Natalia Carolina Miranda

Cálculo en Tiempo Real de Identificadores Robustos para Objetos Multimedia mediante una Arquitectura Paralela CPU-GPU

Tesis Doctoral en Ciencias Informáticas
PREMIO DR. RAÚL GALLARD | Año 2015

Cálculo en Tiempo Real de Identificadores Robustos para Objetos Multimedia mediante una Arquitectura Paralela CPU-GPU

Lic. Natalia Carolina Miranda

UNIVERSIDAD NACIONAL DE SAN LUIS
FACULTAD DE CIANCIAS FÍSICO, MATEMÁTICA Y NATURALES
DEPARTAMENTO DE INFORMÁTICA

Tesis para optar al grado de Doctor en Ciencias de la Computación

Cálculo en Tiempo Real de Identificadores Robustos para Objetos Multimedia mediante una Arquitectura Paralela CPU-GPU

Lic. Natalia Carolina Miranda

Director

Dr. Chávez Edgar Leonel

Co-Director

Dra. Piccoli María Fabiana

La Plata, abril de 2014

Miranda, Natalia Carolina
Cálculo en tiempo real de identificadores robustos para objetos multimedia mediante una arquitectura paralela CPU-GPU / Natalia Carolina Miranda. - 1a ed. - La Plata: Universidad Nacional de La Plata. Facultad de Informática, 2016.
235 p.; 24 x 16 cm.

ISBN 978-950-34-1302-9

1. Informática. 2. Tesis Doctorales. I. Título.
CDD 004.0711

Primera edición, 2016
ISBN N.º 978-950-34-1302-9

Queda hecho el depósito que marca la Ley 11723
Impreso en Argentina

A mis dos soles: Mateo y Enrique.

A mis padres.

Agradecimientos

Han pasado muchos años desde que comencé la tesis y esta es la parte más difícil para mí, expresar mi reconocimiento a todas aquellas personas que lo hicieron posible. Todos, desde el lugar que les tocó participar, contribuyeron a la concreción de éste, hasta ahora el trabajo más importante y de mayor envergadura que haya encarado en mi vida académica. Probablemente me olvidaré de nombrar a algunas personas, y espero que me perdonen.

En primer lugar quiero agradecer a mis directores de tesis, el Dr. Edgar Chávez y la Dra. Fabiana Piccoli. Al Dr. Edgar Chávez por creer en mí aún sin conocerme, por haberme brindado todo lo que estuvo a su alcance y haber hecho posible, a pesar de las distancias y limitaciones, el desarrollo de este trabajo. A la Dra. Fabiana Piccoli por su apoyo infinito, por los aportes, los concejos y el tiempo que me dedicó para resolver todas mis inquietudes. En ella no solo encontré una profesora sino también a una amiga con la que podía confiar.

Mi agradecimiento se hace extensivo a la Fac. Cs. Fca. Matemática y Naturales, especialmente al departamento de Informática, al proyecto de investigación al que pertenezco y a CONICET por todo el apoyo brindado.

Quiero agradecer a aquellas personas que me brindaron su amistad y su apoyo desinteresado:

- A Nora Reyes por estar siempre dispuesta a responder mis consultas, por el tiempo que me dedicó para ayudarme, aconsejarme y brindarme sus valiosos aportes.
- A Gabriela Palacio y Cecilia Palacios por escucharme y apoyarme incondicionalmente, gracias por estar a mi lado.

- A Mariela Lopresti por escuchar y ayudar en la solución de problemas compartidos y a Mercedes Barrionuevo por brindarme aliento y apoyo.
- A Jacqueline Fernández por todo el tiempo que pasamos juntas realizando los trabajos para aprobar los cursos de postgrado.
- A todos aquellos profesores que de una forma u otra me impulsaron a seguir formándome y a seguir en esta disciplina.

Quiero agradecer a mi familia (hermanos, suegros, cuñados y sobrinos) y amigos quienes sin conocer lo que significa realizar un doctorado brindaron su apoyo y concejo.

Por último mi especial agradecimiento a mi esposo Enrique y mi hijo Mateo no sólo por permitirme hacer esta tesis, quitándoles tiempo para disfrutar juntos, sino porque sin ellos mi vida estaría vacía y mis logros académicos carecerían de sentido.

A mis padres que siempre estuvieron conmigo y se alegran de cada uno de mis logros. Por cuidar de Mateo cuando yo no estaba en casa.

A todos y cada uno de los que colaboraron, hasta del lugar más pequeño.

¡¡¡ Muchas Gracias!!!

Índice General

Índice de Figuras

Índice de Cuadros

CAPÍTULO 1

Introducción

El surgimiento de los datos multimedia ha innovado el concepto de las bases de datos. Éstas se han visto siempre como repositorios de información, tanto textual como numérica, regidas por un conjunto de normas que le aportan semántica. Ahora al considerar los objetos multimedia, se necesita hacer un cambio radical en esta concepción a fin de poder representar y recuperar los objetos. Un objeto multimedia puede ser un texto, un video, una imagen, un sonido, entre otros.

Por ser representados en forma diferente, los objetos multimedia no pueden ser tratados de la misma manera que los objetos en una base de datos tradicional. Su representación implica un tratamiento distinto, se debe extraer las características particulares y generales de cada uno de ellos para poder recuperarlos por similitud o proximidad. En el caso de las bases de datos multimediales no es posible hacer búsquedas exactas, la información no siempre se almacena de la misma manera; dos imágenes de la misma escena no necesariamente son iguales, desde el punto de vista perceptual, una persona puede no encontrar diferencias y diría que ambas imágenes son idénticas, pero si se las compara bit a bit se podría determinar que ambas imágenes son totalmente diferentes.

En las bases de datos tradicionales, la recuperación de un registro se realiza a través del dato *clave* (llave del registro), el cual lo identifica unívocamente. En las bases de datos de texto completo (*full text*), para el caso de la recuperación de información, con cualquier segmento significativo de un texto podemos recuperar el texto completo; así, cuando buscamos en internet sólo utilizamos para la búsqueda aquellas palabras que distinguen al texto a recuperar. Visto de otra manera, con una parte significativa, pero arbitraria del objeto almacenado, podemos recuperar el objeto completo.

Por el contrario, las bases de datos multimedia almacenan información cuyas características, como se enunció antes, son completamente diferentes a lo discutido para bases de datos tradicionales. El comportamiento particular de los objetos multimedia se debe al hecho de intentar formalizar la percepción de los mismos. Un ejemplo de este comportamiento se puede encontrar en las canciones comprimidas con pérdida: mp3, ogg, entre otras; la parte central de la compresión reside en modelar la inhabilidad del cerebro para distinguir ciertos sonidos simultáneos, los cuales se enmascaran unos a otros. Si se comparan digitalmente dos canciones, una comprimida en formato mp3 y la otra sin comprimir, difícilmente se encontrarían coincidencias digitales; pero si se escuchan ambas sería complicado poder distinguir una de la otra.

Para poder tratar a los objetos multimedia, en particular a las señales de audio, con la misma eficiencia con las que se tratan los objetos de texto, es necesario obtener una representación de ellos, la cual sea estable y persistente a las posibles degradaciones naturales que puedan sufrir. Esta representación se la conoce como característica de la señal o huella digital de la señal.

Una huella digital es un mecanismo capaz de identificar de manera precisa y única a un objeto o persona. La huella digital humana, es un ejemplo de ello, es el certificado de autenticidad de las personas de manera única e inconfundible.

En el caso de las señales de audio, una huella digital de audio es un fragmento compacto de bajo nivel basado en el contenido de la señal. La huella digital proporciona la capacidad de identificar las señales de audio de una manera rápida y confiable.

Ahora si la necesidad es recuperar señales de audio o algún otro objeto multimedia a partir de grandes repositorios, es necesario contar con un mecanismo eficaz de recuperación de datos. Esto se puede llevar a cabo mediante el uso del modelo de espacio métrico.

El modelo de espacio métrico es un paradigma capaz de modelar los problemas de búsqueda por similitud. Un espacio métrico está compuesto de un conjunto de objetos (un subconjunto de éste se denomina base de datos) y una función de distancia especificada entre los objetos. La función de distancia cumple algunas propiedades establecidas y permite determinar qué objetos son próximos o similares unos de otros. Las bases de datos métricas permiten el almacenamiento de

objetos de un espacio métrico (por ejemplo: documentos, strings, imágenes, secuencias de ADN, entre otros) y la realización de consultas por similitud sobre ellos de manera eficiente. En este tipo de base de datos, las consultas de mayor interés son: la búsqueda por rango y los *k* vecinos más cercanos (*k*-NN). Otras consultas a realizar en un espacio métrico son: la búsqueda de los *k* vecinos más cercanos para todos los objetos de la base de datos (all-*k*-NN) y la unión por similitud (*join*). Para poder responder a las consultas en un espacio métrico existen diferentes métodos, en forma general se los puede clasificar en dos grandes grupos: los métodos basados en fuerza bruta y los basados en índices. En el primer caso, el más fácil de resolver, se caracteriza por examinar la base de datos completa calculando la distancia entre el objeto de consulta y todos los objetos de la base de datos, evaluar las distancias obtenidas y devolver el resultado. Si se resuelve usando una CPU en forma secuencial, este método posee un alto costo computacional debido al número de evaluaciones de distancia a realizar. Los métodos de indexación, en cambio, reducen el número de cálculos de consulta, evitando tratar toda la base de datos. Estos métodos con índices requieren previamente una etapa de pre procesamiento de la base de datos para establecer el índice. En base al índice calculado, estos métodos permiten acelerar las consultas pero no siempre es suficiente, principalmente cuando estamos en un ambiente de tiempo real, donde las respuestas a las consultas deben ser obtenidas en forma rápida. En consecuencia, para lograr mayor velocidad en las respuestas, otras técnicas deben ser incluidas, una de ellas es aplicar computación de alto desempeño en la solución computacional.

Las técnicas de computación de alto desempeño (HPC, sigla en inglés de *High Performance Computing*) han permitido acelerar el tiempo de aquellas aplicaciones secuenciales con un alto costo computacional. Tanto la extracción de características de una señal de audio, como la recuperación de objetos en una base de datos métrica son un ejemplo de este tipo de aplicaciones, es por ello que pensar en soluciones donde se aplican técnicas de HPC es posible. Además, dadas las características de ambas aplicaciones y de las unidades de procesamiento gráfico GPU (sigla en inglés de *Graphics Processing Unit*), aplicar técnicas de HPC usando GPU en las soluciones es una alternativa válida.

Las GPUs son dispositivos masivamente paralelos, procesadores many- core, los cuales permiten ejecutar múltiples unidades de proceso: threads, aplicando un paralelismo de gránulo fino. Tienen características muy diferentes a otras arquitecturas paralelas, como por ejemplo la asignación flexible de los recursos locales (memoria o registros) para los threads, jerarquía de memoria, entre otras.

El modelo completo, denominado GPU computing o GPGPU (sigla derivada del inglés General-Purpose Computing on Graphics Processing Units), consiste en el uso de un sistema integrado por al menos una CPU y una GPU para acelerar las operaciones de cálculo científico de propósito general. La CPU junto con la GPU constituye una potente combinación: la CPU está formada por varios núcleos optimizados para el procesamiento de datos en serie, mientras que la GPU consta de millares de núcleos más pequeños y eficientes diseñados para el procesamiento en paralelo. Por lo tanto, las partes del código con características secuenciales se ejecutan en la CPU mientras que las secciones paralelas se ejecutan en la GPU. Esto permite aumentar el desempeño o la performance de las aplicaciones.

Existen varias herramientas para programar en una GPU, una de ellas es CUDA (siglas en inglés de Compute Unified Device Architecture). CUDA permite a los desarrolladores crear componentes de programación aislados. Cada componente resuelve un problema en una GPU dedicada aplicando un procesamiento masivo de datos en paralelo. Un programa en CUDA está especificado en lenguaje C/C++ extendido con un conjunto de instrucciones, las cuales permiten implementar el paralelismo en el procesamiento de tareas y datos con diferentes niveles de granularidad. Un programa completo CUDA tiene diferentes fases, las cuales se pueden ejecutar ya sea en la CPU o la GPU. Cuando la fase tiene bajo o nulo paralelismo, su ejecución se realiza en la CPU, en cambio si la fase presenta un paralelismo masivo, se implementa y ejecuta en la GPU.

Este trabajo de tesis está dedicado a dar soluciones de alto rendimiento utilizando GPU computing a los problemas de extracción de características de una señal de audio para obtener la huella digital de la misma y su respectiva recuperación en una base de datos. Ambos problemas tienen un alto costo computacional si los mismos se calculan en la CPU, por lo tanto haciendo uso de los beneficios de la GPU para acelerar

este tipo de aplicaciones, las mismas pueden ser obtenidas en tiempo real de forma rápida y eficiente.

En las siguientes secciones se realiza un análisis de los aspectos relevantes de las problemáticas: Huella digital de la señal, extracción de características y recuperación de objetos multimedia usando espacios métricos. Finalmente se detallan los objetivos, aportes y organización de esta tesis.

1.1. Descripción de la Problemática

Las tarjetas de video (GPUs) han evolucionado desde sus orígenes, a mediados del 2001, hasta la actualidad en forma drástica permitiendo a numerosas aplicaciones aprovechar su potencia de cómputo. Las GPUs son arquitecturas paralelas programables diseñadas en un principio para aplicaciones gráficas y de visualización. Con el correr de los años se fue extendiendo su uso a aplicaciones de propósito general. Dada su naturaleza paralela y bajo costo, su popularidad y aceptación como procesador masivo paralelo se debe a varias de sus características entre las cuales se encuentran el gran número de threads a ejecutar en paralelo, la jerarquía de memorias y el gran ancho de banda de memoria.

Todas las razones antes mencionadas junto al bajo costo de las placas de video hicieron que evaluáramos la posibilidad de utilizarlas en la solución computacional para un sistema integral de huella digital, dentro del cual identificamos dos grandes y costosos subproblemas: la extracción de características de una señal de audio (huella digital) para su identificación y la recuperación de objetos multimedia (audio), en una base de datos métrica con el fin de aplicarla en la recuperación de señales de audio.

La indexación e identificación de señales de audio son dos disciplinas que han sido muy estudiadas en los últimos años, especialmente la última. La identificación de audio consiste en la habilidad de agrupar las señales de audio con la misma naturaleza perceptual, por esta razón se requiere una representación del objeto, la cual debe ser estable y persistente a diferentes degradaciones naturales. Esta representación se denomina huella digital de audio: AFP (del inglés AudioFingerPrint). La obtención de la AFP forma parte de un sistema de huella digital.

Un sistema de huella digital, básicamente consiste de dos partes fundamentales: la extracción de características de la señal, AFP, y la aplicación de un algoritmo de búsqueda eficiente para encontrar aquellas AFP similares a una AFP de consulta, en una base de datos.

Hay varios requerimientos prácticos a cumplir por un sistema de huella digital de audio exitoso, ellos son:

- Ser capaz de identificar una señal a pesar de presentar severas degradaciones de la misma.
- Insumir poco tiempo en el proceso de identificación de las señales.
- Ser computacionalmente eficiente, tanto en el cálculo de las huellas digitales como en la búsqueda en una base de datos de la señal o en una similar
- Ser escalable, es decir, seguir operando bien aunque la base de datos incremente su tamaño

En las siguientes subsecciones se describen brevemente los problemas principales que contribuyeron a ser un desafío en este trabajo de tesis.

1.1.1. Huella Digital de Audio: Extracción de Características

Las AFPs son tecnologías usadas como materia prima de software por un gran número de aplicaciones de importancia económica. Algunas de ellas son: monitoreo broadcast [SKKC02]; etiquetado automático de audio [HK02, Wan03] y detección de duplicados [CICI0], entre otras.

En el diseño de las AFPs para las aplicaciones antes mencionadas, existe cierta tensión entre dos objetivos contrapuestos: robustez y rapidez. Por un lado una característica robusta de la señal implica una representación densa del audio, y por otro lado una AFP densa implica más ciclos de computadora para obtener la representación. Por esta razón sería importante encontrar un punto medio entre estos dos objetivos para que la performance del sistema no se vea afectada.

Para obtener las características de la señal es necesario dividir la señal de audio en segmentos cortos de igual tamaño,

los cuales son procesados aplicando varias etapas para finalmente obtener la AFP. Este procesamiento tiene asociado un alto costo computacional. En el capítulo 2 se explica detalladamente, realizando un análisis exhaustivo de los diferentes pasos computacionales involucrados.

1.1.2. Recuperación de Objetos Multimedia: Espacios Métricos

La indexación y recuperación de objetos multimedia también ha captado mucho el interés de los investigadores en los últimos años. Como se mencionó antes, los objetos multimedia no pueden ser tratados del mismo modo que los objetos en una base de datos tradicional, la representación de los objetos multimedia es totalmente diferente. Por ejemplo dos señales de audio de una misma canción pueden resultar idénticas al oído humano pero computacionalmente son totalmente distintas, pueden no coincidir en todos los bytes. Los objetos multimedia son representados por un conjunto de características relevantes y los podemos recuperar por similitud.

El modelo adecuado para los problemas de búsqueda por similitud es el modelo de espacio métrico. Las bases de datos métricas permiten almacenar objetos de un espacio métrico y ejecutar consultas por similitud sobre ellos de forma eficiente. La organización y obtención de los objetos multimedia se puede realizar mediante la aplicación de algún método, ya sea utilizando fuerza bruta (scan secuencial) o mediante la construcción y aplicación de un índice. La selección de uno u otro método está determinado por la clase de objeto y el tipo de aplicación para lo cual se requiere.

Cualquiera sea el método elegido, fuerza bruta o índice, el costo computacional tanto en tiempo como espacio es elevado, debido a las estructuras de datos a administrar y el tiempo de respuesta para una consulta dada. Muchas veces es necesario pensar en aplicar otras técnicas de programación como es la computación de alto desempeño. En el capítulo 3 se explica con más detalle la problemática de los espacios métricos y en el capítulo 4 las características de la computación paralela en GPU.

1.1.3. Limitaciones de GPGPU y CUDA

Desde su creación, la GPU (placa de video) se ha utilizado como un dispositivo dedicado para acelerar aplicaciones de procesamiento gráfico, juegos de videos, visión por computadoras, aplicaciones 3D, entre otras [Buc07]. Debido a su naturaleza altamente paralela los investigadores tuvieron la visión de que la GPU puede ser adecuada para ser usada en aplicaciones de propósito general, dando origen al paradigma GPGPU.

Cuando este paradigma comenzó a expandirse haciendo uso de la tecnología CUDA, los investigadores y desarrolladores se encontraron con muchos obstáculos al momento de programar alguna aplicación general. Algunos ejemplos de ellos son: el acceso de varios threads modificando la misma posición de memoria, el uso de punteros, la aplicación de procesos recursivos, la sincronización entre threads de distinto bloques, entre otros. Si bien hoy muchos de estos problemas se han resuelto a través de la aplicación de las técnicas de coalescencia de memoria, las funciones atómicas, el uso de templates, la incorporación de punteros en memoria global, la admisión de la recursión, entre otros; existen problemas que aún son un desafío al momento de aprovechar el máximo rendimiento de la GPU, uno de ellos es la sincronización entre threads de distintos bloques.

Transformar aplicaciones secuenciales de alto costo computacional a aplicaciones paralelas con alto rendimiento utilizando la GPU es un gran reto, el cual implica gran cantidad de trabajo, además del conocimiento de las fortalezas y debilidades de la GPU para aprovechar los múltiples beneficios de la misma. Estas características constituyen un desafío, el cual vale la pena enfrentar para lograr la optimización y mejora de la performance de las aplicaciones.

1.2. Objetivos

El objetivo principal planteado para esta tesis es: Analizar la validez de utilizar una arquitectura many-core para resolver un sistema de huella digital robusto y eficiente a ser usado en tiempo real.

A partir de este objetivo general, surgen los objetivos específicos, los cuales intentan dar respuesta a los aspectos enunciados en forma genérica en la sección anterior. Los objetivos específicos planteados son:

1. Objetivo 1: Adquirir los conocimientos relevantes de acuerdo a la temática de esta tesis: huella digital de audio, espacios métricos y computación paralela utilizando GPU. Realizar un relevamiento del estado del arte de los espacios métricos y AFP aplicando la GPU para su resolución.
2. Objetivo 2: Analizar diferentes algoritmos de transformación de audio y técnicas de extracción de características, principalmente aquellas basadas en el análisis del contenido de la información.
3. Objetivo 3: Diseñar y codificar las diferentes funciones para la extracción de las características de una señal de audio y generar la huella digital de audio utilizando la arquitectura CPU-GPU.
4. Objetivo 4: Analizar las estructuras necesarias para la indexación y recuperación de audio en espacios métricos.
5. Objetivo 5: Obtener y evaluar los resultados obtenidos, considerando aspectos básicos como Robustez, Compactación, Granularidad, Complejidad de Tiempo y Escalabilidad.

A partir de la problemática planteada en la sección anterior junto con todos estos objetivos cumplidos, derivados del objetivo general, se logró la construcción de esta tesis doctoral.

1.3. Principales Contribuciones y Publicaciones

En esta sección se detallan las principales contribuciones realizadas en el marco de la tesis, en ella enumeramos los artículos directamente relacionados con este trabajo y aquellos donde se transfirieron algunos de los desarrollos realizados.

1.3.1. Huella Digital de Audio

- Publicaciones de trabajos completos y exposiciones en Congresos Internacionales con Referato.

  - Natalia Miranda, Fabiana Piccoli, Edgar Chávez, Antonio Camarena-Ibarrola. "*Using GPU to Speed Up the Process of Audio Identification*". 10th International Information and Telecommunication Technologies Symposium, I2TS'2010. IEEE. Pp: 153-160. Río de Janeiro -Brasil, December 2010.
  - Natalia Miranda, Fabiana Piccoli, Edgar Chávez. "*A Pure GPU Multiband Spectral Entropy Audio Fingerprinting*". Jornadas Chilenas de Computación (JCC), Conferencia Internacional (SCCC-11). Curicó, Chile. Octubre 2011. In Press. http://jcc2011.utalca.cl/index.php/en/accepted-papers.

- Publicaciones de trabajos completos y exposiciones en Congresos Nacionales con Referato.

  - Natalia Miranda, Fabiana Piccoli, Edgar Chávez. "*Considering Pure GPU Model for an Audio Fingerprinting System*". ENIEF 2011. Noviembre 2011. ISNN 1666-6070. Pp 3033-3044. Rosario. Santa Fe.
  - Natalia Miranda, Fabiana Piccoli, Edgar Chávez. "*Finding Audio Fingerprinter Using GPU*". MECOM 2010 CILAMCE 2010. ISSN 1666-6070. Pp 3127-3141. Buenos Aires. Noviembre 2010.
  - Natalia Miranda, Fabiana Piccoli, Edgar Chávez, Antonio Camarena-Ibarrola. "*Fast GPU Audio Identification*". XVI Congreso Argentino en Ciencias de la Computación (CACIC 2010). Buenos Aires. ISBN 978-950-9474-49-9. Pp 229-242. Octubre 2010.

1.3.2. Espacios Métricos

- Capítulos de Libro

  - Natalia Miranda, Edgar Chávez, Fabiana Piccoli, Nora Reyes. "*(Very) Fast (All) k-Nearest Neighbors Without Indexing*". Proceedings del 6th International Conference on Similarity Search and Applications (SISAP 2013) en Lecture Notes in Computer Science series, Nro 8199, pages 300-311, 2013. Editors: Nieves Brisaboa, Oscar Pedreira, Pavel Zezula. Springer-Verlag Berlin Heidelberg 2013. ISBN: 978-3-642-41061-1 (Print) 978-3-642-41062-8 (Online).

1.3.3. Publicaciones de Transferencia de Desarrollo

Los trabajos aquí enumerados corresponden a las transferencias de algunos desarrollos realizados durante esta tesis, algunos de ellos no forman parte de las versiones finales de cada una de las propuestas. Tal es el caso del quicksort, el cual fue desarrollado como parte de la versión de los all-k-NN aproximados.

- *En Revista*

  - Lopresti, Mariela; Miranda, Natalia; Piccoli, Fabiana; Reyes, Nora."*Solving Multiple Queries through a Permutation Index in GPU*". Revista Computación y Sistemas, vol. 17, Nro. 3, julio-septiembre, 2013, Págs. 341-356, Instituto Politécnico Nacional, Distrito Federal, México. ISSN: 1405- 5546. Editores en jefe: Grigori Sidorov, Centro de Investigación en Computación, IPN, México; Ulises Cortés, Universidad Politécnica de Cataluña, España. Publicada por el Centro de Investigación en Computación (CIC) del Instituto Politécnico Nacional (IPN) y es patrocinada por el Consejo Nacional de Ciencia y Tecnología (CONACyT).

- Mercedes Barrionuevo, Luis Britos, Fabricio Bustos, Verónica Gil Costa, Mariela Lopresti, Virginia Mancini, Natalia Miranda, Cesar Ochoa, Fabiana Piccoli, A. Marcela Printista, Nora Reyes. "*New technologies for big multimedia data treatment*". Journal of Computer Science & Technology, Vol. 13. Nro. 3 December 2013, Págs. 111-117 – ISSN 1666-6038, Special Issue on "I Jornadas de Cloud Computing 2013

- *Publicaciones de trabajos completos y exposiciones en Congresos Internacionales con Referato*

  - Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, Nora Reyes. "*Solving Multiple Queries through the Permutation Index in GPU*". 4th International supercomputing Conference in México. Colima México. 5-8 March 2013. Artículo invitado para enviar una versión extendida para un Special Issue dedicado a Supercomputing: Applications and Technologies, en la revista Computación y Sistemas.

- *Publicaciones de trabajos completos y exposiciones en Congresos Nacionales con Referato*

  - Mariela Lopresti, Natalia Miranda, Mercedes Barrionuevo, Fabiana Piccoli, Nora Reyes. "*Evaluating tradeoff between recall and performance of GPU Permutation Index*". XIX Congreso Argentino de Ciencias de la Computación (CACIC 2013). Universidad CAECE y Red UNCI. PP: 194-203. ISBN 978-987-23963-1-2. Octubre 2013, Mar del Plata.
  - Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, Nora Reyes. "*Permutation Index and GPU to Solve Efficiently Many Queries*". Proceedings del VI Latin American Symposium on High Performance Computing (HPC Latam 2013), Sesión: GPU Architecture and Applications. PP. 101-112. July, 2013. Mendoza, Argentina.

- Mariela Lopresti, Natalia Miranda, Fabiana Piccoli, Nora Reyes. "*Efficient Similarity Search on Multimedia Databases*". XVIII Congreso Argentino de Ciencias de la Computación (CACIC 2012). ISBN 978-987-1648-34-4. Pp: 1079-1088. Universidad Nacional del Sur – Bahía Blanca, Buenos Aires, Argentina, Octubre 2012.

1.4. Organización de la Tesis

En la escritura de este trabajo de tesis, establecimos como premisa que el mismo fuera tan auto contenido como sea posible, está organizado en siete capítulos y tres apéndices. Los capítulos 2, 3 y 4 están dedicados a introducir los conceptos generales sobre los principales tópicos vinculados a esta tesis: huella digital de señales de audio, espacios métricos y computación de propósito general y alto desempeño en GPU (GPGPU). En los capítulos 5 y 6 se especifican los aportes de esta tesis y finalmente en el último capítulo se exponen las conclusiones y las líneas de investigación y trabajos futuros a seguir. Cada capítulo trata los siguientes temas:

- Capítulo 2: **Huella Digital de Audio**

  Describe el marco teórico relacionado a la huella digital y las alternativas para obtenerla, enfocándonos en las AFPs calculada a través de la entropía de la señal. Describimos dos Métodos computacionales para obtenerla: *MBSES* y *TES*, ambos con diferentes costos.

- Capítulo 3: **Espacios Métricos: Generalidades**

  Detalla todos los conceptos básicos relacionados a espacios métricos y a la resolución de las diferentes consultas por similitud, describiendo las características de cada una. Finalmente hacemos especial referencia al método basado en índice SAT+ (Árbol de Aproximación Espacial Distante).

- Capítulo 4: **GPGPU**

  Introduce a las GPUs como computadoras paralelas y su programación con CUDA, detallando el modelo de programación, la arquitectura CUDA de la GPU, las características relevantes del modelo de ejecución y sus ventajas respecto al desempeño de las soluciones.

- Capítulo 5: **Huella Digital de Audio en GPU**

  Presenta las soluciones desarrolladas para esta tesis en GPU para obtener la AFP *MBSES* (Trabaja con la señal en el dominio de la frecuencia) y la AFP *TES* (Se basa en una señal en el dominio del tiempo), detallándose los aspectos de diseño e implementación te- nidos en cuenta a fin de lograr soluciones con buen desempeño. Finalmente muestra los resultados empíricos, evaluando el comportamiento de cada solución en distintas GPUs, y analiza sus aspectos distintivos respecto a las soluciones existentes en la bibliografía.

- Capítulo 6: **Espacios Métricos en GPU**

- Explica la solución en GPU propuesta para este trabajo, con el propósito de resolver de manera simple y con buen desempeño consultas por similitud en espacios métricos. Las consultas a resolver son búsqueda por rango, búsqueda de los k vecinos más cercanos y de todos los k vecinos más cercanos de todos los objetos en una base de datos.
  Este capítulo incluye la descripción de *To p - K* y sus tres versiones: *Top-K AA*, *Top-K QSeP* y *Top-K QSeH*; los resultados empíricos, por medio de los cuales se evalúa el comportamiento de cada solución; y un pormenorizado análisis de la bibliografía existente y una comparación con nuestros aportes.

- Conclusiones y Trabajos Futuros

  En este capítulo se resumen las principales contribuciones y las futuras líneas a seguir.

Respecto a los apéndices, estos están dedicados a: detallar las características de la arquitectura de la GPU para las diferentes generaciones (Apéndice A Generaciones de Arquitectura de GPU Nvidia); otras propuestas desarrolladas y desestimadas por su bajo desempeño y/o dificultad de implementación en la GPU (Apéndice B Otros Desarrollos en GPU); y, a la comparación detallada de las propuestas de soluciones usando GPU para resolver consultas en espacios métricos con las desarrolladas en esta tesis (Apéndice C Ventajas y Desventajas del Estado del Arte).

CAPÍTULO 2

Huella Digital de Audio

A medida que la tecnología ha ido avanzando siempre existió el interés por distinguir un elemento de un grupo de ellos mediante sus propias características. Una técnica conocida y bien establecida es la huella digital humana, la cual permite identificar unívocamente a cada persona. Además se puede leer, almacenar y usar fácilmente en distintos ambientes. Siguiendo con esta idea, se desea buscar una forma de simbolizar los objetos multimedia, en particular la señal de audio. Dicha representación al ser única permitirá identificar a los objetos adecuadamente.

Una huella digital de audio (AFP) es una representación compacta de los segmentos perceptualmente relevante del contenido de un audio. Idealmente, una AFP debe ser una invariante de la señal, una característica intrínseca, si la señal ha sufrido severas degradaciones, su huella será similar a la original.

Un sistema de huella digital comprende dos partes fundamentales: la extracción de características relevantes de la señal y su recuperación desde una base de datos de AFPs. En este capítulo nos centraremos en la primera fase, detallando las propiedades a cumplir por una AFP y las características del sistema que permite obtenerla.

2.1. Propiedades deseables en una AFP

Para que una AFP sea exitosa debe cumplir ciertos requisitos, los cuales dependen en gran medida de la aplicación donde se usa. Los requerimientos de una AFP son útiles para valorar y comparar diferentes tecnologías de huellas digitales de audio [CBKH05, HK02]. A continuación se enumeran cada uno de los requerimientos en forma detallada:

1. *Precisión:* Mide el número de objetos recuperados correctamente sobre el total de objetos en una base de datos. Se contabiliza la cantidad de identificaciones correctas, perdidas y erróneas (falsos positivos).
2. *Robustez:* La huella digital debe basarse en características perceptuales invariantes a las degradaciones de la señal. Las señales de audio dañadas severamente deberían poseer una huella digital similar a la señal de audio original. Algunas fuentes de degradación son: la compresión, la distorsión o la interferencia en el canal de transmisión, el ruido de fondo, los codificadores de audio, la variación del volumen o intensidad, el cropping, entre otras.
3. *Granularidad:* Es capacidad para identificar una señal de audio con un pequeño segmento de ésta. Esta característica, también denominada robustez al cropping, es muy utilizada en algunas aplicaciones de recuperación de información musical. Es un parámetro dependiente de la aplicación que la usa, algunas consideran toda la canción para poder identificarla mientras que otras sólo un pequeño fragmento del audio.
4. *Complejidad:* Una AFP debería ser determinada con el menor esfuerzo computacional posible. Para que una AFP cumpla con este requisito se deben tener en cuenta los siguientes factores: costo computacional de la extracción de la AFP, tamaño de la AFP, complejidad de la búsqueda, complejidad de la comparación de AFPs, costo de agregar nuevos elementos a la base de datos, entre otros.
5. *Escalabilidad*: Se define como la capacidad de un sistema de huellas digitales para operar con grandes conjuntos de señales de audio. Esta característica está condicionada por un tiempo de complejidad bajo, una AFP compacta y una buena técnica de indexación. La escalabilidad afecta a la precisión y complejidad del sistema.
6. *Versatilidad:* Es la capacidad de identificar una señal de audio independientemente del formato. En

otras palabras, usa la misma base de datos en diferentes aplicaciones.

7. *Seguridad:* Se refiere a la vulnerabilidad de la solución para la manipulación o cracking. En contraste con el requisito de la robustez, las manipulaciones están diseñadas para engañar al algoritmo de identificación de AFP.
8. *Fragilidad* Algunas aplicaciones, como sistemas de verificación de integridad de contenido, pueden requerir la detección de cambios en el contenido. Esto es contrario a la exigencia de robustez, la AFP debe ser robusta a las transformaciones de contenido pero no a otras distorsiones externas. El objetivo de este requisito es lograr detectar alteraciones en los datos.

La mejora de algunas características, a veces implica la pérdida de performance en otras. Por lo tanto una AFP debería ser:

- *Un resumen perceptual de la señal:* Una AFP debe conservar la máxima información relevante perceptual de modo tal que pueda discriminar una señal de audio entre un gran número de AFPs. Este requisito entra en conflicto con las características de complejidad y robustez.
- *Robusta:* La AFP debería ser invariante a las degradaciones de la señal tales como: ruido, ecualización, cropping, entre otras.
- *Compacta:* La representación de la señal debería ser de un tamaño reducido, disminuyendo la complejidad sin afectar a la robustez, precisión y fiabilidad del sistema.
- *Fácilmente computable:* La AFP debería poder determinarse con el mínimo esfuerzo computacional posible.

Dependiendo de la aplicación en donde usamos la AFP, uno puede priorizar una propiedad respecto de otra o realizar una negociación (trade off) de ellas.

2.2. Aplicaciones de la AFP

Las huellas digitales de audio son empleadas en un amplio número de aplicaciones, algunas de ellas se describen a continuación:

1. *Monitoreo Broadcast:* Ésta es la aplicación más conocida donde se usa AFP. Se refiere a la generación automática de una lista de reproducción de audio de radio, de televisión o de internet con fines de recaudación de regalías, verificación de programas, verificación de publicidad y medición de audiencia [SKKC02].
   Un sistema de control de broadcast a gran escala basada en huellas digitales consta de varios puntos de control y un sitio central donde está ubicado el servidor de AFPs. En los sitios de monitoreo, las AFP son extraídas de todos los canales de transmisión y recolectadas en el servidor central. Luego éste, teniendo en cuenta una base de datos, realiza la búsqueda y produce las listas de reproducción de todos los canales de broadcast.
2. *Audio conectado:* Es un término general para aplicaciones de consumo, donde la música está conectada de alguna forma a la información adicional y de sostén. Una canción puede ser identificada usando un fragmento de audio capturado desde un teléfono celular [HK02, Wan03]. Como ejemplo de esta aplicación, supongamos la siguiente situación: una persona va conduciendo su auto y escucha por la radio una canción que le gusta pero no sabe qué artista la canta y el nombre de la canción ¿Cómo podría averiguar esos datos? Una forma deseable de realizar esta operación sería usar su teléfono celular para realizar la consulta y unos segundos más tarde, a través del teléfono obtener el nombre del autor y de la canción escuchada, o mediante el envío de un correo electrónico con la información a la dirección de la persona. Notar que la señal de audio en este tipo de aplicaciones está severamente degradada debido a la transformación aplicada por las emisoras de radio, la transmisión de F M/AM, el cambio acústico entre el altavoz y el micrófono de los teléfonos celulares, la

codificación de la voz y por último, la transmisión a través de la red móvil. Por lo tanto, desde el punto de vista tecnológico es una aplicación con un gran desafío.

3. *Tecnología de Filtrado para el uso compartido de archivos:* Se refiere a los filtros que se usan para descargar música a través de internet. Cuando la música puede ser transmitida por una red peer-to-peer, las AFPs son determinadas a partir de los paquetes y son buscadas en una lista de canciones legales, esto se realiza para prevenir las copias ilegales y asegurar que los archivos descargados son realmente los buscados.
4. *Organización automática de una biblioteca musical:* Los reproductores de música proveen herramientas para organizar las canciones, uno de ellos es el rotulado de las mismas con el título del álbum, por ejemplo. Cuando estos rótulos están vacíos podrían ser automáticamente completados usando técnicas de AFPs.
5. *Detección de duplicados:* Esta aplicación permite mantener la integridad de las bases de datos multimedia, evitando las copias duplicadas. Este tipo de aplicación es útil para determinar si dos archivos de audio son iguales independientemente del formato de compresión u otra característica [CIC1O].

Por lo expuesto anteriormente, existen muchas aplicaciones donde se puede usar la AFP de audio. Es más, de las aplicaciones antes numeradas podemos ver su naturaleza distinta e intuir el nivel de complejidad de cada una de ellas.

2.3. Estructura General de un Sistema de Huella Digital Basado en el Contenido

Un sistema de huella digital está compuesto por dos etapas fundamentales; una es la extracción de características de la AFP y la otra hace referencia a la búsqueda en una base de datos de la AFP. El resultado de esta última devuelve aquellas AFP similares a la huella de consulta. En la figura 2.1 se muestra el proceso general.

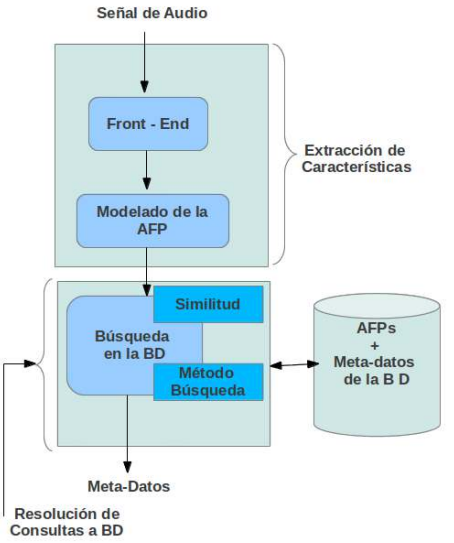


Figura 2.1: Estructura General de un Sistema de AFP Basado en el Contenido según [CBKHO5]

La primera parte del sistema, *Extracción de Características*, permite obtener la AFP a partir de la señal de audio. Para ello se extraen las propiedades relevantes basadas en el contenido de la señal y mediante alguna técnica se define la representación de la AFP. Este proceso se explica en la siguiente subsección.

Luego de obtener la AFP se ejecuta la segunda parte del sistema de huella digital, el algoritmo de búsqueda de la AFP en una base de datos de huellas digitales. Este algoritmo busca la mejor coincidencia o aquellas AFPs similares a la de consulta, para ello es necesario contar con una medida de similitud para comparar las huellas digitales. Con el objeto de acelerar las búsquedas en la base de datos, diversos métodos han sido propuestos, dado que el número de comparaciones de AFPs es alta en una gran base de datos y la similitud o cálculo de distancias es costoso de calcular. Algunos sistemas de huellas digitales utilizan una medida de similitud más simple y con ella descartan rápidamente los candidatos, otros usan una medida de similitud más precisa pero es costosa para el conjunto reducido de candidatos. Hay métodos que examinan todos los objetos de la base de datos y obtienen aquellos cuya

distancia al objeto de consulta es mínima. Otros métodos, a partir de la construcción de un índice, realizan un pre procesamiento de algunas distancias fuera de línea (off-line) y arman una estructura de datos, la cual permite reducir el número de cálculos a hacer on-line.

Para resolver esta etapa donde se tiene un conjunto de datos, AFPs, y una medida de similitud para responder a las consultas, el modelo que mejor se adapta es el modelo de espacio métrico. En el capítulo 3 se describen las características importantes del modelo de espacio métrico y los tipos de consultas que podemos realizar.

2.3.1. Extracción de Características

La extracción de AFPs se obtiene a partir de un conjunto de características perceptuales relevantes de un objeto, en este caso una señal de audio, en forma precisa y robusta. En la figura 2.2, se muestra el modelo presentado por [CBKH05], el cual consta de dos módulos bien diferenciados: el módulo *Front-End* y el módulo *Modelado de la AFP* (ver figura 2.1). El primer módulo, se encarga de calcular un conjunto de mediciones de la señal y obtener las características de la AFP. En el segundo se define la representación final de la AFP, ésta puede ser un vector, un conjunto de vectores, una serie de códigos, una secuencia de índices de Markov, entre otros. Ambos módulos son explicados en las secciones siguientes.

2.3.1.1. Módulo *Front-End*

En este módulo, la señal de audio se convierte en una secuencia de características relevantes, las cuales son la entrada al módulo *modelado de la AFP*, según la figura 2.2. La mayoría de los algoritmos de extracción de características se basan en el esquema mostrado en la figura, realizando todas o algunas fases. Cada una de ellas son detalladas a continuación:

1. En un primer paso, *fase pre-procesamiento*, la señal de audio es convertida a un formato general, es decir de estéreo a monoaural. También se incluyen las tareas de normalizar la amplitud de la señal (limitar el rango entre -1 y 1), codificar/decodificar el sistema de identificación de un teléfono celular, digitalizar la señal, entre otros.

