/OmpSs.
- Evaluar las estructuras y estrategias de distribución sobre plataformas heterogéneas con otros aceleradores, como son las FPGAs.
- Adaptación y aplicación de las soluciones propuestas en la tesis sobre plataformas heterogéneas para resolución de problemas específicos, como por ejemplo, de tecnologías ómicas y del área biológica (genómica, proteómica), y su comparación con soluciones existentes.

8.4. Publicaciones

Durante el periodo de estancia en el doctorado en la Universidad de Castilla-La Mancha, la difusión de los resultados del trabajo de investigación desarrollado se realizó mediante una serie de publicaciones científicas, todas detalladas a continuación.

Conferencias Internacionales de relevancia en las áreas de Bases de Datos y Paralelismo:

1. Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, Diego Cazorla, Pedro Valero-Lara. “Improving the Performance for the Range Search on Metric Spaces using a Multi-GPU Platform”. In 23rd International Conference on Database and Expert Systems Applications (DEXA 2012), LNCS volume 7447, pages 442-449, Springer Berlin Heidelberg, Vienna, Austria, Sept. 2012. (Core B, indexada en Scopus y DBLP)
2. Roberto Uribe-Paredes, Pedro Valero-Lara, Enrique Arias, José L. Sánchez, Diego Cazorla. “Similarity Search Implementations for Multi-core and Many-core Processors”. In International Conference on High

Performance Computing and Simulation (HPCS 2011), Workshop on Exploitation of Hardware Accelerators, pages 656-663, Istanbul, Turkey. July 2011. (Core B, indexada en Scopus y DBLP)

3. Roberto Uribe-Paredes, Pedro Valero-Lara, Enrique Arias, José L. Sánchez, Diego Cazorla. "A GPU-Based Implementation for Range Queries on Spaghettis Data Structure". In International Conference on Computational Science and Its Applications (ICCSA 2011), LNCS volume 6782, pages 615-629, Santander, Spain, June 2011. (Core C, indexada en Scopus y DBLP)

   Otras Conferencias Internacionales:

4. Roberto Uribe-Paredes, Diego Cazorla, Enrique Arias, José Luis Sánchez. "An approach to an Efficient Scheduling Scheme for Delivering Queries to Heterogenous Clusters in the Similarity Search Problem". In 13th International Conference Computational and Mathematical Methods in Science and Engineering. Almería, Spain, June 2013.

5. Roberto Uribe-Paredes, Enrique Arias, Diego Cazorla, José L. Sánchez. "Analysing Metric Data Structures Thinking of an Efficient GPU Implementation". IAENG Transactions on Engineering Technologies, Lecture Notes in Electrical Engineering 229, pages 81-91, 2013. Springer Netherlands. (Chapter book)

6. Roberto Uribe-Paredes, Diego Cazorla, José L. Sánchez, Enrique Arias. "A comparative study of different Metric Structures: Thinking on GPU implementations". The 2012 International Conference of Computational Statistics and Data Engineering (ICCSDE'12), part of The World Congress on Engineering (WCE2012). London, England, July 2012.

   Conferencias Nacionales:

7. Roberto Uribe-Paredes, Diego Cazorla, Enrique Arias, José Luis Sánchez. "Estudio de Diferentes Esquemas de Planificación para el Reparto de Consultas en una Plataforma Heterogénea". Actas XXIV Jornadas de Paralelismo (JPAR2013). Madrid, España, Sept. 2013.

8. Roberto Uribe-Paredes, Diego Cazorla, Enrique Arias, José Luis Sánchez. "Acelerando la Búsqueda por Rango con un Sistema Híbrido de Memoria Compartida". Actas XXIII Jornadas de Paralelismo (JP2012). Sept. 2012, Elche España.

9. Roberto Uribe-Paredes, Enrique Arias, Diego Cazorla, José Luis Sánchez. "Una estructura Métrica Genérica para Búsquedas por Rango sobre una Plataforma Multi-GPU". XVII Jornadas de Ingeniería

del Software y Bases de Datos (JISBD2012). Almería, España, Sept. 2012

10. Roberto Uribe-Paredes, Pedro Valero-Lara, Enrique Arias, José L. Sánchez, Diego Cazorla. "Búsquedas por Similitud en Espacios Métricos sobre Plataformas Basadas en GPUs". En XXII Jornadas de Paralelismo, La Laguna, España, Septiembre 2011.

Reportes Técnicos:

1. Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, and Diego Cazorla. "Metric Data Structures Suppported by Heterogeneous Systems". Computing Systems Dept, University of Castilla-La Mancha, Spain, Technical Report, May 2013. (UCLM - DIAB-13-05-2)

2. Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, and Diego Cazorla. "GPU-based implementation for range queries on spaghettis data structures". Computing Systems Dept, University of Castilla-La Mancha, Spain, Technical Report, December 2010. (UCLM - DIAB-10-12-1)

Una descripción general de los artículos es la siguiente:

- Los artículos 3 y 10 corresponden a una muestra de las primeras soluciones presentadas para estructuras métricas sobre plataformas basadas en GPU, que incluye la determinación y comparación entre las estructuras con diferentes números de pivotes y versus búsqueda exhaustiva, además, de las primeras comparaciones en términos de consumo energético. El artículo 2 corresponde a una profundización y a la acomparación con otras plataformas, como es el caso multicore. Estos tres artículos incluyen parte del trabajo presentado en el reporte técnico número 1.

- Los artículos 5 y 6 corresponden a los experimentos y comparativas en el contexto secuencial, orientadas hacia la búsqueda de mejores alternativas para implementar sobre GPUs. Estos corresponden a lo presentado en la sección 5.1 (el trabajo 5 es la versión capítulo de libro del artículo 6).

- Los artículos 1 y 9 corresponden a desarrollos y optimizaciones de búsquedas sobre estructuras métricas sobre plataformas basadas en GPUs y multi-GPU. Las implementaciones presentada en estos trabajos correspondieron a tres kernels.

- Los artículos 4 y 7 desarrollan lo presentado en la sección 7.2 analizando distintos esquemas de planificación sobre plataformas heterogéneas.

- El artículo 8 presenta una primera versión de un sistema híbrido de memoria compartida (sistema heterogéneo).

Artículos recientemente enviados y en proceso de evaluación:

1. Roberto Uribe-Paredes, Enrique Arias, Diego Cazorla, José Luis Sánchez. "Metric Data Structures Supported by Heterogeneous System". INFORMATICA, Vilnius University, Institute of Mathematics and Informatics. Lithuania. (ISI Journal, factor de impacto JCR 2011 1.627, Q1: 30/135; Q1: 18/245)

2. Roberto Uribe-Paredes, Diego Cazorla, Enrique Arias, José Luis Sánchez. "A Multicore/GPU heterogeneous system to accelerate the similarity search over metric structures". INGENIARE, Universidad de Tarapacá, Chile. (Scielo Journal) (En español)

3. Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, Diego Cazorla. "From secondary memory to GPU: Towards overlapping CPU and GPU computing". 6th International Conference on Similarity Search and Applications (SISAP). A Coruña, Spain, 2013.

4. Roberto Uribe-Paredes, José L. Sánchez, Diego Cazorla, Enrique Arias. "Optimizing a pivot-based algorithm for similarity search algorithm on a GPU-based platform: A qualitive approach". International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar'2013). Aachen, Germany, 2013.

Respecto de los artículos presentados y en estado de revisión, se puede decir que:

- El artículo 1 presenta gran parte de lo desarrollado en el capítulo 6 de la Tesis, es decir, una completa y detallada descripción de un sistema de búsqueda por similitud sobre una plataforma heterogénea. Este artículo corresponde también a lo presentado en el reporte técnico 2.

- El artículo número 2 presenta una versión preliminar de un sistema heterogéneo con dos kernels como núcleo central para procesar las consultas que llegan al sistema.

- El artículo 3 presenta parte de lo desarrollado en el capítulo 7.1.

- El artículo 4 presenta distintas optimizaciones y comparaciones de kernels. Este trabajo muestra como las diferentes propiedades y características de las actuales GPUs son utilizadas para obtener versiones optimizadas de la estructura genérica utilizada en la Tesis. El contenido de este artículo no alcanzó a ser incluido en la Tesis.

Artículos publicados con otros grupos de investigación en el área de espacios métricos, durante la estancia en el doctorado:

1. Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. Journal of Information Systems, Volume 36, issue 4, pp. 734-747, June, 2011. (ISI Journal)

2. E. Peña-Jaramillo, R. Ovando y R. Uribe-Paredes. Recuperación de Imágenes Basadas en Contenido Sobre Estructuras Métricas. Jornadas Chilenas de Computación (JCC2010). Antofagasta, Chile, Noviembre 2010. (Conferencia Nacional)

Referencias

[1] GPGPU. general-purpose computation using graphics hardware. http://www.gpgpu.org.