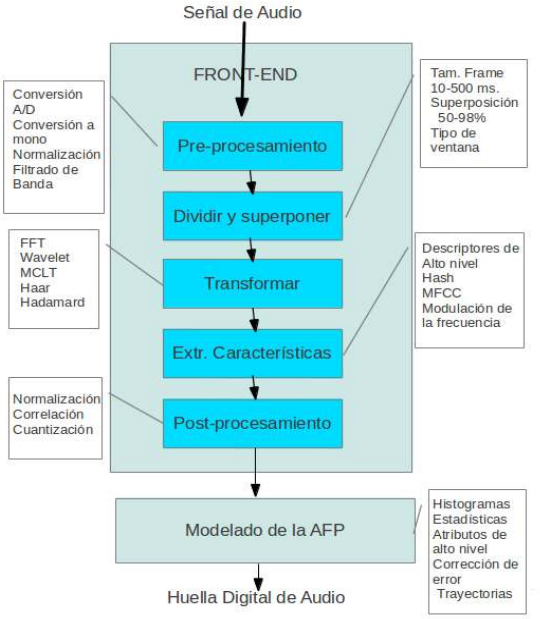


Figura 2.2: Estructura de la Extracción de Características de una AFP según [CBKH05]

2. En la *fase dividir y superponer*, la señal puede ser considerada como estacionaria en un intervalo de unos pocos milisegundos, esto es un supuesto clave en la medición de las características. Aquí la señal es dividida en *frames* de un tamaño fijo, este tamaño debe ser comparable con la velocidad de variación de los eventos acústicos adyacentes. Un frame es un segmento corto de audio y el número de frames por segundo calculados se llama *tasa de frame*. A cada frame se le aplica una función ventana para reducir al mínimo las discontinuidades en su comienzo y fin, por lo tanto, la señal está centrada en la ventana. El frame debería contener al menos dos períodos de la componente de menor frecuencia de la señal que está siendo considerada, razón por la cual el frame nunca es menor a 10 ms. Además, el frame nunca debería ser mayor que 500 ms, a fin de considerar la señal como estacionaria. Con el objeto de asegurar la robustez con respecto al desplazamiento (esto sucede cuando el dato de entrada no

está alineado con la señal que fue usada para generar la AFP), se aplica una superposición en la señal. Esto es necesario para asegurar una baja variación de las Características extraídas, el rango que generalmente se usa es entre el 50 % y el 98 % de superposición.

3. En la fase *Transformar*, se realiza una proyección de un conjunto de mediciones para un nuevo conjunto de características. Cuando la transformación es elegida adecuadamente la redundancia se reduce significativamente. Hay transformaciones óptimas en el sentido del encapsulamiento de las propiedades de la señal de audio y propiedades de no correlación como las transformaciones KL (Kerhunen-Loéve) [BYR10] o las SVD (siglas en inglés de Singular Value Decomposition) [Woo12]. Estas transformaciones son dependientes del problema y computacionalmente complejas, por esta razón las transformaciones a partir de vectores bases fijos son comunes. Existen sistemas para extraer las características de la señal en el dominio del tiempo como en [KS03], otros en el dominio de la frecuencia usando una variedad de transformaciones lineales, tales como la Transformada Discreta de Coseno (DCT) [BYR10], la Transformada de Fourier (FFT) [Woo12, Sun01], la transformada de la modulación de la frecuencia (MFT) y algunas transformadas *wavelet* como las de Haar y Walsh-Hadamard [Woo12, Sun01]. Las transformaciones basadas en el dominio de la frecuencia son las más utilizadas debido a que facilitan la compresión eficiente, la eliminación de ruido y el procesamiento posterior.
4. En la fase *Extracción de Características* se aplican transformaciones adicionales a la señal representada en el dominio tiempo-frecuencia, con el objeto de generar los vectores finales. En esta fase se encuentran una gran variedad de algoritmos, los cuales tienen por objetivo reducir la dimensionalidad y al mismo tiempo incrementar la invariancia a las distorsiones. Los algoritmos más comúnmente usados son: MFCC (siglas en inglés de Mel Frecuency Cepstral Coefficients) [Ler12], coeficientes LPC (en inglés Linear Predictive Coding) [KLT13], modulación de la frecuencia, coeficientes SFM (siglas en inglés de Spectral Flatness Measure) [SPA06], división en bandas, entre otros.

5. Fase *Post-procesamiento*: En esta fase, algunos sistemas de AFP realizan algún tipo de post-procesamiento de las características obtenidas. Hasta el momento las propiedades de la señal descriptas son medidas absolutas. Por lo tanto, con el fin de caracterizar mejor la señal se considera por ejemplo las variaciones temporales en la señal, la información de qué tan rápido esas características cambian con el tiempo.

Al finalizar este módulo se obtiene un vector de características que representa la AFP, el cual es el dato de entrada para el módulo *Modelado de la AFP*. Dependiendo de la aplicación que se está implementando se incluyen algunas o todas las etapas de este proceso propuesto por [CBKH05].

2.3.1.2. Módulo Modelado de la AFP

El módulo *Modelado de la AFP* por lo general recibe una secuencia de vectores de características calculada sobre una base frame a frame. El aprovechamiento de las redundancias en la vecindad del frame con respecto al tiempo, dentro de una señal de audio y a través de toda la base de datos, es útil para reducir aún más el tamaño de la huella digital. El tipo de modelo elegido condiciona la medida de similitud y el método de búsqueda de la AFP para favorecer la recuperación del resultado en forma rápida. Además la AFP debe ser modelada según la forma que mejor se ajusta a la aplicación para la cual es diseñada. Algunos modelos son los siguientes:

1. *Secuencias de Vectores Características*: Este tipo de AFP también se conoce como trayectorias o trazas. Las características extraídas en períodos equidistantes de tiempo simplemente se almacenan en una lista de vectores o en una tabla, donde una fila contiene la AFP de un frame (una fila por frame de la señal total). Las AFP diseñadas en [FRM94, HK02] son ejemplos de este modelo.

2. *Estadísticas:* En lugar de almacenar un vector de características, se almacena un conjunto de datos estadísticos sobre la AFP. Por ejemplo, la AFP diseñada para MPEG-7 calcula medias, varianzas, mínimos y máximos cada 32 frames. Los mínimos y máximos se

utilizan para delimitar la búsqueda y los promedios y las varianzas se emplean para la búsqueda real usando alguna medida de distancia como la distancia del Mahalanobis [MWI0]. Un ejemplo de la aplicación de este modelo, es el propuesto en [HAC+0I].

3. *Codebooks:* La secuencia de vectores de características extraídas se sustituye por un pequeño número de vectores de código representativos almacenados en un codebook, el cual representa la señal de audio. Este modelo no tiene en cuenta la evolución temporal de la señal de audio. En [AHH+0I] se presenta una AFP siguiendo este modelo.

4. *Modelos de Markov* (HMM - siglas en inglés Hidden Markov Models): Para cada señal de audio de la colección se construye un HMM. Las características extraídas de la señal de consulta son considerados como una secuencia de eventos acústicos, para luego usarlos como entrada de los HMM de las señales candidatas. Éstas últimas a su vez, informan de la probabilidad de coincidencia con la candidata. Dicha probabilidad se utiliza como una medida de proximidad para la elección de las respuestas adecuadas (señales de audio similares a la de consulta) [BMG04, BMGC04].

5. *Modelos Gaussianos* (GMM - siglas en inglés de Gaussian Mixture Models): La señal de audio es modelada por una función de densidad de probabilidad (PDF). Si asumimos que tales PDF es el resultado de una combinación de componentes gaussianas o mezclas de ellas, entonces los parámetros (media y varianza) de todos los componentes tienen que ser estimados [VA04]. La estimación de los parámetros se realiza mediante la maximización de la probabilidad de los frames realmente presentes en la señal de audio. Para esta maximización se utiliza normalmente el algoritmo de Baum-Welch o expectativa de maximización (EM) [MZ97].

6. *Modelos Basados en la Entropía:* La entropía es la información de contenido, en una secuencia, que el cerebro percibe. Esto sirvió de motivación para usar la entropía como característica perceptual en sistemas de identificación de audio. En [CI07, CICI0, CIC06] se

propone un método muy eficaz, usando la entropía de Shannon, para identificar sin error secuencias de audio sujetas a degradaciones severas.

Los modelos presentados anteriormente son algunos de los usados para obtener las características perceptuales de la señal con el objeto de construir AFPs robustas a diversas degradaciones. Estos son aplicados según la realidad que se está modelando y pueden ser incluidos en forma pura o ser una combinación de ellos. Para poder realizar esto, se necesita analizar en forma exhaustiva el costo, los pros y los contras de cada modelo. Específicamente en la siguiente sección se presentan dos AFPs basadas en la entropía siguiendo el modelo propuesto en [CI07, CICI0, CIC06].

2.4. Proceso Secuencial AFP basado en Entropía

La primera tarea a realizar por un sistema de huella digital es extraer las características a partir de la señal. La señal de audio se procesa en base a frames, es decir la señal de audio se divide en frames de igual tamaño (fase Dividir en Frame). Como dijimos antes, un frame es un segmento corto de audio, al cual se aplican varios pasos para obtener la AFP [CI07]. La figura 2.3 muestra el proceso completo realizado por cada frame y posteriormente se explica cada una de las fases del mismo.

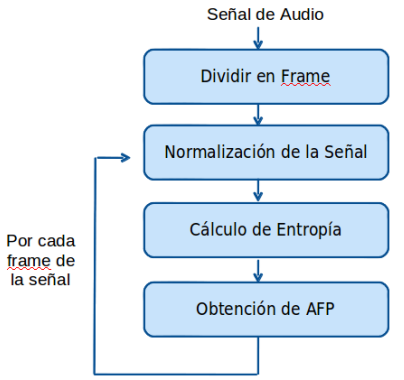


Figura 2.3: Proceso Secuencial AFP

Cada una de las etapas involucradas en el cálculo de la AFP, de cada frame, tienen las siguientes características:

- *Normalización de la señal*: Las señales de audio estéreo se con- vierten a mono-aural, utilizando frecuentemente una normalización de amplitud para hacer la AFP robusta a cambios en el volumen. Para asegurar una baja variación de las Características extraídas, una superposición del 50 % es aplicada.

- *Cálculo de Entropía*: Los valores del frame son continuos, por lo tanto se deben discretizar, es decir, los valores continuos se con- vierten en valores discretos. Para representar cada elemento de un frame se usan 8 bits, por lo tanto, cada dato discreto tendrá un valor entre 0 y 255. Con estos valores discretos, se calcula el histograma del frame para obtener la estimación de la función de densidad de probabilidad (PDF). Esta función es necesaria en el Cálculo de la Entropía. La Entropía, se calcula según la fórmula de Shannon (ecuación 2.1) [SW49]. Por consiguiente, con la entropía se obtiene la componente *i* de la AFP, donde *n* es la cantidad de componentes de la AFP.

$$-\sum_{I=1} nln(pi)x\ pi \qquad (2.1)$$

- *Obtención de la AFP:* Determina el valor de la AFP. La ecuación (2.2) establece cómo se determina el bit correspondiente de la AFP utilizando los valores de entropía de los frames n y n-1. Sólo se necesitan 3 bytes (es decir, 24 bits) para la AFP de la señal de audio.

$$F(n) = \begin{cases} 1\ if\ \ [h(n) - \ h(n-1)] > 0 \\ \\ 0\ en\ otro\ caso \end{cases} \qquad (2.2)$$

Donde la función h(x) representa el cálculo de la entropía para el frame x

En las siguientes secciones se presentan la definición de entropía y dos AFPs: *TES* y *MBSES*, las cuales utilizan la entropía en el proceso de extracción de características de la señal. Ambas difieren en las dos últimas etapas del proceso, debido a que *TES* obtiene una única AFP de la señal, mientras que m *MBSES* obtiene un conjunto de AFPs, una por cada frame, es decir que si la señal tiene *N* frames entonces se tienen *N*- 1 AFP

2.4.1. Entropía

Como se expuso anteriormente, la obtención de la AFP de la señal de audio está basada en la entropía, por esta razón es importante conocer su definición formal. La entropía de una señal es la medida de la cantidad de información que la señal posee. Si *X* es una variable al azar representando la señal, y se desea un único valor para identificarla, luego la entropía de Shannon [SW49] es una buena candidata.

Sea $X = x_1, x_2, ..., x_n$ los posibles valores de amplitud de una muestra de una señal de audio, cada x_i tiene una probabilidad p_i de ocurrir. Toda la secuencia $p_1, p_2,..., p_n$ representa a la PDF. Esto se muestra en la ecuación (2.3).

$$\sum_{i=1}^{n} p(x_i) = 1 \qquad (2.3)$$

La entropía H de una secuencia X es la información de contenido expresada en ella, es decir, es el promedio de todas las informaciones de contenido pesadas por sus probabilidades de ocurrir. Ésta se define en la ecuación (2.4)

$$H(X) = -\sum_{i=1}^{n} p(x_i)\ln\big(p(x_i)\big), \qquad (2.4)$$

La entropía de una señal es una medida para determinar cuán impredecible es la misma. Si es constante en un valor fijo k, luego su PDF (ecuación 2.3) está ubicada alrededor de k, esto es $p_i = \delta(k)$, por lo tanto su entropía es cero. Por el contrario si la señal presenta una distribución uniforme, su entropía debería

ser máxima, esto es si $p_i = \frac{1}{n}$ para n valores posibles entonces su entropía es ln(n). Esto se muestra en las ecuaciones (2.5) y (2.6).

$$H_{min} = -\sum_i \delta(k) \ln(\delta(k)) = \ln(1) = 0 \qquad (2.5)$$

$$H_{max} = -\sum_i \frac{1}{n} \ln\left(\frac{1}{n}\right) = -\ln\left(\frac{1}{n}\right) = \ln(n) \qquad (2.6)$$

En la próxima sección se muestran dos AFPs obtenidas a partir del uso de la entropía.

2.4.2. AFPs Basadas en la Entropía

Calcular la entropía de una señal, requiere la estimación de la PDF (ecuación 2.3). Tal estimación puede ser realizada por tres métodos: paramétricos, no paramétricos e histogramas. Los métodos paramétricos son adecuados cuando la distribución de probabilidad se conoce a priori y la cantidad de datos involucrados no es grande [BV00]. En los métodos no paramétricos, no se conoce la distribución de los datos, éstos son computacionalmente costosos y por lo tanto no se utilizan con frecuencia para aplicaciones de reconocimiento de patrones en tiempo real. Por último, el histograma es un método rápido y sencillo para calcular, es un buen Método cuando es necesaria la determinación on-line de la PDF de un stream de audio.

La probabilidad p_i para el valor v_i de la muestra leída en un stream de audio se calcula utilizando la fórmula de Laplace (ecuación 2.7) donde f_i es el número de veces que el valor v_i ocurre en la secuencia $x = x_1, x_2, .., x_N$; N es el tamaño del frame [Dyk99].

$$p_i = \frac{f_i}{N} \qquad (2.7)$$

La certeza del Método de histograma está garantizada por el hecho de que miles de muestras de audio se utilizan en la construcción del histograma. Además la precisión de las muestras está reducida a 8 bits solamente, por lo tanto el histograma resultante es una tabla con 256 entradas.

En [CI07, CICl0, CIC06] se han propuesto dos AFPs basadas en la Entropía, una se denomina TES (siglas en inglés de *Time-Domain Entropy Signature*) y la otra es MBSES (siglas en inglés de *Multi-Band Espectral Entropy Signature*). Ambas siguen el proceso mostrado en la figura 2.3, pero difieren en las últimas dos etapas. Éstas AFPs se describen a continuación.

2.4.2.1. Time-Domán Entropy Signature: TES

La AFP TES es obtenida a partir de la señal de audio, usando el signo de la derivada para construir un string binario a partir del cálculo de entropía.

Luego de haber dividido la señal de audio en frames (fase *Dividir en Frames* en la figura 2.3), se ejecuta la fase *Cálculo de Entropía*. Esta fase se encuentra dividida en 3 subfases, como se muestra en la figura 2.4.

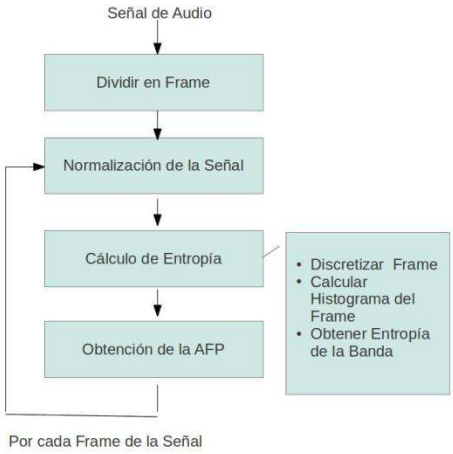


Figura 2.4: Proceso Secuencial AFP para TES

En la subtarea *Discretizar Frame* los valores continuos se convierten en valores discretos. Cada elemento de un frame se representa en 8 bits, por lo tanto tendrá un valor entre 0 y 255. Estos valores son la entrada a la tarea *Calcular Histograma del Frame*, el cual utiliza este método para obtener la estimación de la PDF (ecuación 2.3). Dicha función es necesaria en el Cálculo de la entropía. Luego de obtener un valor de entropía por cada

frame, se determina el bit correspondiente de la TES según la ecuación 2.2. Al finalizar el proceso se obtiene una AFP de N bits, donde N es la cantidad de frames que posee la señal.

Como se presentó en [CIC06], TES no sólo es compacta y fácil de calcular, también es robusta a degradaciones específicas de filtrado, escala y compresión con pérdida.

2.4.2.2. Multl-Band Spectral Entropy Signature: MBSES

La información de contenido en la señal se debe medir en la perspectiva del oído humano. La escala de Bark define 25 bandas críticas, las cuales se muestran en la tabla 2.7, cada una de ellas corresponde a una sección de la cóclea de alrededor de 1,3 mm [ZF90, Fuj96]. La banda crítica 25 se descarta debido a que sólo los oídos más jóvenes y más sanos son capaces de percibir. Si la entropía de los coeficientes espectrales correspondientes a la banda crítica b se calcula para cada frame de una señal de audio, se obtiene una secuencia de valores de entropía. Sea esta secuencia denotada como $SE_b(t)$ para $t = 0, ...,(N\text{-}1)$ y N el número de frames en la señal. Si se grafica $SE_b(t)$, se obtiene la curva espectral de entropía para la banda crítica b o simplemente la curva de SE_b , la figura 2.5 muestra las curvas de algunas SE_b.

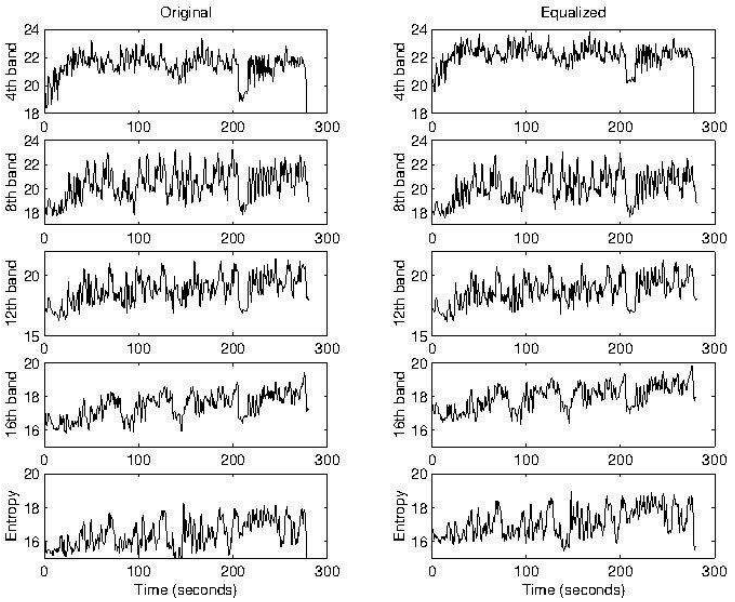


Figura 2.5: Distintas curvas según las bandas de la escala de Bark

Las curvas de SE son robustas a las degradaciones de ecualización y esta característica fue el objeto de diseño de una AFP basada en las distintas bandas espectrales de entropía. A ésta AFP se denomina MBSES.

A diferencia de TES, la fase *cálculo de entropía* mostrada en la figura 2.3, está dividida en varias subfases como se muestra en la figura 2.6. En primer lugar, el frame se divide en dos partes, la parte real y la parte imaginaria. Cada parte se discretiza, es decir, los valores continuos se convierten en valores discretos (subfase *Discretizar Frame*). Cada elemento de un frame se representa en 8 bits, por lo tanto tendrá un valor entre 0 y 255. Con estos valores discretos, se calcula el histograma del frame para obtener la estimación de la PDF (ecuación 2.3). Esta función es necesaria en el Cálculo de la entropía (subfase *Calcular Histograma del Frame*).

En la subtarea *Dividir Frame en Bandas y Calcular sus Histogramas*, el frame se divide en bandas de acuerdo a la escala de Bark, de las cuales se originan 48 histogramas, 24 bandas de la parte real y 24 bandas de la parte imaginaria.

Para cualquier banda *b*, los elementos del frame correspondiente a *b* se utilizan para construir dos histogramas,

uno para la parte real y otro para la parte imaginaria. Después de obtener el histograma complemento (subfase *Obtener el Complemento del Histograma*), el cual es la diferencia entre el histograma del frame y el histograma de la banda, el histograma resultante se utiliza para estimar la función de distribución de probabilidad. La entropía *h* de la banda *b*, se calcula según la fórmula de Shannon (ecuación 2.4), y por consiguiente la entropía de partes reales e imaginarias se calculan por separado y operan en conjunto para obtener la componente *i* de la AFP (subfase *Calcular Entropía de la Banda*).

La idea central de MBSES consiste en almacenar para cada banda un indicador si la entropía espectral es creciente o no en el frame actual. La ecuación (2.8) establece cómo el bit correspondiente a la banda *b* y el frame *n* de MBSES está determinado usando las entropías de los frames *n* y *n* - 1 para la banda *b*. Solamente 3 bytes (24 bits) son necesarios para cada frame de la señal de audio. En la ecuación 2.8 la función *hb* representa la entropía de la banda *b*

$$F(n) = \begin{cases} 1 \; if \;\; [h_b(n) - \; h_b(n-1)] > 0 \\ \\ 0 \; en \; otro \; caso \end{cases} \qquad (2.8)$$

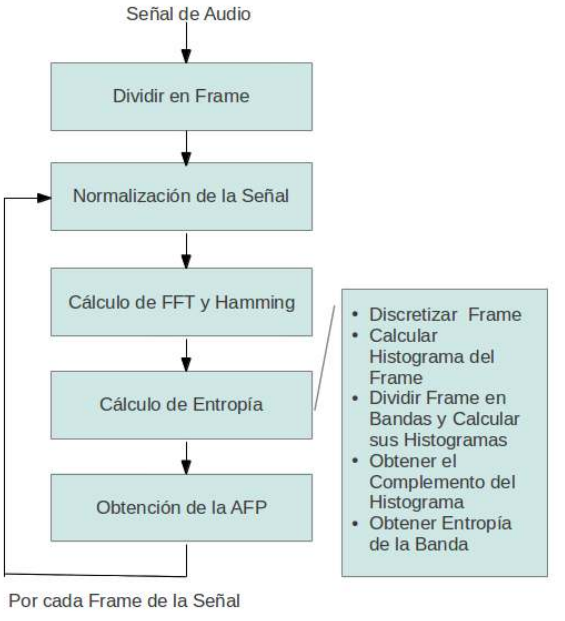


Figura 2.6: Proceso Secuencial AFP para MBSES

Por lo tanto la MBSES de una señal de audio es una matriz binaria donde cada fila representa la AFP binaria de un frame.

2.5. Resumen

Los objetos multimedia no tienen una coincidencia digital; comparados bit por bit, dos objetos considerados iguales perceptualmente pueden no coincidir digitalmente (a menos que uno sea una copia digital del otro, lo cual es un caso poco interesante). Para poder identificar a los objetos multimedia, en particular a las señales de audio, con la misma eficiencia con las que se tratan a los objetos de texto, es necesario obtener su representación, estable y persistente a las diversas degradaciones, tanto de origen natural como proveniente de ataques maliciosos. Esta representación es llamada la huella digital de la señal, AFP

Idealmente la AFP debe ser una invariante de la señal; aquellas características intrínsecas, no alterada por su constante manipulación. Su determinación permite, por ejemplo identificar objetos de audio a partir de segmentos, detectar duplicados, plagios, rotulado automático (MP3 modernos), consulta por ejemplos y filtrados en redes p2p, entre otros.

Existen diferentes formas de obtener una AFP, calcularla mediante la entropía ha demostrado ser una buena opción. Generalmente se considera a la entropía como la cantidad de información que tiene una señal. En este capítulo describimos las características a cumplir por una huella digital y los tipos existentes. Finalmente nos enfocamos en las AFPs calculada a través de la entropía de la señal, describiendo dos métodos computacionales para obtenerla, ambos con diferentes costos computacionales.

Banda (Bark)	Rango Frec. (Hz)	Ancho de Banda	Coef. Inicial	Coef. Espectral
1	0 – 100	-	2	37
2	100 – 200	100	39	37
3	200 – 300	100	76	37
4	300 – 400	100	113	37
5	400 – 510	110	150	41
6	510 – 630	120	191	45
7	630 – 770	140	236	52
8	770 – 920	150	288	55
9	920 – 1080	160	343	66
10	1080 – 1270	190	409	70
11	1270 – 1480	210	479	78
12	1480 – 1720	240	556	90
13	1720 – 2000	280	646	105
14	2000 – 2320	320	751	112
15	2320 – 2700	380	863	148
16	2700 – 3150	450	1011	161
17	3150 – 3700	550	1172	210
18	3700 – 4400	700	1382	260
19	4400 – 5300	900	1642	335
20	5300 – 6400	1100	1977	408
21	6400 – 7700	1300	2385	509
22	7700 – 9500	1800	2894	643
23	9500 – 12000	2500	3537	923
24	12000 – 15500	3500	4460	1300

Figura 2.7: Escala de Barks para la Estimación de las Bandas Críticas

CAPÍTULO 3

Espacios Métricos: Generalidades

En las últimas décadas la cantidad y variedad de información disponible digitalmente ha crecido drásticamente, en particular los objetos multimedia, entre los cuales podemos mencionar texto, audio, secuencias biológicas, imágenes, videos.

Debido a la naturaleza de estos objetos, los cuales se representan por un conjunto de características relevantes, éstos no pueden ser almacenados, recuperados e indexados como se hace con los datos en una base de datos tradicional. En este tipo de base de datos, los objetos eran recuperados de manera exacta a través de una clave que los identificaba unívocamente. Los objetos multimedia en cambio, tienen que ser recuperados por similitud. Cuando se realiza una búsqueda de un objeto multimedia, a partir de un objeto de consulta, el resultado son todos aquellos objetos similares o "próximos" al de consulta.

Los problemas de similitud se modelan a través de espacios métricos [CNBYM0l, Sam05, ZADB06] y las consultas se resuelven mediante búsquedas por similitud. En este capítulo se definen los conceptos básicos de espacios métricos y los tipos de consultas que se pueden realizar a estos espacios.

3.1. Espacios Métricos

Un espacio métrico (U, *d*) está compuesto por un universo de objetos válidos U y una función de distancia *d*: U *x* U $\rightarrow$ R definida entre ellos. Esta función determina la similitud o distancia entre dos objetos. Para poder ser considerada una métrica debe cumplir con las siguientes propiedades:

$\forall x, y, z \in$ U

- Positividad Estricta: $d(x, y) > 0$ y si $d(x, y) = 0$ entonces $x = y$
- Simetría: $d(x, y) = d(y, x)$
- Desigualdad Triangular: $d(x, z) \leq d(x, y) + d(y, z)$

El conjunto finito $X, X \subseteq U$, con tamaño $|X|=n$ se denomina base de datos y representa la colección de objetos sobre los cuales se responden las consultas.

3.1.1. Funciones de Distancia

Existen distintos tipos de funciones distancia o métricas a usar en un espacio métrico, éstas dependen de las características y el tipo de objetos de la base de datos.

En el caso de los espacios vectoriales, existen algunas distancias usadas comúnmente, cuál aplicar depende de la aplicación. Dados v y q dos vectores con dimensión m, donde $v = (v_1, v_2, \ldots, v_m)$ y $q = (q_1, q_2, \ldots, q_m)$, la métrica usada es la Minkowski de orden p o L_p, definida como:

$$L_p = d_p(v, x) = \sqrt[p]{\sum_{1 \leq i \leq m} (|v_i - q_i|^p)} \qquad (3.1)$$

donde los casos especiales de esta métrica son:

- Distancia L_1: representa la suma de las diferencias a lo largo de las coordenadas. También se la conoce como *Distancia en Bloque* o de *Manhattan* porque cuando se trabaja en dos dimensiones se corresponde con caminar entre dos puntos en una ciudad de manzanas rectangulares. Se define con la siguiente ecuación:

$$L_1 = d_1(v, x) = \sum_{1 \leq i \leq m} (|v_i - q_i|) \qquad (3.2)$$

- Distancia L_2: también conocida como *Distancia Euclideana*, corresponde a la noción de la distancia espacial. Es la longitud del segmento de recta cuyos extremos son dos puntos del plano. La expresión que la representa es:

$$L_2 = d_2(v,x) = \sqrt{\sum_{1 \le i \le m} (|v_i - q_i|^2)} \qquad (3.3)$$

- Distancia L_∞: denominada también como *Distancia Máxima*. Es una generalización de las distancias L_1 y L_2, denotándose según la ecuación 3.4. Como se puede observar se corresponde con la ecuación 3.1 pero considerando al límite para *p* cuando tiende a infinito.

$$L_\infty = d_\infty(v,x) = \max_{1 \le i \le m} (|v_i - q_i|) \qquad (3.4)$$

Cuando el tipo de objeto son palabras de un diccionario, la función comúnmente utilizada para determinar la similitud entre dos palabras es la *Distancia de Levenshtein* o *Distancia de Edición* [Lev66]. Esta función determina el número mínimo de operaciones de edición necesarias para transformar una cadena de caracteres en otra. Hay tres operaciones de edición: inserción, eliminación y sustitución de un carácter. Entre más próxima a cero es la distancia de Levenshtein más parecidas son las palabras. Por ejemplo, la distancia de edición entre las palabras *casa* y *calle* es 3 porque son necesarias 3 operaciones (dos sustituciones y una inserción) para cambiar la palabra *casa* en *calle*.

El cálculo de distancia es una operación compleja y supone un costo mayor al de la comparación con una palabra clave, como se hace en las estructuras de búsqueda tradicionales.

Las funciones de distancia presentadas anteriormente poseen un alto costo computacional; la distancia de Minkowski es de $O(n)$, siendo *n* el número de coordenadas de los vectores, y la distancia de Edición tiene $O(n * m)$, donde *n* y *m* son las longitudes de las palabras a comparar.

Como puede observarse, la evaluación de la distancia es costosa, razón por la cual se concentró el interés de los investigadores en los últimos años, quienes intentan encontrar alguna estrategia capaz de reducir el número de evaluaciones de distancia.

3.2. Búsquedas por Similitud

Los métodos tradicionales de búsqueda están orientados al tratamiento de datos con una estructura determinada, generalmente mediante atributos que reflejan las propiedades de los objetos y son representados mediante registros almacenados en la base de datos. Las búsquedas en este contexto, donde los atributos son generalmente de tipo estructurado (numérico, alfanumérico, de tiempo, entre otros), están basadas en la búsqueda exacta y sus variantes de búsqueda por rango. Ésta última se realiza entre valores determinados, restringiéndose la igualdad a unas posiciones determinadas, por ejemplo consultar por una cadena alfanumérica con algunos caracteres específicos.

Los objetos multimedia utilizados en las aplicaciones actuales son tipos de datos no estructurados, éstos pueden ser imágenes, videos, audio, documentos de texto, secuencias de ADN, entre otros. Generalmente no poseen una estructura determinada, razón por la cual una búsqueda exacta es inaplicable. Los datos multimedia presentan una gran dificultad para reproducirse en las mismas condiciones (por ejemplo: dos imágenes tomadas de una misma escena casi nunca son exactamente iguales), siendo muy sensibles a situaciones de ruido, distorsiones, entre otras degradaciones. Por estos motivos, si se desea buscar un objeto multimedia en una base de datos, se debe realizar a través de una búsqueda por similitud, es decir determinar cuáles son los elementos más semejantes, parecidos o "cercanos" a algún otro, definido como objeto de consulta.

Dado un espacio métrico (U, d), se tiene que la base de datos o colección de objetos sobre la que se realizan las búsquedas es un subconjunto finito $X \subseteq$ U de tamaño $n = |X|$. Una consulta se expresa mediante un objeto de búsqueda o de consulta $q \in X$ y un criterio de proximidad a ese objeto. El conjunto resultado es el conjunto de objetos de la colección X que cumplen dicho criterio.

En las bases de datos multimedia, las consultas de mayor interés son: la búsqueda por rango y la de los k vecinos más cercanos (k-NN) [CNBYMOI]. Otras consultas a realizar en un espacio métrico son: la búsqueda de los k vecinos más cercanos para todos los objetos de la base de datos (*all-* k-NN) y la unión por similitud o *join* por similitud. Cada una de estas consultas tiene las siguientes características:

- *Búsqueda por Rango*: Es uno de los tipos de búsqueda más intuitivos y el más importante, en función de ésta se pueden expresar el resto de las consultas [CNBYMOI]. El criterio de proximidad está determinado por el objeto de consulta q y un radio o rango r, de forma tal que la búsqueda consiste en obtener todos los objetos $x \in X$, los cuales se encuentran a una distancia de q menor o igual a r. La expresión 3.5 define la búsqueda por rango

$$R_{(q,r)} = \{x \in X / d_{(q,r)} \leq r\} \qquad (3.5)$$

Los objetos reportados o de interés son aquellos que se encuentran dentro del radio de búsqueda, el resto se descarta. El valor del radio influye directamente en el costo de la resolución de la consulta. Si r es igual a cero estamos ante el caso particular de una búsqueda exacta sobre q. La figura 3.1 muestra gráficamente un ejemplo de este tipo de consulta, considerando puntos en R^2 y distancia Euclidiana.

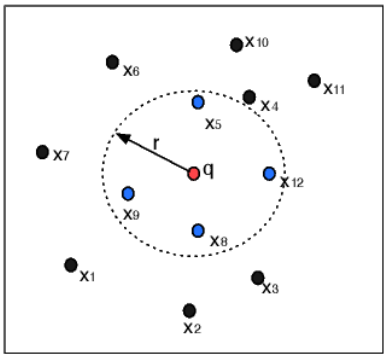


Figura 3.1: Ejemplo de una búsqueda por Rango.

- *Búsqueda de los k-NN:* Este tipo de consulta determina los k objetos más cercanos al objeto de consulta. Consiste en recuperar un conjunto de k objetos, los cuales son los más cercanos al objeto de consulta q. La expresión 3.6 define esta consulta.

$$K - NN(q) = \{A \subseteq X \,/\, |A| = k, \forall\, x \in A \;\; y \;\; \forall\, u \in (X - A), d(q,x) \leq d(q,u)\} \quad (3.6)$$

Al indicar un número determinado de vecinos a buscar y como puede haber varios objetos a la misma distancia de q, puede ocurrí que se obtenga un número mayor de candidatos a los inicialmente estipulados. En este caso, primero se eligen aquellos que están a menor distancia de la consulta, luego se van seleccionando indistintamente los siguientes hasta completar los k vecinos más cercanos. En la figura 3.2 se muestra un ejemplo de los k-NN cuando k es igual a 2

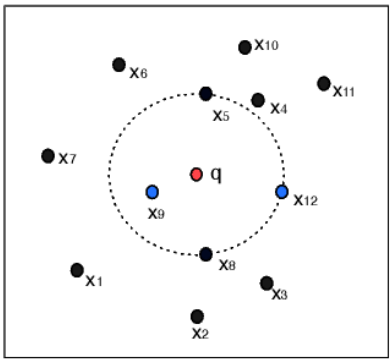


Figura 3.2: Ejemplo de una Búsqueda de los k-NN con k=2.

- *Búsqueda de los all-k-NN*: Este tipo de consulta se usa frecuentemente en aplicaciones de cluster, diseño de VLSI, reconocimiento de patrones, entre otros. Resuelve el problema de los k-NN para todos los objetos $x \in X$, es decir, se obtienen los k vecinos más cercanos de cada objeto de la base de datos. Esta consulta puede expresarse como:

$$all - k - NN = U_{x_i \in X}\, k - NN(x_i) \quad (3.7)$$

Un ejemplo de esta búsqueda se muestra en la figura 3.3 donde k es igual a 2. Esta consulta no tiene la propiedad de simetría, se puede modelar como un grafo dirigido, donde una flecha con doble sentido

entre dos objetos x_1 y x_2, significa que x_2 es uno de los dos vecinos más cercanos de x_1 y viceversa. La flecha unidireccional indica que el destino es uno de los vecinos más cercanos al origen, por ejemplo x_9 es uno de los 2-NN(x_{10}), pero x_{10} no pertenece a los 2-NN(x_9).

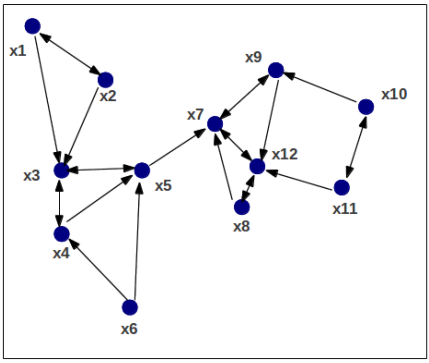


Figura 3.3: Ejemplo de una búsqueda de los all-k-NN para k=2.

- *Join por Similitud:* Este tipo de consulta es utilizada frecuentemente en minería de datos, detección de duplicados, validación de datos, entre otros. Se resuelve sobre dos conjuntos de bases de datos A y B, tal que A y B ⊆ U . La idea es encontrar todos los pares de objetos en el producto cartesiano de A x B (es decir, pares de objetos donde cada una de las componentes pertenecen a cada una de las bases de datos) que satisfacen algún predicado o criterio de similitud. Si ambas bases de datos coinciden A = B, se denomina auto-join por similitud. Algunas de las posibles variantes del join por similitud entre A y B son: *join por rango*, dado un radio o umbral r ≥ 0 se devuelven todos los pares de objetos a distancia a lo más r entre sí; *join de k pares de vecinos más cercanos*, se buscan los k pares de elementos más cercanos entre sí; o *join de k-vecinos más cercanos*, se encuentran los pares de vecinos formados por cada elemento de A con sus k vecinos más cercanos en B [PR09].

  Nos enfocamos en el *join de k-vecinos más cercanos*, el cual, más formalmente, se puede definir como: dadas dos bases de datos A, B ⊆ X, el join de k-vecinos más cercanos $A_{\bowtie kNN}B$ permite encontrar los pares de

elementos de AxB tales que $\forall$ a $\in$ A, existen k pares de la forma: $(a, x_i), \forall\, x_i \in k - NN(a) \subseteq B$ Claramente, $|A_{\bowtie kNN}B| = n\ k$ si $|A| = n$ y $|B| \geq k$

En el ejemplo de la figura 3.4 se realiza $A \bowtie_k B$, donde los objetos de la base de datos de A se representan con puntos azules y los objetos de la base de datos B se simbolizan en color verde para un valor de k = 2.

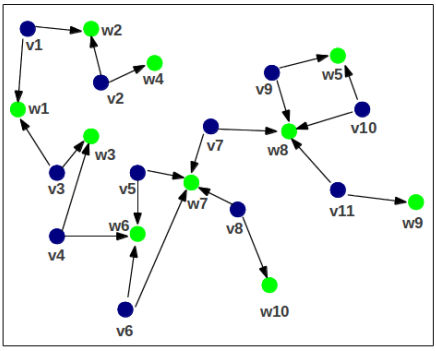


Figura 3.4: Ejemplo de una búsqueda join k-NN para k=2

Todas las consultas antes mencionadas pueden ser resueltas computacionalmente de distintas maneras, implicando distintos grados de dificultad y costo.

3.3. Estrategias Utilizadas en Espacios Métricos

Como se ha expuesto anteriormente, en la búsqueda por similitud se plantea la obtención de objetos, los cuales no son exactamente iguales al objeto de búsqueda. El cálculo de distancia entre el objeto de consulta y cada objeto de la base de datos requiere un gran costo computacional. Existen diversos métodos para resolver este problema, entre ellos se encuentran el método trivial denominado *Método de Fuerza Bruta* y aquellos que intentan reducir los cálculos de distancia mediante indexación, llamados *Métodos basados en índices*. Ambos se explican en las siguientes subsecciones.

3.3.1. Método de Fuerza Bruta

El método de *Fuerza Bruta*, también llamado búsqueda exhaustiva o *scan secuencial*, es la manera trivial de implementar la búsqueda por similitud. Consiste en comparar el objeto de consulta con todos los objetos de la base de datos, realizando un recorrido secuencial de la misma. Si bien es un método simple, no es adecuado debido al costo computacional elevado producto de todos los cálculos de distancia a realizar. Este problema se incrementa si se trabaja con bases de datos de gran tamaño.

Particularmente para la búsqueda de los k vecinos más cercanos, el algoritmo realiza las siguientes tareas:

1. Calcular las distancias entre el objeto de consulta q y cada elemento de la BD.

2. Ordenar las distancias computadas.

3. Seleccionar los k elementos, los cuales se corresponden con aquellos cuyas distancias son las k mínimas.

4. Repetir los tres pasos anteriores para todos los elementos de consulta.

El método de *Fuerza Bruta* tiene una gran complejidad si se resuelve en forma secuencial, para m consultas es $O(m*n*logn)$, considerando la comparación con todos los objetos es $O(n)$ y el ordenamiento $O(n*logn)$. Como alternativa, existen otras soluciones menos costosas, las cuales intentan reducir la complejidad, disminuyendo los cálculos de distancias haciendo un pre-procesamiento de la base de datos. Estas son presentadas en la próxima sección

3.3.2. Métodos Basados en Índices

El objetivo principal de los métodos basados en índices consiste en resolver las consultas reduciendo el número de evaluaciones de distancia. Para ello se propone realizar un pre-procesamiento de la base de datos, construyendo un índice con información, la cual será utilizada posteriormente en el proceso de búsqueda para descartar objetos del conjunto resultado sin

necesidad de compararlos directamente con el objeto de consulta y evitar así cálculos de distancia.

La complejidad de una búsqueda está dada principalmente por el número de evaluaciones de distancia necesarias para resolver una consulta. Sin embargo, hay otros factores que influyen en el costo total de una búsqueda, como son el tiempo de entrada/salida necesario para acceder al índice (si éste está en memoria secundaria y la totalidad no puede almacenarse completamente en memoria principal), y el tiempo de CPU dedicado a procesar los datos del índice. Frecuentemente estos factores adicionales se consideran despreciables, centrándose los métodos en reducir el número de evaluaciones de distancia.

Un método de indexación implica un proceso para construir una estructura de datos denominada índice, normalmente previo al proceso de consulta, y luego almacenarlo. Estos métodos requieren un costo inicial generalmente no vinculado al proceso de búsqueda.

El índice permite realizar búsquedas por similitud evitando tener que revisar toda la base de datos. Para llevar a cabo este proceso, los algoritmos utilizan la desigualdad triangular. Ésta permite dar un límite inferior a la distancia real y en consecuencia estimar si es posible descartar objetos, evitando calcular la distancia real con el objeto de consulta. Esto da una idea de la información que debería contener el índice para llevar a cabo la tarea.

Con la finalidad de evitar dichas evaluaciones de distancia, los métodos de búsqueda se basan principalmente en dividir el espacio empleando la distancia a uno o más objetos seleccionados. Para particionar la base de datos existen dos grandes enfoques: los *algoritmos basados en pivotes* y los *algoritmos basados en clustering o particiones compactas* [CNBYMOl].