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

[3] Adil Alpkocak, Taner Danisman, and Ulker Tuba. A parallel similarity search in high dimensional metric space using m-tree. In *Advanced Environments, Tools, and Applications for Cluster Computing*, volume 2326 of *LNCS*, pages 247–252. Springer Berlin / Heidelberg, 2002.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[5] Ricardo Baeza-Yates. Searching: An algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[6] Ricardo Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *5th Combinatorial Pattern Matching (CPM'94)*, volume 807 of *LNCS*, pages 198–212. Springer Berlin Heidelberg, 1994.

[7] Ricardo Baeza-Yates and Gonzalo Navarro. Fast approximate string matching in a dictionary. In *5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.

[8] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto, and Mauricio Marín. kNN query processing in metric spaces using GPUs. In *17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*, volume 6852 of *LNCS*, pages 380–392, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] Ricardo J. Barrientos, José I. Gómez, Christian Tenllado, Manuel Prieto, and Mauricio Marín. Range query processing in a multi-GPU environment. In *10th IEEE International Symposium on Parallel and*

Distributed Processing with Applications (ISPA 2012), pages 419–426, Madrid, Spain, July 2012.

[10] J. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM*, pages 18(9):509–517, 1975.

[11] J. Bentley. Multidimensional binary search trees in database applications. In *IEEE Trans. on Software Engineering*, pages 5(4):333–340, 1979.

[12] S. Berchtold, D. Keim, and H. Kriegel. The x-tree: an index structure for high-dimensional data. In *Proc. 22nd Conference on Very Large Database (VLDB'96)*, pages 28–39, 1996.

[13] Stefan Berchtold, Christian Böhm, and Bernhard Braunmüller. Fast parallel similarity search in multimedia databases. In *In Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, 1997.

[14] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627, 2009.

[15] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *ACM Conference on Management of Data (ACM SIGMOD'97)*, Sigmod Record 26(2), pages 357–368, 1997.

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

[17] Nieves Brisaboa, Oscar Pedreira, Diego Seco, Roberto Solar, and Roberto Uribe-Paredes. Clustering-based similarity search in metric spaces with sparse spatial centers. In *34rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, volume 4910 of *LNCS*, Nový Smokovec, Slovakia, 2008. Springer.

[18] Luis Britos, A. Marcela Printista, and Nora Reyes. Dsacl+-tree: A dynamic data structure for similarity search in secondary memory. In *5th International Conference on Similarity Search and Applications (SISAP)*, pages 116–131, Toronto, Canada, August 2012.

[19] W. Burkhard and R. Keller. Some approaches to best-match file searching. *Communication of ACM*, 16(4):230–236, 1973.

[20] 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.

[21] Benjamín Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot selection techniques for proximity search in metric spaces. In *XXI Conference of the Chilean Computer Science Society*, pages 33–40. SCCC, IEEE Computer Science Press, 2001.

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

[23] 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.

[24] Edgar Chávez, José L. Marroquín, and Ricardo 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.

[25] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, June 2001.

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

[27] B. Chazelle. Computational geometry: A retrospective. In *26th ACM Symposium on the Theory of Computing (STOC'94)*, pages 75–94, 1994.

[28] T. Chiuch. Content-based image indexing. In *The 20th Conference on Very Large Databases (VLDB'94)*, pages 582–593, 1994.

[29] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree : An efficient access method for similarity search in metric spaces. In *the 23st International Conference on VLDB*, pages 426–435, 1997.

[30] E. F. Codd. A relational model of data for large shared data bank. *Communications of ACM*, 13(6):377–387, June 1970.

[31] Arturo Crespo and Hector Garcia-molina. Routing indices for peer-to-peer systems. 2002.

[32] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[33] Veronica Gil de Costa. *Consultas sobre Espacios Métricos en Paralelo.* PhD thesis, Universidad Nacional de San Luis, San Luis, Argentina, Junio 2009.

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

[35] Christos Doulkeridis, Akrivi Vlachou, Yannis Kotidis, and Michalis Vazirgiannis. Peer-to-peer similarity search in metric spaces. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 986–997. VLDB Endowment, 2007.

[36] Zhe F., Feng Q., Arie K., and S. Yoakum-Stover. Gpu cluster for high performance computing. In *ACM / IEEE Supercomputing Conference 2004*, Pittsburgh, PA, USA, November 06–12 2004.