Los *algoritmos basados en pivotes* realizan una preselección de objetos de la base de datos. Dichos objetos, denominados *pivotes*, sirven para filtrar objetos en una consulta utilizando la desigualdad triangular, sin medir realmente la distancia entre el objeto de consulta y los objetos descartados. El procedimiento es el siguiente:

- Sea $(p_1, p_2, ..., p_l) \in U$ un conjunto de pivotes. Se almacena para cada elemento $x \in X$, su distancia a los l pivotes $(d(x, p_1), ..., d(x, p_l))$. Dada una consulta q, se calcula su distancia a los l pivotes $(d(q, p_1), ..., d(q, p_l))$.

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

Algunos algoritmos hacen una implementación directa de este concepto, diferenciándose básicamente en su estructura extra para reducir el costo de CPU de encontrar los objetos candidatos (puntos no descartados), pero no en la cantidad de evaluaciones de distancia. Ejemplos de éstos son: SSS-Index [BFPR06], SSS-Tree [BPS+08], AESA [VR86], LAESA [MOV94], Spaghettis y sus variantes [CMBY99, NN97], FQT y sus variantes [BYCMW94] y FQA [CMN0l].

Los *algoritmos basados en particiones compactas* dividen el espacio en áreas, donde cada una tiene un centro. El índice almacena información sobre el área que representa para permitir descartar mediante la comparación únicamente de la consulta con el centro del área. Existen dos criterios para delimitar las áreas en las estructuras basadas en clustering: regiones de Voronoi o hiperplanos y radio cobertor. El primero divide el espacio usando hiperplanos y determina la partición a la cual pertenece la consulta según la región a la que corresponde. El criterio de radio cobertor divide el espacio en esferas, las cuales pueden intersectarse. Una consulta puede caer en varias regiones de Voronoi y también puede intersectar varias esferas (por radio cobertor).

Ejemplos de métodos basados en particiones compactas usando hiperplanos son: GHT y sus variantes [NVZ92, Uhl9l] y los árboles Voronoi- trees [DN88, Nol92]; y usando radio cobertor son: los algoritmos M-tree [CPZ97] y lista de clusters [CN00]. Algunas estructuras combinan ambas técnicas, como el caso del GNAT [Bri95] y el EGNAT [NUPll, Uri05] y el SAT [Nav02].

En la próxima sección se explica el método basado en el índice SAT +, una nueva y muy eficiente variante del SAT, el cual es utilizado en esta tesis como punto de referencia para comparar los resultados obtenidos.

3.4. Algoritmo SAT +

Una de las variantes más eficientes del Árbol de Aproximación Espacial (SAT por su sigla en inglés) [Nav02] es el *Árbol de Aproximación Espacial Distante*, denominado *SAT*+ [CLnRRll, Rey02]. El SAT se basa en una nueva técnica, la cual es específica de la búsqueda espacial, propuesta en [Nav02]: más que dividir el conjunto de candidatos durante la búsqueda, se trata de comenzar la búsqueda en algún punto del espacio y *acercarse espacialmente* a la consulta *q*, encontrando elementos cada vez más cercanos a ella. Cuando no se puede aproximar más, se está posicionado en el elemento más cercano a la consulta en toda la base de datos.

Las aproximaciones se realizan sólo a través de los vecinos. Cada elemento de la base de datos *a* tiene un conjunto de vecinos *N*(*a*), y sólo se permite mover a los vecinos. La estructura natural para representar esta restricción es un grafo dirigido. Los nodos son los elementos de la base de datos y están conectados a sus vecinos por arcos. Más específicamente, existe un arco desde *a* hasta *b* si es posible moverse en un solo paso. Cuando el grafo está definido adecuadamente, el proceso de búsqueda para una consulta q ∈ X es simple: se comienza la búsqueda a partir de un nodo aleatorio *a* y se consideran todos sus vecinos. Si ningún vecino está más cerca de *q* que *a* entonces *a* es el vecino más cercano a *q*. En otro caso, se selecciona algún vecino *b* más cercano a *q* que *a* y se mueve a *b*. Se puede elegir a *b* como el vecino más cercano de *q* (*best fit*) o como el primero más cercano a *q* que *a* (*first fit*). Para que el algoritmo funcione, el grafo debe contener el menor número posible de arcos, lo cual permite responder correctamente a todas las consultas. Un ejemplo de este tipo de grafo es un árbol.

Al considerar realmente un árbol, la búsqueda se inicia desde un elemento en particular, la raíz del árbol, y se combina la aproximación espacial con *"backtracking"* para responder a cualquier consulta q ∈ *U* (no sólo elementos q ∈ X), para consultas por rango y de *k*-NN [Nav02].

En las siguientes secciones se explican los aspectos básicos de *SAT* +: cómo se construye el árbol y la forma en las que se resuelven los distintos tipos de consultas.

3.4.1. Construcción del SAT+

Si se considera que la base de datos a indexar es un conjunto $X \subseteq U$, se selecciona aleatoriamente un elemento $a \in X$ como la raíz del árbol. A continuación se elige un conjunto adecuado de vecinos $N(a)$, el cual satisface la siguiente propiedad:

$$\text{Dados a y S } \forall x \in X, x \in N(a) \Leftrightarrow \forall y \in N(a) - x, d(x,y) > d(x,a)$$

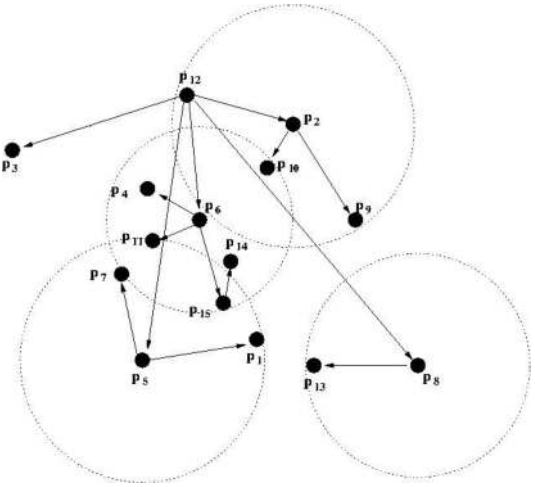
Es decir, los vecinos de *a* forman un conjunto en el cual cualquier vecino está más cerca de *a* que de cualquier otro vecino. La parte *sólo sí* (⇐)) de la definición garantiza que si es posible acercarse a algún elemento $b \in X$ entonces un elemento en $N(a)$ está más cerca de *b* que de *a*, por ser vecinos directos es decir, son todos aquellos elementos cumpliendo no estar más cerca que cualquier otro vecino. La parte "si" (⇒) apunta a obtener además sólo los vecinos necesarios [Rey02].

En este índice, una vez elegida la raíz del árbol *a*, el orden en que se consideran los restantes elementos de la base de datos *S* determina el conjunto de vecinos. En el SAT+ se utiliza un ordenamiento para los elementos de *S- {a}* y se genera un árbol, el cual permite obtener mejores costos de búsqueda gracias a la partición en hiperplanos generada por este nuevo orden beneficia la separación de las regiones y consigue que las mismas sean más compactas; es decir, los radios de cobertura son menores.

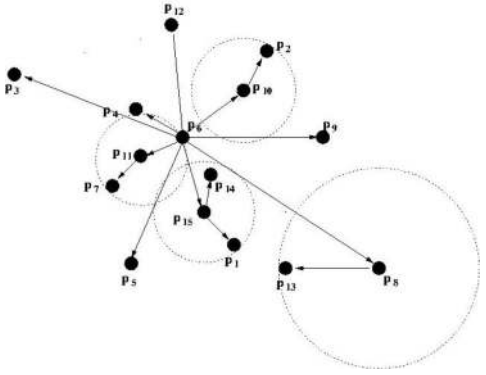
Cuando se trabaja con hiperplanos, para lograr separación de los datos, es recomendable utilizar pares de objetos alejados entre sí; tal como se documenta en [CNBYM0l] para estructuras como el *GNAT* [Bri95] y el *GHT* [Uhl9l]. Teniendo en cuenta estas observaciones, se puede asegurar una buena separación entre los hiperplanos implícitos, seleccionando el primer vecino como el elemento más alejado a la raíz. Claramente, es aconsejable seleccionar recursivamente de esta manera, en cada nivel del árbol. En la Figura 3.5 se ilustran los SAT+s obtenidos seleccionando como raíz a p_{l2}(Figura 3.5(a)) y a p_6 (Figura 3.5(b)). El procedimiento de construcción implica un árbol fijo para cada selección de la raíz.

Más detalladamente, el conjunto de vecinos inicial de la raíz *a*, $N(a)$, es vacío. Esto implica que es posible seleccionar *cualquier* elemento de la base de datos como el primer vecino.

Una vez fijado este elemento, la base de datos se divide en dos mitades por los hiperplanos definidos por la proximidad a *a* y al vecino recientemente seleccionado. Cualquier otro elemento en el lado de *a* puede ser seleccionado como segundo vecino. La selección del siguiente vecino continúa mientras la zona de la raíz no se vacíe, es decir contenga aquellos elementos más cercanos a la raíz que a los vecinos previamente seleccionados.



(a) p_{12} seleccionado como raíz



(b) p_6 seleccionado como raíz

Figura 3.5: Ejemplos de SAT+.

Ordenar los elementos en forma creciente de distancia a la raíz es sólo uno de los $n!$ posibles permutaciones de los elementos de la base de datos, siendo la utilizada por el SAT original. Cualquiera de las permutaciones de la base de datos se puede usar como un orden posible para la construcción del *SAT+*. Cada orden de inserción producirá una versión *correcta* del *SAT+*, pudiéndose usar los mismos algoritmos de búsqueda. Claramente es muy probable obtener un desempeño diferente para cada permutación. En [CLnRR14] se intenta responder a la pregunta natural ¿Cuál es la mejor permutación de una base de datos? Para responderla, en lugar de probar ciegamente con cada permutación, se intenta determinar aquel ordenamiento capaz de optimizar las reglas de descarte del SAT. Un subárbol se evita en las búsquedas usando dos reglas: hiperplanos y radios de cobertura, la característica clave en el descarte por hiperplanos es la separación entre los puntos que los definen, es más probable que la "*bola*" de consulta caiga en un lado del hiperplano, descartando así todos los objetos del lado opuesto. Una buena separación de los hiperplanos en los niveles superiores del árbol implican a su vez radios de cobertura más chicos en los niveles bajos del árbol. El *SAT+* explota estas dos observaciones mejorando significativamente el desempeño en las búsquedas utilizando una heurística exactamente opuesta a la del SAT original.

3.4.2. Búsqueda por Rango

Desde luego es de poco interés buscar sólo elementos $q \in X$. El árbol descripto puede, sin embargo, ser utilizado como dispositivo para resolver consultas de cualquier tipo $q \in U$. Se comienzan analizando las consultas por rango con radio r.

Cuando se conoce el q buscado, es decir cuando $q \in X$, se va directamente al vecino de a más cercano a q. En otro caso, cuando se busca un q_i desconocido y no se sabe cuál es el vecino de a más cercano a q_i, se deben explorar varios vecinos. Para algunos vecinos, afortunadamente, es posible determinar su irrelevancia, esto se hace porque q_i no los pudo haber elegido en la construcción del árbol si se cumple que $d(q, q_i) \leq r$.

Finalmente, el radio de cobertura $R(a)$ se usa para reducir más el costo de búsqueda. Nunca se considera un subárbol con raíz a en donde $d(q, a) > R(a) + r$, porque esto implicaría que

$d(q_i, a) > R(a)$ para cualquier q_i tal que $d(q, q_i) \leq r$. La definición de $R(a)$ implica que q_i no puede pertenecer al subárbol de a [Rey02].

3.4.3 Búsqueda de Vecinos más Cercanos

Se realizan las búsquedas del vecino más cercano simulando una búsqueda por rango, donde se va reduciendo el radio de búsqueda a medida que se obtiene más información.

Para resolver las consultas del vecino más cercano 1-NN, se comienza con $r = \infty$, y se reduce r cada vez que una distancia menor a r es obtenida. Finalmente, se informa el elemento más cercano encontrado durante toda la búsqueda.

Para la consulta por los k vecinos más cercanos k-NN se almacena todo el tiempo una cola de prioridad con los k elementos más cercanos a q vistos hasta el momento. El radio r es la distancia entre q y el elemento más lejano de la cola (infinito cuando todavía se tienen menos que k candidatos). Cuando aparece un nuevo candidato se lo inserta en la cola, pudiendo desplazar a otro elemento y por lo tanto reducir r. Al final, la cola contiene los k elementos más cercanos a q [Rey02].

3.5. Resumen

Cuando se trabaja con datos multimedia utilizar un modelo de espacios métricos es una buena opción. En situaciones donde se quiere responder a consultas sobre estas bases de datos, no tiene sentido la búsqueda exacta, la idea es medir la similitud (o diferencia) entre el elemento de consulta dado y cada uno de los objetos de la base de datos para recuperar los objetos similares a la consulta. Para determinar la similitud entre los objetos se deben obtener las distancias entre un elemento particular (consulta) y cada objeto de la base de datos. La similitud es modelada con una función de distancia, la cual debe satisfacer las propiedades de positividad, simetría y desigualdad triangular. El conjunto de elementos u objetos y la función de distancia entre ellos forman un espacio métrico.

Mientras más pequeña sea la distancia entre los objetos más similares son. Varios tipos de consultas por proximidad en espacios métricos pueden ser de interés, entre las consultas a

resolver se encuentran las consultas: por rango (recupera los elementos que están a lo más a una determinada distancia), por k vecinos más cercanos k-NN (q) (recupera los k vecinos más cercanos al elemento de consulta q en X), y todos los k vecinos más cercanos de todos los elementos de la base de datos, all-k-NN.

Existen distintas formas de resolver las consultas por similitud en espacios métricos, la más trivial pero costosa (en tiempo y/o recursos) es el método de fuerza bruta o scan secuencial (primero se calcula la distancia del elemento de consulta con todos los elementos de la base de datos para luego responderla). Una alternativa son los métodos basados en índices, los cuales permiten localizar rápidamente los elementos relevantes sin tener que realizar muchos cálculos de distancia. Un buen índice es el Árbol de Aproximación Espacial Distante, SAT+.

En este capítulo detallamos todos los conceptos básicos relacionados a espacios métricos y a la resolución de las diferentes consultas por similitud, describiendo las características de cada una. Finalmente hacemos especial referencia al método basado en índice SAT+

CAPÍTULO 4

GPGPU

La constante demanda de mayores prestaciones hizo que la industria de los mono-procesadores se encontrara en una situación límite respecto al cumplimiento de la ya conocida *Ley de Moore* sobre rendimiento del hardware. Como los mono-procesadores alcanzaron su rendimiento máximo, se debió buscar otras alternativas, una de ellas fue los multiprocesadores, computadoras de propósito general con dos o más núcleos.

Al mismo tiempo que los multiprocesadores eran aceptados por la sociedad en general y ante el auge de la industria de los videojuegos, grandes y relevantes avances tecnológicos fueron hechos en las Unidades de Procesamiento Gráfico (Graphic Processing Units, GPU) con el objetivo de liberar a la CPU del proceso de renderizado propio de las aplicaciones gráficas. La gran demanda de gráficos de alta calidad motivó el incremento de la potencia de cálculo transformando a las GPU en potentes coprocesadores paralelos. Si bien su origen fue brindar asistencia en aplicaciones gráficas, su uso como coprocesador paralelo de la CPU para resolver aplicaciones de propósito general constituye uno de los tópicos más actuales en la computación de alto desempeño. Si se observa la lista de computadoras más poderosas en el ToP500 [MSDS13], la mayoría son una combinación de elementos *multi-core* (CPU) y *many-cores* (GPU).

A partir del 2005, se comenzó a utilizar la gran potencia de cálculo y el número de procesadores de las GPUs como arquitectura masivamente paralela para resolver tareas no vinculadas con actividades gráficas, es decir utilizarlas en programación de aplicaciones de propósito general (GPGPU). En este ámbito surgieron varias técnicas, lenguajes y herramientas para la programación de la GPU como coprocesador genérico a la CPU. La evolución de éstas fue tan rápida como su popularización. Una de las herramientas más difundidas es CUDA (*Compute Unified Device Architecture*),

desarrollada por Nvidia para sus GPUs a partir de la generación G80.

En este capítulo se describe la computación de alto desempeño utilizando GPU y los conceptos básicos aplicados en esta tesis.

4.1. Computación de Alto Desempeño

La constante demanda de mayor poder computacional, no sólo de la comunidad científica y académica, ha determinado grandes avances en el hardware de computadoras; hoy en día es muy difícil encontrar computadoras con un único procesador. El software no fue ajeno a estos avances, cada día se proponen más herramientas, metodologías y técnicas que facilitan la tarea de los desarrolladores de software paralelo y les permiten obtener buen desempeño en sus aplicaciones.

La computación de alto desempeño o de alto rendimiento (High Performance Computing, HPC) se relaciona con el desarrollo de hardware y software de supercomputación. Dentro de HPC se encuentran diferentes tecnologías tal como los sistemas distribuidos y los sistemas paralelos; entre ellos los cluster de computadoras, las tarjetas gráficas y las computadoras masivamente paralelas. Esta tesis hace énfasis en HPC considerando a los sistemas paralelos y más específicamente a los sistemas paralelos en tarjetas gráficas.

Los Sistemas Paralelos implican simultaneidad espacial y temporal en la ocurrencia de eventos, todos relacionados e involucrados en la resolución de un único problema. Diseñar programas paralelos no es una tarea fácil, si bien actualmente existen herramientas que ayudan en los desarrollos, se debe tener en cuenta varias características propias de paralelismo, las cuales determinarán el desempeño o performance del sistema. Al igual que en la programación secuencial, para lograr buenos programas paralelos es necesario seguir una metodología asegurando de esa manera un software portable, escalable y con buen rendimiento.

Al referirse a un sistema paralelo, se hace referencia tanto al hardware como al software, un sistema paralelo es la combinación de ambos. El hardware tiene distintas clasificaciones, Flynn dividió a las computadoras según el flujo de datos y el flujo de sentencias [Fly95, FR96]. Si bien esta clasificación es una de las más populares, otra, no menos

popular, es aquella donde se considera la disposición de la memoria respecto a los procesadores. De ella surgen dos tipos de arquitecturas: las arquitecturas con memoria compartida y las arquitecturas con memoria distribuida [Gra03, Leo0l, Pacll, Qui02, Qui04].

El software paralelo tiene, también, su clasificación, ella depende de varios factores: la división del trabajo en las distintas tareas, la asignación de las tareas a los procesadores, la relación de los procesadores y la organización de las comunicaciones. Surgen modelos estándares, también llamados paradigmas: Paralelismo de Datos y Paralelismo de Tareas [Gra03, Pacll, Qui04].

Tanto el hardware como el software han evolucionado en forma independiente, existen aplicaciones paralelas desarrolladas para un tipo de arquitectura, y también existen computadoras paralelas fabricadas para resolver un tipo específico de aplicación. En las próximas secciones abordaremos un caso de arquitectura especial: la Unidad de Procesamiento Gráfico, mostrando su capacidad de cómputo paralelo y las características básicas del modelo de programación asociado.

4.2. Unidad de Procesamiento Gráfico: GPU

Las unidades de procesamiento gráfico, GPU, se han convertido en una parte integral de los sistemas actuales de computación. El bajo costo y marcado incremento del rendimiento permitieron su fácil incorporación al mercado. En los últimos años, su evolución implica un cambio, deja de ser un procesador gráfico potente para convertirse en un coprocesador apto para el desarrollo de aplicaciones paralelas de propósito general con demanda de anchos de banda de procesamiento y de memoria sustancialmente superiores a los ofrecidos por la CPU. La rápida adopción de las GPU como computadora paralela de propósito general se vio favorecida por el incremento tanto de sus capacidades como de las facilidades y herramientas de programación. Actualmente la GPU se ha posicionado como una alternativa atractiva a los sistemas tradicionales de computación paralela.

En esta sección se describen las características fundamentales de las GPUs, como fueron sus inicios, su filosofía de diseño y todos los aspectos relevantes relacionados

a obtener mejores soluciones paralelas con buen desempeño. Finalmente se realiza un análisis de la Computación Paralela de Propósito General sobre GPU (GPGPU), mostrando su evolución desde las primeras GPUs hasta las actuales.

4.2.1. CPU y GPU

La evolución de los sistemas de computación con multiprocesadores ha seguido dos líneas de desarrollo: las arquitecturas *multi-core* (multi-núcleos) y las arquitecturas *many-cores* (muchos-núcleos o muchos-cores). En el primer caso, los avances de la arquitectura se centraron en el desarrollo de mejoras con el objetivo de acelerar las aplicaciones, por ejemplo la incorporación de varios núcleos de procesamiento. Ante los límites físicos alcanzados por las computadoras personales, la industria tomó la idea de las supercomputadoras existentes e incorporó más procesadores a sus desarrollos. Es así que surgieron las computadoras de 2, 3, 4, 8 y más procesadores por unidad central (multi-core). Los multi-cores se iniciaron con sistemas de 2 núcleos y con cada generación se duplicó este número, actualmente el procesador Intel Core i7 cuenta con versiones de 2 a 8 núcleos y el Intel Xeon E7 con 10 núcleos. En [HKM08, SL05] se enuncian las características fundamentales de las nuevas arquitecturas, entre las cuales se encuentra el poder de procesamiento. En la actualidad ya no existen computadoras con un único procesador.

En el caso de las arquitecturas many-cores, los desarrollos se centraron en optimizar el desempeño de las aplicaciones. Dentro de este tipo de arquitectura están las GPUs. En su inicio, la primer GPU, la GeForce 256, se caracterizó por superar ampliamente a sus antecesores, en ella se aumentó el número de pipelines y se podían realizar cálculos de iluminación y de transformación de la geometría (T&L) como así también la adición de características de compensación de movimiento en formato MPEG-2. Con cada nueva generación, los núcleos se duplicaron, la actual GeForce GTX 780 Ti cuenta con 2880 cores desde sus comienzos, este tipo de arquitecturas many-core le llevó ventaja a los procesadores de propósito general multi-cores, principalmente respecto a las mejoras en la performance, a partir del 2009 la diferencia de las velocidades provistas por ambas arquitecturas es de 10 a 1 (GPU-CPU), 1 TB versus 100GB.

Uno podría preguntarse por qué existe una brecha tan grande entre el rendimiento de una CPU multi-core y una GPU many-core. La respuesta la tiene la diferencia en la filosofía de diseño de ambos. Las optimizaciones de las arquitecturas multi-core son hechas para proveer mejor desempeño a las soluciones secuenciales, es por ello que las optimizaciones se basan en, por ejemplo, proveer lógica de control compleja para la ejecución paralela de código secuencial o incluir memorias caché más rápidas y grandes a fin de disminuir la latencia de las instrucciones. Si bien la intención es muy buena, logrando tener computadoras rápidas, con 4 o más núcleos, éstas no necesariamente mejoran la performance de las aplicaciones.

La filosofía de diseño de las arquitecturas many-cores, como la GPU, y sus avances están regidos por la industria del videojuego y su constante demanda de mejores prestaciones. La idea subyacente es optimizar el throughput de muchos threads ejecutando en paralelo, de manera tal que si alguno de ellos está esperando por la finalización de una operación, se le asigne trabajo y no permanezca ocioso. Las memorias caché son pequeñas, su función es ayudar a mantener el ancho de banda definido para todos los threads paralelos. Estas características determinan por ello que la mayor parte de la arquitectura está dedicada a cómputo y no a técnicas para disminuir la latencia. En la figura 4.1 se muestran las diferencias de filosofías en ambos desarrollos.

Figura 4.1: Arquitectura CPU vs Arquitectura GPU

Otro punto de discrepancia entre ambos tipos de arquitecturas es el ancho de banda de la memoria, las GPUs mantuvieron siempre una brecha en el ancho de banda diez veces superior al de las CPUs contemporáneas. Esto obedece a que las arquitecturas de propósito general deben optimizar el ancho de banda para atender a todas las aplicaciones,

operaciones de entrada/salida y funciones del sistema operativo coexistentes en el sistema. Por el contrario en la GPU con su modelo de memoria más simple, es más fácil lograr mayor ancho de banda de memoria. Uno de los más recientes chip de Nvidia GeForce GTX 780 Ti soporta alrededor de 336 GB/s, mientras que para los Intel Xeon E7 en multi-core, el ancho de banda de la memoria es de 102GB/s.

Como se desprende de lo expuesto, la filosofía de diseño para las arquitecturas de las CPUs y las GPUs es distinta, estas últimas son diseñadas como computadoras especializadas en cómputo numérico. Es de esperar que las aplicaciones secuenciales trabajen bien en las CPUs, mientras que aquellas con intensivo cómputo numérico lo hagan mejor en la GPU. Poder contar con una arquitectura de trabajo híbrida permite aprovechar las ventajas de la CPU y las GPUs.

Además de las prestaciones ofrecidas por las arquitecturas, otros dos factores que influyen en la selección de una arquitectura para una determinada aplicación son: la aceptación popular y las herramientas adecuadas para el desarrollo de software. En el primer caso se refiere a la presencia en el mercado. Así como las CPUs existentes hoy en el mercado son multi-cores (es muy raro encontrar computadoras con un único core), la mayoría de los sistemas de computación tienen una GPU incorporada. Esto hizo que arquitecturas masivamente paralelas de bajo costo lleguen a la mayoría de los usuarios y, en consecuencia, se piense en usarlas para el desarrollo de soluciones paralelas de problemas comunes. Anteriormente el alto costo de las computadoras paralelas determinaba que los desarrollos fueran exclusivos de los gobiernos, las grandes empresas o los ámbitos académicos. Con la presencia de GPU en la mayoría de los sistemas de computación, esto cambia.

Respecto a los desarrollos de software, todas las características de las nuevas arquitecturas multi-core no tienen sentido si se continúan desarrollando aplicaciones siguiendo la metodología de programación secuencial von Neumann [BGvN46]. Si bien las aplicaciones se ejecutan más rápido en las nuevas tecnologías, éstas no aprovechan todo el potencial de la arquitectura subyacente. Todo lo contrario ocurre con aquellas aplicaciones desarrolladas siguiendo otros modelos de computación como es el modelo paralelo.

Como se mencionó antes la industria de las computadoras basó su evolución en dos aspectos, por un lado en las prestaciones que ofrecían al usuario común, lo cual determinó

la evolución de la computadora personal. Por el otro, se basó en ofrecer mejor performance en los ambientes de supercomputadoras. En el primer caso, la evolución desde la década de los 80, y por 30 años, estuvo marcada por el incremento de la velocidad del reloj del procesador, la cual fue desde 1MHz hasta llegar a velocidades que oscilan entre 1 y 4 GHz. Respecto a los avances en las supercomputadoras si bien obtuvieron ganancias de los avances de la computación personal, la demanda de mayor velocidad fue resuelta mediante la incorporación de más procesadores a su supercomputadora, llegando a tener miles de ellos; y de la inclusión de otros dispositivos más simples y con mayores prestaciones, como es el caso de la GPU. Varias de las diez supercomputadoras más rápidas del mundo [MSDS13], cuentan entre sus componentes con GPUs conectadas por una red de alta performance, siendo esto un indicador de la potencialidad y competitividad de la GPU en la computación de alta performance

4.2.2. Evolución Histórica de la GPU

La historia de las GPUs se inicia en la década del 60, cuando se pasa de la impresora como medio de visualización de resultados a los monitores. Si bien las primeras tarjetas gráficas tuvieron capacidades muy reducidas, su crecimiento no se detuvo con el correr de los años. Así como la velocidad del procesador avanza a través de dos líneas: incrementando la velocidad del reloj y/o incrementando el número de cores, la evolución de las unidades de procesamiento gráfico tuvo su origen a inicio de los 80 cuando se inicia la gran demanda de mejores interfaces gráficas por parte de los sistemas operativos, por ejemplo de Microsoft Windows. Esto implica que a principios de los 90 se comenzara a vender aceleradores gráficos 2D para computadoras personales.

La evolución de las tarjetas gráficas dio un giro importante en 1995 con la aparición de las primeras tarjetas 2D/3D, fabricadas por Matrox, Creative, S3 y ATI, entre otros. Dichas tarjetas cumplían el estándar SV-GA, pero incorporaban funciones 3D. En 1997, *3dfx* lanza el chip gráfico Voodoo, con una gran potencia de cálculo, así como nuevos efectos 3D (Mip Mapping, Z-Buffering, Antialiasing, entre otros [FHvD+13, McC06a]). A partir de allí los desarrollos no se detuvieron, día a día la potencia de cada nuevo producto fue mayor llegando al

punto que el puerto PCI-Express mediante el cual se conecta la GPU a la CPU (host) constituyera un cuello de botella por su bajo ancho de banda.

Al mismo tiempo, en la comunidad profesional y mediante la empresa Silicon Graphics, se introduce al mercado el uso de gráficos 3D en una gran variedad de ámbitos. Se desarrolla la librería openGL para ser usada como método independiente de la plataforma y poder escribir aplicaciones gráficas 3D [SG09].

La demanda de aplicaciones gráficas 3D tuvo un gran crecimiento, el cual fue alentado primero, por el desarrollo de juegos en primera persona, FPS [WFH04] como *Doom, Duke Nukem 3D* y *Quake*, los cuales no sólo le dieron el puntapié para los desarrollos gráficos sino que constantemente demandaron mayores prestaciones para lograr mayor realismo en la interface.

Compañías como Nvidia, ATI Technologies y 3fdx Interactive inician una competencia feroz para lograr buenos aceleradores gráficos. En 1999 Nvidia lanza al mercado la tarjeta gráfica Geforce 256, la cual permitía realizar transformaciones e iluminación a través del hardware, además de brindar mejores condiciones de visualización. Este desarrollo es considerado la piedra basal para los posteriores desarrollos en tarjetas gráficas. La siguiente generación de Nvidia constituyó un gran paso en la tecnología de las GPUs, fue considerada la primer GPU con implementación nativa de la primera versión de DirectX8 [Sin01]. Por primera vez los desarrolladores tienen el control de la computación a realizarse en la GPU.

Desde 1999 hasta 2002, Nvidia domina el mercado de las tarjetas gráficas con las GeForce. En ese período, las mejoras se orientaron hacia el campo de los algoritmos 3D y la velocidad de los procesadores gráficos. Sin embargo, las memorias también necesitaban mejorar su velocidad, por lo que se incorporaron las memorias DDR [JNW07] a las tarjetas gráficas. Las capacidades de memoria de video en esa época pasan de los 32 MB de las Geforce2 a los 64 y 128 MB de la GeForce4.

A partir del 2006, Nvidia y ATI (comprada por AMD) constituyen los competidores directos, repartiéndose el liderazgo del mercado con sus series de chips gráficos GeForce y Radeon, respectivamente.

4.2.3. GPU y Computación de Alto Desempeño

La GPU desde sus inicios fue un procesador con muchos recursos computacionales (ver figura 4.1). Actualmente ha adquirido notoriedad por su uso en aplicaciones de propósito general, pasó de ser un procesador con funciones especiales a ser considerado un coprocesador masivamente paralelo, capaz de ser la arquitectura base para ejecutar aplicaciones paralelas.

Aunque surgía como un coprocesador paralelo constituido por varios pipelines gráficos donde cada etapa resolvía un problema específico, a partir de la G80 de Nvidia o de la serie R600 de ATI se adopta la solución de crear arquitecturas unificadas a nivel de procesadores o *shaders*. En este tipo de arquitecturas, no existe ya división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Cada unidad de procesamiento que la integra es capaz de trabajar tanto a nivel de vértice como a nivel de fragmento, sin estar especializado en el procesamiento de ninguno de los dos en concreto. Dichas unidades reciben el nombre de procesadores de *Stream* (*Stream Processors*)

En la arquitectura unificada, el número de etapas del pipeline se reduce de forma significativa, pasa de un modelo secuencial a un modelo clásico. El pipeline clásico utiliza tipos de shaders distintos a través de los cuales los datos fluyen de forma secuencial. En la arquitectura unificada, con una única unidad de shaders no especializada, los datos que llegan a la misma (en forma de vértices) en primera instancia son procesados por la unidad, enviando los resultados de salida a la entrada para ser procesados nuevamente, en cada ciclo se realiza una operación distinta. De esta manera se imita el comportamiento del pipeline clásico. La acción se repite hasta que los datos han pasado por todas las etapas del pipeline. Finalmente son enviados hacia la salida de la unidad de procesamiento.

El cambio de arquitectura implica también un cambio en el pipeline gráfico: con la arquitectura unificada, ya no existen partes especificas del chip asociadas a una etapa concreta del pipeline, sino que una única unidad central de alto rendimiento será la encargada de realizar todas las operaciones, sea cual sea su naturaleza.

4.2.4. GPGPU: Computación de Propósito General en GPU

Las GPUs comenzaron siendo pipelines de procesamiento gráfico con funciones fijas. Con el paso de los años, estos chips se fueron haciendo más programables. Entre los años 1999 y 2000, científicos e investigadores de disciplinas como el diagnóstico por imagen o electromagnetismo, empezaron a usar las GPUs para aplicaciones de cálculo de propósito general, descubriendo que su gran rendimiento en operaciones de punto flotante implicaba un extraordinario aumento de la velocidad de ejecución en la gran variedad de aplicaciones científicas. Este fue el nacimiento de un nuevo concepto denominado GPGPU o GPU de propósito general.

Dadas las características antes enunciadas de la GPU, ésta constituye una alternativa válida a las arquitecturas masivamente paralelas. Cada día más desarrolladores de software paralelo las están teniendo en cuenta como arquitectura paralela para computación de propósito general, es decir utilizar el hardware formulado para aplicaciones gráficas en cualquier otro tipo de aplicaciones. Aunque las operaciones son las mismas, la terminología de la computación gráfica es diferente a la de la computación de propósito general.

En [Mar05] se brinda una excelente descripción de la evolución de la programación de la GPU desde su utilización para problemas de gráficos, hasta su empleo en la programación de propósito general, considerando como se resolverá teniendo en cuenta el pipeline gráfico y como se logra hoy sobre las GPUs actuales en forma sencilla y directa. Los tres casos son:

1. *Programación de una GPU para Aplicaciones Gráficas*:
   En este caso se hace referencia a los aspectos programables del pipeline gráfico de la GPU. Para el desarrollo de una aplicación gráfica se deben realizar los siguientes pasos:
   - *a)* El programador especifica la geometría que cubre una región en la pantalla. La rasterización genera un fragmento en cada pixel cubierto por la geometría.
   - *b)* Cada fragmento es sombreado por el programa de fragmentos.
   - *c)* El programa fragmento calcula el valor del fragmento mediante una combinación de

operaciones matemáticas y lecturas en la memoria global desde una memoria de textura.

d) La imagen resultante de las acciones anteriores se puede utilizar como textura para futuros pasos a través del pipeline gráfico.

2. *Programación de propósito general sobre las primeras GPUs:*

   La computación de propósito general se llevaba a cabo siguiendo los mismos pasos en el pipeline gráfico, pero usando diferentes terminologías. Si se considera como aplicación la simulación de fluidos sobre una malla: en cada paso del tiempo, se calcula el estado del líquido de cada punto de la malla para el siguiente instante considerando el estado actual del punto y de sus vecinos. Los pasos a seguir son:

   a. El programador especifica una primitiva geométrica para cubrir todo el dominio de la computación. El proceso de rastering genera un fragmento en cada pixel cubierto por la geometría definida. Para el ejemplo, la geometría debe cubrir una malla de fragmentos del mismo tamaño que el dominio de la simulación de fluidos considerada.
   b. Cada fragmento es sombreado mediante un programa SPMD de fragmentos de propósito general. Según el ejemplo considerado, en forma paralela se ejecuta el mismo programa a cada punto de la malla para actualizar su estado.
   c. El programa de fragmentos calcula el valor del fragmento mediante la combinación de operaciones matemáticas y accesos a la memoria global para la obtención (gather) de los resultados. En el ejemplo, cada punto de la malla accede al estado de sus vecinos para computar su próximo estado
   d. Los resultados de una etapa son almacenados en la memoria global y pueden ser usados para el cómputo de los siguientes estados. En el ejemplo, el estado actual de los fluidos será utilizado para el cómputo de los próximos estados.

3. *Programación de propósito general sobre GPUs modernas:*
   Entre las principales dificultades a las que se enfrentaron los programadores de aplicaciones de propósito general en GPU (GPGPU) fue pensar en cómo resolver una aplicación que no tiene nada en común con los gráficos y programarla con APIs gráficas. Los programas deberían seguir el pipeline gráfico como describimos en el punto 2, no se podía acceder a estados internos del pipeline sin antes pasar por los anteriores. Se propusieron muchos entornos de programación, los cuales facilitaban la programación de aplicaciones no gráficas para GPU, entre ellos podemos encontrar BrookGPU [BFH+04], Sh [MTP+ 04], Microsoft's Accelerator [TP006], RapidMind [McC06b], PeakStream [Pea06], CTM [Hen07] de AMD. Hoy uno de los más populares y al cual se hará referencia en esta tesis es CUDA de Nvidia [Farll, Hwull, KHl0b, NVil3b, Sanl0]. Las aplicaciones de propósito general para GPU se estructuran de la siguiente manera:

   a) El programador define el dominio de la computación como una estructura grid de bloques de threads.
   b) Un programa de propósito general es ejecutado por cada thread en paralelo sobre distintos datos según el modelo de programación SPMD.
   c) Cada thread realiza cálculos mediante una combinación de operaciones matemáticas y de accesos de escritura y lectura en la memoria global. La diferencia con las dos formas de programación anteriores está en que el mismo buffer puede ser usado tanto para leer como para escribir, por ejemplo se pueden programar algoritmos in-place.
   d) Los resultados de la operación que se guardan en la memoria global pueden ser usados por otras computaciones de la misma aplicación en la GPU.

El modelo de programación propuesto es muy poderoso por las siguientes razones:

- Permite al hardware tomar ventaja del máximo paralelismo de datos de la aplicación. Esto se logra

mediante la especificación explicita del paralelismo en el programa.

- Ofrece un balance entre la generalidad en la programación a costa de algunas restricciones como son el modelo de programación SPMD, la comunicación de los datos entre las unidades de computación (threads), entre los kernels (funciones ejecutadas en la GPU e invocadas desde el host), los mecanismos de sincronización y la jerarquía de memoria.
- Elimina la complejidad que enfrentaban los programadores de las primeras GPUs, los cuales deberían programar su aplicación de propósito general utilizando la interfaz grafica. Los programas actuales son expresados en un lenguaje de programación familiar, generalmente resultado de la extensión de un lenguaje secuencial, C por ejemplo [KR88]. Además son más simples y fáciles de construir y depurar, llegando a tener algunas herramientas que favorecen estas tareas.

La adopción de los sistemas CPU-GPU obedece principalmente a que comparados con las supercomputadoras clásicas (mainframes) estos proveen:

- Un poder de cálculo comparable con supercomputadoras y hardware de bajo costo.
- Bajo costo de mantenimiento.
- Alta escalabilidad. Además de las características ya enunciadas de las arquitecturas, varios sistemas CPU-GPU pueden ser conectados con redes de alta performance
- Reducido costo de programación y portabilidad. Aunque la CPU y la GPU están en constante avance, tanto el código secuencial como el paralelo desarrollado para arquitecturas anteriores, se pueden seguir ejecutando en las nuevas versiones.

Desde la modificación de la arquitectura propuesta por Nvidia, donde la GPU se convirtió en un dispositivo totalmente programable, y con el desarrollo de herramientas como CUDA para su programación mediante la extensión de lenguajes de alto nivel como C, C++ y Fortran [Geh96], la GPU es considerada una arquitectura apta para resolver problemas masivamente paralelos.

4.3. Programación de GPUs: CUDA

Como se mencionó antes, en su inicio la GPU fue usada para aplicaciones específicas: aplicaciones gráficas, tridimensionales o videojuegos. En los últimos años su evolución permitía el uso en un sin número de soluciones computacionales altamente paralelizables, constituyéndose en una alternativa válida a las supercomputadoras. El éxito de su notable expansión se basa fundamentalmente en su gran potencia de cálculo, bajo costo, reducido consumo relativo a su poder computacional y la posibilidad de complementarse con la CPU. Todas estas características le permiten comportarse como un coprocesador para resolver tareas altamente paralelas, mientras que el código menos paralelizable puede continuar ejecutándose en la CPU.

Si bien existen diferentes alternativas para programar en GPU, CUDA (*Compute Unified Device Architecture*) es uno de los más populares. Esto se debe a que una de las placas más ampliamente utilizada es la tarjeta Nvidia y CUDA es un kit de programación en C desarrollado para estas placas. CUDA ha sido diseñado para simplificar el trabajo de sincronización y comunicación entre threads en la GPU, y entre la CPU y la GPU. CUDA define: una arquitectura de GPU, un modelo de programación asociado y un modelo de ejecución de los programas CPU-GPU.

Antes de CUDA la programación de GPU no era tan simple, ella implicaba conocer detalles de la arquitectura del pipeline gráfico y utilizar APIs como de OpenGL [SWH13] o DirectX [RB99, Wal10].

En el año 2006, NVIDIA presenta a CUDA para su última generación de tarjetas gráficas, la serie 8 (G80). Ésta propone una filosofía integradora, con un lenguaje de programación genérico como C y la arquitectura paralela de una GPU, desvinculándose del pipeline gráfico. La tecnología CUDA permite considerar a la GPU como una arquitectura paralela para la resolución de problemas de propósito general. El desarrollo de dichas aplicaciones paralelas es posible, principalmente, por dos razones:

1. Las tarjetas gráficas NVIDIA, por las características antes enunciadas, son una componente común en la mayoría de las computadoras personales actuales.

2. Es de fácil aprendizaje, más para aquellos programadores cercanos a lenguajes C o C++.

En esta sección introducimos la programación de la GPU mediante CUDA, detallando las características básicas de la arquitectura CUDA y del modelo de programación: *kernel*, *threads*, *bloques*, *grid*, la jerarquía de memoria y las sincronizaciones.

4.3.1. Arquitectura de GPU según CUDA

CUDA presenta a la arquitectura de la GPU como un conjunto de multiprocesadores MIMD (Múltiple Instrucciones-Múltiple Datos) [Coo13]. Cada multiprocesador posee un conjunto de procesadores SIMD (Simple Instrucción-Múltiples Datos). Respecto a la memoria, existen numerosos modelos conviviendo en esta arquitectura: cada procesador SIMD posee una serie de registros a modo de memoria local (sólo accesible por el procesador), a su vez cada multiprocesador posee una memoria compartida o *shared* (accesible por todos los procesadores SIMD del multiprocesador) y finalmente la memoria global, a la cual acceden todos los multiprocesadores y, por ende, todos y cada uno de los procesadores SIMD. En la figura 4.2 esquematizamos la arquitectura de la GPU para CUDA.

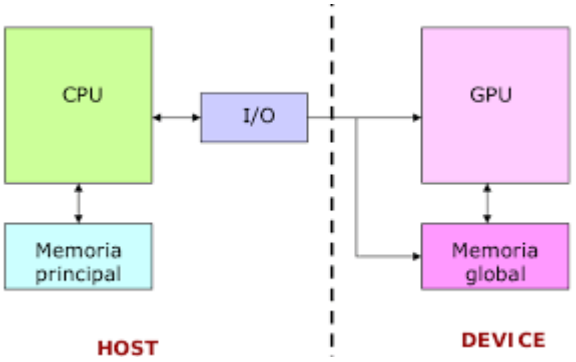


Figura 4.2: Arquitectura CUDA de la GPU

4.3.2. Modelo de Programación CUDA

CUDA propone un modelo de programación SPMD con funcionalidades de procesamiento de vector [Fly95][Leo01]. La programación de GPU se realiza a través de una extensión del lenguaje estándar C/C++ con constructores y palabras claves. La extensión incluye dos características principales: la organización del trabajo paralelo a través de threads concurrentes y la jerarquía de memoria de la GPU con sus diferentes costos de acceso.

Los *threads* en el modelo CUDA son agrupados en *Bloques*, los cuales se caracterizan por:

- El tamaño del bloque: es la cantidad de threads que lo componen y lo determina el programador
- Todos los threads de un bloque se ejecutan sobre el mismo SM.
- Los threads de un bloque comparten la memoria *shared*, la cual pueden usar como medio de comunicación entre ellos.

Varios bloques forman un *Grid* y los threads de diferentes bloques de un grid no se pueden comunicar entre sí, esto permite que el administrador de bloques sea rápido y flexible, no tiene en cuenta el número de SM utilizados para la ejecución del programa.

Además de las variables en la memoria shared, los threads tienen acceso a otros dos tipos de variables: *Locales* y *Globales*. Las variables locales son privadas a cada thread. Las variables globales residen en la memoria DRAM de la tarjeta, se diferencian de las locales en que pueden ser accedidas por todos los threads aunque pertenezcan a distintos bloques. Esto lleva a una manera de comunicación global de los threads. Como la memoria DRAM es más lenta que la memoria shared, los threads de un bloque se pueden sincronizar mediante una instrucción especial, la cual es implementada en memoria shared. La figura 4.3 muestra la relación de los threads y la jerarquía de memoria.

Además de la memoria *local*, *shared* y *global*, existen dos espacios adicionales de memoria de sólo lectura, ambos pueden ser accedidos por todos los threads de un grid. Éstas son la *memoria de constantes* y la *memoria de texturas*, ambas optimizadas por caches y para determinados usos.

En un programa CUDA se diferencian dos ámbitos: el procesador (CPU) y el dispositivo (GPU). El host es el ordenador al cual está conectada la tarjeta gráfica y será quien rija su comportamiento. El dispositivo es la tarjeta gráfica. La comunicación de datos entre el host y el dispositivo se lleva a cabo a través de la memoria global, de constante y de textura. Toda la comunicación entre la CPU y la GPU se realiza a través del *PCI-Express* (I/0 en la figura 4.2).

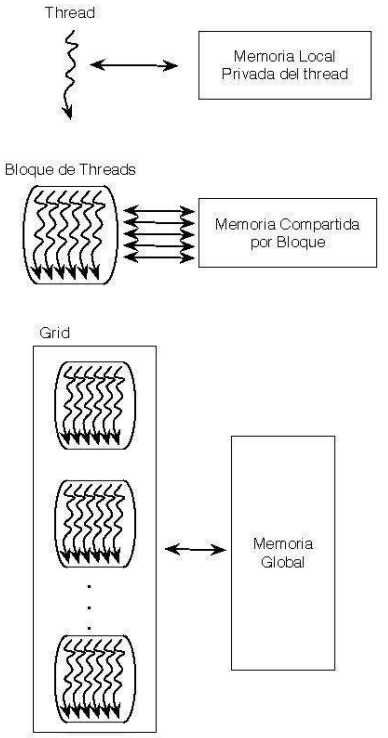


Figura 4.3: Threads y jerarquía de memoria

4.3.3. Generalidades de la Programación con CUDA

No todos los problemas pueden ser resueltos en la GPU, los más adecuados son aquellos que pueden resolverse mediante la aplicación del paradigma paralelo de datos, es decir aplican la

misma sentencia o secuencia de código a todos los datos de entrada. Se puede decir que una solución de un problema en GPU será más ventajosa respecto a la solución en la CPU si la aplicación tiene las siguientes propiedades:

- El algoritmo tiene un orden de ejecución cuadrático o superior: El tiempo necesario para realizar la transferencia de datos entre la CPU y la GPU tiene un gran costo, el cual no suele verse compensado por el bajo costo computacional de un método lineal.
- Es mayor la carga de cálculo computacional en cada thread: Para compensar el tiempo de transferencia de información es conveniente que cada thread posea una carga computacional considerable.
- Es menor la dependencia entre los datos para realizar los cálculos, esto es posible si cada SM sólo necesita de los datos de su memoria local o compartida y no necesita acceder a memoria global, la cual tiene un acceso más lento.
- Es menor la transferencia de información entre CPU y GPU. La situación óptima es cuando la transferencia sólo se realiza una vez, al comienzo (desde la CPU a la GPU con los datos de entrada) y al final del proceso (desde la GPU a la CPU para obtener los resultados). Es bueno no tener transferencias intermedias, ya sea de resultados parciales o datos de entradas intermedios.
- No existen secciones críticas, es decir, varios procesos no necesitan escribir en las mismas posiciones de memoria, las lecturas de memoria global y compartida puede ser simultánea, pero las escrituras en la misma posición plantean un acceso a un recurso compartido, lo cual implica contar con mecanismo de acceso seguro a los datos. Este proceso hace más lenta la solución del proceso global

Además, es necesario que las estructuras de datos en la aplicación secuencial se adapten o puedan transformarse a estructuras más simples del tipo matriz o vector a fin de poder ser compatibles con las estructuras administradas por la GPU.

Como se mencionó en la sección anterior, el modelo de programación CUDA asume que los threads CUDA se ejecutan en una unidad física distinta, la cual actúa como coprocesador al procesador. Como CUDA C es una extensión del lenguaje de

programación C, permite al programador definir funciones C, llamadas *kernels*, las cuales al ser invocadas son ejecutadas en paralelo por *N* threads diferentes en la GPU. En la figura 4.2 se mostró un diagrama de la arquitectura del sistema CPU-GPU donde se ejecutarán los programas CUDA C.

Los kernels son el componente principal del modelo de programación de CUDA, son funciones invocadas desde el host y ejecutadas en el dispositivo. Cuando se invoca un kernel, se ejecuta *N* veces en *N* threads diferentes. Cada thread se diferencia de los demás por su identificador, el cual es único y accesible en el kernel a través de una variable interna y predefinida de CUDA (built-in) llamada *threadldx*. A través de *threadldx* se puede definir el comportamiento específico de cada uno de los threads.

Para la definición de un kernel se deben respetar varias condiciones, las cuales se enuncian a continuación:

- Es una función que no retorna ningún valor.
- Todos los threads que se activen durante la ejecución del kernel, ejecutan el mismo programa, el cual coincide con el kernel que lo activó.
- El número de threads es conocido antes de la ejecución del kernel, ellos serán agrupados, según se indica en la invocación. Todos los bloques tienen igual número de threads.

Existe una jerarquía perfectamente definida sobre los threads de CUDA. Los threads se agrupan en bloques, los cuales se pueden ver como vectores (una dimensión) o matrices (dos o tres dimensiones), ver figura 4.4. Como se mencionó antes, los threads de un mismo bloque pueden cooperar entre sí, compartiendo datos y sincronizando sus ejecuciones. Sin embargo, threads de distintos bloques no pueden cooperar directamente entre ellos. Los bloques a su vez, se organizan en grid, éste puede ser de una o dos dimensiones, ver figura 4.5. La cantidad de bloques y threads por bloque de un grid son valores establecidos antes de la invocación del kernel, los cuales permanecen invariables durante toda la ejecución del mismo.

Así como los threads se pueden identificar mediante una variable predefinida, los bloques y grid también tienen sus variables predefinidas para permitir al programador tomar decisiones.

Además de las anteriores generalidades, otras de las características importantes de CUDA son:

- Estructura de un programa
  Un programa CUDA está compuesto de una o más fases, las cuales son ejecutadas o en el host o en el dispositivo. Aquellas partes que exhiben poco o nada de paralelismo se implementan en el código a ejecutar sobre el host, no así las que pueden ser resueltas aplicando paralelismo de datos, éstas son implementadas a través de código paralelo a ejecutarse en el dispositivo, en este caso la GPU. Si bien en el programa CUDA existen dos partes bien diferenciadas, será el compilador el responsable de su identificación.

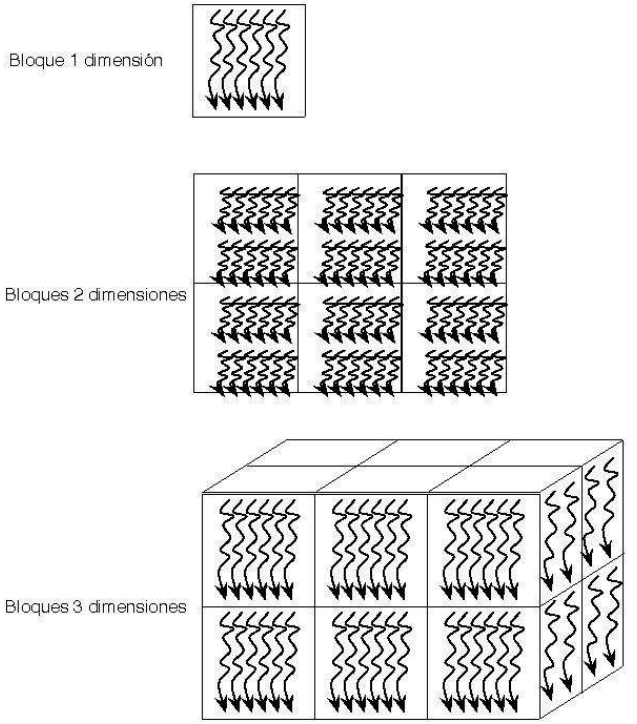


Figura 4.4: 0rganización de los threads en un bloque

Para ello, la parte de código secuencial será compilado con el compilador estándar de C (o del lenguaje secuencial utilizado) y ejecutado en la CPU como un

proceso común. En cambio el código paralelo a ejecutarse en el dispositivo, escrito en C extendido con palabras claves para expresar el paralelismo de datos y las estructuras de datos asociadas, será compilado con el compilador propio de CUDA (*nvcc* por ejemplo [NVI13b]).

Un kernel se diferencia de una llamada a una función común en el código secuencial porque además de las especificaciones enunciadas anteriormente, en su invocación se establecen los parámetros necesarios para configurar la ejecución. En la figura 4.6 se muestra la estructura de un programa CUDA y la forma de cómo se realiza la invocación a un kernel, en este caso particular se invocan dos kernel: *kernelA* y *kernelB*, indicando para cada caso cuántos bloques (#BlA, #BlB) y threads por bloques (#ThA, #ThB) resolverán el problema respectivamente; *args* es la lista de parámetros de las funciones *kernelA* y *kernelB*.

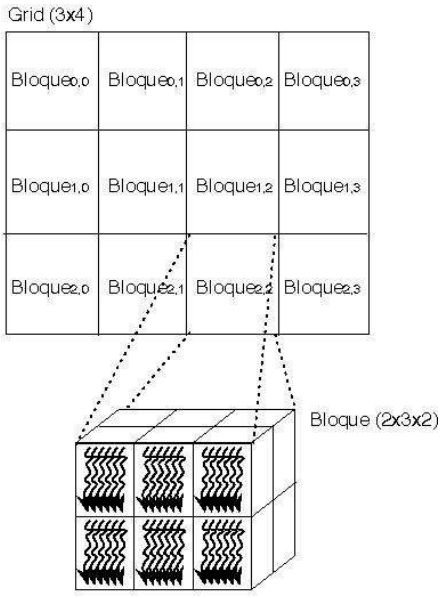


Figura 4.5: 0rganización de los Bloques en un Grid

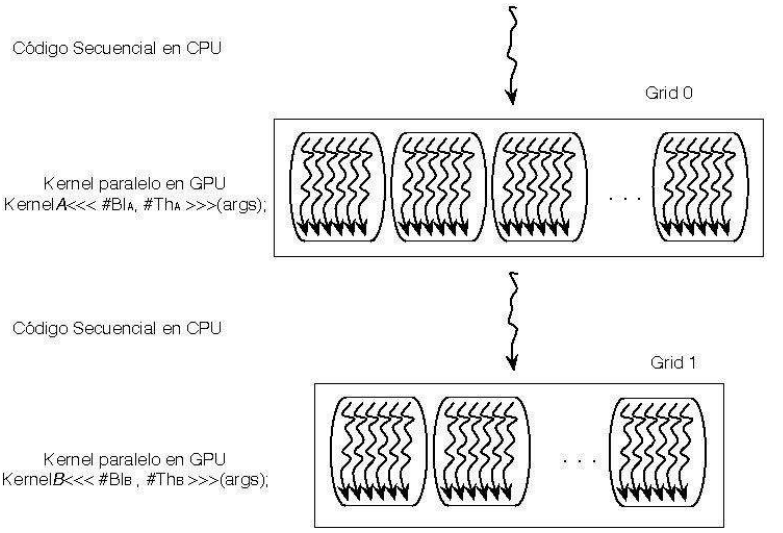


Figura 4.6: Esquema de la estructura de un programa en CUDA

Cuando se invoca una función kernel, es en ese momento cuando se establece la estructura que tendrá el grid y los bloques, además del número de threads con los cuales se resolverá en paralelo el problema. El número de threads por bloque y la cantidad de bloques en un grid pueden afectar la performance de la aplicación.

- Transferencia de datos CPU-GPU
  En el sistema CPU-GPU, cada una de las componentes, host y dispositivo, tienen su propio espacio de memoria, las cuales son independientes. Para resolver un problema en la GPU, el programador necesita transferir los datos de entrada del problema a la GPU y, una vez obtenidos los resultados, transferirlos a la CPU. CUDA provee funciones para realizar estas tareas. Las transferencias de datos entre la CPU y la GPU se dan a nivel de la memoria principal de cada uno. En la figura 4.7 se muestra la visión general del modelo de memoria CUDA de la GPU, detallando las posibles transferencias y accesos a los distintos tipos de memoria.

Se puede observar el acceso desde:

- El host
  - A la *memoria global* del dispositivo, tanto para lectura como para escritura.
  - A la *memoria de constante* del dispositivo, sólo para escritura.
  - A la *memoria de textura* del dispositivo para sólo escritura.
- El dispositivo
  - A la *memoria de registros* y a la *local*, propias de cada thread.
  - A la *memoria shared,* por todos los threads de un bloque. Cada thread puede leer o escribir en esta memoria.
  - A la *memoria de constante* y de *texturas*. Estos accesos son de sólo lectura.
  - A la *memoria global*, a la cual pueden acceder todos los threads tanto para lectura como para escritura.

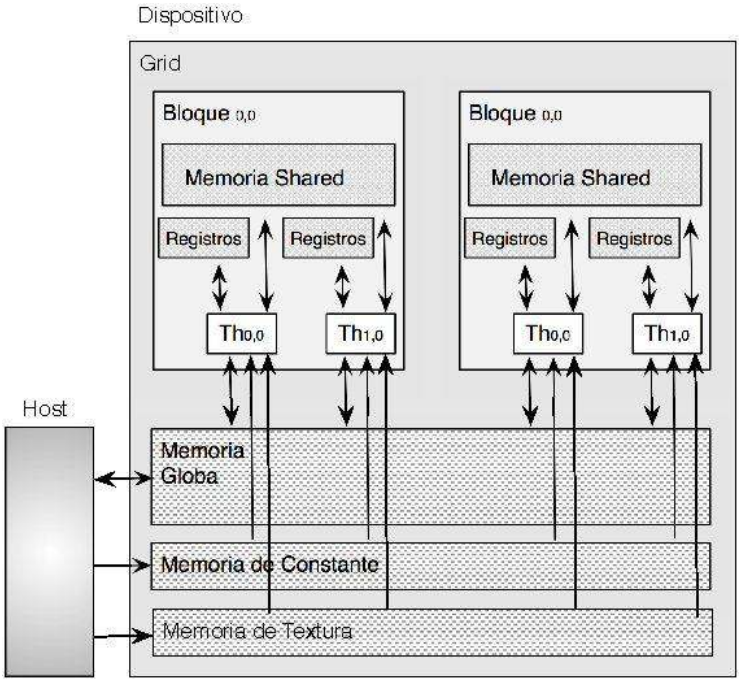


Figura 4.7: Jerarquía de memoria y accesos

La cantidad de *memoria shared*, *local* y *de registro* solicitada por los threads puede limitar el grado de paralelismo del problema. Además los accesos a la memoria global son mucho más lentos que los accesos a cualquiera de las otras memorias.

- Sincronización de threads

  La sincronización por barrera es un método simple y popular para coordinar actividades paralelas. Un ejemplo de la vida real donde se aplica una sincronización de este tipo es cuando 2 o más personas van a un lugar en un mismo auto. Al llegar a destino, cada una puede realizar una actividad distinta (actividades en paralelo). La sincronización es necesaria a la salida del lugar, las personas deben esperar por todas, es decir sólo pueden marcharse cuando todas hayan finalizado sus respectivas tareas. Si así no lo hicieran se está en presencia de un grave problema, por ejemplo dejar olvidado a alguien.

  Existen muchas aplicaciones donde una sincronización de actividades entre las distintas tareas es vital para la resolución del problema. CUDA provee la sincronización por barrera de threads de un mismo bloque a través de la invocación a la función *syncthreads()*. Cuando un kernel hace una sincronización, todos los threads del kernel permanecerán en el lugar de la invocación hasta que todos los threads del bloque lleguen a ejecutar dicha sentencia. Esto ase gura que todos los threads de un bloque completen la fase anterior a la invocación de la sincronización.

  Para una buena programación paralela, las sincronizaciones no deben llevar mucho tiempo. En el caso de CUDA, esto es posible por la asignación de todos los recursos requeridos por los threads antes de su ejecución. Es decir, al momento de ejecutarse un bloque, se le asignan los recursos necesarios para todos los threads como una unidad, todos los threads de un bloque tendrán los mismos recursos de ejecución. De esta manera se asegura la proximidad en el tiempo, evitando así largas esperas por sincronización.

Todas las características enunciadas establecen la independencia entre el desarrollo de programas CUDA y su ejecución. La solución paralela debe pensarse como un conjunto de unidades de proceso independientes (bloques y threads) donde no existe ningún requerimiento de orden de planificación ni de ejecución; los threads de los bloques se pueden ejecutar en cualquier orden y los bloques se planifican de igual manera.

Por todo lo expuesto, es claro que no todas las aplicaciones son adecuadas para el desarrollo en la GPU. Para lograr implementaciones paralelas exitosas, éstas deben cumplir con ciertas condiciones, las más relevantes son: poder resolverse aplicando paralelismo de datos, tener gran cantidad de cálculos aritméticos independientes, ser capaces de tomar ventaja de los anchos de banda y la jerarquía de memoria de la GPU, y requerir una mínima interacción entre la CPU y la GPU.

4.4. Modelo de Ejecución

El modelo de programación CUDA asume que los threads se ejecutan en una unidad física distinta, la cual actúa como coprocesador (device o dispositivo) al procesador (host) donde se ejecuta el programa. Gráficamente se puede ver la ejecución de un programa CUDA C como lo muestra la figura 4.8.

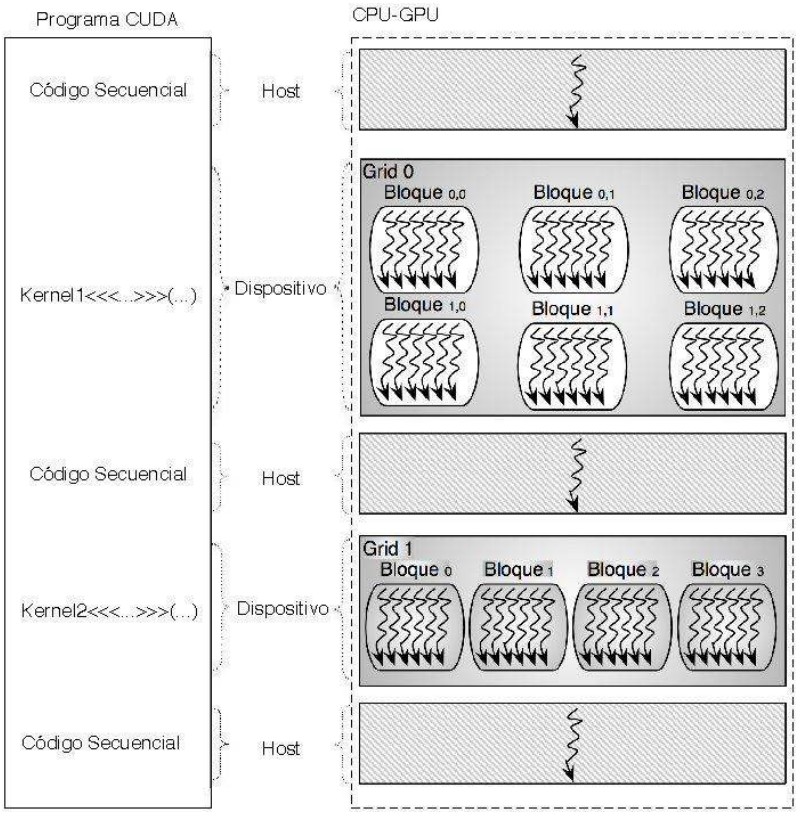


Figura 4.8: Ejecución de un programa CUDA C en un Sistema CPU- GPU

La arquitectura del sistema CPU-GPU se muestra en la figura 4.9. En ella se observa el flujo de procesamiento de un kernel en la GPU, pudiendo verse las distintas acciones a realizarse antes, durante y después de su ejecución. En orden cronológico, ellas son:

1. Copia de datos de la memoria de la CPU a la memoria global de la GPU.
2. La CPU ordena la ejecución del kernel en la GPU
3. En cada uno de los cores (SP) se ejecutan los threads indicados en la invocación del kernel. Los threads acceden a la memoria global del dispositivo para leer o escribir datos.
4. El resultado final se copia desde la memoria del dispositivo a la memoria de la CPU.

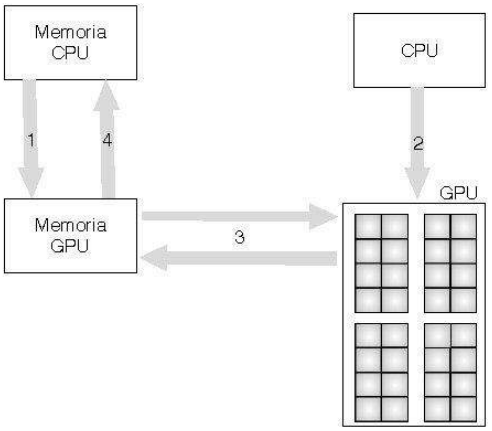


Figura 4.9: Flujo de procesamiento de un kernel en la GPU

El modelo de ejecución de los programas CUDA está íntimamente ligado a la arquitectura de la GPU, la arquitectura se basa en un arreglo escalable de SM.

Los threads se asignan a los recursos de ejecución en base a los bloques. Al estar los recursos de ejecución organizados por SM, dependiendo de la arquitectura y el modelo de la placa (En el apéndice A se detallan las características de las distintas generaciones de arquitectura GPU NVIDIA) se determina la cantidad de threads y de bloques por cada SM a ejecutar en paralelo.

Cuando un programa CUDA invoca un kernel con una configuración específica (estructura del grid, cantidad de bloques, estructura de los bloques y cantidad de threads por bloque), los bloques del grid son enumerados y distribuidos a los SM disponibles. Todos los threads de un bloque se ejecutan concurrentemente en un único SM. Cuando un bloque finaliza, un nuevo bloque es ejecutado en su lugar. La ejecución de los bloques es paralela/concurrente, en paralelo se ejecutan tantos bloques como SM tenga el dispositivo y concurrente porque si existen más bloques que SM, estos se ejecutan concurrentemente entre sí. En otras palabras en paralelo se ejecutan grupos de tantos bloques como SM tenga la GPU.

Cada SM crea, gestiona y ejecuta los threads concurrentemente en hardware, sin sobrecarga de planificación. Para poder administrar cientos de threads en ejecución, el SM emplea el modelo denominado SIMT (*Simple Instrucción-Múltiples Threads*) [PH13]. SIMT es similar al modelo SIMD

pero se aplica a nivel de instrucciones, las cuales especifican la ejecución y comportamiento de un único thread. A diferencia de las máquinas SIMD, SIMT permite al programador describir paralelismo a nivel de thread para threads independientes, así como paralelismo a nivel de datos para threads coordinados. El programador puede ignorar el comportamiento SIMT para el correcto funcionamiento de su código, sin embargo es un detalle muy importante para lograr un buen rendimiento.

El SM asigna un thread a cada SP, ejecutando cada uno el código en forma independiente según sus registros de estado. El multiprocesador SIMT crea, administra y organiza la ejecución. Ejecuta los threads en grupos de 32 threads paralelos llamados *warps*. Los threads que componen un warp empiezan a ejecutar el código de forma conjunta en la misma dirección de programa, siendo libres para ejecutar sentencias condicionales.

La pregunta ahora es ¿Cómo divide un bloque en warps? Los threads son divididos en grupos según sus identificadores y en orden creciente, el thread 0 pertenece al primer warps. En la figura 4.10 se muestra la división de *N* bloques en warps.

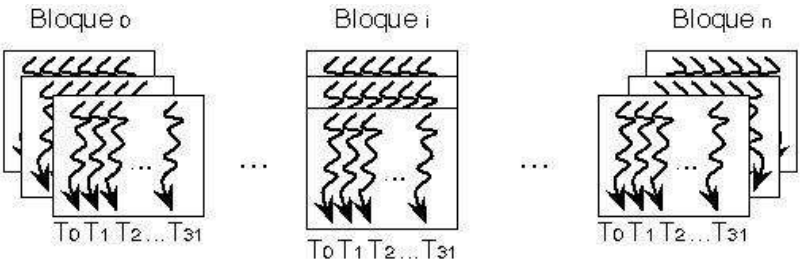


Figura 4.10: División de N bloques en warps.

La unidad SIMT selecciona un warp listo para ejecutar y prepara la siguiente instrucción para los threads activos del warp. Un warp ejecuta una instrucción por vez, de modo que la mayor eficiencia se obtiene cuando los 32 threads del warp coinciden en el mismo camino de ejecución, es decir no hay sentencias condicionales por la cual se produce una bifurcación en la ejecución. Si la ejecución de los threads toma diferentes caminos, el warp ejecuta en forma serializada cada una de las ramas, deshabilitando los threads no pertenecientes a dicha rama; cuando todas las ramas finalizan, los threads se reúnen en el camino común. La divergencia de saltos ocurre únicamente

entre los threads de un warp; warps diferentes ejecutan código de forma independiente.

Los SM pueden ejecutar varios bloques a la vez, el número depende de cuánta memoria shared necesita cada bloque. Si los registros o la memoria shared del SM no son suficientes para procesar como mínimo un bloque, el kernel fallará en su ejecución.

La administración de los threads en warps permite realizar diferentes optimizaciones, entre las más importantes se encuentran:

- Ejecutar eficientemente operaciones de gran latencia como son los accesos a la memoria global del dispositivo. Si una instrucción debe esperar por otra iniciada antes y con mayor latencia, el warp no es seleccionado para su ejecución, otro sin estas características será el elegido. Si existen varios warps listos para ejecutar, se selecciona uno según una prioridad establecida.
- Llevar a cabo ejecuciones más eficientes de operaciones de alta latencia como son las operaciones aritméticas de punto flotante y las instrucciones de *branch* (salto). Si hay muchos warps, es posible encontrar alguno listo para ejecutar mientras se resuelven las operaciones más lentas.

Todas estas características son posibles porque la selección de un warp listo no implica costo, esto es conocido como scheduling de threads con overhead cero (costo cero). Esta optimización de la ejecución de threads, evitando las demoras de unos ejecutando otros es lo que en la bibliografía se conoce como ocultamiento de la latencia (*latency hiding*) [WHW93]. Esta propiedad es una de las principales razones por las que la GPU no dedican áreas del chip a memoria caché y mecanismos de predicción de salto como lo hace la CPU, dichas áreas pueden ser utilizados como recursos para la ejecución de operaciones de punto flotante por ejemplo.

Conocer todas estas características de ejecución permite, a la hora de resolver un problema en la GPU, determinar el valor óptimo de bloques y de threads por bloques para resolver el problema en consideración, pudiendo establecer la subutilización del dispositivo por ejemplo.

4.5. Resumen

Desde sus inicios, la GPU fue diseñada como un procesador paralelo, se diferencia de la CPU por su filosofía de desarrollo completamente distinta. Los avances en la arquitectura de la CPU obedecen a ofrecer mejores prestaciones y reducir el tiempo de las aplicaciones secuenciales existentes, mientras que para la GPU estos obedecen a la necesidad de optimizar el throughput de las aplicaciones paralelas.

Respecto a la programación de la GPU, ésta ha cambiado mucho desde sus inicios, principalmente para su utilización como coprocesador en la resolución de aplicaciones de propósito general, GPGPU. Actualmente y ante el desarrollo de una arquitectura más general de la GPU, existen diferentes herramientas, las cuales permiten una programación genérica igual que ocurre con la CPU. Una de las herramientas más populares para la programación de propósito general es CUDA, definido por Nvidia para sus GPUs a partir de la serie G80.

En este capítulo se realiza una introducción básica a las GPUs y su programación con CUDA: el modelo de programación, la arquitectura de la GPU para CUDA, las características relevantes del modelo de ejecución y sus ventajas respecto a la performance.

CAPÍTULO 5

Huella Digital de Audio en GPU

Mediante la huella digital de audio, AFP, podemos identificar unívocamente a una señal, aunque ésta haya sufrido modificaciones. Una buena AFP debe representar los aspectos perceptuales del audio, y modelarla mediante entropía está demostrado que es una buena elección.

El proceso computacional para obtener la AFP basada en la entropía de la señal es costoso, por ello resulta necesario emplear otros paradigmas de computación, tal como aquellos relacionados a la computación de alto desempeño para acelerarlo.

En este capítulo se presentan las soluciones para obtener la AFP mediante GPGPU de las huellas TES (del inglés Time-Domain Entropy Signature) y MBSES (del Inglés Multi-Band Spectral Entropy Signature), explicadas en el capítulo 2. El capítulo se organiza de la siguiente manera, primero realizamos una revisión de la bibliografía relacionada a AFP y GPU. Luego, explicamos las soluciones propuestas para TES y MBSES, trabajo de esta tesis, y por último se muestran los resultados obtenidos.

5.1. Estado del Arte

Del análisis de la bibliografía existente relacionada a obtener la AFP de una señal de audio aplicando técnicas de HPC en GPU, los trabajos publicados son [GBCOl, MVSPll, SPVll, SOBll]. A continuación se muestran las distintas soluciones propuestas y cómo han abordado el uso de la GPU para mejorar el desempeño de sus soluciones y reducir el alto costo computacional de la aplicación. Las principales características de cada uno son:

1. En [MVSPll, SPVll, SOBll] se propone un algoritmo para la búsqueda de audio por contenido basado en la extracción de las componentes frecuenciales. Para ello usa un algoritmo de correlación para la comparación y posterior identificación. Éste se compone de dos partes, una es la *Extracción de la AFP* y la otra la *identificación de la señal de audio*. La identificación se realiza mediante la comparación de la AFP con las AFPs almacenadas en una base de datos.

   En la primera fase, la señal de audio es digitalizada y almacenada. Luego es dividida en frames de l28 ms de duración (analizaron diferentes longitudes de frame, desde 20 ms a 256 ms) y se procede con el análisis. Se usa la Transformada de Fourier (FFT) para obtener los componentes en el dominio de la frecuencia, eliminando la dependencia temporal. Una vez realizados todos los pasos anteriores, calculan la autocorrelación para el espectro obtenido en cada frame. Este proceso permite determinar la similitud de una señal de audio con la señal original almacenada en la BD. La división en frame de la señal de consulta debe coincidir con la división de la original para hacer la correlación. Este conjunto de datos conformará la AFP de la señal.

   En la etapa de identificación, la nueva señal de audio está digitalizada con una frecuencia de muestreo igual a la señal original, se divide en frames de igual tamaño y se calculan sus componentes de frecuencia. Posteriormente para cada frame se realiza la correlación de las componentes de frecuencia con el espectro de la señal buscada y el conjunto de correlaciones se compara con la AFP del audio original. Este procedimiento se repite con cada frame durante la búsqueda en tiempo real.

   Luego de haber realizado un análisis del algoritmo, los autores determinaron qué partes consumían mayor cantidad de recursos computacionales, y las implementaron aplicando técnicas paralelas en la GPU. Estas partes son el cálculo de la FFT y la correlación cruzada.

   En el caso de la solución para la FFT, ésta se implementa mediante el uso de la librería CUFFT propuesta por CUDA [NVIl2], la cual está basada en los algoritmos de la biblioteca FFTW [FJ98].

Con respecto al cálculo de la correlación cruzada, cada vez que un frame *w* es considerado es necesario calcular la correlación para los *N* frames que compone la AFP. El número de operaciones realizadas para dos vectores *f* y *g* de longitud *j*, en un intervalo de *i* implica $\mathrm{N} \times \mathrm{j} \times (\mathrm{i} \times 2 + 1)$ multiplicaciones, como se muestra en la fórmula $R_{f,g}(w,i) = \sum_j f^*_{w,j} \circ g_{w,i+j}$ donde f* denota el conjugado complejo de f. La independencia de datos se da en el cálculo de correlación en cada intervalo *i*, significa que el sistema se puede dividir de manera tal que cada producto escalar de los vectores *f* y *g* puede ser implementado por un thread *t*. Por lo tanto, se deberán ejecutar $N * (i * 2 + 1)$ threads, cada uno con una carga de *j* multiplicaciones. Si cada bloque de threads ejecuta la correlación cruzada entre dos vectores *f* y *g*, son necesarios *N* bloques para obtener las *N* correlaciones de las dos señales de audio y así determinar si son similares o no.

Los resultados reportados en estos trabajos fueron pobres, la solución en GPU fue dos veces más rápida que la correspondiente en CPU. Las razones de este desempeño son las demasiadas comunicaciones entre CPU y GPU, y el trabajo dependiente de los threads en la etapa de correlación.

2. En [GBCO1] se calcula la AFP con el objeto de realizar detección de copias de audio. La señal de audio analógica o digital es la entrada al sistema, para obtener la AFP de la señal se divide en frames. La AFP es un valor entero que caracteriza al frame. Una vez calculadas las AFPs de una señal, se procesan para determinar si coinciden con las AFPs de la base de datos. En el contexto de la detección de copia, y dependiendo de la aplicación, la base de datos contiene las AFPs de las canciones con derecho de autor (originales) o los anuncios publicitarios siendo supervisados, entre otras.

   Se han considerado dos tipos de AFPs, uno se basa en las diferencias de energía entre sub-bandas, y el otro en la clasificación de cada frame con respecto al frame más cercano a la consulta (vecino más cercano). Esta última proporciona un rendimiento mejor que la primera pero resultan más lentas para calcular también.

El autor expone ambas huellas, pero la huella de interés es la del vecino más cercano, por ser la que usa la GPU en su computación. Para ello se instancian los frames de audio con los frames más cercanos de la consulta. Para el cálculo de esta medida de aproximación se calculan: 12 coeficientes cepstrales y 12 de energía normalizada con sus coeficientes delta. La distancia entre los frames del audio de consulta y los frames del audio se define como la suma de la diferencia absoluta entre los parámetros cepstrales correspondientes. Si $a_1 \cdot \cdot \cdot a_n$ son los parámetros cepstrales para el frame de audio de consulta, y $p_1 \cdot \cdot \cdot p_n$ los correspondientes al frame de la BD, la distancia se calcula como $\sum_{i=1}|p_i - a_i|$. Para cada frame de audio se asocia el frame del audio de consulta más cercano y k es el coeficiente cepstral k-ésimo.

Para el cálculo del vecino más cercano en GPU, se usan bloques de 128 threads donde el número de bloques se determina por $\frac{N}{128}$ siendo N el número de frames de la señal de audio Los 128 threads fueron elegidos para asegurar la utilización de toda la memoria shared y la transferencia eficaz de los datos desde la memoria global a la memoria shared. Cada thread de un bloque es responsable de calcular el frame más cercano a uno de los frames de la señal de consulta. Una vez que todos los threads han finalizado, los próximos 128 frames en la BD son considerados, repitiendo el proceso anterior. Para aumentar el rendimiento se procesan simultáneamente varios audios y/o consultas.

La búsqueda con el vecino más cercano de las AFPs usando GPU sigue teniendo un tiempo de proceso demasiado largo cuando se considera un gran conjunto de datos. Por lo tanto una solución sería combinar las dos AFPs propuestas. Por lo tanto el proceso se divide en dos fases. En la primera fase la búsqueda utiliza las AFPs de la diferencia de energía y luego en la segunda fase de la búsqueda las instancias se encuentran utilizando las AFPs del vecino más cercano. Esto reduce significativamente el tiempo de cálculo mientras se mantiene la precisión de la búsqueda de las AFPs del vecino más cercano.

En todos los trabajos analizados las ganancias obtenidas no fueron muy prometedoras. En la sección 5.5, comparamos las soluciones antes descriptas con las desarrolladas en esta tesis.

5.2. Cálculo de la AFP $MBSES_{GPU}$

Las AFPs expuestas en el capítulo 2 sección 2.4.2.2 se basan en el cálculo de la Entropía y la división en bandas según la escala de Bark.

El proceso en GPU de $MBSES_{GPU}$ posee tres etapas principales, figura 5.1, todas ellas aplicadas en secuencia sobre cada frame de la señal. Como puede observarse, el primer paso para la obtención de $MBSES_{GPU}$ es dividir la señal de audio en frames de longitud fija, en este caso los frames son de 16 KB equivalente a 370 ms de duración. Esta longitud es adecuada para el cálculo de la Entropía [CI07]. Al igual que en el proceso secuencial, todos los frames se superponen un 50 % para asegurar una lenta variación de las características extraídas.

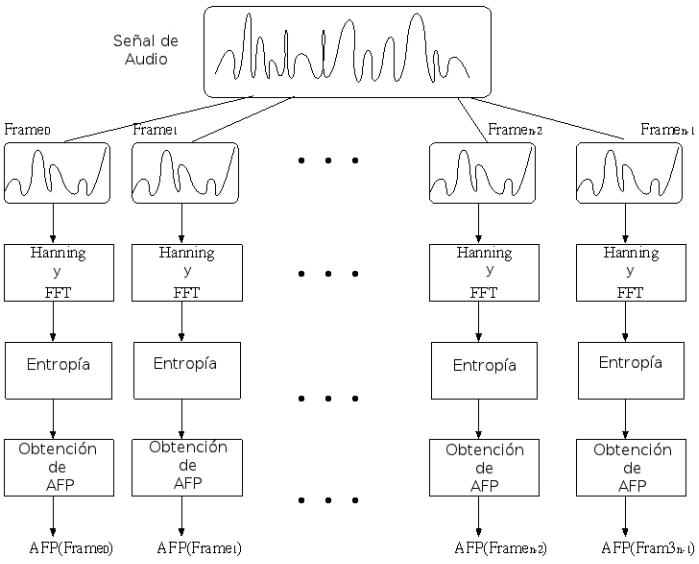


Figura 5.1: Proceso Paralelo para la obtención de la AFP $MBSES_{GPU}$

El problema ahora se aplica independientemente a cada frame, esta características hace del problema particularmente adecuado para el procesamiento paralelo masivo.

Una vez dividido en frames, cada uno de ellos es asignado a un grupo de unidades de procesamiento, las cuales aplican secuencialmente las etapas: *Hanning y FFT*, *Entropía* y *Obtención AFP*. En las próximas secciones se explican los aspectos de diseño e implementación.

5.2.1. Aspectos de Diseño

En esta sección se detallan las características principales de diseño de las tareas involucradas en el proceso para obtener la AFP $MBSES_{GPU}$.

5.2.1.1. Etapa Hanning y FFT

En esta etapa se calcula la FFT por cada frame de audio, transformándolo desde el dominio del tiempo al dominio de la frecuencia. El cómputo se divide en dos fases principales, la primera relacionada con la aplicación de la ventana de *Hanning* y el cálculo del *Bit-Reverse*, y la segunda parte se corresponde con el cálculo de la FFT propiamente dicho. Cada fase tiene las siguientes características:

1. Fase *Ventana de Hanning y Bit-Reverse*: Previo al cálculo de la FFT se aplica por cada frame la ventana de Hanning. Esto es necesario para reducir los efectos de borde en los extremos del frame, así se ubica al frame en el centro de la ventana. Luego para poder computar la FFT en forma iterativa debemos ubicar cada uno de los componentes del frame según la permutación determinada por el método *Bit-Reverse*. Como resultado de su aplicación se ubicará cada componente del frame en la posición establecida, de esta manera las componentes en las posiciones pares son movidas a un extremo y las impares a otro extremo del frame. El resultado del *Bit-Reverse* es un vector con las nuevas posiciones, el cual es común a todos los frames.

2. Fase *FFT*: El diseño del algoritmo se basó en el algoritmo original de Cooley y Tukey [CT65], tanto la FFT directa como la inversa pueden ser calculadas mediante el cambio de un único parámetro. La muestra se divide a la mitad usando el teorema de Danielson Lanczos [DL42], obteniendo dos sublistas. Este proceso se repite recursivamente o de forma iterativa hasta que el problema es trivial (el vector sobre el que se aplica la FFT tiene sólo dos componentes). En la solución presente, el algoritmo es iterativo (Cuando fue desarrollado, CUDA no permitía recursión).
   Para resolver la FFT también son necesarios otros datos auxiliares además del vector *Bit-Reverse*, estos son los coeficientes de cada uno de los elementos sobre los cuales se calcula la FFT. Los mismos pueden ser pre-calculados y dispuestos como en un vector con la misma característica del vector de *Bit-Reverse*, común a todos los frames.

La salida de esta etapa es un frame en el dominio de la frecuencia, por lo tanto es un vector de números complejos. Las siguientes etapas trabajan sobre ambas componentes, el vector de la parte real y el de la parte imaginaria.

5.2.1.2. Etapa Entropía

Como hemos dicho en el capítulo 2 específicamente en la sección 2.4.2, la entropía de una señal es una medida de la cantidad de información que conlleva la señal [CI07]. Para calcularla en este caso se considera la entropía basada en histogramas.

Determinar la entropía implica varias tareas o fases, éstas son: *Traducción a Valores Discretos*, *Histograma Final de la Banda* y *Entropía de la Banda*. Como la entrada a esta etapa es una secuencia de valores complejos, todas las fases se aplican a dos vectores, el vector de las componentes reales y el de las componentes imaginarias. La figura 5.2 muestra la etapa de entropía y todas sus fases.