[37] K. Figueroa, G. Navarro, and E. Chávez. Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

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

[39] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[40] V. Gil-Costa, Printista M., and M. Marín. Parallel strategies for multimedia web services using static sat. In *International Conference on Internet Technologies and Applications (ITA 2007)*, pages 164–172, Wrexham, North Wales, UK, Sep. 4-7 2007. IEEE Computer Society.

[41] Veronica Gil-Costa, Ricardo Barrientos, Mauricio Marín, and Carolina Bonacic. Scheduling metric-space queries processing on multicore processors. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 0:187–194, 2010.

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

[43] W. Gropp, E. Lusk, and A. Skelljum. *Using MPI:Portable Parallel Programming with the Message Passing Interface.* Scientific and Engineering computation Series. MIT Press, Cambridge, MA, 1994.

[44] A. Guttman. The r-trees: a dynamic index structure for spatial search. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[45] Magnus Hetland. The basic principles of metric indexing. In Carlos Coello, Satchidananda Dehuri, and Susmita Ghosh, editors, *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*, pages 199–232. Springer Berlin / Heidelberg, 2009.

[46] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[47] Mouna Kacimi and Kokou Yetongnon. Similarity search in a hybrid overlay p2p network. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 460–466, Washington, DC, USA, 2006. IEEE Computer Society.

[48] Iraj Kalantari and Gerard McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5):631–634, September 1983.

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

[50] King-Ip Lin, H. Jagadish, and C. Faloutsos. The tv-trees - an index structure for high-dimensional data. *The Very Large Databases Journal (VLDBJ)*, 3:517–542, Oct. 1994.

[51] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, pages 33+, New York, NY, USA, 2004. ACM.

[52] Margarida Mamede. Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. In *Computer and Information Sciences - ISCIS 2005*, volume 3733 of *Lecture Notes in Computer Science*, 2008.

[53] Mauricio Marín, Veronica Gil Costa, and Roberto Uribe-Paredes. Hybrid index for metric space databases. In *The International Conference on Computational Science (ICCS)*, volume 5101 of *LNCS*, pages 327–336, Kraków, Poland, June, 23-25 2008. Springer.

[54] Mauricio Marin, Veronica Gil-Costa, and Cecilia Hernandez. Dynamic p2p indexing and search based on compact clustering. In *SISAP '09: Proceedings of the 2009 Second International Workshop on Similarity Search and Applications*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[55] Mauricio Marín, Roberto Uribe, and Ricardo Barrientos. Searching and updating metric space databases using the parallel egnat. In *Computational Science - ICCS 2007*, volume 4487 of *LNCS*, pages 229–236. Springer Berlin / Heidelberg, 2007.

[56] Dylan McGrath. Power consumption hurdles litter path to exascale computing. In *EETimes: News and Analysis* , 7/11/2012.

[57] Luisa Micó and Jose Oncina. A constant average time algorithm to allow insertions in the laesa fast nearest neighbour search index. In *Proc. of the 20th International Conference on Pattern Recognition, ICPR 2010, Istanbul, Turkey*, pages 23–26, August 2010.

[58] Luisa Micó, José Oncina, and Rafael C. Carrasco. A fast branch & bound nearest neighbour classifier in metric spaces. *Pattern Recognition Letters*, 17(7):731–739, June 1996.

[59] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, January 1994.

[60] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

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

[62] Gonzalo Navarro and Nora Reyes. Fully dynamic spatial approximation trees. In *the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, pages 254–270, Springer 2002.

[63] Gonzalo Navarro and Roberto Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734 – 747, 2011.

[64] 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.

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

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

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

[68] David Novák, Michal Batko, and Pavel Zezula. Metric index: an efficient and scalable solution for precise and approximate similarity search. *Information Systems*, 36(4):721 – 733, 2011.

[69] Eduardo Peña Jaramillo, Rodrigo Ovando, and Roberto Uribe-Paredes. Recuperación de imágenes basadas en contenido sobre estructuras métricas. In *Jornadas Chilenas de Computación (JCC2010)*, Antofagasta, Chile, Nov. 2010.

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

[71] P. Jonathon Phillips, Patrick J. Flynn, Todd Scruggs, Kevin W. Bowyer, Jin Chang, Kevin Hoffman, Joe Marques, Jaesik Min, and William Worek. Overview of the face recognition grand challenge. In *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1 of *CVPR '05*, pages 947–954, Washington, DC, USA, 2005. IEEE Computer Society.

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

[73] Nora Reyes. índices dinámicos para espacios métricos de alta dimensionalidad. Master's thesis, Universidad Nacional de San Luis, Argentina, 2002.