Cada fase tiene las siguientes características:

1. *Traducción a Valores Discretos*: Los valores continuos de cada frame son convertidos a valores discretos, necesarios para calcular el histograma. Esto implica obtener el intervalo al que pertenecen todos los valores del frame, dividirlo en tantas partes como valores distintos serán considerados en el histograma y, finalmente, asociar cada elemento del frame a la partición correspondiente. Para llevar a cabo cada una de estas tareas es necesario primero determinar los límites del intervalo (valor máximo y mínimo), luego dividir el intervalo en 256 subintervalos (el proceso considera un histograma de 256 valores distintos), y finalmente instanciar cada elemento del frame a uno de los 256 intervalos.

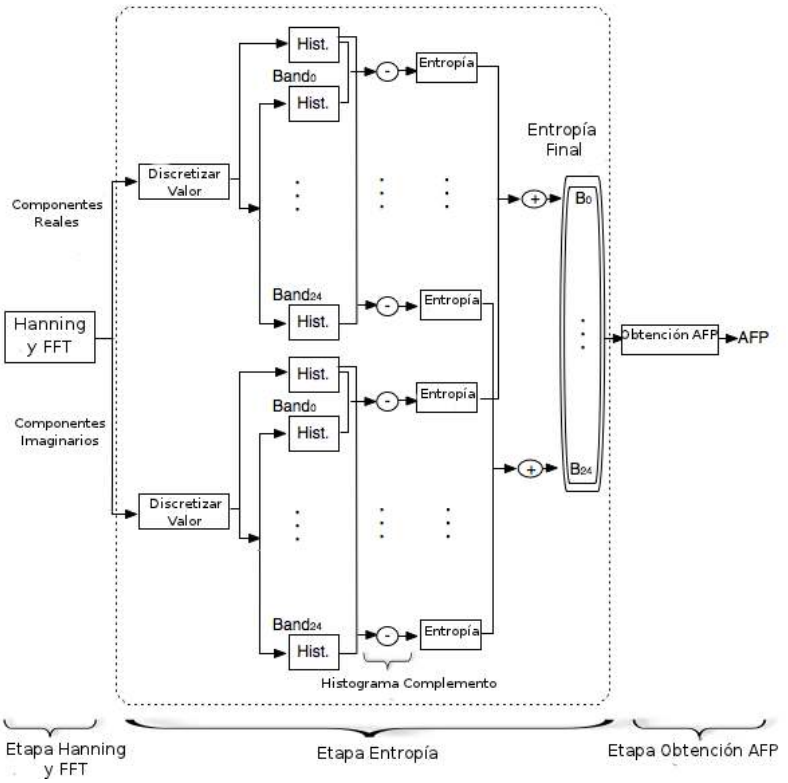


Figura 5.2: Etapa Entropía para cada Frame.

2. *Histograma Final de la Banda*: Para obtener el histograma final de cada banda se deben realizar varios pasos. La entrada de esta fase son dos vectores (parte real e imaginaria) de valores discretos entre 0 y 255, sobre los cuales se calcula el histograma del frame.
   Luego que el histograma total del frame es calculado, se divide el frame en las 24 bandas críticas según la escala de Bark (ver sección 2.4.2.2 del capítulo 2) y se procede a computar el histograma de cada banda. La división en bandas es hecha aplicando un filtro de banda estándar, sintonizado con las frecuencias correspondientes de la escala de Bark. Los límites de cada banda son los mismos para cada frame. Esta propiedad permite calcularlos de antemano.
   A partir del histograma total y los histogramas de cada banda se calcula el histograma complemento.
   Todo el proceso se aplica dos veces, una para los componentes reales y otro para las componentes imaginarias, ver figura 5.2. En consecuencia, la salida de esta fase son los histogramas de la parte real e imaginaria de cada banda. Estos histogramas son la entrada de la etapa *Entropía de la Banda*.

3. *Entropía de la Banda*: Cada componente x_i del histograma complemento representa la frecuencia del i-ésimo elemento ($f(x_i)$), el cual es necesario para obtener la probabilidad según la fórmula de Laplace $\frac{f(x_i)}{N'}$ donde N' es el total de elementos del frame (N =16384) menos la cantidad de elementos en la banda actual. Cada thread calcula una probabilidad, $p(x_i)$ y la parte interna del cálculo de entropía $p(x_i) \times \ln(p(x_i))$. Finalmente los resultados de cada thread se suman. Esta operación es una suma reducción en paralelo por suma (Ver ecuación 2.8 del capítulo 2).
   La salida es un vector de veinticuatro elementos, donde cada uno de sus componentes es la suma de las entropías real e imaginaria de cada banda.

Los datos de salida de la etapa de la entropía son *N* vectores (uno por frame) de 24 componentes. Cada componente se corresponde con la entropía de cada banda.

5.2.1.3. Obtención AFP

Una vez calculados los valores de entropía para cada frame, la AFP es computada. Como cada grupo de unidades de procesamiento calcula la AFP de un frame, para esta etapa final se requieren N 1 unidades de procesamiento, donde cada una será responsable de calcular la AFP del frame a partir de sus entropías y según la ecuación 5.1.

Para el frame *i* y toda banda *b*

$$F(i,b) = \begin{cases} 1, & si\ [h_b(i) - h_b(i-1)] > 0 \\ 0, & en\ otro\ caso \end{cases} \qquad (5.1)$$

Entonces

$$AFP_i =< F(i,0), F(i,1), \dots, F(i,23) >$$

Al finalizar este proceso, se obtiene la *MBSE* S_{GPU} de la señal de audio, la cual será igual a

$$MBSES_{GPU}(z) =<< F(0,0), \dots, F(0,23) >, \dots, \\ < F(i,0), F(N-1,0), \dots, F(N-1,23) >>$$

Cuando la señal de audio *z* es dividida en N frames.

5.2.2. Aspectos de Implementación

Al inicio de la computación, la señal de audio es normalizada, convertida de estéreo a monoaural promediando ambos canales. Posteriormente, es transferida a la GPU. Además la CPU computa todos los datos auxiliares: *Bit-Reverse*, *Coeficientes* y límites de las bandas de la escala de Bark.

Dos transferencias de datos entre la CPU y la GPU son hechas, desde:

1. La CPU a la GPU: Antes de iniciar el proceso, la CPU transfiere a la memoria global de la GPU la señal de audio normalizada y los datos auxiliares necesarios para la etapa de la FFT: vector de *Bit- Reverse* y de *Coeficientes*; y a la memoria de constante los límites de las bandas de la escala de Bark para un frame de 16KB.

2. La GPU a la CPU: Esta transferencia se realiza al final del proceso de AFP, la GPU envía a la CPU la AFP resultante de la señal de entrada

El trabajo en la GPU está organizado en bloques de 256 threads. Cada bloque es responsable de calcular la AFP de un frame. Si la señal de audio es dividida en N frames, entonces N bloques trabajan en paralelo en la GPU para obtener la AFP de la señal. El número máximo de threads activados es igual a $N \times 256$.

El proceso de MBSESGPU se implementa en tres kernels, cada uno de ellos se corresponden a cada una de las etapas en la figura 5.1. Los kernels se ejecutan en secuencia y ningún movimiento de datos es necesario entre ellos. Los dos primeros kernels se resuelven con N bloques de 256 threads cada uno, mientras que para el último kernel son activados N-1 bloques de un thread cada uno.

En las dos primeras etapas, cada bloque de 256 threads se encarga de aplicar los kernels a un frame. Como el número de threads es menor al tamaño del frame (16 KB= 16384 componentes), cada thread trabaja sobre una fracción de 64 componentes ($\frac{16384}{256}$) del frame. La asignación de los datos a los threads se realiza mediante una distribución cíclica, favoreciendo de esta manera los accesos coalescentes a memoria.

Respecto al cómputo del histograma, varias implementaciones fueron desarrolladas para calcularlo, todas ellas utilizando la memoria global. Aunque los resultados obtenidos eran aceptables, pudimos mejorar su calidad utilizando la memoria shared, reduciendo drásticamente los accesos de memoria global [MPC1O, MPCCI1Oa, MPCCI1Ob]. En esta propuesta, el histograma se calcula directamente sobre un vector de 256 elementos residentes en la memoria shared del bloque. Como los valores oscilan entre 0 y 255, cada thread accede directamente a la posición del histograma, el cual es indicado por el valor del elemento que le corresponde, los accesos pueden producir conflictos (dos o más threads pueden operar simultáneamente en la misma dirección de memoria). Lo resolvimos utilizando las funciones atómicas [San1O], si dos o más threads acceden a la misma dirección de memoria, las funciones atómicas serializan el acceso y los resultados son determinísticos.

Cuando la etapa está dividida en fases, éstas están separadas por sincronizaciones por barrera en el bloque.

En la ú ltima etapa son activados *N*-1 bloques con un thread cada uno. Cada thread calcula la AFP de un frame a partir de la entropía propia y la del frame anterior.

5.3. Cálculo de la AFP TES_{GPU}

Tanto la huella *TES* como la *MBSES* se basan en la entropía. Su implementación secuencial se explicó en la sección 2.4.2.1 del capítulo 2. En esta sección detallaremos las características algorítmicas y de implementación de la solución para obtener *TES* en la GPU, TES_{GPU}.

La figura 5.3 muestra el proceso paralelo para la obtención de la AFP de una señal de audio. Por sus características, el problema es altamente paralelizable y adecuado para ser resuelto en la GPU.

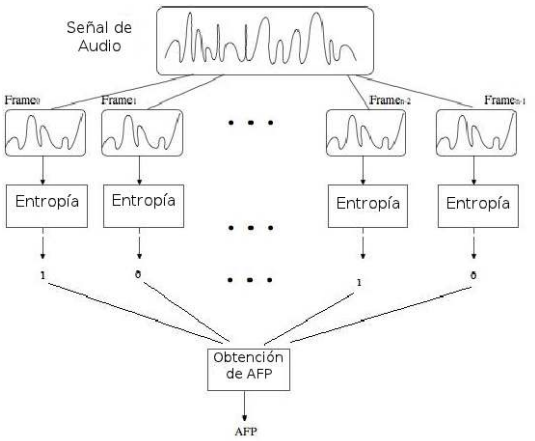


Figura 5.3: Proceso Paralelo para la obtención de la AFP TES

Al inicio del proceso, la señal de audio es normalizada, la señal estéreo es convertida a monoaural promediando ambos canales. Posteriormente, es dividida en frames de longitud fija, en este caso la longitud es igual a 64 KB, teniendo en cuenta una superposición del 50 %. Cada frame es procesado en paralelo por un grupo de unidades de procesamiento, significando que si la señal tiene N frames entonces existen N grupos trabajando en paralelo, cada uno con la misma cantidad

de unidades de procesamiento. El resultado final es una AFP, TES_{GPU}, la cual es una secuencia de N bits.

Para obtener la TES_{GPU}, el procesamiento en paralelo de cada frame implica resolver en secuencia las siguientes tareas: *Cálculo del valor mínimo y máximo*, *Cálculo de Entropía*, y *Obtención de la huella digital binaria*. En las siguientes secciones se especifican las características de diseño e implementación de cada una.

5.3.1. Aspectos de Diseño

Los aspectos básicos de diseño de cada una de las etapas involucradas en el proceso para obtener la AFP TES_{GPU} son:

1. *Cálculo del valor mínimo y máximo:* Para obtener esta AFP debemos computar el histograma total de la señal, la cual está dividida en *N* frames trabajados en paralelo. Calcular el histograma implica convertir los valores continuos de la señal a valores discretos. Esto implica obtener el intervalo al que pertenecen todos los valores de la señal, dividirlos en tantas partes como valores distintos serán considerados en el histograma y, finalmente, asociar cada elemento de la señal a la partición correspondiente. Para llevar a cabo cada una de estas tareas es necesario primero determinar los límites del intervalo (valor máximo y mínimo) para la señal completa, luego dividir el intervalo en 256 subintervalos (el proceso considera un histograma de 256 valores distintos), y finalmente instanciar cada elemento de la señal a uno de los 256 intervalos.
   Como puede observar, al necesitar obtener la amplitud total del intervalo de valores de la señal completa, la cual está dividida en frames paralelos, la determinación del intervalo implica un trabajo a nivel local para cada frame y luego a nivel global de la señal. Estas dos actividades tienen las siguientes características:

   a) *Cálculo de máximo y mínimo local:* Cada grupo calcula el valor máximo y mínimo local al frame sobre el que está trabajando.
   b) *Cálculo de máximo y mínimo global:* Una vez calculados los valores máximo y mínimo locales de cada frame, mediante reducción paralela por

valor máximo y mínimo se obtienen ambos valores para toda la señal.

Con ambos valores, cada unidad de procesamiento realizará la discretización de sus respectivos valores, instanciándolos con uno de los 256 posibles valores discretos.

2. *Cálculo de Entropía:* Su cómputo se basa en la entropía de Shannon [SW49] mediante histograma. Cada grupo de unidades de procesamiento trabaja sobre un frame distinto implicando 3 fases, las cuales son:
    a) *Traducción en Valores Discretos:* Los valores continuos del frame son convertidos a valores discretos teniendo en cuenta los mínimos y máximos globales determinados en la etapa anterior, necesarios para calcular el histograma.
    b) *Calcular el histograma del frame:* Cada unidad de procesamiento del grupo trabaja sobre un grupo de elementos del frame. Sobre los datos locales se computa localmente un histograma, para luego realizar una reducción global por suma de cada uno de los histogramas locales de todos los grupos de trabajo.

    c) *Obtener un valor de entropía:* Cada componente x_i del histograma representa la frecuencia del i-ésimo elemento, el cual es necesario para obtener la probabilidad del elemento x_i según la fórmula de Laplace ($p(x_i)= \frac{f(x_i)}{N}$ donde $f(x_i)$ es la frecuencia de x_i y N es el tamaño de la señal) cada unidad de procesamiento es responsable del cómputo de la parte interna de la fórmula de entropía ($p(x_i) \times \ln(p(x_i))$). Los resultados son sumados en paralelo según ecuación 2.8.
    Una vez obtenidos los valores de entropía (h) del frame, se calcula la n-ésima componente de la huella según la ecuación 5.2

$$F(i,b) = \begin{cases} 1, & si\ [h_b(i) - h_b(i-1)] > 0 \\ 0, & en\ otro\ caso \end{cases} \qquad (5.2)$$

    La salida de esta etapa es una secuencia de bits donde cada bit corresponde al resultado de aplicar la expresión 5.2 para un frame.

3. *Cómputo de la huella digital binaria:* Para obtener la AFP final se utiliza un único grupo de unidades de procesamiento, donde la cantidad de unidades depende del número de frames y el tamaño de la representación computacional de un número entero. Cada unidad de procesamiento participa en la composición del número entero que formará parte de la huella final, aportando el valor binario de la huella de un frame.

Todas estas etapas son aplicadas en secuencia, sin poder alterar el orden.

5.3.2. Aspectos de Implementación

Al inicio de la computación, al igual que en $MBSES_{GPU}$, la señal de audio es normalizada, convertida de estéreo a monoaural promediando ambos canales. Posteriormente, es transferida a la GPU.

Dos transferencias de datos entre la CPU y la GPU son hechas, desde:

1. La CPU a la GPU: Antes de iniciar el proceso, la CPU transfiere a la memoria global de la GPU la señal de audio normalizada. Ningún otro dato es requerido.
2. La GPU a la CPU: Esta transferencia se realiza al final del proceso de AFP, la GPU envía a la CPU la AFP resultante de la señal de entrada.

Cada uno de los diferentes frames son procesados en paralelo por un bloque de 256 threads, esto significa que si la señal tiene N frames entonces se tienen N bloques de 256 threads cada uno. El resultado es una AFP, la cual es un valor binario de N bits.

El trabajo en la GPU está organizado en los bloques de threads. Cada bloque es responsable de colaborar con el cálculo de la AFP de un frame. Si la señal de audio es dividida en N frames, entonces N bloques trabajan en paralelo en la GPU para obtener la AFP de la señal. El número máximo de threads activados es igual a $N \times 256$. Recordemos que los frames tienen una longitud fija de 64KB y una superposición del 50 %.

El proceso de TES_{GPU} se implementa en tres kernels, cada uno de ellos se corresponden a cada una de las etapas en la

figura 5.3. Los kernels se ejecutan en secuencia y ningún movimiento de datos entre la CPU y la GPU es necesario.

Los dos primeros kernels se resuelven con *N* bloques de 256 threads cada uno, mientras que para el último kernel es activado un bloque de tantos threads como bits tenga la representación de un entero en la GPU.

En el primer kernel, los 256 threads de cada bloque son responsables de determinar los valores mínimos y máximos locales. Luego, mediante el uso de funciones atómicas se realiza una reducción entre todos los bloques por el valor máximo y mínimo de la señal. Estos valores se almacenan en memoria global de la GPU y serán accedidos por todos los threads de la siguiente etapa. Con el objeto de lograr aprovechar al máximo los recursos de la GPU, hemos desarrollado y evaluado tres versiones distintas, cada una con las siguientes características:

1. Primera Versión: Para el cálculo de los valores máximos y mínimos de la señal se definen dos kernels. El primer kernel obtiene ambos valores en forma local, es decir, cada bloque adquiere el máximo y mínimo del frame de la señal. El segundo kernel obtiene los valores máximo y mínimo de la señal, a partir de los valores locales obtenidos de cada bloque. Este proceso se realiza mediante una reducción paralela y el resultado es almacenado en memoria global de la GPU.
2. Segunda Versión: Los valores máximo y mínimo de la señal son calculados en la CPU y luego copiados en la memoria constante de la GPU para ser accedido por los threads de la aplicación.
3. Tercera Versión: Para el cálculo se utilizan las funciones atómicas globales. En un principio cada bloque calcula en forma local cada valor y luego se obtiene el valor máximo y mínimo de la señal mediante las funciones atómicas *atomicMax* y *atomicMin*.

Como puede observarse, las tres versiones consideran diferentes aspectos de la GPU y su relación con la CPU: Activación de dos kernels, Cómputo en la CPU y uso de la memoria de constante en la GPU, o Activación de un único kernel, memoria global y funciones atómicas. Como no

mostraron diferencias significativas con respecto al tiempo, se optó por la versión tres.

La segunda etapa, los bloques de 256 threads continúan trabajando sobre un frame cada uno. El cálculo del histograma se realiza sobre la memoria shared, cada thread incrementa la posición del histograma que le corresponde a la componente con la cual está trabajando. Como varios threads pueden acceder a la misma posición de la memoria shared, funciones atómicas son utilizadas.

Una vez calculado el histograma local, se computa el histograma global. Para ello, cada thread del bloque es responsable de una componente del histograma, y mediante funciones atómicas sobre la memoria global, realiza la reducción por suma del histograma total.

Para la última etapa, un bloque de tantos threads como la representación de un número entero de la GPU es activado, en este caso se fijo un bloque de 32 threads. Los 32 threads colaboran en el armado de un número entero aportando el bit resultado de la etapa anterior para un frame. El bit es ubicado en la posición indicada por el identificador del thread.

El resultado es una secuencia de bits, agrupados en valores enteros, los cuales constituyen la AFP de la señal de audio.

5.4. TES_{GPU} vs $MBSES_{GPU}$

Los desarrollos descriptos anteriormente permiten obtener la AFP binaria de una señal de audio. Éstas poseen diferencias y semejanzas, las cuales se explican a continuación. Las semejanzas encontradas son:

1. La señal de audio se divide en frames de longitud fija con una superposición del 50% y cada frame es tratado por un bloque de 256 threads.
2. Se basan en la entropía para la obtención de la AFP y usan el Método del histograma de 256 valores. Los valores continuos de los frames son discretizados en un rango entre 0 y 255.
3. Los puntos de comunicación entre la GPU y la CPU: al inicio y al final de la aplicación.

Las diferencias entre ambas versiones de AFP son:

1. La longitud del frame es distinta, en MBSES es de 16KB mientras que en TES es de 64KB.

2. $MBSES_{GPU}$ usa la FFT para convertir la señal del dominio del tiempo al dominio de la frecuencia, mientras que TES_{GPU} se calcula directamente en el dominio del tiempo.

3. En $MBSES_{GPU}$, cada frame se divide en bandas, obteniendo una AFP de 24 bits por cada frame. Por lo tanto si la señal tiene N frames, la AFP de la señal está formada por (N-1) vectores binarios de 24 bits cada uno. En la implementación de TES_{GPU}, cada frame aporta un valor binario, por lo tanto la AFP de la señal es un vector binario de N bits.

En ambos casos, las soluciones en GPU tomaron ventaja de las características de la arquitectura y la herramienta de desarrollo: uso de la jerarquía de memorias, control de acceso a memoria, funciones atómicas y número de threads por bloques.

5.5. Estado del Arte vs TES_{GPU} y $MBSES_{GPU}$

Del análisis comparativo entre las características de diseño e implementación de las AFPs desarrolladas en esta tesis con respecto a las presentadas en el estado del arte, se observa que ambas huellas, TES_{GPU} y $MBSES_{GPU}$, realizan dos transferencias entre la CPU y GPU, una al inicio y la otra al finalizar el proceso. Por otra parte se aprovechó la jerarquía de memoria, utilizando aquellas de rápido acceso y baja latencia como es la memoria shared y la memoria de constante, logrando una reducción en los tiempos del proceso. Otra característica importante es que se tomó ventaja de la independencia de datos, desarrollando soluciones donde cada thread trabaja en forma autónoma del resto de los threads, teniendo una mínima necesidad de sincronización. Todas estas particularidades permitieron obtener mejoras significativas con respecto al tiempo de procesamiento.

En las soluciones presentadas en [MVSP11, SPV11, SOB11], el rendimiento obtenido utilizando la GPU fue pobre, resultando dos veces más rápido que en CPU. La razón de ello

fue la necesidad de realizar varias comunicaciones entre la CPU y GPU. Además, en la propuesta donde se calcula la AFP mediante el algoritmo de correlación, existe una gran dependencia entre los threads afectando drásticamente el tiempo de computación.

En [GBCO1] los resultados obtenidos en la solución con GPU fueron 3 veces más rápido que en la CPU. Si bien no es una aceleración muy alta constituyó un gran avance debido al costo computacional del problema. Los autores no dan grandes detalles respecto a: la cantidad de comunicaciones existentes entre la CPU y la GPU, y a la dependencia entre los datos. Por lo tanto no se puede determinar las razones por las cuales la aceleración no es mayor.

Del análisis del problema en la literatura existente, se puede inferir que el problema no ha sido ampliamente estudiado, más el caso del uso de la GPU y el aprovechamiento sus beneficios como arquitectura paralela.

5.6. Resultados Experimentales

En esta sección se presentan los resultados del desempeño de las distintas propuestas en la GPU. Los experimentos fueron desarrollados en diferentes escenarios; para las versiones secuenciales se consideró una computadora con las siguientes características: intel core i3, 2.13 GHz y 3 GB de memoria. Los escenarios para los experimentos paralelos fueron diferentes GPUs, las cuales pertenecen a distintas generaciones de arquitecturas: G8O y GF1OO, o tienen distinta cantidad de recursos. En la tabla 5.1 se detallan las características de cada una de las GPUs utilizadas.

Cuadro 5.1: Características básicas de las GPUs

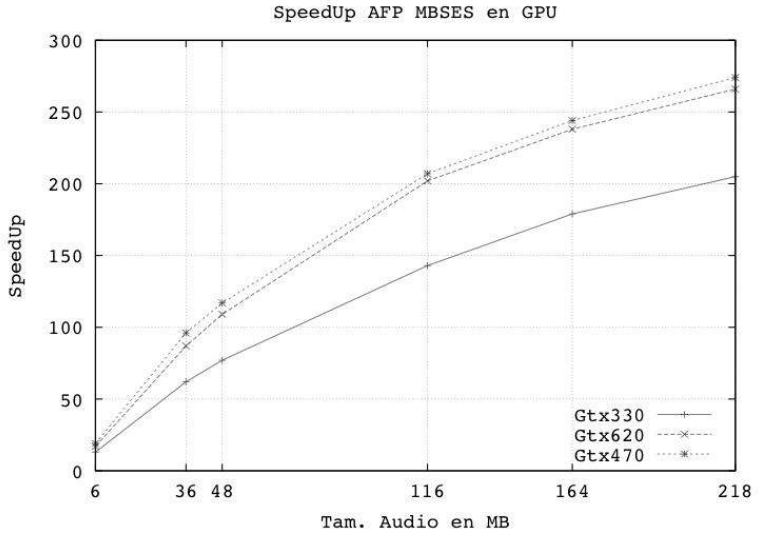
GPU	GT-330M	GTX-470	GTX-620
Memoria. Global	900 MB	1280 MB	1024 MB
SM	6	14	1
SP	8	32	48
Clock – Rate	1040MHz	1215MHz	1400MHz
Capacidad Computación	1.2	2.0	2.1

Para analizar el comportamiento de las dos soluciones en GPU de la AFP de audio explicadas en las secciones anteriores, las evaluamos para distintos tamaños de señales. En la tabla 5.2 se detallan las características de cada una de las señales: tamaño y número de frame tanto para $MBSES_{GPU}$ como para TES_{GPU}. Recordemos que el tamaño de frame para $MBSES_{GPU}$ es de 16KB y el de TES_{GPU} de 64Kb.

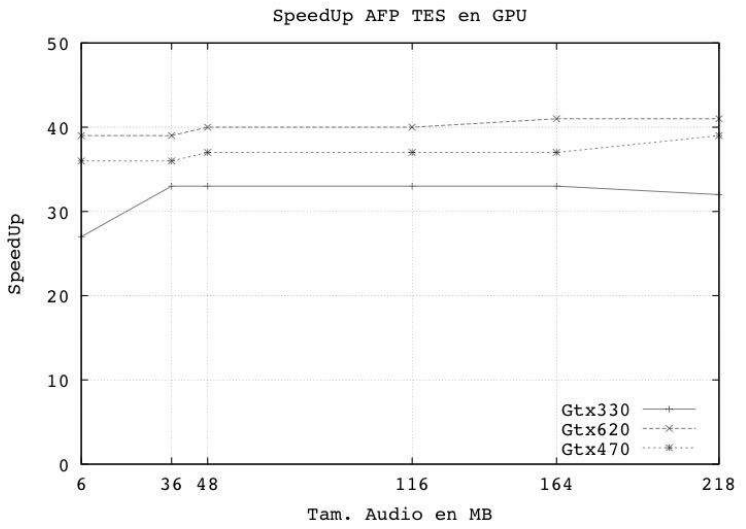
Cada uno de los valores reportados es el promedio de 100 ejecuciones. La figura 5.4 muestra la aceleración (speedup) $MBSES_{GPU}$ y TES_{GPU} para todas las señales de audio detalladas en la tabla 5.2.

Cuadro 5.2: Características de los audios de prueba

Audio ID	A-6	A-36	A-48	A-116	A-164	A-218
Tamaño en MB	6,2	36,4	48	116,1	164,4	218,1
Frames_MBSES	189	1112	1462	3540	5015	6654
Frames_TES	46	277	364	884	1253	1662



(a) $MBSES_{GPU}$



(b) TES_{GPU}

Figura 5.4: Aceleración de $MBSES_{GPU}$ y TES_{GPU} en 3 GPU

Como se puede observar en la figura 5.4(a), $MBSES_{GPU}$ muestra un buen desempeño, el cual es más alto cuando los tamaños de las señales son más grandes. Claramente se puede observar que la señal A-218 obtiene una aceleración aproximada a 270x mientras que para la señal A-6 es de 19x. Esto obedece a la influencia de los costos de las transferencias entre CPU y GPU. La relación es mostrada en las tablas 5.3 y 5.4, donde especificamos para cada señal los tiempos del proceso secuencial (tabla 5.3), y los correspondientes a la transferencia y al procesamiento en dos GPUs de distinta generación, la Gtx330 y la Gtx470 (tabla 5.4).

Cuadro 5.3: Tiempo de procesamiento secuencial para cada AFP en ms

ID Audio	MBSES	TES
A-6	842,949	82,069
A-36	5041,108	492,597
A-48	6543,625	648,422
A-116	16058,321	1572,022
A-164	22892,291	2219,707
A-218	29992,852	2974,001

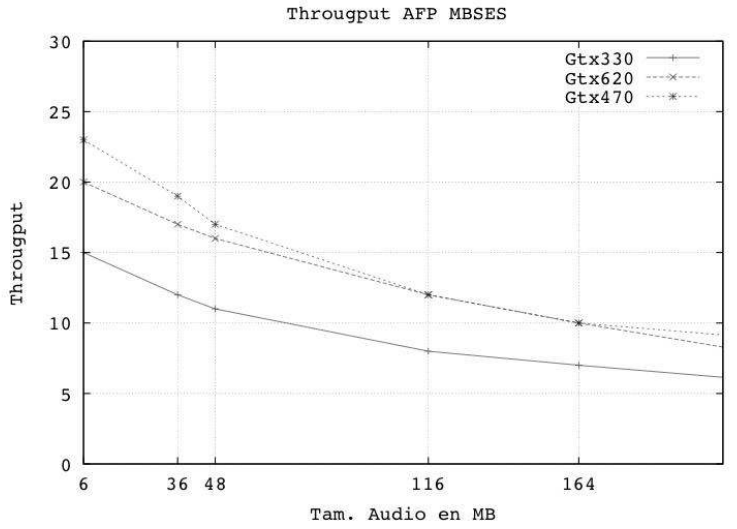
En todos los casos podemos observar buenos desempeños, ya sea en GPUs con arquitecturas antiguas o con menor cantidad de recursos.

Cuadro 5.4: Tiempo de procesamiento vs. Tiempo de transferencia para cada AFP en ms

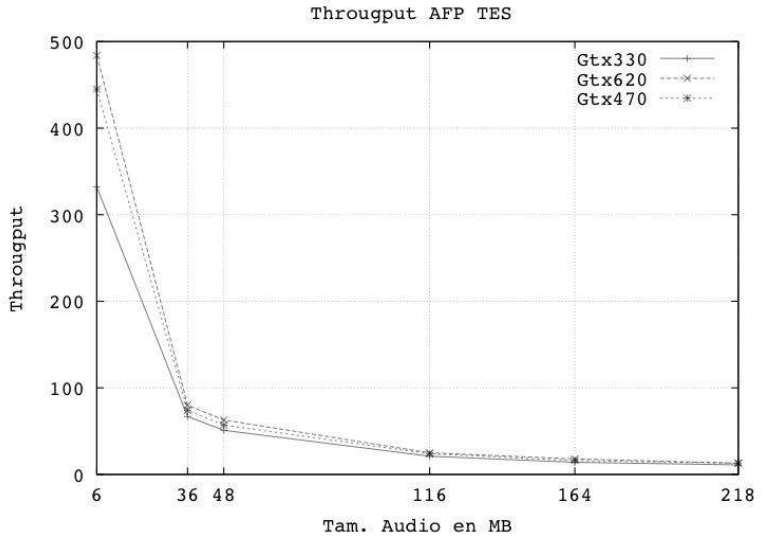
GPU	ID Audio	$MBSES_{GPU}$		TES_{GPU}	
		Tpo Proc.	Tpo Transf.	Tpo Proc.	Tpo Transf.
Gtx330	A-6	0,0457	63,973	0,0176	2,9938
	A-36	0,0461	80,0509	0,0179	14,8935
	A-48	0,0529	84,8137	0,0177	19,5494
	A-116	0,0468	112,1794	0,0178	47,4665
	A-164	0,0449	127,3961	0,0184	67,1403
	A-218	0,0473	145,8579	0,0176	90,5793
Gtx470	A-6	0,0274	42,538	0,0074	2,447
	A-36	0,0283	52,384	0,0084	13,268
	A-48	0,0294	55,706	0,0095	17,467
	A-116	0,0278	77,426	0,0073	41,527
	A-164	0,0281	93,518	0,0076	58,520
	A-218	0,0287	109,225	0,0080	77,951

En el caso de TES_{GPU}, su comportamiento fue distinto al de $MBSES_{GPU}$. Se puede observar (Figura 5.4 (b)) cómo influyen las características del problema (fases de cómputo con menor costo) en la aceleración alcanzada, la cual osciló entre 30x y 40x dependiendo de la GPU. Además es bueno destacar que el tamaño de la señal no incide en el desempeño, éste se mantiene cercano al constante, independiente de las dimensiones de la entrada.

La figura 5.5 muestra el througput (cantidad de señales procesadas por segundo). En ambas AFP, el througput muestra un comportamiento dependiente del tamaño de la señal, a medida que las dimensiones del audio se incrementan, menor cantidad de señales son procesadas por segundo.



(a) $MBSES_{GPU}$



(b) TES_{GPU}

Figura 5.5: Througput de $MBSES_{GPU}$ y TES_{GPU} en 3 GPU

En la figura 5.6 mostramos la relación entre el througput de ambas AFPs. Se puede observar que la $MBSES_{GPU}$ al poseer una complejidad mayor de computación que TES_{GPU}, el tamaño

de la señal no afecta drásticamente su comportamiento. Todo lo contrario ocurre en TES_{GPU} donde queda expuesta la influencia.

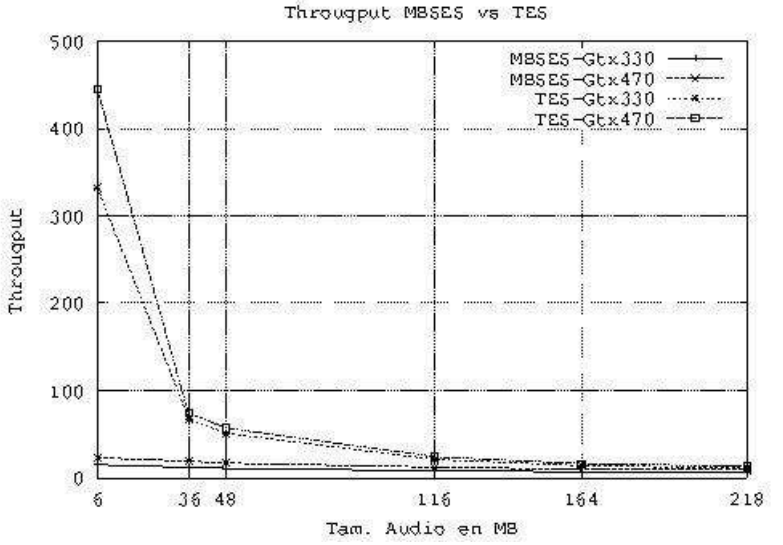
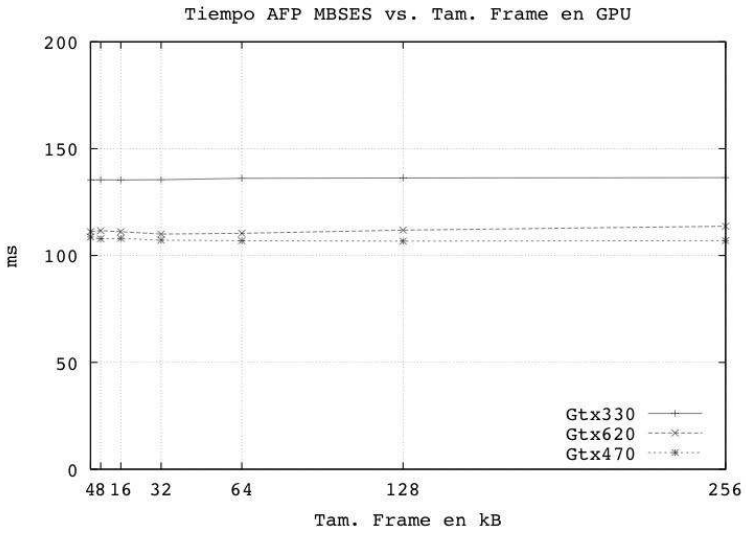


Figura 5.6: Througput de AFP $MBSES_{GPU}$ vs TES_{GPU}

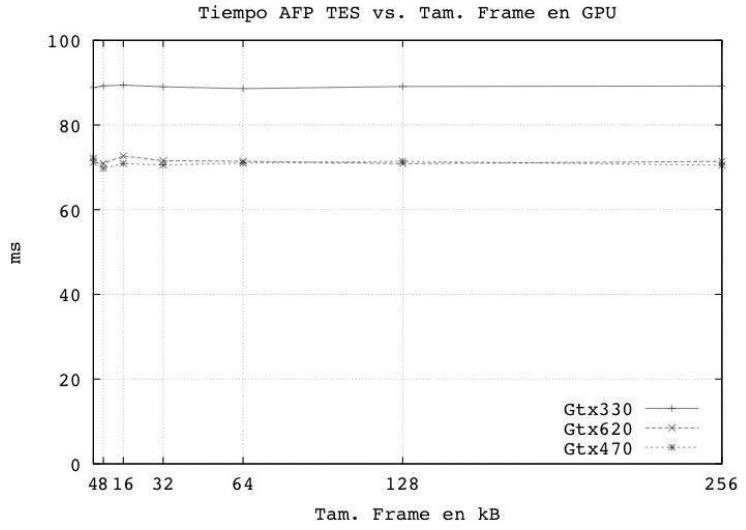
La figura 5.7 muestra los tiempos de ejecución de la señal de audio más grande del lote de prueba (A-218) para distintos tamaños de frame. Podemos observar que el tamaño del frame no influye significativamente en el tiempo de ejecución de ambas huellas, razón por la cual se tomaron los estudios realizados por [CI07] estableciendo los tamaños en 16KB para $MBSES_{GPU}$ y 64KB para TES_{GPU}.

5.7. Resumen

El proceso para obtener la AFP de una señal de audio insume muchos recursos computacionales. Obtener la AFP mediante la entropía, explicado en el capítulo 2, es un buen problema para resolverlo aplicando técnicas de computación paralela, HPC. Particularmente en esta tesis nos enfocamos en la computación paralela en GPU.



(a) $MBSES_{GPU}$



(b) TES_{GPU}

Figura 5.7: Tiempos de Ejecución de $MBSES_{GPU}$ y TES_{GPU} en 3 GPU para A-218 y distintos tamaños de frame

En este capítulo se presentan las soluciones en GPU para obtener la AFP *MBSES* (trabaja con la señal en el dominio de la frecuencia) y la AFP *TES* (se basa en una señal en el dominio

del tiempo), detallándose los aspectos de diseño e implementación tenidos en cuenta a fin de lograr soluciones con buen desempeño. Finalmente se muestran los resultados empíricos, evaluando el comportamiento de cada solución en distintas GPUs, y se analizan sus aspectos distintivos respecto a las soluciones existentes en la bibliografía.

Espacios Métricos en GPU

La Recuperación de objetos multimedia, en particular señales de audio, en una base de datos modelada a través de un espacio métrico demanda muchos recursos, uno de ellos es el tiempo de resolución de las consultas. Poder acelerar la obtención de las respuestas a través de la aplicación de técnicas de computación de alto desempeño es una solución válida. Particularmente aquellas técnicas relacionadas a GPGPU (expuestas en el capítulo 4).

En este capítulo se presenta el estado del arte considerando los espacios métricos y la GPU, y las características del método propuesto en esta tesis, Top-K, detallando tanto los aspectos algorítmicos como los de implementación. Finalmente exponemos los resultados obtenidos y realizamos un análisis de los mismos.

6.1. Estado del Arte

En esta sección mostramos el contexto y los avances realizados respecto a la aplicación de la GPU como solución a distintos problemas concernientes a los espacios métricos. Estos problemas, como por ejemplo el cálculo de distancias y la búsqueda de los k vecinos más cercanos (k-NN), tienen un alto costo computacional.

Como se expuso en el capítulo 3, la búsqueda por similitud o proximidad consiste en encontrar los elementos u objetos más similares a un elemento buscado. Una de las consultas por proximidad más utilizada es la búsqueda de los k-NN. Dada una base de datos (BD) y un elemento de consulta q, el problema de búsqueda k-NN consiste en encontrar los k elementos de la BD que se encuentran más cerca o a una distancia menor de q.

El problema de encontrar los *k*-NN de un objeto es ampliamente estudiado en diversas áreas como en grandes repositorios de datos para la recuperación de información y búsquedas de objetos; en minería de datos para desarrollar técnicas de clasificación, especialmente en clasificación de texto; en procesamiento de imágenes, aplicado en algoritmos geométricos de visión por computadora y en el contexto de visión 3D, para la registración de una nube de puntos 3D; en reconocimiento de patrones y aprendizaje de máquina, para métodos de regresión y clasificación; entre otras. Particularmente nosotros nos enfocamos en el área de base de datos.

La solución directa para obtener los *k*-NN es simple pero computacionalmente costosa, debido a que involucra la comparación entre el objeto de consulta y todos los elementos de la BD. Si tenemos una BD de cardinalidad *n* se tiene para cada consulta *n* evaluaciones de distancia y, generalmente, este número mide la complejidad del algoritmo. En [CNBYMOl] se proponen distintos algoritmos, los cuales representan una solución al problema *k*-NN de manera secuencial.

Cuando el tamaño de la BD y el número de consultas a ser resuelto aumenta, el tiempo de ejecución en una máquina secuencial es demasiado alto. Una solución a este problema es aplicar técnicas de computación de alto desempeño usando tecnologías emergentes como lo son las GPUs.

Existen distintos métodos en GPU para abordar el problema de búsqueda por *k*-NN, algunos aplican el Método de fuerza bruta ([BGTPlO, Barll, GDBO8, GDNBlO, KHlOa, KZO9, LWLJO9, LLWJO9, LLWJlO]), otros se basan en la construcción de un índice para reducir el número de evaluaciones de distancia y así encontrar la solución de los *k*-NN ([BGT+ll, Barll, QMNO9]).

En las próximas secciones se explicarán cada uno de los métodos propuestos en la literatura para resolver el problema de *k*-NN utilizando GPU. En todos los casos consideramos tanto los aspectos de diseño como de implementación.

6.1.1. Método de Fuerza Bruta

Debido al alto costo computacional que posee el método de *Fuerza Bruta*, ejecutado de manera secuencial (explicado en el capítulo 3), existen varias propuestas que resuelven el problema

de los *k*-NN aplicando este método en la GPU, ellas son [BGTPlO, GDBO8, GDNBlO, KHlOa, KZO9, LWLJO9, LLWJO9, LLWJlO]. Todas tienen aspectos comunes como el cálculo de distancias y se diferencian principalmente en el método de ordenamiento. A continuación se realiza un análisis de las distintas propuestas.

6.1.1.1. Aspectos Algorítmicos

Como se expuso anteriormente, todas las soluciones analizadas de la bibliografía, básicamente coinciden en el cálculo de la función distancia, primera etapa en todas las propuestas. Para obtener las distancias entre un objeto de consulta q y los n objetos de la BD, n unidades de procesamiento son activadas, donde cada una calcula la distancia entre un objeto de la BD o_i y q. Este proceso se realiza en forma paralela, por lo tanto cada distancia es calculada en forma simultánea con el resto.

En [BGTP10, Bar11, LWLJ09, LLWJ09, LLWJ10] se resuelve en forma paralela una única consulta por vez, es decir en esta etapa se obtiene como resultado un vector de distancias. Mientras que en [GDB08, GDNB10, KH10a, KZ09] proponen resolver múltiples consultas en paralelo, consecuentemente el resultado es una matriz de distancias de tamaño $m \times n$, donde n es la cantidad de objetos en BD y m el número total de consultas a resolver, cada fila es el vector de distancias de una consulta.

En las etapas de ordenamiento y obtención de los *k*-NN las distintas propuestas presentan diferencias, las cuales se detallan a continuación:

1) En [BGTP10, Bar11] se obtienen los *k*-NN según tres versiones distintas:
   a) Versión 1 *"Ordering Reduction"*: Esta versión aplica el *Quicksort* paralelo propuesto por Cerdeman en [CT08]. Éste consta de dos etapas: la etapa de *Partición* y la de *Ordenamiento*. La primera se inicia seleccionando un pivote global a partir del cual, en paralelo, se divide la secuencia en 2 subsecuencias, una con los elementos menores al pivote y la otra con los mayores. Este proceso se repite iterativamente sobre cada subsecuencia hasta que el tamaño de las mismas sea menor o igual a un límite establecido por el desarrollador. En cada iteración y para

cada subsecuencia se selecciona un pivote de la misma. *P* particiones implican *P-1* iteraciones de selección de pivotes (total de pivotes=*P-1*) y la partición de la subsecuencia. Las particiones obtenidas al finalizar esta etapa, son de diferentes tamaños, dependiendo de la selección del pivote. Esto puede llevar a un desbalance de la carga de trabajo de los distintos grupos. Luego de la etapa de partición, en la etapa de ordenamiento, cada subsecuencia es ordenada aplicando *quicksort* paralelo iterativo. Cada unidad de procesamiento ordena, en paralelo a las demás, su respectiva secuencia. Cuando el tamaño de la subsecuencia generada tiene menos elementos que los establecidos como límite inferior para aplicar el *quicksort*, se finaliza el ordenamiento usando el *bitonic sort* propuesto en [GRH+05].

b) Versión 2 *"Heap Reduction"*: El ordenamiento se basa en la administración de heaps de máximos. Un heap es una estructura de datos de tipo árbol binario donde su contenido mantiene un orden. La característica principal de un heap de máximos es que el nodo raíz posee el elemento mayor en el heap y en todo subárbol se cumple que el padre es mayor a sus hijos. En este caso el heap contiene las distancias mínimas encontradas en la secuencia. Para llevar a cabo este proceso, el algoritmo se estructura en tres fases, donde cada una de ellas realiza las siguientes tareas:

   i) Fase 1: Las distancias entre el objeto de consulta y los elementos de la base de datos son distribuidos en *T* unidades de procesamiento, siguiendo una distribución circular. Cada una de ellas, almacena sus elementos en un heap (cada unidad de procesamiento trabaja con $\frac{n}{T}$ elementos). Una vez que el heap tiene *k* elementos, se insertará uno nuevo sólo se éste es menor que la raíz. Al finalizar esta etapa se tienen *T* heaps de tamaño *k*. esta fase tiene una complejidad de $O(\frac{T}{Tamaño\ de\ Secuencia} \times \log(k))$ en el peor de los casos y de $O(\frac{T}{Tamaño\ de\ Secuencia})$ en el mejor.

   ii) Fase 2: Toma como entrada los $k \times T$ elementos resultantes de la fase anterior (*T* heaps de tamaño *k*) y construye 32 nuevos heaps de tamaño *k*, según la fase 1 (el valor 32 se corresponde con el número de threads por *warp*).

iii) Fase 3: En base a los 32 heaps de k elementos, siguiendo el método de la fase 1, se construye un heap de tamaño k, el cual contendrá el resultado de la consulta.

c) Versión 3 *"Bulk Heap Reduction"*: Este algoritmo sigue el procedimiento descripto en la versión 2, para resolver múltiples consultas en paralelo. En la versión 2 se usa un único grupo de unidades de procesamiento para resolver una consulta, en esta versión varios grupos trabajan para resolver en paralelo una consulta cada uno.

2) En [GDB08, GDNB10] se propone realizar el ordenamiento de las distancias entre el objeto de consulta y los elementos de la BD según los algoritmos *Comb Sort* y una variante del *Insertion Sort*.

El algoritmo *Comb Sort* es uno de los algoritmos más rápidos y fáciles de implementar, el cual tiene una complejidad $O(n \times \log(n))$ para el caso promedio y el peor caso. Éste es una variante del algoritmo de ordenamiento *por burbuja*, pero a diferencia de éste realiza las comparaciones entre dos elementos que están a una longitud mayor o igual a 1. La distancia entre ambos elementos está determinada por un factor de 1.3 (se ha demostrado empíricamente que el algoritmo tiene mejor desempeño con dicho valor). Inicialmente, el algoritmo establece la distancia o espacio entre los elementos a comparar como $\frac{Tamaño\ Lista}{1{,}3}$ y ordena la lista según este valor. Iterativamente el espacio se va dividiendo por el mismo factor y la lista se va ordenando hasta obtener un espacio de 1. El *Comb Sort* continúa usando el inter-espacio de 1 hasta que la lista este completamente ordenada (esta etapa es equivalente al algoritmo *por burbuja*).

El algoritmo que propone una variante del *Insertion Sort*, asume que las primeras k distancias están ordenadas e inserta una nueva distancia i en la posición correcta, es decir $d[i] < d[k]$. Para valores pequeños de k este algoritmo es más rápido que el *Comb Sort*.

La diferencia entre [GDB08] y [GDNB10] es que este último [GDNB10] propone el uso de la biblioteca CUBLAS [NVI13a], para acelerar el proceso de obtención de los k-NN. CUBLAS es una biblioteca de álgebra lineal para GPU, especializada en operaciones de vector y matrices. Ésta es utilizada para mejorar la etapa de cálculo de distancias, etapa con mayor tiempo de computación.

La complejidad de esta propuesta, si es resuelto en forma secuencial, los autores la expresan como $O(n \times m \times d)$ (donde m consultas, n elementos de la BD y d dimensión de los objetos) más $O(n \times m \times \log(m))$ para los m ordenamientos requeridos, aplicando tanto el algoritmo *Comb Sort* como *Insertion Sort*.

3) En [KH10a], en la fase de obtención de los *k*-NN, se utiliza una estructura de heap de al menos *k* elementos, organizada en orden descendente. Cada grupo de unidades de procesamiento calcula los *k*-NN de una consulta de Q (Q es el conjunto de consultas). Cada unidad de un grupo chequea si su elemento es más pequeño que el elemento más grande en el heap (raíz), si lo es lo almacena en un buffer local. Al finalizar su procesamiento local, las unidades modifican el heap según sus respectivos buffer locales. Para esta operación son necesarios mecanismos de sincronización. En el trabajo de divulgación no se especifica cómo se arma la estructura heap.
4) En [KZ09], luego de haber calculado las distancias entre los objetos de la BD y las consultas en Q, proponen aplicar el algoritmo de ordenamiento *Radix-sort* desarrollado por [SHG09]. Este algoritmo ordena los elementos procesando sus dígitos en forma individual desde el menos al más significativo. Por lo tanto cada grupo de unidades de procesamiento ordena individualmente sus elementos y realiza un histograma de dígitos locales a cada grupo. Esto es necesario para conocer las posiciones de los elementos. Por último, aplica la suma de prefijos para calcular los desplazamientos globales y ordenar las distancias en orden creciente. Los *k*-NN resultantes de la consulta son los primeros *k* elementos de la secuencia ordenada final.
5) En [LWLJ09, LLWJ09, LLWJl0], para obtener los *k*-NN de la consulta, las distancias calculadas en la fase *cálculo de distancias* se dividen en *P* secuencias de igual tamaño, es decir que existen *P* grupos con *t* unidades de procesamiento cada uno. Cada unidad de procesamiento conoce la distancia entre el objeto o_i que le corresponde y la consulta q. Esta distancia es comparada con el resto de las distancias calculadas en el bloque, obteniendo así un ranking de distancias. Una vez obtenidos los *k*-NN de las *P* secuencias, una única unidad de procesamiento de los *P* grupos selecciona el elemento más pequeño de las secuencias, el cual formará parte del resultado. Este proceso se realiza iterativamente hasta obtener los *k*-NN de la consulta.

Los autores proponen también resolver múltiples consultas siguiendo el mismo procedimiento de resolución de una única consulta, para ello dividen el proceso en dos etapas:

a) Etapa 1 *Cálculo de distancias*: Cada unidad de procesamiento realiza el cálculo de *r* distancias entre *r* objetos de consulta y cada objeto o_i de la BD. De esta forma se logra realizar el proceso de evaluación de múltiples distancias en paralelo.
b) Etapa 2 *Ordenamiento*: El resultado de la etapa anterior es una matriz de $r \times n$ donde *r* es el número de consultas realizadas en paralelo ($r <= |Q|$) Las *n* columnas de la matriz de distancias se dividen en *P* grupos, cada uno posee *t* unidades de procesamiento ($t = \frac{n}{p}$). Cada grupo está asignado a una porción de la matriz, una submatriz de $r \times t$ donde cada unidad de procesamiento aplica iterativamente el método explicado para una consulta. Se obtiene como resultado los *k*-NN del grupo de las *r* consultas. Por último para obtener los *k*-NN de cada consulta, *r* unidades de procesamiento (cada unidad trabaja en una consulta) seleccionan los *k* elementos más pequeños de la consulta asignada.

6.1.1.2. Aspectos de la Implementación

Como se manifestó anteriormente, todos los algoritmos coinciden en el cálculo de distancias. Previo a este cálculo, en [BGTPl0, BGT+ll, Barll, KHl0a, KZ09, LWLJ09, LLWJ09, LLWJl0] se transfiere la BD a la memoria global de la GPU para luego proceder con la evaluación de distancias. Sólo en [GDB08, GDNBl0], la BD se almacena en la memoria de texturas de la GPU.

En [BGTPl0, Barll, LWLJ09, LLWJ09, LLWJl0] para la computación de las distancias, se ejecuta un kernel donde cada thread resuelve la distancia entre el objeto de consulta q y un objeto de la BD, logrando coalescencia en la memoria. Por lo tanto este kernel se ejecuta con *B* bloques de un número de threads; particularmente [BGTPl0, Barll] utiliza l28 threads por bloque. El resultado de este kernel es un vector de distancias almacenado en la memoria global de la GPU.

En [GDB08, GDNBl0, KHl0a, KZ09] se resuelven múltiples consultas en paralelo, tanto los elementos de la BD como el conjunto de consultas están organizados en matrices de $n \times d$ (matriz MDB) y $m \times d$ (matriz MQ) elementos cada una (siendo *n* el tamaño de la BD, *d* la dimensión de los elementos y m la cantidad de consultas), el resultado es una matriz de distancias de $n \times m$. Esta matriz resultante se divide en submatrices de $T \times T$, donde cada bloque de threads computa las distancias de la submatriz. Por lo tanto, cada bloque transfiere las filas necesarias de las matrices *MBD* y *MQ* a memoria compartida y cada thread del bloque resuelve una distancia entre el objeto de la BD y el objeto de consulta que le corresponde, almacenado el resultado en la submatriz correspondiente. El número de bloques necesarios para resolver todas las distancias de $Q\ es\left(\frac{m}{T}\right) \times \left(\frac{n}{T}\right)$. Específicamente en [KZ09] el valor de T es 16 entonces cada bloque trabaja con 256 threads cada uno. En [LWLJ09, LLWJ09, LLWJ10] se propone también una solución para múltiples consultas, seleccionando *r* consultas para el cálculo de distancias ($r \geq 1\ y\ r \leq Q$) y el proceso de obtención de distancias es el mismo, la diferencia está en que cada thread es el encargado de computar las *r* distancias entre un objeto de la BD y los *r* objetos de consulta.

Una vez calculadas las distancias, éstas se ordenan por consulta, mediante algún método de ordenamiento, los cuales se implementan en uno o más kernels de la GPU. A continuación se detalla cada uno:

1. En [BGTP10, Bar11], como se explicó en el aspecto algorítmico, se propone tres soluciones al problema de ordenar las distancias. Estas soluciones reciben como parámetro un vector conteniendo las distancias entre un objeto de consulta *q* y los elementos de la BD. Los aspectos relevantes de cada implementación son:
    a) Versión 1 *"Ordering Reduction"*: En esta propuesta se aplica el algoritmo de ordenamiento quicksort (enunciada por [CT08]), el cual está dividido en dos etapas: *partición* y *ordenamiento*. Cada etapa se corresponde con un kernel en la GPU. En la primera etapa, el vector de distancias se divide en *P* particiones, las cuales pueden ser ordenadas independientemente, para esto se ejecuta un kernel con *B* bloques de threads (*B* es igual al número de particiones actuales) y cada bloque genera dos

nuevas particiones a partir de un pivote seleccionado previamente, una con los elementos mayores al pivote y otra con los menores. Este proceso es repetido iterativamente hasta alcanzar un umbral, el cual indica la cantidad adecuada de particiones para ejecutar la segunda etapa. Para establecer la posición donde guardar los elementos se usa la función atómica *atomicAdd*, ésta permite a los threads del bloque cooperar y encontrar la posición correcta donde deben almacenar su elemento respectivo en la secuencia. Para obtener P particiones, al menos P-1 pivotes deben ser seleccionados y el kernel de partición se ejecuta P-1 veces, lo cual involucra P-1 sincronizaciones por barrera y P-1 comunicaciones entre la CPU y la GPU.

Una vez inicializada la etapa de partición, se ejecuta el kernel de ordenamiento, cada bloque de threads se encarga de ordenar una partición usando la estrategia *quicksort*, la recursión se implementa mediante una iteración usando una pila en memoria compartida. Cuando la secuencia a ordenar es pequeña se aplica *bitonic sort* para completar el ordenamiento.

Además de las sincronizaciones necesarias por este algoritmo, uno de los problemas principales es el desbalance de la carga entre los bloques de threads en la segunda etapa, debido a que el tamaño de las particiones depende del pivote seleccionado y éste no puede ser controlado. Por lo tanto aquellas particiones con pequeñas cantidades de elementos deben esperar aquellas que poseen muchos elementos.

b) Versión 2 *"Heap Reduction"*: Cada bloque de threads se encarga de resolver una consulta por completo. El kernel está estructurado en tres fases, las cuales se deben realizar en secuencia y la fase i se inicia cuando finaliza la fase i-1. Por ello son necesarias sincronizaciones entre las fases. La sincronización es hecha a través de sincronizaciones por barrera.

En la primera fase, se distribuyen los elementos del vector de distancias entre los threads del bloque, siguiendo una distribución circular. Cada thread mantiene en su heap los k elementos más cercanos

a la consulta. Para esto es necesario mantener en memoria compartida, un heap de tamaño *k* por cada thread. Este conjunto de heaps es almacenado en una matriz de $k \times T$, donde *T* es la cantidad de threads del bloque. Cada columna de esta matriz representa un heap, de forma tal que la i-ésima columna tendrá los elementos correspondientes al heap del i-ésimo thread.

Una vez que el heap tiene *k* elementos, se insertará un elemento sólo si éste es menor que la raíz del heap (la raíz mantiene el elemento con mayor distancia a la consulta de los *k* posibles). Cuando la distancia del nuevo elemento es menor a la distancia almacenada en la raíz del heap, se intercambia esta última por el nuevo elemento y lo hace descender a la posición correcta en el heap. Al finalizar esta etapa, existen *T* heaps de tamaño *k*.

En la segunda fase, los heaps de la etapa anterior son asignados a los threads del primer warp del bloque según una distribución circular. Cada thread guarda los elementos (si corresponde) en su heap almacenado en memoria compartida. Al finalizar esta etapa se obtiene una matriz de $k \times 32$, donde 32 es la cantidad de thread por warp.

Finalmente en la tercera fase, el primer thread del bloque es el encargado de recorrer, de forma circular, los elementos de los $SIZE_{warp}$ heaps de tamaño *k*, almacenados en memoria compartida para obtener los *k*-NN resultado de la consulta.

El kernel de este algoritmo es ejecutado *Q* veces, siendo *Q* el número de consultas a resolver.

c) Versión 3 *"Bulk Heap Reduction"*: La estrategia explicada anteriormente explota el paralelismo para una única consulta. Este nuevo algoritmo se basa en la estrategia empleada en la versión anterior sólo que explota el paralelismo inter-consultas, a partir de varias consultas en paralelo. El kernel de esta versión posee *B* bloques de *T* threads cada uno. Cada consulta es resuelta por un bloque, por lo tanto este kernel es ejecutado $\frac{Q}{B}$ veces. El tamaño del grid está restringido por las necesidades de la memoria para almacenar los heaps de todos los threads asociados.

2. En [GDB08, GDNB10], luego de haber obtenido la matriz de distancias, se invoca el kernel encargado de realizar el ordenamiento usando la memoria global con accesos coalescentes. En el proceso, cada thread es el encargado de ordenar, en forma secuencial, cada fila de la matriz de distancias ya sea aplicando el algoritmo *Comb Sort* o *Insertion Sort*, según el caso.

3. En [KH10a] la salida de la etapa *Cálculo de distancias* es una matriz de $n \times m$ (*n* tamaño de la BD y *m* cantidad de consultas a resolver). En esta etapa cada bloque de threads está asignado a una fila de la matriz. Las *k* distancias más pequeñas de la consulta son almacenadas en orden descendente en una estructura de heap de tamaño *k*. Una vez que el heap se completa con los *k* elementos, cada thread chequea si su elemento es más pequeño que el *k*-ésimo elemento en el heap, si esto sucede el elemento es almacenado en un buffer en memoria compartida. Al finalizar el chequeo de su secuencia, cada thread modifica el heap según su buffer local. Para realizar este proceso son necesarias sincronizaciones por barrera.
4. En [KZ09] se aplica el algoritmo de ordenamiento *CUDA based Radix Sort* propuesto por [SHG09], en este algoritmo se asume que las claves son de d números digitales y ordena un dígito desde la posición menos significativa a la más significativa. La implementación del *Radix-Sort* está dividido en cuatro pasos:
    a. Cada bloque carga y ordena su baldosa (tile) en memoria compartida usando b iteraciones de división por un bit.
    b. Cada bloque escribe los resultados en memoria global, incluyendo su histograma de dígitos de entrada $2 * b$ y la baldosa ordenada.
    c. Se ejecuta una suma de prefijos sobre la tabla $q \times (2 * b)$, donde q es el objeto de consulta, quien está almacenado por columna para calcular los desplazamientos de los dígitos globales.
    d. Usando los resultados de la suma de prefijos de cada bloque, se copian sus elementos en la posición de salida correspondiente.

    Esta fase de ordenamiento se convierte en el cuello de botella afectando directamente el desempeño de la solución.

5. En [LWLJ09, LLWJ09, LLWJ10] las distancias entre un objeto de consulta q y los objetos de la BD son almacenados en la memoria compartida de cada bloque. Para un thread dado, se obtiene el ranking de su distancia comparándola con el resto de las distancias del bloque. Tales rankings son generados simultáneamente, de esta manera los *k*-NN del bloque son obtenidos (*k*-NN locales), los cuales se ordenan según sus rankings en una cola de *k* elementos en orden ascendente. Por lo tanto se tienen *B* colas de *k* elementos cada uno, donde *B* es la cantidad de bloques. Para obtener los *k*-NN de q, un único thread realiza la tarea en forma iterativa. En la primera iteración, el thread selecciona el elemento más pequeño de las *m* colas, éste es eliminado de su posición actual y forma parte del resultado. El segundo elemento asciende a la posición donde se encontraba el primero. Este proceso se ejecuta iterativamente, hasta obtener los k-NN del resultado.
   El autor propone también resolver múltiples consultas en paralelo. Luego de haber calculado las distancias de las *r* consultas, en forma paralela, se obtiene como resultado una matriz de distancias de $r \times n$ (donde *r* es el número de consultas a resolverse en paralelo, ($r <= |Q|$) sobre la cual se realiza el ordenamiento por consulta.
   Para llevar a cabo el ordenamiento se tienen *P* bloques de *t* threads cada uno ($t = \frac{n}{p}$) Cada bloque trabaja con una porción de la matriz de distancias, una submatriz de $r \times t$, donde cada thread realiza iterativamente el mismo procedimiento aplicado en una consulta. Luego de haber obtenido los *k*-NN locales al bloque de las *r* consultas, *r* threads de los *P* bloques son seleccionados. Cada uno de estos thread se encarga de una única consulta obteniendo los *k*-NN correspondientes.

6.1.2. Método Utilizando Índices Métricos

Como se explicó en el capítulo 3, el uso de un índice métrico evita realizar una búsqueda exhaustiva en la BD, ahorrando de esta manera cálculos de distancia. En las siguientes secciones se detallan los aspectos algorítmicos y de implementación de las soluciones basadas en índice usando GPU ([BGT+11, Bar11, QMN09]).

6.1.2.1. Descripción Algorítmica

En [BGT+11] se proponen dos índices métricos distintos para obtener los *k*-NN de la consulta *q*. Estos algoritmos son: *Lista de cluster* (LC) y *Sparse Spatial Selection* (SSS-Index). Fueron seleccionados por el autor por ser los más populares entre los que no usan una estructura de árbol para podar el espacio de búsqueda en forma eficiente. Además, estos índices se administran en matrices, razón por la cual los hacen adecuados para GPU.

LC se basa en clustering y radio cobertor. SSS-Index, en cambio, es un algoritmo basado en pivotes, el cual realiza una preselección de objetos de la BD (pivotes), quienes sirven para filtrar los objetos en una consulta utilizando la desigualdad triangular, sin medir realmente la distancia entre el objeto de consulta y todos los objetos descartados. Ambos algoritmos se explicarán en detalle a continuación:

1. La forma de resolver los *k*-NN con LC es usando consultas por rango creciente. Se inicia la ejecución de la consulta con un radio inicial R_{ini} y repite la búsqueda por rango incrementando el radio un Δ hasta que los *k*-NN de la consulta son encontrados (Los valores de R_{ini} y Δ son establecidos empíricamente con un análisis off-line de la BD). El ajuste del rango se realiza cuando se encuentra el valor mínimo.

   La resolución de k-NN utilizando LC se realiza en una secuencia de cinco etapas. Esta solución utiliza heaps, los cuales son reducidos aplicando el método heap reduction explicado en [BGTPlO]. Cada etapa realiza las siguientes tareas:

   a. Primera etapa: Establece el rango inicial y cada unidad de procesamiento colabora para copiar la consulta a resolver en memoria.
   b. Segunda etapa: Cada unidad de procesamiento, siguiendo una distribución circular, obtiene la distancia entre un centro del cluster y la consulta q.
   c. Tercera etapa: Los elementos de los clusters son asignados a las unidades de procesamiento siguiendo una distribución circular. Cada elemento perteneciente a un cluster que no puede ser descartado usando la desigualdad triangular, es comparado con q. Cuando el elemento se encuentra dentro del rango de búsqueda, se inserta en el heap de la unidad de procesamiento a la que pertenece.

Luego los centros son distribuidos entre las unidades de procesamiento, por cada uno se controla si se encuentra dentro del rango de búsqueda, si es así se inserta en el heap de la unidad de procesamiento respectiva.

d. Cuarta etapa: Las primeras 32 unidades de procesamiento acceden a los elementos de los heaps de la etapa anterior. Cada una de estas unidades almacenan los elementos en su heap respectivo (fase 2 del algoritmo explicado en [BGTPlO])
e. Quinta etapa: La primera unidad de procesamiento de cada grupo, se encarga de recorrer los 32 heaps de la etapa anterior y encontrar los k-NN de la consulta.

2. Al igual que LC, utiliza el método de rango creciente. Además cada unidad de procesamiento administra un heap y realiza la reducción por heap explicada en [BGTPlO].
   Los elementos de la BD son asignados a las unidades de procesamiento siguiendo una distribución circular y cada una se encarga de descartar los elementos que le corresponden utilizando todos los pivotes. De no ser posible el descarte, la misma unidad de procesamiento realiza la evaluación de distancia entre la consulta *q* y el elemento para verificar si es parte de la respuesta.
   La solución de los *k*-NN utilizando el índice *SSS-Index* se realiza siguiendo la siguiente secuencia de etapas, donde las tareas de cada una son:
   a. Primera etapa: Las unidades de procesamiento colaboran para copiar la consulta a resolver en memoria.
   b. Segunda etapa: Cada unidad de procesamiento (siguiendo una distribución circular) obtiene la distancia entre un pivote y la consulta.
   c. Tercera etapa: Cada unidad de procesamiento, siguiendo una distribución circular, tiene en cuenta una de las distancias calculada en la etapa anterior. Cada una intenta descartar el elemento mediante desigualdad triangular y en caso de no ser posible, la misma unidad de procesamiento realiza la evaluación de distancia entre el elemento y *q*. Además cada unidad de procesamiento, si corresponde, almacena los elementos en su heap.

d. Cuarta y Quinta etapa: Son iguales a las correspondientes de la LC.

3. En [QMN09] propone resolver el problema de *k*-NN utilizando *kd-tree*, para ello y antes de resolver la consulta, se construye el árbol *kd-tree* balanceado hacia la izquierda.
Propone usar colas de prioridad. La solución se realiza en tres etapas, en la primera se extrae de la cola el elemento con distancia mínima al objeto de consulta. Luego, esta segunda etapa, se expande el nodo extraído, insertando el mayor en la cola y expandiéndose el menor, este paso es repetido hasta que se alcance un nodo hoja. Por último en la tercera etapa se actualiza la lista de *k*-NN para obtener el resultado de la consulta. No se dan detalles del proceso de obtención de los *k*-NN propiamente dichos.

6.1.2.2. Descripción de la Implementación

Las características de implementación de cada propuesta son:

1. *LC*: Son necesarias dos matrices y un vector para implementarlo en la GPU, estas estructuras son: *CLUSTERS*, *CENTROS* y *RC*. *CENTROS* es una matriz $d \times size_{centros}$ (d es la dimensión de los elementos y $size_{centros}$ es la cantidad de centros de clusters), cada columna en la matriz representa un centro del cluster; *CLUSTER* es una matriz $d \times size_{clusters}$ ($size_{clusters}$ es el número de elementos por cada cluster, el cual es fijo), cada columna representa un elemento del cluster y los elementos de un mismo cluster se encuentran en columnas contiguas. Por último, RC es un vector de tamaño $size_{centros}$, el cual contiene los radios cobertores de cada cluster. Las matrices son alojadas en memoria global y la disposición detallada obedece a favorecer el acceso contiguo a memoria global.
Con respecto al método de rango creciente, se deben realizar I iteraciones para resolver una consulta $(1 \leq I)$. Debido a esto y teniendo en cuenta que cada bloque resuelve una consulta completa, la carga de trabajo de cada bloque varía según el valor de I en cada consulta. Esto produce un desbalance de carga significativo entre los multiprocesadores de la GPU, degradando el

rendimiento. Para intentar reducir este problema, se realizan dos lanzamientos de kernels, en el primero se resuelven todas las consultas con $rango = r_{ini}$ y las que necesitan más iteraciones se dejan en una cola de pendientes para ser resueltas en un segundo lanzamiento de kernel. Esto permite a los bloques realizar la misma cantidad de trabajo en el primer kernel, mientras que en el segundo sólo habrá una diferencia con aquellos bloques que resuelvan consultas que requieran $I \geq 3$ (siendo estas consultas un porcentaje menor). Con esta nueva solución y al ser menor la cantidad de bloques distribuidos entre los multiprocesadores, mejora el rendimiento.

Al igual que la implementación basada en heaps ([BGTP1O]), también utiliza un heap por cada thread para mantener los k elementos más cercanos a la consulta.

Como una consulta es resuelta en un bloque, las etapas del algoritmo están delimitadas por sincronizaciones por barrera. Antes de comenzar se establece el rango inicial, el cual toma en cuenta a qué lanzamiento de kernel corresponde (si es el primer o segundo lanzamiento del kernel). Luego, cada thread, siguiendo una distribución cíclica, obtiene la distancia entre un centro de cluster y q. Esta distancia es almacenada en memoria compartida. En la tercera etapa, tanto los elementos de los clusters como sus centros son asignados a los threads siguiendo también una distribución cíclica. Si se encuentran dentro del rango actual de búsqueda son insertados en el heap del thread. En la cuarta etapa sólo trabajan los 32 primeros threads del bloque. Finalmente en la última etapa, el primer thread de cada bloque se encarga de recorrer y almacenar los elementos de los 32 heaps de la etapa anterior en memoria compartida y obtener los *k*-NN de q. En esta etapa también se determina si se han encontrado los *k*-NN y si es el primer o segundo lanzamiento de kernel.

2. *SSS-Index*: En este índice se requieren tres matrices denominadas *PIVOTES*, *DISTANCIAS* y *MBD*, las cuales son almacenadas en memoria global.

   Al igual que en la LC, se utilizó el método de rango creciente y las consultas son resueltas en dos lanzamientos de kernels. También, cada thread utiliza un

heap y se realiza una reducción de éstos como en la LC. Cada bloque se encarga de resolver una consulta completamente.

Los elementos de la BD se asignan a los threads siguiendo una distribución circular y cada thread se encarga de descartar los elementos que le corresponden utilizando todos los pivotes.

Para obtener los *k*-NN de *q* usando SSS-Index, el algoritmo se divide en las etapas descriptas en la sección anterior. Estas etapas están delimitadas por sincronizaciones por barrera. Como primer paso, los threads del bloque colaboran para copiar la consulta a resolver en memoria compartida. Luego, siguiendo una distribución cíclica, obtienen la distancia entre un pivote y *q*, la cual almacenan en memoria compartida. En la tercera etapa, cada elemento de la matriz *DISTANCIAS* es asignado a un thread siguiendo la misma distribución anterior. Si el thread determina que uno o más de sus elementos son candidatos, los almacena en su heap en memoria compartida. Las siguientes dos etapas son iguales a las etapas de LC, salvo que en la última, el primer thread del bloque además evalúa si se han encontrado los *k*-NN de *q* y si es el primer o segundo lanzamiento de kernel.

3. En [QMN09], como el *kd-tree* es un árbol balanceado hacia la izquierda, es posible representarlo con un vector residente en la memoria global de la GPU. El *kd-tree* se construye en la CPU y se transfiere a la memoria global de la GPU. Para el caso de las colas de prioridad, las mismas se implementan en los registros de la GPU. Como la memoria de registro es escasa en la GPU, se debió fijar la longitud de las colas, cuando éstas se completan, la inserción de los nodos es ignorada.

Todos los algoritmos descriptos anteriormente han utilizado la GPU con el objeto de acelerar los procesos de búsqueda de *k*-NN, mediante la aplicación de algún método como fuerza bruta o de índice, para obtener mejores resultados que los logrados si se ejecutaran de manera secuencial.

En la siguiente sección se presenta el algoritmo propuesto en esta tesis en sus tres versiones, denominado *Top-K*, mostrando tanto los aspectos algorítmicos como los de implementación, además de los resultados obtenidos.

6.2. Consulta por k-NN: Top-K

El algoritmo propuesto, denominado *Top-K*, aplica el método de fuerza bruta para la resolución del problema de *k*-NN usando GPU. Por su naturaleza, este método es altamente paralelizable y por lo tanto es adecuado para resolverse en este tipo de arquitectura.

Siguiendo la filosofía del método de fuerza bruta, el proceso se inicia con el cálculo de las distancias entre el objeto de consulta y todos los objetos de la BD. Una vez realizado este paso, los objetos con las distancias más pequeñas tienen que ser identificados, lo cual se considera un desafío a superar. La idea es seleccionar los objetos en forma independiente, evitando interrelaciones entre ellos (*crosstalk*).

El problema a resolver es equivalente a la siguiente situación: en una habitación con *N* personas, supongamos 1000, cada una recibe un número en un trozo de papel. ¿Cómo se puede determinar si una persona está entre las *k* personas con el número más pequeño en el papel? Un algoritmo de ordenamiento sería costoso, cada persona debe interactuar con las otras personas, ver sus números, compararse y precisar, entre todos, la nueva posición donde se ubicaría. Como restricción de la propuesta hemos establecido que no existe un proceso central de selección de los *k* elementos más pequeños y la ubicación de un elemento en el orden de todos se debe determinar sin interactuar con los demás. Por lo tanto, el mecanismo debe obtener los *k* números más pequeños sin intercambio de información entre todos los presentes. Además la otra condición a tener en cuenta es que la solución computacional debe ser simple, un programa CUDA CPU/GPU funciona mejor si el código es sencillo, con la menor cantidad de condiciones. Por todo lo expuesto, esencialmente necesitamos definir un algoritmo simple para encontrar los *Top-K* elementos más pequeños de una secuencia sin un supervisor central y evitando la comunicación entre las unidades de procesamiento.

Retomando el ejemplo de la gente en la sala, la idea es reunir las personas en pequeños grupos, entre ellos se *"muestran"* el papel. Si todos los números son mayores al mío, entonces tengo el número más pequeño por lo tanto sé que voy a estar entre los finalistas y en silencio paso a la etapa finalista sin hablar con nadie. En la solución computacional, se tiene una única secuencia de acciones, simple y sin bifurcaciones. Estas características permiten una ejecución más rápida, sin necesidad de

sincronizaciones entre las unidades de procesamiento. El proceso se puede repetir hasta que sólo el más pequeño permanece. Extender el algoritmo a los *k* elementos más pequeños es sencillo.

El proceso completo y general se muestra en la figura 6.1. Como los espacios de memoria de la CPU y la GPU son disjuntos, previo al inicio de la búsqueda debe transferirse la BD y las consultas a la GPU. Al finalizar el proceso la transferencia de los resultados es a la inversa, desde la GPU a la CPU.

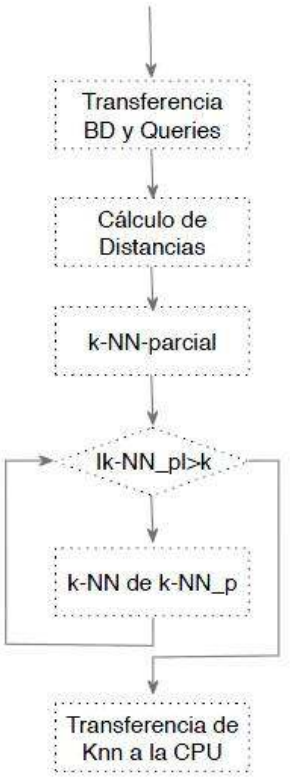


Figura 6.1: Proceso Completo Top-K.

Hemos desarrollado y definido 3 versiones para la resolución del problema, las cuales se diferencian en el proceso de determinación de los menores. Las tres alternativas propuestas las denominamos: *Top- K AA* (Todos contra todos, *All to All*), *Top-K QSeP* (*Quickselect*), *Top- K QSeH (Híbrido: Quickselect + All to All)*. Todas ellas tienen una fase común, el cálculo de las distancias. La diferencia radica en la forma de obtener los *k* elementos más similares a la consulta. La figura 6.2 muestra el

proceso de obtención de los k-NN en forma genérica, donde la diferencia está en la fase de Selección k-NN. Una explicación más detallada es realizada en las secciones 6.2.1 y 6.2.2. Una explicación general de la selección de los k-NN es esquematizada en la figura 6.2.

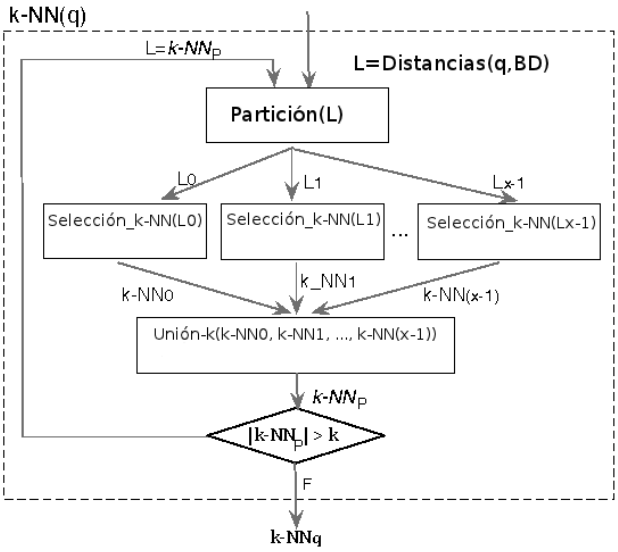


Figura 6.2: Proceso Genérico para obtener los k-NN de una consulta q

Como puede observarse, el proceso itera mientras el tamaño de la lista de las *k*-NN*p* locales es mayor que *k*, al inicio la lista *L* está formada por todas las distancias de la consulta a cada objeto de BD. La lista es dividida, etapa de *Partición*, y asignada a un grupo de unidades de procesamiento. Cada unidad de procesamiento determina sobre qué elemento de la lista L_i debe trabajar según la información local provista por el grupo al que pertenece. La etapa *Selección k-NN* es una unidad de cálculo, su responsabilidad es la elección de los objetos *k*-NN de la lista L_i, donde $\bigcup L_i = L \forall i = 0 \ldots x-1$ y x es la cantidad de grupos trabajando, para luego guardarlos en la lista parcial *k-NN p* (etapa *Unión- k*). Después de la primera iteración, *L* contiene sólo las distancias de los objetos *k*-NN de cada grupo. A partir de esta nueva lista, se continúa aplicando el proceso.

A continuación se explican los aspectos algorítmicos y de implementación de cada una de las propuestas de *Top-K*.

6.2.1. Aspectos Algorítmicos

El algoritmo *Top-K* propuesto en sus tres versiones, se divide en dos etapas bien definidas. En la primera etapa, las distancias entre cada elemento de la BD y el objeto de consulta son procesadas en paralelo. Esto significa que existen *n* unidades de procesamiento calculando al mismo tiempo una distancia correspondiente a la distancia entre un objeto de la BD y el elemento de consulta.

La segunda etapa se encarga de la iteración mostrada en la figura 6.2, en la cual se han desarrollado las tres posibles soluciones. Cada una de las soluciones presenta diferentes características, particularmente en la etapa *Selección k-NN*. Todas las alternativas toman como entrada las distancias de todos los elementos de la BD al objeto de consulta ($L =< d_0, d_1, \dots, d_{n-1} > donde\ d_i = distancia(q, o_i) \forall o_i \in BD$). El proceso se inicia, particionando las distancias en L_i sublistas ($i = 0 \dots x - 1$), las cuales son trabajadas en forma independiente y en paralelo. Cada algortimo desarrollado se caracteriza por:

1. *Top-K AA*: En la etapa *Selección k-NN* se eligen los *k*-NN elementos a partir de la lista local L_i, cada elemento se compara con todos los elementos del grupo para determinar si su distancia está entre los *k* más pequeños del grupo. Una vez determinados los *k* elementos menores de cada una de las L_i sublistas, se unifican en una única lista (*Unión-k* de la figura 6.2). El resultado de la unificación será la entrada de la próxima iteración.
   Este procedimiento se realiza iterativamente hasta obtener los *k*- NN de la consulta (iteración en la figura 6.2). La iteración finaliza cuando la secuencia a tratar tiene *k* elementos, los cuales constituyen los *k*-NN resultados de la consulta. Después de la primera iteración, la lista parcial tiene $k \times x$ elementos donde *x* es el número de grupos de trabajo.
2. *Top-K QSeP*: En esta versión aplicamos la metodología propuesta por el algoritmo *Quickselect* [Hoa61, KubO6], el cual encuentra los *y* elementos más pequeños de una secuencia no ordenada *z*. En la etapa *Selección k-NN* de la figura 6.2 se aplica el algoritmo *Quickselect* en cada una de las L_i listas locales para elegir los *k*-NN de cada grupo. Este proceso se aplica iterativamente mientras que la posición del elemento distinguido, pivote, no es

igual a *k*. En cada iteración, seleccionamos el pivote, el cual es la mediana de tres valores de la lista local (primero, último y el valor que se encuentra en la posición centro). A partir del pivote se realiza la partición de la lista local. Si la posición de pivote es igual a *k*, la partición con los elementos más pequeños que el pivote son los *k*-NN de la lista local. De lo contrario, pueden suceder dos casos posibles: la posición del pivote es mayor que *k*, o es menor que *k*. En el primer caso, se ejecuta el *QuickSelect* sobre la partición con los elementos más pequeños. En otro caso, se trabaja sólo sobre la segunda partición de la lista local, la cual tiene los elementos más grandes que el pivote. Al finalizar la iteración, el pivote se encuentra en la posición *k*, obteniéndose los *k*-NN de la consulta.

3. *Top-K QSeH*: Este algoritmo aplica una combinación de los dos anteriores, razón por la cual se llama *Top-K QSeH*, *H* por híbrido. En esta propuesta, en la primera iteración, mostrada en la figura 6.2, se aplica el algoritmo *Quickselect* a cada L_i resultante de la partición de la lista con todas las distancias. Como resultado se obtienen los *k*-NN de cada lista local al grupo. Luego en los pasos siguientes, se determinan los *k*-NN locales aplicando el algoritmo *Top-K AA* iterativamente hasta obtener los *k*-NN resultados de la consulta.

Los algoritmos planteados anteriormente han sido diseñados respetando la consigna inicial de formular un algoritmo simple y de bajo costo. Como puede observarse en todas las propuestas se mantiene la independencia entre los datos, de manera tal que la interacción entre las unidades de procesamiento es nula.

6.2.2. Aspectos de Implementación

En esta sección se presenta la descripción de la implementación del algoritmo *Top-K* en sus tres versiones. Un aspecto a considerar son las transferencias entre la CPU-GPU, éstas se realizan en dos momentos, primero, previo a la resolución de las consultas en la GPU, se transfiere la BD y las consultas a la memoria global para proceder con la evaluación de las distancias; y al finalizar el proceso *Top-K* para transferir los *k*-NN a la CPU.

El proceso completo de *Top-K* para cualquiera de sus alternativas implica el lanzamiento de dos kernels, uno responsable de realizar el cálculo de distancias y la búsqueda de los primeros *k*-NN en forma local; el otro a cargo de la selección en forma iterativa de los *k*-NN finales. El primer kernel se ejecuta sobre x bloques de 256 threads cada uno, en consecuencia x es igual a $\left\lceil \frac{n}{256} \right\rceil$. El segundo kernel se encarga de obtener los *k*-NN de la consulta y se implementa con un bloque de a lo sumo 512 threads.

Para el cálculo de distancias, se activan en paralelo por los x bloques de threads. Cada thread calcula la distancia entre el objeto de consulta y un objeto de la BD almacenándola en la memoria compartida del bloque al que pertenece el thread. Una vez realizado este proceso, cada bloque obtiene los *k*-NN locales (primera iteración mostrada en la figura 6.2).