[74] H. Samet. The quadtree and related hierarchical data structures. In *ACM Computing Surveys*, pages 16(2):187–260, 1984.

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

[76] M. Shapiro. The choice of reference points in best-match file searching. *Communications of ACM*, 20(5):339–343, 1977.

[77] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J Dongarra. *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996.

[78] Raisa Socorro, Luisa Micó, and Jose Oncina. A fast pivot-based indexing algorithm for metric spaces. *Pattern Recognition Letters*, 32(11):1511–1516, August 2011.

[79] Caetano Traina, Agma Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *VII International Conference on Extending Database Technology*, pages 51–61, 2000.

[80] Caetano Traina, Jr., Roberto F. Filho, Agma J. Traina, Marcos R. Vieira, and Christos Faloutsos. The omni-family of all-purpose access methods: a simple and effective way to make similarity search more efficient. *The VLDB Journal*, 16(4):483–505, 2007.

[81] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience*, 3(1):71–86, January 1991.

[82] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. In *Information Processing Letters*, volume 40, pages 175–179, 1991.

[83] Roberto Uribe, Gonzalo Navarro, Ricardo Barrientos, and Mauricio Marín. An index data structure for searching in metric space databases. In *Computational Science - ICCS 2006*, volume 3991 of *LNCS*, pages 611–617. Springer Berlin / Heidelberg, 2006.

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

[85] Roberto Uribe-Paredes, Enrique Arias, José L. Sánchez, Diego Cazorla, and Pedro Valero-Lara. Improving the performance for the range search on metric spaces using a multi-GPU platform. In *Database and Expert Systems Applications (DEXA)*, volume 7447 of *Lecture Notes in Computer Science*, pages 442–449. Springer Berlin Heidelberg, 2012.

[86] Roberto Uribe-Paredes, Diego Cazorla, José L. Sánchez, and Enrique Arias. Acelerando la búsqueda por rango con un sistema híbrido de memoria compartida. In *Actas XXIII Jornadas de Paralelismo (JP2012)*, Elche, Spain, Sept 2012.

[87] Roberto Uribe-Paredes, Claudio Márquez, and Roberto Solar-Gallardo. Construction strategies on metric structures for similarity search. *CLEI Electronic Journal*, 12(3), Dic. 2009.

[88] Roberto Uribe-Paredes and Gonzalo Navarro. Egnat: A fully dynamic metric access method for secondary memory. In *Proc. 2nd Internatio-*

nal Workshop on Similarity Search and Applications (SISAP), pages 57–64, Prague, Czech Republic, August 2009. IEEE CS Press.

[89] Roberto Uribe-Paredes, Pedro Valero-Lara, Enrique Arias, José Luis Sanchez, and Diego Cazorla. Similarity search implementations for multi-core and many-core processors. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 656–663, 2011.

[90] URL. Bsponmpi. http://bsponmpi.sourceforge.net/.

[91] URL. Maryland cpu-gpu cluster. http://www.umiacs.umd.edu/research/GPU/.

[92] URL. NCSA cluster project. http://www.ncsa.uiuc.edu/Projects/GPUcluster/.

[93] URL. Top 500 list. http://www.top500.org/.

[94] URL. Yokogama pz4000 power analyzer. http://tmi.yokogawa.com/es/.

[95] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[96] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.

[97] K. Verbarg. The c-tree: A dinamically balanced spatial index. *Computing Suppl.*, 10:323–340, 1995.

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

[99] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Skypeer: Efficient subspace skyline computation over distributed data. In *Proceedings of ICDE 07*, pages 416–425, 2007.

[100] Wu-Feng and Dinesh Manocha. High-performance computing using accelerators. *Parallel Computing*, 33:645–647, 2007.

[101] P. Yianilos. Data structures and algoritms for nearest neighbor search in general metric spaces. In *4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*, pages 311–321, 1993.

[102] P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.

[103] P. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. In *11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, 2000.

[104] Pavel Zezula, Giuseppe Amato, Vlastislav Dohnal, and Michal Batko. *Similarity Search - The Metric Space Approach*, volume 32. Springer, 2006.

[105] Pavel Zezula, Pasquale Savino, Fausto Rabitti, Giuseppe Amato, and Paolo Ciaccia. Processing m-trees with parallel resources. In *RIDE '98: Proceedings of the Workshop on Research Issues in Database Engineering*, pages 147–, Washington, DC, USA, 1998. IEEE CS.