A continuación se explica el proceso de obtención de los *k*-NN de la consulta, el cual se corresponde con un segundo kernel. Dependiendo de la versión *Top-K*, las características son:

1. *Top-K AA*: Cada bloque posee en su memoria compartida las distancias que fueron calculadas y almacenadas cooperativamente por los threads del bloque. Hay tantos threads como distancias en la lista local. Cada thread determina si su distancia está entre los *k*-NN de su bloque. La comparación es de todos a todos, cada thread en el bloque compara su distancia frente a todas las distancias en el bloque. Una vez que un thread establece su orden, analiza si su distancia es uno de los candidatos para formar parte de la lista *k*-NN local. Si es verdad, reporta que su objeto pertenece a *k*-NN*p*, de lo contrario finaliza. Al finalizar la primera iteración, el tamaño de la lista *k*-NN*p* es igual a $k \times x$. La posición que van a ocupar los *k* objetos con menor distancia de cada uno de los bloques en la lista *k*-NN*p* se determina mediante el uso de funciones atómicas de CUDA, particularmente en este caso es la función atómica *AtomicSub*. Como la lista local se trabaja en la memoria shared, la función atómica es hecha sobre ésta. El pseudo-código 1 muestra las características básicas de este kernel.
   Al finalizar el primer kernel, la lista *k*-NN*p* contiene los $k \times x$ elementos, los cuales son la entrada al segundo kernel. Este kernel obtiene como resultado los *k*-NN de la consulta, quienes al finalizar el proceso de selección

serán reportados a la CPU. Este proceso es ejecutado por un bloque de a lo sumo 512 threads. Si $k \times x$ es menor o igual a 512 entonces el bloque tendrá $k \times x$ threads.

Algorithm 1 Pseudo-código del Primer Kernel de *Top-K_AA*

```
Kernel k-NN_Parcial (BD, q, L_k-NN_p){
1.  threadID //Identificador global del thread en la aplicación
2.  Copiar memoria shared obj_threadID
3.  dist = Calcular_distancia(q, obj_threadID)
4.  dist_local[threadId.x] = dist //almacena dist en memoria shared
                              //según identificador thread local
5.  Sincronizar Threads()
6.  Compara dist con todos los elementos de dist_local[]
7.  Determinar si dist es menor a todos en dist_local[]
8.  If (Verdadero) entonces
9.    Almacenar dist en L_k-NN_p
}
```

Las distancias seleccionadas son almacenadas en la memoria shared del bloque, cada thread se corresponde con un elemento de la lista *k*-NN*p*. Caso contrario cada thread trabajará con más de un elemento. El procedimiento para obtener los *k*-NN de la consulta es el mismo que se explicó anteriormente para obtener los *k*-NN de cada bloque. El vector es dividido en *L* sublistas, donde cada $|L| = \left|\frac{(L*k)}{512}\right|$ (la primer lista *L* contiene todos los *k*-NN de cada uno de los *x* bloques del kernel anterior, en consecuencia $|L| = x \times k$), de las cuales se obtienen $L * k$ elementos cuyas distancias son las menores. Este proceso se repite iterativamente hasta obtener los *k*-NN, resultado final de la consulta (figura 6.2). Los *k*-NN del resultado de la consulta se encuentran ordenados en forma ascendente. La implementación de este kernel se muestra en el pseudo-código 2.

2. *Top-K QSeP*: Como en *Top-K-AA*, las distancias calculadas en el bloque se encuentran almacenadas en memoria compartida. Este vector de distancias es la entrada del algoritmo *Quickselect* detallado en la sección 6.2.1. Al finalizar este kernel se obtiene como resultado la lista *k*-NN*p* con los $x \times k$ elementos cuyas distancias

son las k menores de cada bloque. El pseudocódigo 3 muestra la implementación del kernel.

Algorithm 2 Pseudo-código del Segundo Kernel de *Top-K_AA*

```
Kernel k-NN_Final(L_k-NN_p, k, Lista_k-NN-Final){
1.   tam = |L_k − NN_p|
2.   While (tam >k){
3.      nro_sublistas = ⌈tam/512⌉
4.      list=0;
5.      While (list < nro_sublistas)
6.         Copiar en mem. shared la sublista de distancias en L_local
7.         Sincronizar Threads()
8.         Comparar dist local con todos los elementos de L_local
9.         Determinar si dist es menor que todos en L_local
10.         If (Verdadero) entonces
11.            Almacenar dist en L_k-NN_p
12.         Sincronizar Threads()
13.         list + +;
        }
14.      tam = (nro_sublistas * k)
     }
15.   Guardar en Lista_k-NN-Final k-NN de q
}
```

El segundo kernel se ejecuta tomando como entrada el resultado del primer kernel, es decir un vector con $x \times k$ elementos. Al igual que la implementación de *Top-K AA*, un único bloque de no más de 512 threads ejecuta este kernel. La diferencia está en la determinación de los k menores, siendo en este caso realizada a través de la aplicación del algoritmo *Quickselect* en cada sublista. Los k-NN de la consulta no necesariamente se encuentran ordenados. Este kernel se muestra en el pseudo-código 4

3. *Top-K QSeH*: Este algoritmo combina las versiones descriptas anteriormente. Cada bloque carga en memoria shared las distancias calculadas por sus threads, le aplica el algoritmo *Quickselect* para obtener los k-NN de cada

bloque y determinar la lista *k*-NN*p*, donde $|k - NN_p| = x \times k$ El pseudo-código 3 muestra la implementación del primer kernel.

Algorithm 3 Pseudo-código del Primer Kernel de *Top-K_QSEP*

```
Kernel k-NN_Parcial (BD, q, L_k-NNp,k){
1.  threadID //Identificador global del thread en la aplicación
2.  Copiar memoria compartida obj_threadID
3.  dist = Calcular_distancia(q, obj_threadID)
4.  dist_local[threadId.x] = dist  //almacenar dist en memoria shared
                                   //según identificador thread local
5.  Sincronizar Threads()
6.  Quickselect(dist_local, k)
7.  Almacenar k objetos en L_k-NN_p
}
```

La lista *k*-NN*p* es la entrada del segundo kernel, el cual aplica la metodología propuesta en *Top-K AA* en forma iterativa hasta obtener los *k*-NN de la consulta. La implementación de esta versión se muestra en el pseudo-código 2.

Al finalizar la ejecución, los *k*-NN de la consulta quedan almacenados en memoria global. Una vez que se procesaron todas las consultas, se transfieren a la CPU los resultados.

6.3. Otras Consultas resueltas con Top-K

Además de resolver consultas por *k*-NN como se explicó en las secciones anteriores, se han implementado las consultas por rango y los *all-k*-NN. Las características de cada una son:

- *Consultas por Rango*: la consulta por rango tiene la particularidad de que su solución es altamente paralelizable, y su implementación en la GPU es trivial. Una vez transferida la BD a la memoria de la GPU, cada thread calcula la distancia entre el objeto de consulta q y un objeto o_i de la BD. Si la distancia es menor al radio requerido, $r \in \mathbb{R}$, entonces o_i forma parte del resultado.

Algorithm 4 Pseudo-código del Segundo Kernel de *Top-K_QSEP*

```
Kernel k-NN_Final(L_k-NNp, k, Lista_k-NN-Final){
1.   tam = |L_k − NN_p|
2.   While (tam >k){
3.      nro_sublistas = ⌈tam/512⌉
4.      list=0;
5.      While (list < nro_sublistas)
6.         Copiar en mem. shared la sublista list de distancias en L_local
7.         Sincronizar Threads()
8.         Comparar dist local con todos los elementos de L_local
9.         Quickselect(L_local,k)
10.        Almacenar dist en L_k-NN_p
11.         Sincronizar Threads()
12.         list++;
       }
13.      tam = (nro_sublistas * k)
    }
14.  Guardar en Lista_k-NN-Final k-NN de q
}
```

Este proceso se repite iterativamente por cada objeto de consulta q que pertenece al conjunto Q de consultas, el cual se muestra en la figura 6.3.

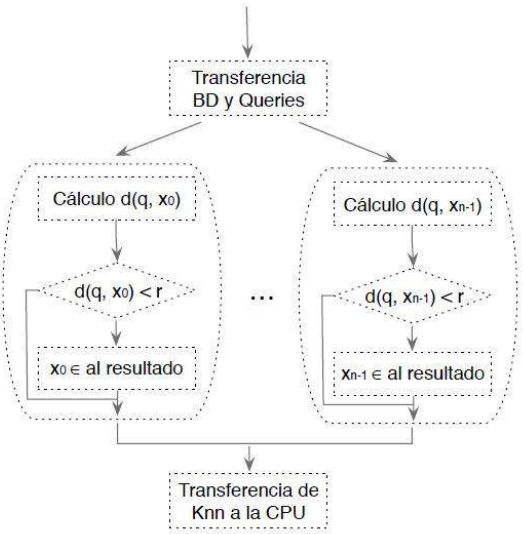


Figura 6.3: Búsqueda por rango

Para resolver una consulta por rango, x bloques de 256 threads son activados, donde $x = \frac{n}{256}$

- *Consulta por all-k-NN*: Se la puede considerar como una generalidad de una consulta por los *k*-NN. En este caso, la BD y el conjunto de consultas coinciden (Q = BD), todos los objetos de BD son considerados objetos de consultas y objetos de la BD con los cuales se debe comparar cada consulta. Todos los resultados son transferidos conjuntamente a la CPU, por ello se debe considerar cómo es la representación y el orden para recuperar un resultado particular.

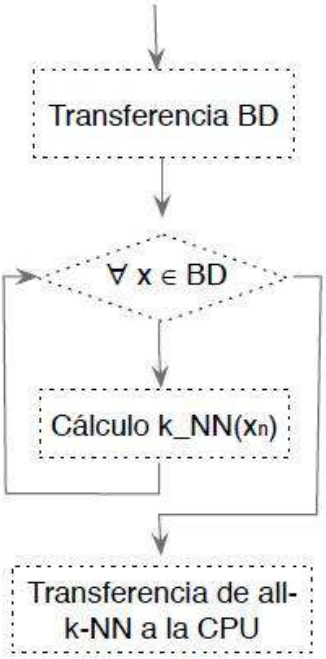


Figura 6.4: Búsqueda por All-k-NN

6.4. Resolviendo Múltiples Consultas k-NN

En sistemas de gran escala no es suficiente acelerar una consulta a la vez, también es necesario aprovechar las capacidades de la GPU para responder en paralelo varias consultas. Con el objeto de cumplir con este fin, la GPU recibe un conjunto de consultas y las resuelve todas a la vez. Cada consulta, en paralelo, aplica el proceso de obtención de los *k*-NN explicados en las secciones anteriores.

El número de recursos necesarios para resolver q consultas en paralelo, con $q \subset Q$, es igual a la cantidad de recursos requeridos para calcular una consulta multiplicado por el número de consultas resueltas en paralelo. La cantidad de consultas a resolver, $|q|$, en paralelo se debe determinar considerando los recursos disponibles de la GPU, principalmente su memoria.

Resolver Múltiples consultas en paralelo implica gestionar cuidadosamente los bloques y sus threads. Si *x* es la cantidad de bloques necesarios para resolver una consulta y se resuelven $|q|$ consultas en paralelo, $|q| \times x$ bloques son activados al mismo tiempo. Como se accede a bloques, los cuales están resolviendo diferentes consultas en paralelo, es importante tener en cuenta una buena administración de ellos y sus threads: qué consulta se está resolviendo y qué elemento de la base de datos se corresponde con el resultado de la consulta. Esto es posible y fácil de lograr estableciendo una relación entre los identificadores del thread, del bloque, de la consulta y del elemento de la BD.

La resolución de múltiples consultas k-NN en paralelo se aplica a la solución de los all-k-NN. Si se responden q consultas en paralelo, podemos calcular los all-k-NN en X iteraciones, donde $X = \frac{n}{|q|}$. La misma filosofía se puede aplicar para resolver múltiples consultas por rango.

6.5. Resultados Experimentales

En esta sección se presentan los resultados del desempeño del algoritmo propuesto *Top-K* y todas sus versiones para distintas GPUs y tipos de consultas. Al igual que en el caso de las AFPs, los experimentos fueron desarrollados en diferentes escenarios, en las versiones secuenciales se consideró una

computadora con las siguientes características: intel core i3, 2.13 GHz y 3 GB de memoria, mientras que para los experimentos paralelos se tuvieron en cuenta diferentes GPUs, las cuales pertenecen a diferentes generaciones de arquitecturas: G80 y GF100, o tienen distinta cantidad de recursos. En la tabla 6.1 se detallan las características de cada una de las GPUs utilizadas.

Cuadro 6.1: Características básicas de las GPUs

GPU	GT-330M	GTX-470	GTX-620
Memoria. Global	900 MB	1280 MB	1024 MB
SM	6	14	1
SP	8	32	48
Clock - Rate	1040MHz	1215MHz	1400MHz
Capacidad Computación	1.2	2.0	2.1

Para el análisis del comportamiento, hemos seleccionado bases de datos de referencia de *SISAP Metric Library* (www.sisap.org), cada una de ellas tiene las siguientes propiedades:

- Imágenes de la NASA: Es un conjunto de 40150 vectores de características de dimensión 20, generadas a partir de imágenes descargadas desde la NASA. La función de distancia para medir la similitud de los objetos es la distancia euclideana.
- Histogramas de Color: Está formado por 112682 vectores de características con dimensión 112. El origen de los datos son histogramas de color a partir de una base de datos de imágenes. Para este espacio, la función de distancia puede ser cualquier función de forma cuadrática, nosotros elegimos la distancia euclideana como la alternativa significativa más simple.
- Imágenes COPHIR: Es un conjunto de 10 millones de imágenes, cada objeto tiene dimensión 208. Es un subconjunto de una base de datos más grande, que cuenta con alrededor de 106 millones de imágenes de *Flickr*, proporcionada por la colección *SAPIR*. La función de distancia utilizada es euclideana.

- English: Es un diccionario de palabras inglesas (69069 elementos). La función de distancia es la distancia de edición (número mínimo de inserciones, eliminaciones y sustituciones necesarias para hacer a dos palabras iguales).
- Vocab: es el vocabulario de un índice invertido, tiene 494048 elementos. Se utiliza la distancia de edición como función de distancia. La particularidad en esta colección es que la distancia de edición tiene complejidad cuadrática en el tamaño de las cadenas, mientras que en el ejemplo anterior la distancia tiene complejidad lineal.

Para evaluar el comportamiento de nuestras soluciones con respecto a un algoritmo secuencial, seleccionamos el índice métrico *SAT* +, el cual es una versión mejorada del árbol de Aproximación Espacial (*SAT*), este método está explicado en la sección 3.4.

Cada uno de los valores reportados es el promedio de 100 ejecuciones de 1000 consultas sobre la base de datos. El número de consultas en paralelo resueltas en la GPU fue de 80 para todas las bases de datos a excepción de Vocab, para la cual se consideraron 30 consultas en paralelo por el tamaño de la BD.

Dividimos el análisis de los resultados en los distintos tipos de consultas, cada uno detallado en las siguientes secciones.

6.5.1. Consulta por Rango

Como se expuso en la sección 6.3 la resolución de la búsqueda por rango es trivial, cada thread responde por un objeto de la BD, si la distancia del objeto es menor al radio establecido, el thread lo reporta como uno de los resultados. Los radios dependen de la BD con que se trabaja, para el caso de los vectores los radios de consulta fueron los mostrados en la tabla 6.2 y para los strings tuvimos en cuenta los valores 1, 2, 3 y 4.

Cuadro 6.2: Rangos para Nasa y Colors

Radios BD Nasa	Radios BD Colors
0,12	0,051768
0,285	0,082514
0,53	0,131163

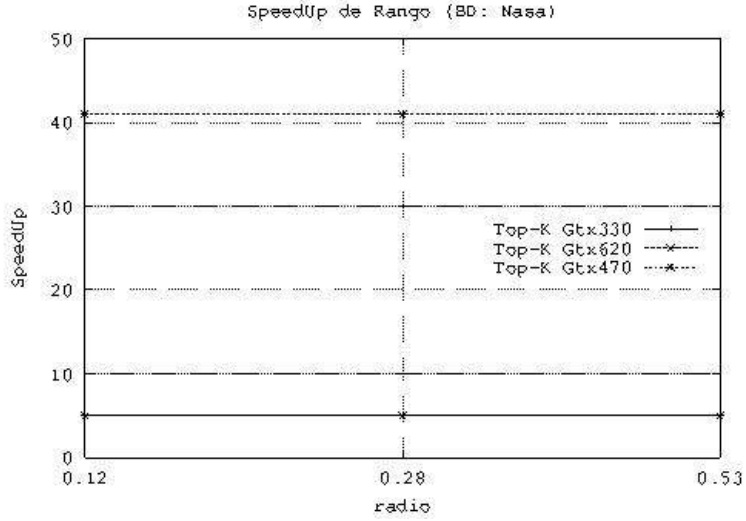
En la tabla 6.3 mostramos los tiempos de ejecución en milisegundos para la búsqueda por rango secuencial y paralela en la GPU Gtx470. Podemos ver en ella cómo el tiempo del proceso de la búsqueda por rango en la GPU (Tiempo Proc.) es casi constante, no influye la cantidad de objetos en la BD, ni la dimensión de los mismos, ni el radio considerado. Además en dicha tabla discriminamos los tiempos de transferencia, quienes en todos los casos dominan ampliamente el tiempo total del proceso. Si analizamos el proceso secuencial completo de búsqueda por rango (Tpo *SAT* + con árbol) respecto al paralelo (Tiempo Total) se puede observar una gran diferencia en los tiempos.

Cuadro 6.3: Tiempos para resolver consulta por rango con SAT+ y *Top-K*

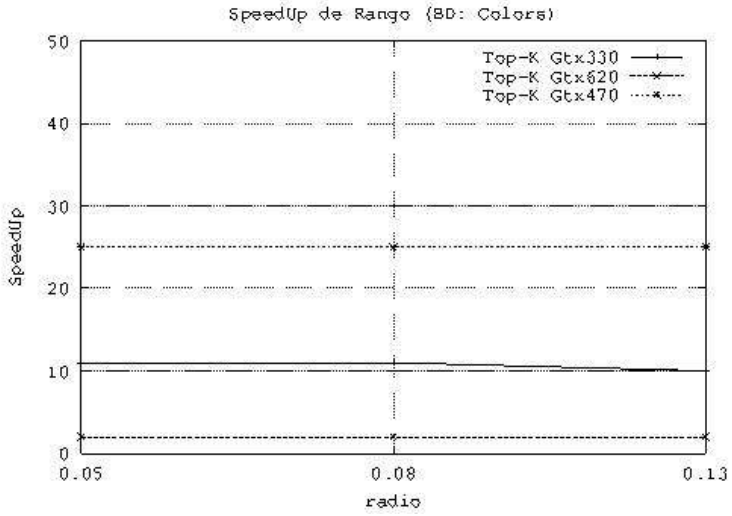
BD	Radio	SAT+		Tiempo			
		Con árbol	Sin árbol	Proc.	Transf. Res.	Transf. BD	Total
Nasa	0,12	995,902	0,163	0,0050	23,005	0,86	23,87
	0,285	996,351	0,612	0,0053	23,045	0,86	23,91
	0,53	997,422	1,683	0,0052	23,007	0,857	23,873
Colors	0,051	17284,73	0,273	0,0054	672,19	15,334	687,54
	0,082	1784,79	0,332	0,0051	672,191	15,335	687,53
	0,13	17285,16	0,7	0,0055	672,502	15,331	687,842
English	1	5,776	11456,223	0,00530	2,444	0,274	2,728
	2	12,166	11462,613	0,00507	2,448	0,275	2,731
	3	16,632	11467,079	0,00515	2,445	0,281	2,728
	4	20,875	11471,322	0,00515	2,445	0,272	2,728
Vocab	1	12,132	295566,523	0,00458	10,297	2,141	12,445
	2	67,181	295621,572	0,00458	10,305	2,149	12,453
	3	120,334	295674,725	0,00444	10,287	2,133	12,435
	4	163,824	295718,215	0,00470	10,293	2,132	12,442

En la figura 6.5 mostramos las aceleraciones obtenidas para las BD de vectores: Nasa y Colors (En todos los casos consideramos procesos completos). El comportamiento es similar en todas las GPUs utilizadas, se mantiene constante en cada uno, las diferencias entre ellas obedecen a la cantidad de recursos que poseen. Las aceleraciones alcanzadas en la GPU con más recursos son de aproximadamente 40x para Nasa y 25x para Colors.

Para el caso de las BDs de string: English y Vocab, las aceleraciones logradas son mostradas en la figura 6.6. Al igual que en vectores se consideran los procesos completos tanto en el secuencial (*SAT* + con construcción de árbol) como en el paralelo. En ambos casos exhiben un comportamiento similar al caso de los vectores, no importa el valor del radio, el tiempo casi es el mismo



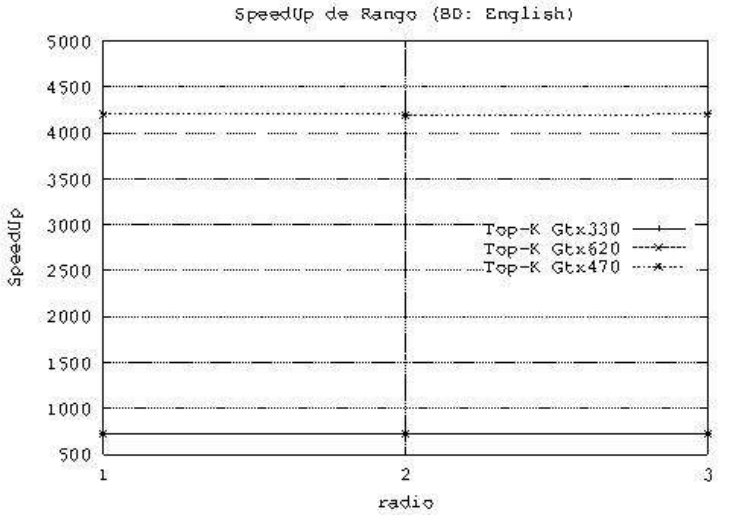
(a) Nasa



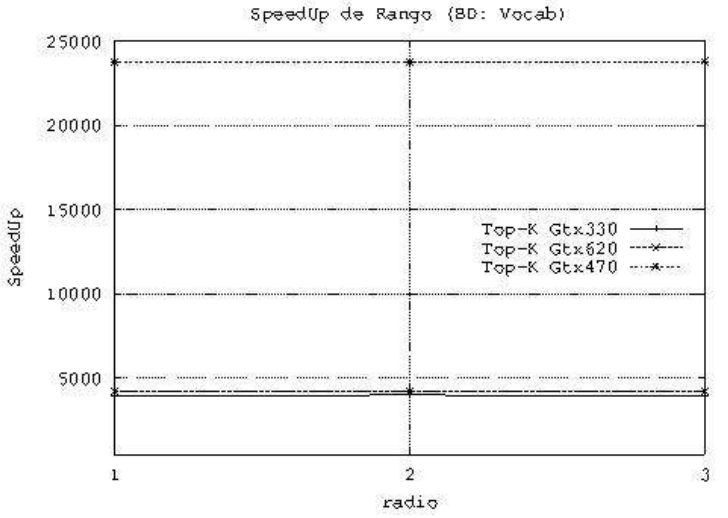
(b) Colors

Figura 6.5: Aceleración de la Consulta por Rango para BD de Vectores en 3 GPUs

(la diferencia son a lo más 5ms). La aceleración alcanzada es muy buena, por ejemplo en el caso de la Gtx470 es de 4000x para English y de 23000x en Vocab, mientras que en el caso de la GPU con menor cantidad de recursos y arquitectura inferior, la Gtx330, las aceleraciones fueron de 700x y 4000x respectivamente.



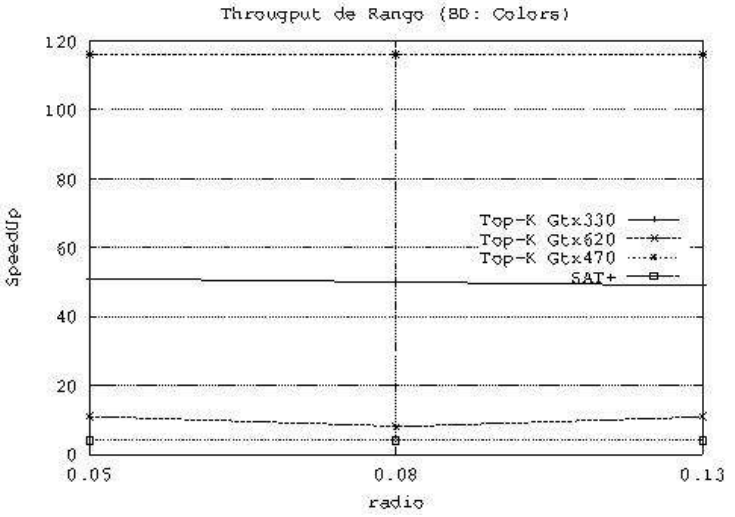
(a) English



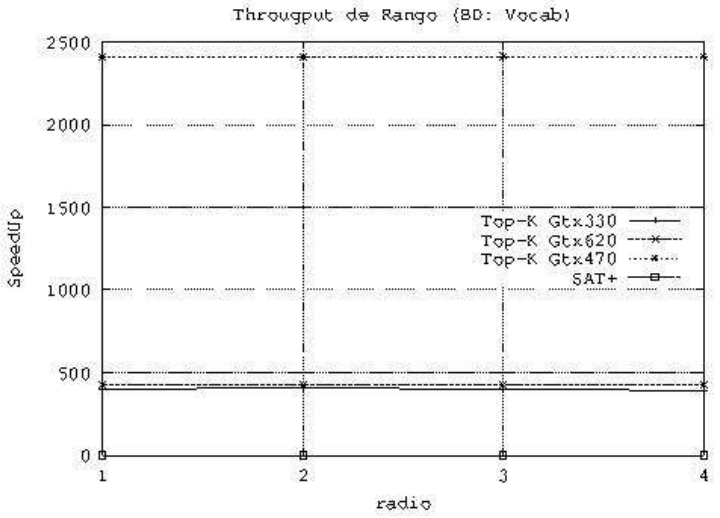
(b) Vocab

Figura 6.6: Aceleración de la Consulta por Rango para BD de String en 3 GPUs

En la figura 6.7 mostramos el througput (cantidad de consultas respondidas por segundo) para Colors y Vocab, tanto del proceso paralelo como del secuencial. Seleccionamos estas BDs por ser las de mayor cantidad de objetos (vectores y cadenas de caracteres respectivamente). Del análisis de los resultados se puede observar que el algoritmo secuencial puede resolver 4 consultas en Colors pero requiere más de un segundo para resolver una consulta en Vocab. En el caso del algoritmo paralelo resuelve 116 consultas en un segundo en Colors y 2400 en Vocab. Todos estos en la GPU Gtx470.



(a) Colors



(b) Vocab

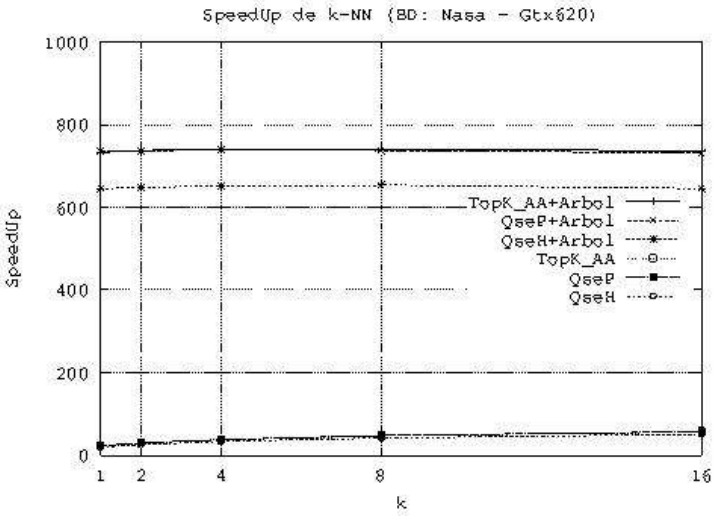
Figura 6.7: Througput de la Consulta por Rango paralela y secuencial para Colors y Vocab

6.5.2. Consultas k-NN

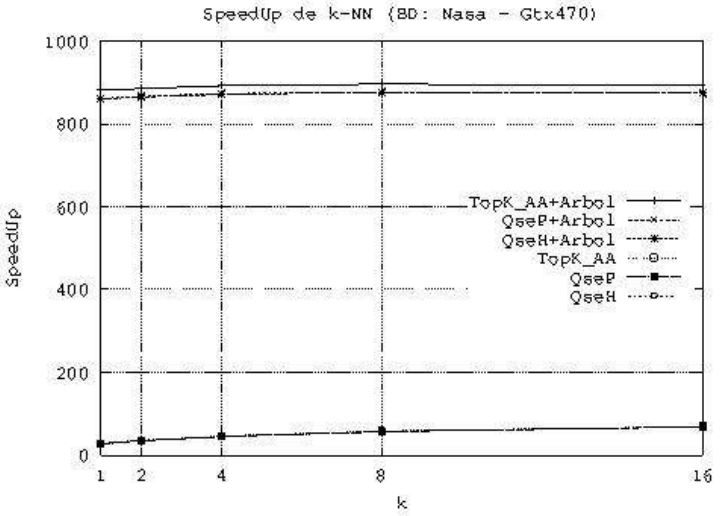
Para este tipo de consultas se evaluó a *Top-K* y todas sus versiones considerando a *k* igual a 1, 2, 4, 8 y 16.

En las figuras 6.8, 6.9, 6.10 y 6.11 mostramos las aceleraciones obtenidas para las respectivas BDs: Nasa, Colors, English y Vocab en las tres GPUs. Hemos comparado el proceso secuencial teniendo en cuenta la construcción del árbol ({*TopK_AA, QSeP, QSeH*}+Árbol) y sin ella con cada una de las versiones de *Top-K*.

En todos los casos y dadas las características de la GPU, la GTX470 es donde se obtienen los mejores speedups.

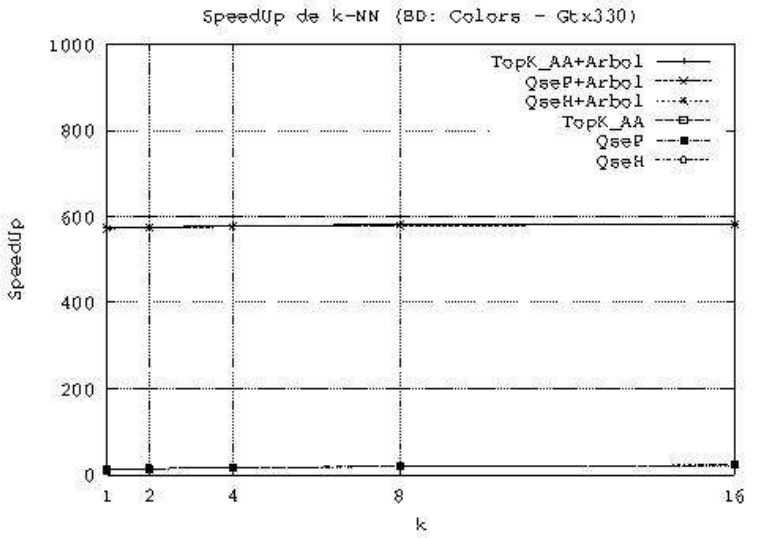


(a) GTX620

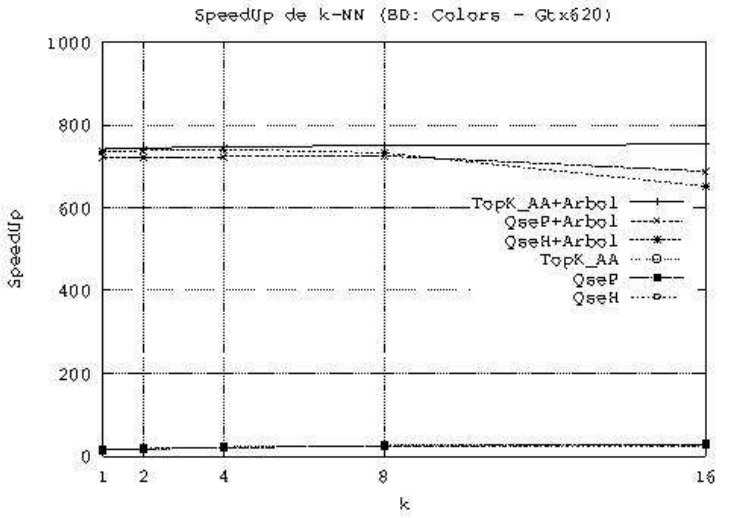


(b) GTX470

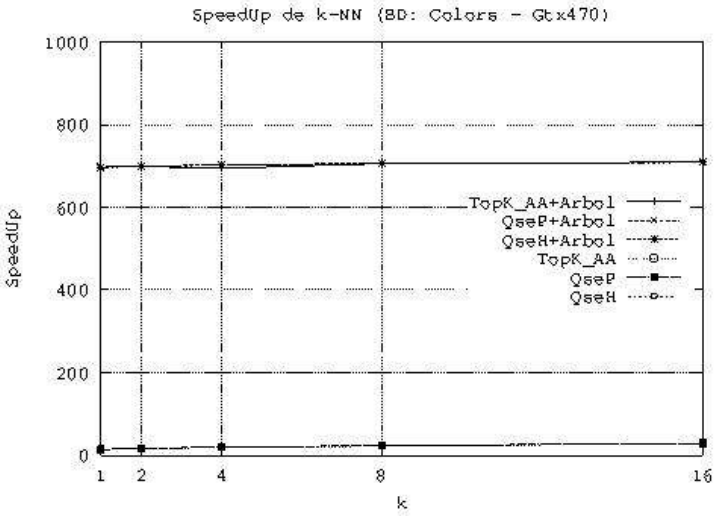
Figura 6.8: Aceleración de la Consulta por k-NN para Nasa en 2 GPUs



(a) 330M



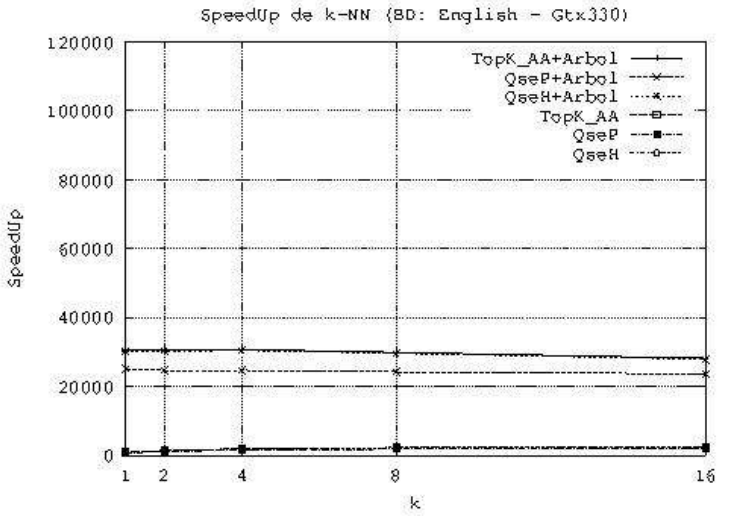
(b) GTX620



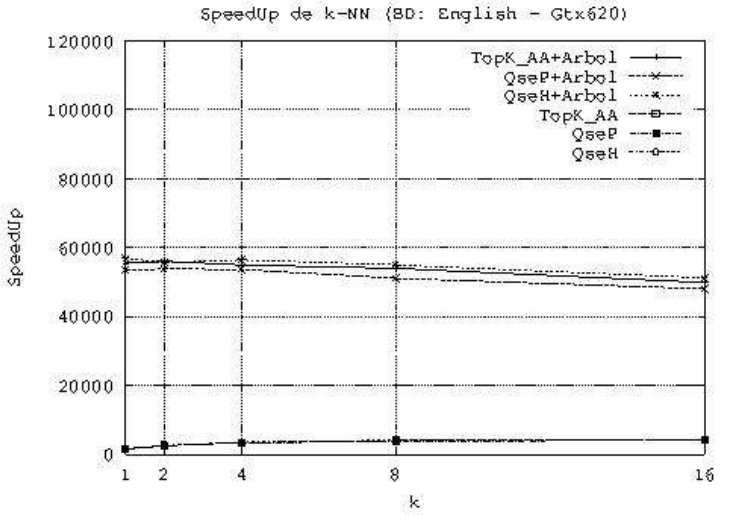
(c) GTX470

Figura 6.9: Aceleración de la Consulta por k-NN para Colors en 3 GPUs

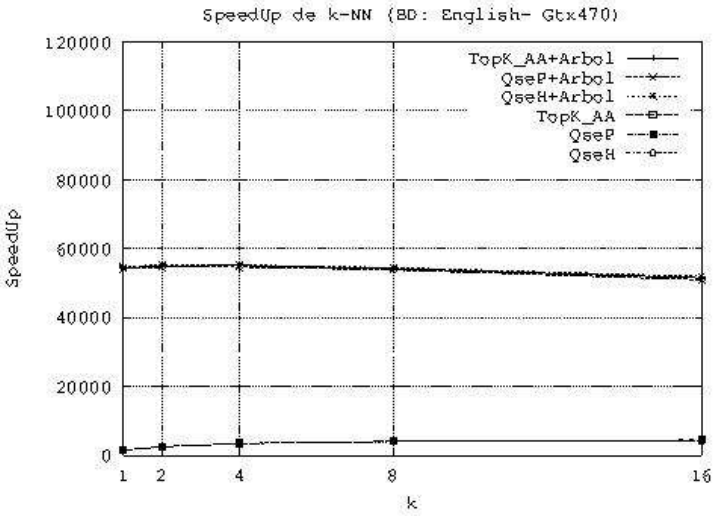
También podemos observar que en caso de considerar el proceso completo en secuencial, la aceleración es mucho mayor. Si no consideramos la construcción del árbol, la aceleración es levemente creciente según *k*. El comportamiento se invierte si se considera al árbol. Además no existe una versión de *Top-K* que muestre una superioridad respecto a las otras para todas las BDs, ni un patrón para una BD o una GPU.



(a) 330M

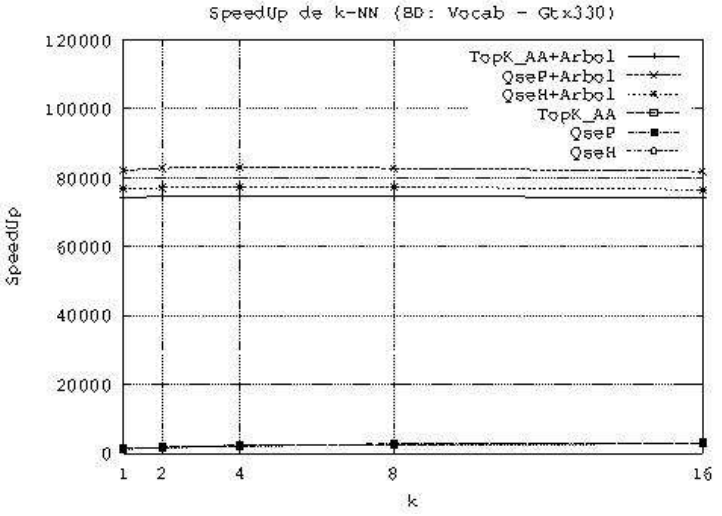


(b) GTX620

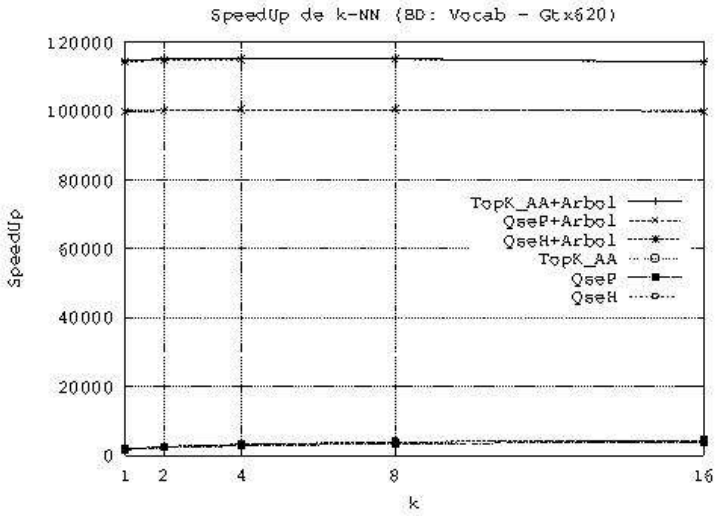


(c) GTX470

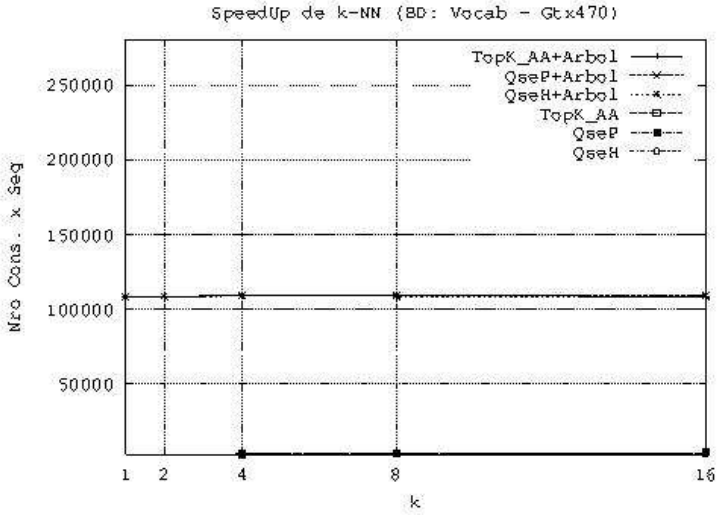
Figura 6.10: Aceleración de la Consulta por k-NN para English en 3 GPUs



(a) 330M



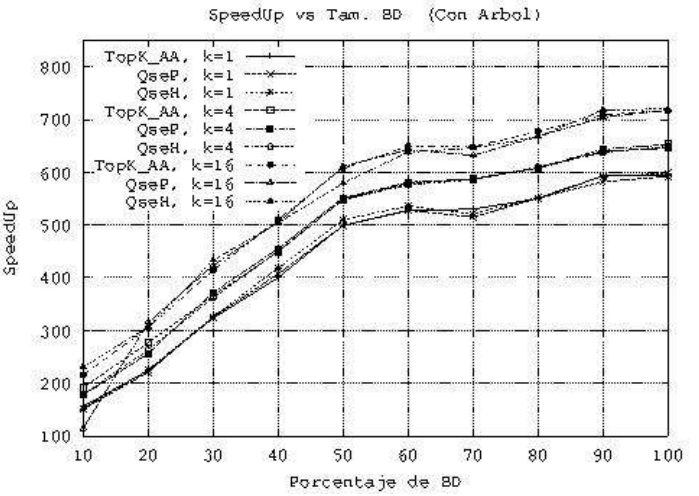
(b) GTX620



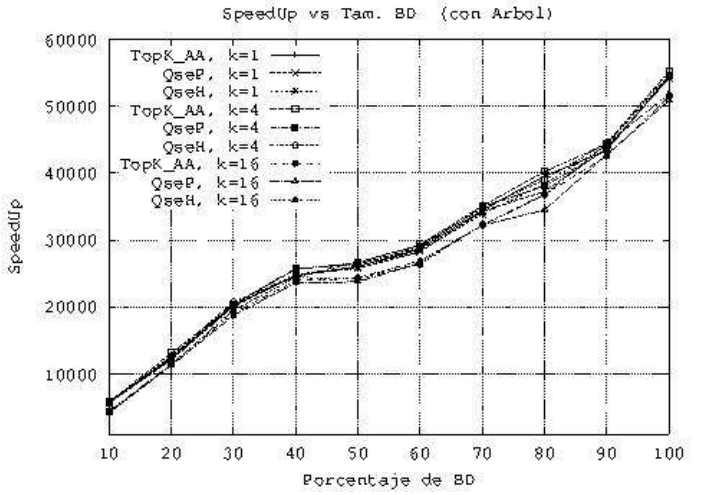
(c) GTX470

Figura 6.11: Aceleración de la Consulta por k-NN para Vocab en 3 GPUs

Otro aspecto analizado fue la escalabilidad de *Top-K* respecto al tamaño de la BD. Para ello consideramos dos BDs y variamos su tamaño desde el 10 % hasta el 100 %, calculando los $k = 1$, $k = 4$ y $k = 16$ vecinos. En la figura 6.12 mostramos la aceleración del proceso completo para las consultas *k*-NN de Colors y English. En ellas se puede observar la importante influencia de la construcción del árbol y cómo va aumentando el speedup a medida que aumenta el tamaño de la BD, siendo más pronunciado el crecimiento para los tamaños más pequeños (Cuando la cantidad de objetos es más pequeña ocurren dos cosas, se subutiliza los recursos de la GPU y las transferencias tienen mayor influencia que el tiempo de procesamiento).



(a) Colors



(b) English

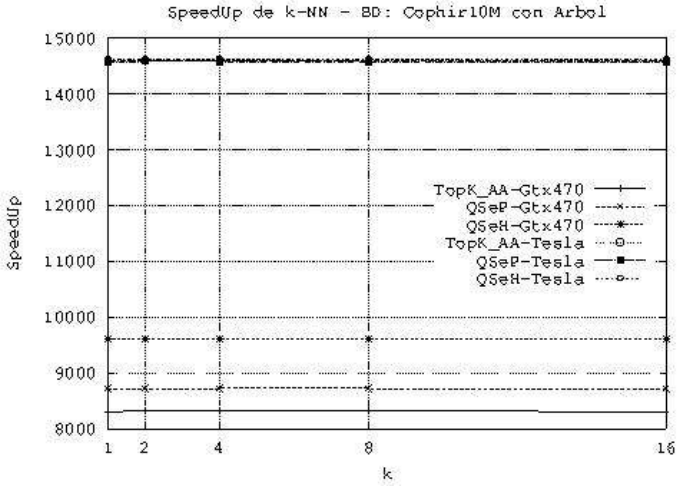
Figura 6.12: Escalabilidad para Colors y English en Gtx470

Otro análisis realizado fue respecto al comportamiento de *Top-K* cuando la BD tiene un tamaño más grande que la memoria global de la GPU. Cuando tenemos una BD muy grande, para responder las consultas es necesario realizar el tratamiento en segmentos o trozos de igual tamaño.

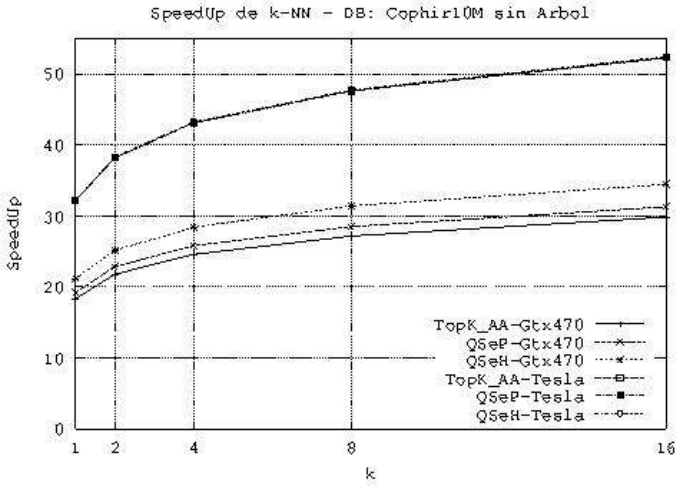
El proceso de *k*-NN realizado en la GPU es el mismo aplicado para las BDs más pequeñas (explicado en la sección 6.2), la diferencia radica en las transferencias realizadas al principio y al final de la computación de *k*-NN. Al inicio se transfieren las consultas y un segmento de la BD, se computan los *k*-NN parciales del segmento para todas las consultas y se carga el siguiente segmento a la GPU para aplicarle el mismo proceso. Los *k*-NN de cada parte de la BD son parciales, al finalizar el proceso sobre todos los segmentos, se transfieren a la CPU los *k* parciales de todas las consultas, donde se los ordena y computan los *k*-NN finales. El ordenamiento se realiza aplicando el algoritmo quicksort provisto en la librería de C.

Para esta BD se resolvieron 30 consultas en paralelo, los segmentos son de 364MB y las GPUs utilizadas fueron la Gtx470 y una Tesla C2075 con las siguientes características: tamaño de memoria global= 5375MB, número de SM= 14, número de SP= 32, Clock rate= 1.15GHz y capacidad de computación 2.0.

En la figura 6.13 mostramos la aceleración obtenida para Cophir. Podemos observar un speedup levemente creciente si no consideramos el tiempo de construcción del árbol, si éste es tenido en cuenta la aceleración es casi constante por su costo significativo, el más alto del proceso. Los resultados en la Tesla fueron mejores respecto a la Gtx470, logrando una aceleración superior a 50x.



(a) Con Árbol

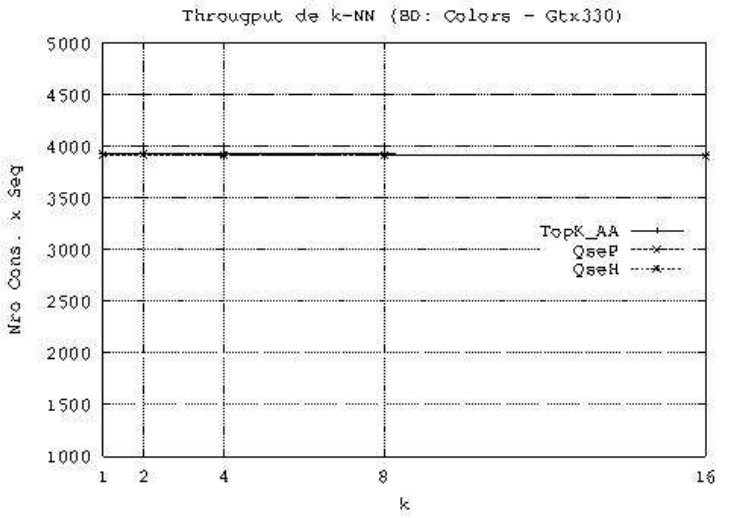


(b) Sin Árbol

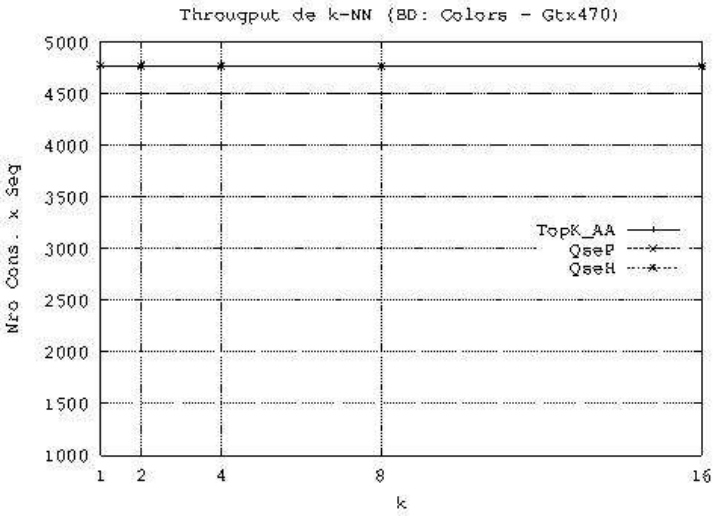
Figura 6.13: Aceleración de Top-K para BD grandes en 3 GPUs

Respecto a la cantidad de consultas en paralelo, hemos evaluado si la cantidad de ellas influye en el desempeño del problema. Para ello evaluamos el comportamiento de Top-K y todas sus versiones para las BDs Nasa, Colors e English en las GPUs Gtx330 y Gtx470. Se tomaron los tiempos involucrados en resolver 1, 10, 20, 30, 40, 50, 60, 70 y 80 consultas en paralelo. En todos los casos, los tiempos se mantuvieron casi constantes, variando entre ellos a lo sumo ±0,2ms.

En las figuras 6.14 y 6.15 medimos la cantidad de consultas k-NN por segundo para las BDs más grandes: Colors y Vocab. El mayor número de consultas resueltas por segundo se corresponden a los valores más chicos de *k*, (1 y 2) en los otros casos desciende débilmente. Por las características de la GPU y su arquitectura, en la Gtx330 se resuelven aproximadamente 3900 consultas por segundo mientras que en la Gtx470 se responden 4700 consultas por segundo aproximadamente para Colors. En el caso de los strings, para Vocab estos números se incrementan llegando a 9500 consultas por segundo en la Gtx330 y 13500 consultas en la Gtx470.

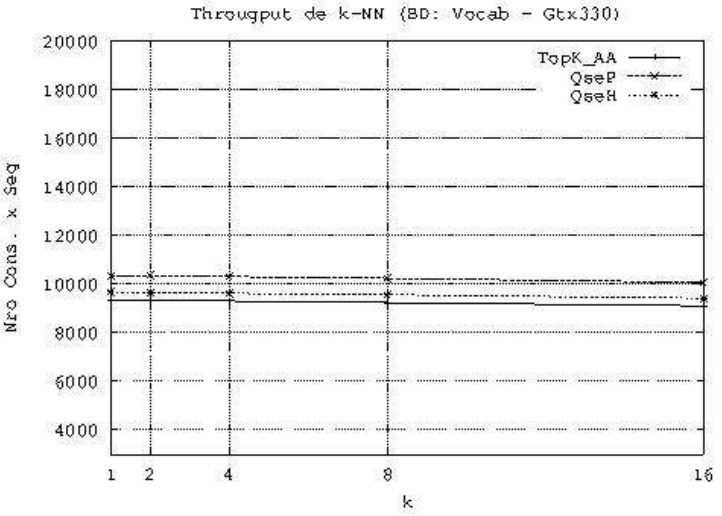


(a) 330M

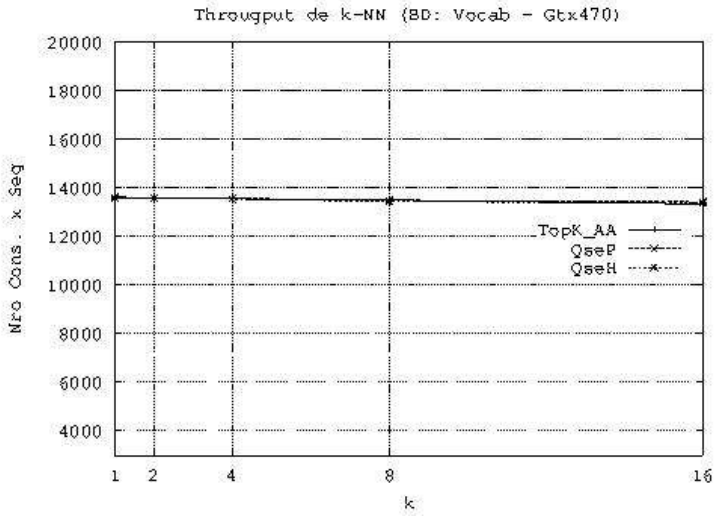


(b) GTX470

Figura 6.14: Througput de los k-NN para Colors en 2 GPUs



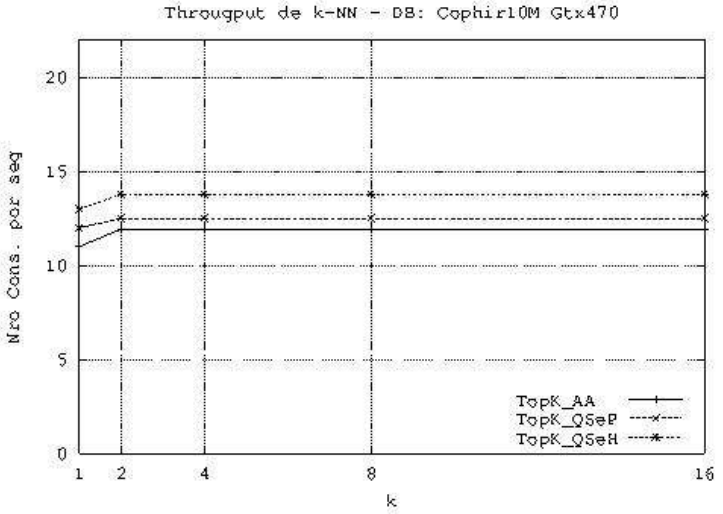
(a) 330M



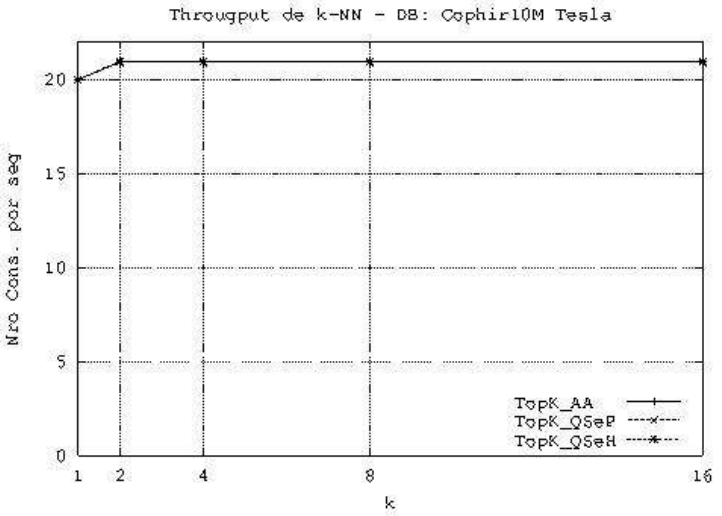
(b) GTX470

Figura 6.15: Througput de los k-NN para Vocab en 2 GPUs

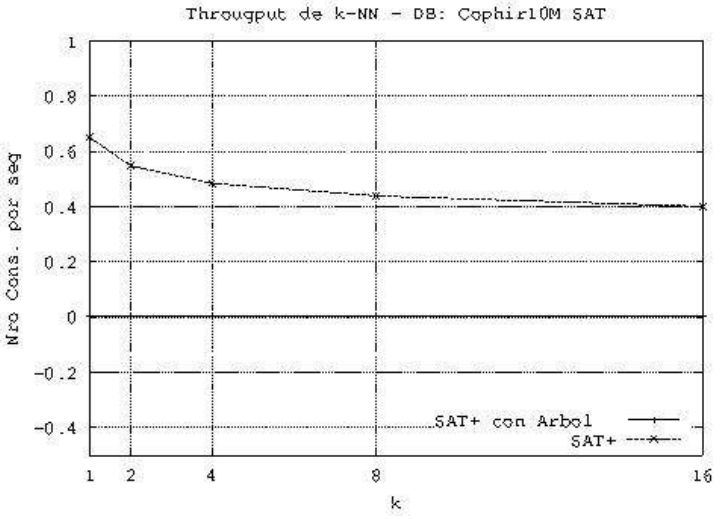
En la figura 6.16 mostramos el througput para la BD grande, donde existen transferencias intermedias entre cómputos. Con *Top-K* se pueden resolver 12 consultas por unidad de tiempo en la Gtx470 y 21 en la Tesla, mientras que en *SAT* + se necesita más de un segundo para resolver una consulta.



(a) GTX470



(b) Tesla

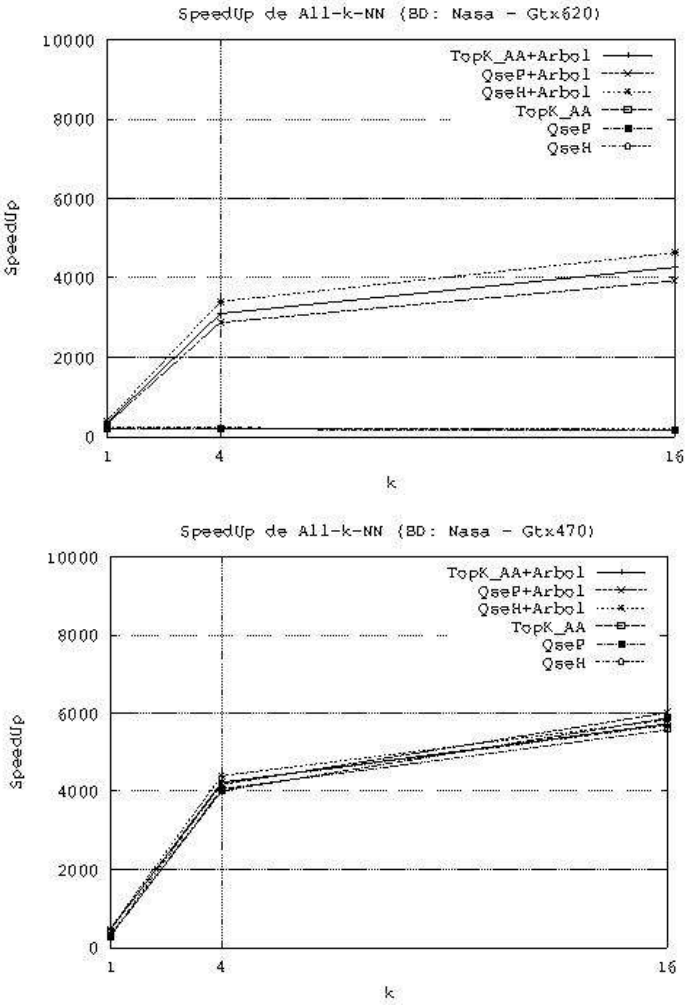


(c) SAT

Figura 6.16: Througput de Top-K y SAT+ para BD grandes

6.5.3. Consulta all-k-NN

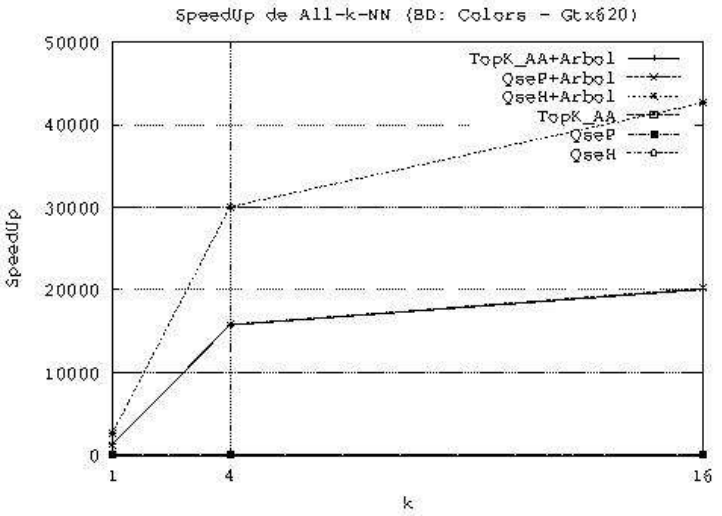
Para esta consulta, en las figuras 6.17 y 6.18 mostramos la aceleración alcanzada para las BDs de vectores en dos GPUs, Gtx620 y Gtx470. En ambos casos y a pesar de las diferencias de recursos, podemos observar un buen desempeño, creciendo en forma pronunciada hasta k = 4 y luego en forma más moderada.



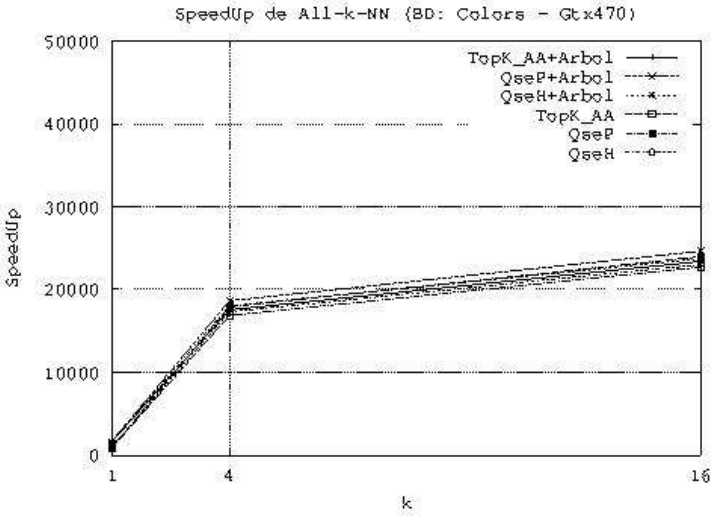
(a) GTX620

(b) GTX470

Figura 6.17: Aceleración de all-k-NN para Nasa en 2 GPUs



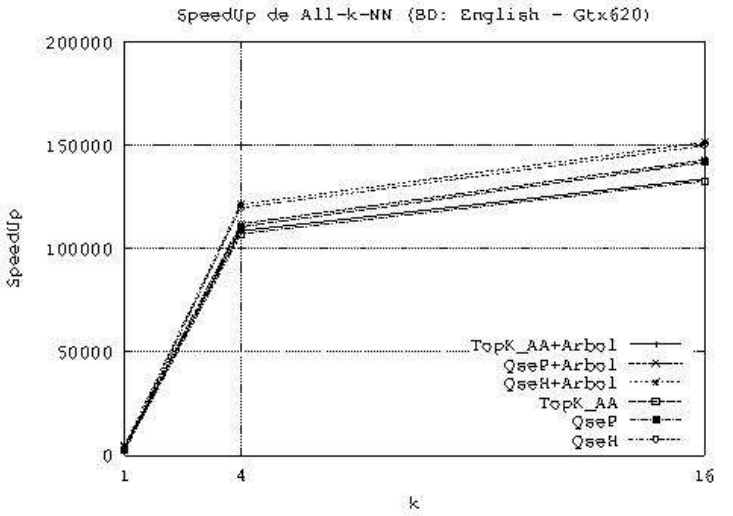
(a) GTX620



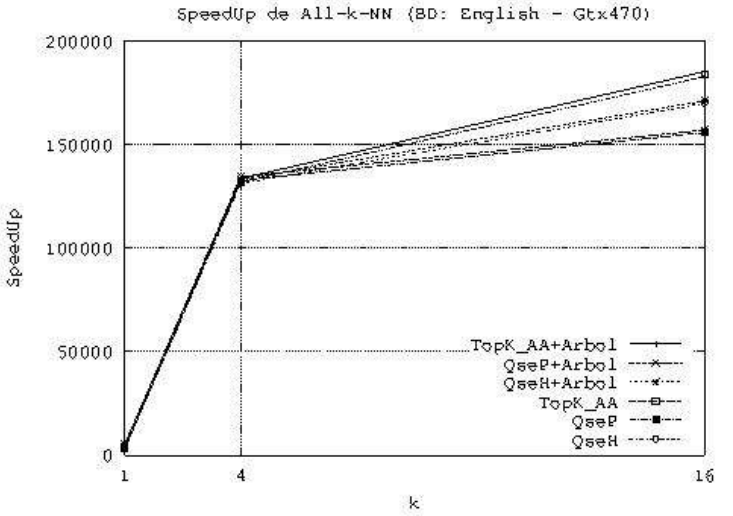
(b) GTX470

Figura 6.18: Aceleración de all-k-NN para Colors en 2 GPUs

En las figuras 6.19 y 6.20 se muestran las aceleraciones para los string. Al igual que en los vectores, se observa un crecimiento más rápido hasta k = 4. Esto obedece a que el costo de las transferencias se incrementa según k y n.

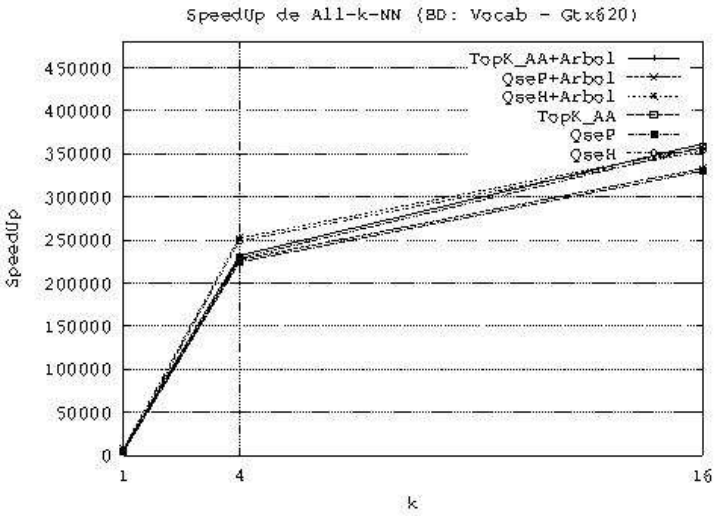


(a) GTX620

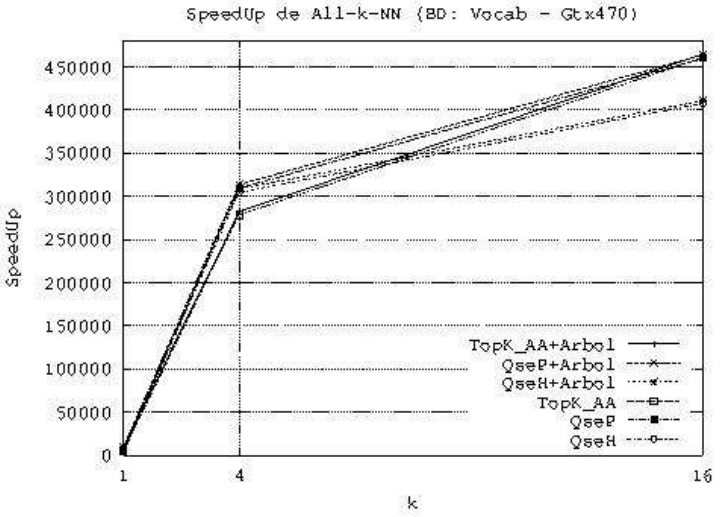


(b) GTX470

Figura 6.19: Aceleración de all-k-NN para English en 2 GPUs



(a) GTX620



(b) GTX470

Figura 6.20: Aceleración para all-k-NN para Vocab en 2 GPUs

Además podemos destacar el comportamiento similar de las aceleraciones cuando se considera la construcción del árbol y cuando no, esto obedece a que el tiempo de encontrar todos los *k*-NN de todos los objetos de la BD es mucho mayor que el de construcción del árbol.

La consulta por all-k-NN es una generalización de la búsqueda por *k*-NN, por ello y al igual que en los *k*-NN, en todos los casos pudimos observar un excelente desempeño de las soluciones en la GPU respecto a su contraparte secuencial, aún si las características de la GPU son limitadas. Además podemos ver que independientemente del número de consultas a resolver, siempre se logran buenas aceleraciones.

6.6. Top-K vs Estado del Arte

En esta sección realizamos un cuidadoso análisis de las ventajas y desventajas de nuestra propuesta, Top-K, respecto a las soluciones detalladas en la sección 6.1.

Entre las características sobresalientes de Top-K podemos encontrar:

- Mínima interrelación entre la CPU y la GPU, ya sea a través de transferencias de datos o de invocación de kernels;
- Máxima utilización de los recursos de la GPU, logrado mediante la definición de múltiples bloques de thread, quienes cooperan en la resolución de una consulta (paralelismo intra-consulta) y de múltiples consultas (paralelismo inter-consulta);
- Buen aprovechamiento de la jerarquía de memorias de la GPU, se priorizó el uso de la memoria shared y de registro por sobre la memoria global;
- Correcta administración de los accesos a la memoria global, éstos se planificaron en forma coalescente a fin de mejorar el rendimiento;
- Trabajo independiente de cada uno de los threads involucrados en la resolución del problema.
- Aplicación de funciones atómicas para resolver los inconvenientes derivados de compartir recursos como la memoria y coordinar el acceso de los threads;
- Utilización de sincronizaciones por barrera locales a un bloque, las cuales tienen un costo cercano a 0 en este tipo de arquitectura.

Luego de haber realizado un análisis exhaustivo a las soluciones existentes en la bibliografía para el problema de los *k*-NN en la GPU (En el apéndice C se realiza un cuidadoso análisis de las ventajas y desventajas de cada una de las propuestas), se ha

podido observar que en todos los algoritmos analizados [BGTP10, Bar11, GDB08, GDNB10, KH10a, KZ09, LWLJ09, LLWJ09, LLWJ10, QMN09, BGT+11], las soluciones tienen una alta complejidad respecto a las estructuras de datos utilizadas y granularidad gruesa respecto al trabajo de los threads, lo cual lleva a obtener bajo rendimiento. Además, la mayoría de ellas no aprovechan las características propias de la GPU, en algunos casos la gran demanda de recursos de la propuesta limita drásticamente el tamaño del problema a resolver. Por ejemplo en [BGTPl0, Barll, BGT+ll] usan un único bloque de threads para realizar el proceso de búsqueda de k-NN, implicando una sobrecarga en el trabajo de cada thread del bloque y un mal aprovechamiento de los recursos de la GPU, la mayoría permanecen ociosos. En [GDB08, GDNBl0, KHl0a, KZ09] proponen resolver varias consultas a la vez, pero lo hacen usando la misma cantidad de threads usada para resolver una consulta, esto implica una sobrecarga de trabajo para cada thread, el incremento de los recursos elementales como la memoria y un desaprovechamiento de las características de la GPU. En [LWLJ09, LLWJ09, LLWJl0] también subutilizan los recursos de la GPU, un único thread finaliza con el proceso de búsqueda de *k*-NN, lo cual implica que el resto de los threads permanecen ociosos, provocando un desbalance de la carga de trabajo.

Observando las falencias mostradas, nuestra propuesta, en cualquiera de sus tres versiones, es simple y aprovecha al máximo las características de la GPU: varios bloques de threads, accesos coalescentes a memoria global, uso de memoria compartida, independencia del trabajo en paralelo y mínima interrelación entre la CPU y la GPU; logrando obtener excelentes resultados a través del paralelismo intra-consulta e inter-consulta.

6.7. Resumen

El costo computacional de resolver consultas en espacios métricos se mide en la cantidad de evaluaciones de funciones de distancia necesarias para responder a una consulta. Existen diferentes alternativas de optimización, una de ellas es el desarrollo de las soluciones computacionales aplicando técnicas de computación de alto desempeño para tecnologías emergentes como son las GPUs.

En este capítulo presentamos una solución en GPU para resolver consultas por similitud en espacios métricos. Las

consultas a resolver son búsqueda por rango, búsqueda de los *k* vecinos más cercanos y de todos los *k* vecinos más cercanos de todos los objetos en una base de datos. Básicamente nos enfocamos en resolver los k-NN, ya que la solución para las búsquedas por rango es trivial y los *all-k*-NN es una generalización de los *k*-NN. Hemos propuesto un algoritmo simple, llamado *Top-K*, el cual tiene tres versiones, donde cada una aplica un método diferente para determinar los *k*-NN de un objeto de consulta. Los métodos aplicados son: comparación todos con todos (*Top-K_AA*), Quickselect (*Top-K QSeP*) y una combinación de los dos anteriores (*Top-K QSeH*). En forma detallada se explican las características consideradas de diseño e implementación. Finalmente se muestran los resultados empíricos, evaluando el comportamiento de cada solución en distintas GPUs, considerando diferentes parámetros como aceleración, througput y escalabilidad. Este capítulo también incluye un pormenorizado análisis de la bibliografía existente y una comparación con nuestros aportes, un mayor detalle de este análisis es realizado en el apéndice B.

Conclusiones y Trabajos Futuros

La aplicación de técnicas de alto desempeño para la resolución de problemas de propósito general en la GPU ha evolucionado desde sus orígenes, iniciando con la programación mediante primitivas gráficas hasta la programación actual usando un lenguaje secuencial como C o Fortran extendido con unas pocas palabras claves para expresar el paralelismo.

El enfoque adoptado en esta tesis doctoral se basa fundamentalmente en el uso de técnicas de computación de alto desempeño considerando la GPU como una computadora paralela de propósito general, GPGPU. Los problemas abordados son los involucrados en un sistema de huella digital: la extracción de características de los objetos multimedia (en particular las señales de audio) y la identificación de los mismos en grandes repositorios de información, bases de datos métricas.

Antes de iniciar el desarrollo de este trabajo, hemos realizado una amplia exploración del estado del arte en los tres aspectos fundamentales involucrados en esta tesis. Primero consideramos el uso de la GPU como arquitectura paralela para resolver problemas generales. Luego nos dedicamos al estudio de: la huella digital de audio, conocer cómo se obtiene y calcula una huella robusta; y el modelo de espacios métricos, los distintos métodos usados para dar solución al problema de las búsquedas y las técnicas adoptadas para resolver consultas por similitud. Finalmente, se relacionaron estos temas mediante la evaluación de la aplicación de técnicas paralelas en GPU en las soluciones de ambos problemas.

Respecto a GPGPU, se realizó un análisis detallado, enfocándonos principalmente en cómo es la arquitectura de la GPU y el modelo de programación propuesto por CUDA a fin de hacer uso de todos los beneficios y lograr resultados óptimos. Este proceso fue continuo, a medida que se incorporaban nuevos conceptos y mejoras en la GPU, se los iba incluyendo en nuestros desarrollos.

Las huellas digitales (AFPs) de audio son ampliamente utilizadas en diversas aplicaciones con impacto económico, tales como detección de duplicado de canciones, monitoreo broadcast, etiquetado de audio entre otros. El proceso para obtener una huella robusta conlleva diversas etapas, desde la digitalización de

la señal de audio hasta la obtención final de la AFP. Todas estas etapas tienen un alto costo computacional, aspecto muy relevante dada la necesidad actual de obtener las AFPs en tiempo real. Respecto al análisis bibliográfico, el uso de la GPU como medio para acelerar el proceso de la determinación de la AFP de una señal no ha sido muy abordado aún.

En nuestro trabajo hemos desarrollado las huellas digitales MBSES y TES propuestas en [CI07], basadas en la entropía de la señal. Ambas fueron desarrolladas usando la GPU como medio para acelerar el proceso de extracción de características. Los excelentes resultados de ambos han mostrado que el empleo de estas técnicas es una buena decisión. En el caso de MBSES, la AFP es obtenida a partir del dominio de la frecuencia, implicando un sin número de etapas cada una con un elevado costo computacional (Cálculo de FFT, Obtención de Histogramas, División de la Señal en Bandas, entre otras). Debido a esto, el tamaño de la señal influye en el tiempo de ejecución del proceso, en consecuencia y mediante el empleo de la GPU hemos logrado acelerar el proceso en un factor de 200x respecto al proceso secuencial cuando la señal es grande (Superior a los 110MB). En el caso de TES, como la AFP se obtiene en el dominio del tiempo, el proceso es más simple. Los resultados alcanzados son casi constantes, independiente del tamaño de la señal. Además las ganancias se ven afectadas por las transferencias entre la CPU y GPU. Aún así, y a pesar de esta dificultad, el proceso en la GPU es más rápido que el respectivo secuencial, obteniendo aceleraciones entre 30x y 40x. A partir de los resultados observados y de su análisis es posible afirmar que el uso de la GPU en este tipo de aplicaciones es relevante principalmente si se quiere aplicar en ambientes de tiempo real.

Respecto a la recuperación de datos, el modelo de espacios métricos es adecuado para aplicar en la resolución de consultas por señales de audio cuando éstas son representadas por sus características relevantes. Las bases de datos métricas permiten almacenar las representaciones de objetos multimedia para luego poder recuperarlos mediante búsquedas por similitud. Dichas búsquedas tienen un alto costo computacional, encontrar los objetos similares a un objeto dado en grandes repositorios de datos implica realizar muchas evaluaciones de distancia. Si bien se han propuesto diversos algoritmos secuenciales para reducir los costos tanto de tiempo como de espacio, esto no siempre es suficiente. Por ello pensar en la aplicación de técnicas de computación de alto desempeño, especialmente el uso de GPU, es

una alternativa viable. Dicho problema ha sido ampliamente estudiado en la bibliografía existente, específicamente la búsqueda de los vecinos más cercanos *k*-NN, encontrando algunos muy buenos resultados.

En este trabajo hemos propuesto un algoritmo, llamado *Top-K* y tres versiones distintas del mismo. *Top-K* es una versión paralela del método de fuerza bruta, el cual es altamente paralelizable y adecuado para resolverse aplicando el modelo de programación de CUDA, permitiendo tomar ventaja de los beneficios de la GPU. Del análisis de los resultados experimentales realizados podemos establecer que sin importar la versión de *Top-K* utilizada, los resultados obtenidos son muy buenos. Se puede ver claramente que sin importar la naturaleza de la base de datos y las características de cada objeto de la misma, se logró acelerar el tiempo de resolución de las búsquedas, aún cuando la GPU es limitada tanto en espacio de memoria como capacidad de computación y generación de la arquitectura. Además observamos la influencia del tamaño de la base de datos en la aceleración, esto nos permitió determinar que no sólo se obtienen buenos resultados en los casos de bases de datos con tamaños adecuados para alojarse en la memoria principal de la CPU o de la GPU, sino también para aquellas cuya cardinalidad es mucho mayor a las respectivas capacidades de memoria. En el caso de la CPU, se resuelve el problema utilizando la memoria secundaria, hecho que no es posible en la GPU donde el problema lo resolvimos haciendo múltiples transferencias entre la CPU y la GPU. Aunque las transferencias afectan el desempeño de la aplicación, mediante los resultados experimentales hemos mostrado que siguen siendo eficientes las soluciones en la GPU, hasta cuando se trata de bases de datos de gran tamaño.

En ambas líneas de desarrollos hemos obtenido aceleraciones muy buenas, algunas de ellas sobre el límite lineal. Esto se debe al aprovechamiento de todas las capacidades propias de la GPU, las cuales permitieron estos logros, entre ellas destacamos el uso de las memorias de acceso más rápido como la memoria shared, de constante y de registro, como así también el acceso controlado a la memoria global.

Por todo lo expuesto se puede concluir que el objetivo general de la tesis fue cumplido, los resultados alcanzados nos permiten validar el uso de la GPU para dar solución a los problemas derivados de un sistema de huella digital robusta y eficiente. Con los desarrollos realizados logramos reducir significativamente el

tiempo de las soluciones, permitiendo transferir estos logros a la solución de otros problemas.

Trabajos Futuros

Como posibles trabajos futuros de esta tesis se pueden señalar los siguientes:

- Las señales de audio consideradas en el presente trabajo están en formato *wav*. Si bien este formato posee alta calidad de audio, es costoso debido a que requiere mucho espacio de almacenamiento, y consecuentemente también lo es su transferencia. Una posible mejora sería transferir las señales de audio a la GPU en formato comprimido, por ejemplo en MP3. Dicho formato tiene un extraordinario grado de compresión y alta calidad, logrando reducir el tamaño de un archivo con un factor de 1/10 a costa de una mínima pérdida de calidad. La idea sería utilizar una señal MP3, transferirla a la GPU, luego allí descomprimirla y obtener la señal en formato WAV para aplicarle el proceso desarrollado.
- En el caso de la huella *TES*, se pudo observar que los tiempos de transferencias tienen un costo significativo, dominando ampliamente el proceso total. Una solución a este problema sería procesar tantas señales de audio como sea posible, teniendo en cuenta sólo los límites propios de la arquitectura. La idea consiste en aplicar paralelismo intra-señal (emplear la misma filosofía de la resolución de múltiples consultas en paralelo), y transferir al inicio del proceso tantas señales como la memoria lo permita, retornando al final del cómputo a la CPU todas las AFPs calculadas.
- En la jerarquía de memorias de la GPU, existen otras con características especiales como es la memoria de textura, la cual tiene asociada una cache permitiendo un acceso más rápido (los threads sólo pueden leer en ella). Usar esta memoria para almacenar la señal y/o la base de datos o parte de ella nos permitirá analizar una posible reducción en los costos de transferencia y de accesos a los datos.
- Con la nueva generación de GPU, las keplers introducen varios aspectos no disponibles en las GPUs empleadas en esta tesis, entre ellos se encuentran el paralelismo

dinámico, hyper-Q y recursión. La aplicación de todas o alguna de ellas nos permitirán mejorar los desarrollos actuales.

- Debido a los beneficios aportados por la GPU como co-procesador paralelo, por sus características y buen desempeño, una de las tendencias actuales es trabajar con múltiples GPUs. Considerar esta arquitectura implicará realizar un análisis exhaustivo de sus características y adaptar las aplicaciones para trabajar no sólo con una única GPU, sino con múltiples, ya sea conectadas a través de una red o en un mismo procesador.
- Del punto anterior se deriva la posible aplicación de otro paradigma, la computación paralela híbrida o heterogénea, combinando OpenMP y/o MPI con GPU. Para llevar a cabo esta acción debemos primero analizar la factibilidad de su aplicación, considerando la posible influencia de las comunicaciones entre procesos o threads en diferentes procesadores. En esta dirección también podemos considerar la aplicación de otras herramientas como CUDA-Aware MPI, GPUDirect, entre otras.

Generaciones de Arquitectura de GPU Nvidia

La principal ventaja de las arquitecturas unificadas de GPU es la posibilidad que brindan del autoequilibrio en la carga computacional. El conjunto de procesadores pueden ahora asignarse a una tarea u otra dependiendo de la carga exigida por la aplicación. De esta manera, a cambio de mayor complejidad en los procesadores de la GPU y de mayor generalidad en su capacidad de procesamiento, se consigue reducir el problema del equilibrado de carga y de la asignación de unidades de procesamiento a cada etapa del pipeline gráfico. En las siguientes secciones se presentan las distintas generaciones de arquitecturas unificadas de GPU, desde la G80 hasta las actuales GKll0, todas ellas aptas para la programación de aplicaciones de propósito general.

A.1. Arquitectura G80

Las arquitecturas anteriores a la G80, la serie GeForce6 y GeForce7, se pueden definir como arquitecturas divididas a nivel de shaders o procesadores programables; existe en ellas hardware especializado para ejecutar programas que operan sobre vértices o están dedicados exclusivamente a su ejecución sobre píxeles. La diferencia entre ambas series de arquitecturas es la potencia, mayor en la serie GeForce7. El principal inconveniente de ambas fue su arquitectura no unificada, por un lado estaban los procesadores de vértices y por el otro los dedicados al cálculo a nivel de píxeles, si la aplicación no tenía un trabajo equilibrado para todos estos tipos de procesadores, algunos eran subutilizados.

La serie G80, en cambio, es una arquitectura unificada a nivel de shaders. Ya no existe una división a nivel de hardware entre procesadores de vértices y procesadores de fragmentos. Las

unidades de procesamiento, llamadas Stream Processors, son capaces de trabajar tanto a nivel de vértice como a nivel de fragmento, sin especializarse en ninguno de los dos. Esto implica un cambio en el pipeline gráfico: la arquitectura unificada. En ella no existen componentes específicas asociadas a una etapa concreta del pipeline, sólo una única unidad central de alto rendmiento será la responsable de realizar todas las operaciones, cualquiera sea su naturaleza. Entre las ventajas de esta arquitectura se encuentra la posibilidad de equilibrar la carga entre los distintos procesadores, los cuales se pueden asignar a una tarea u otra dependiendo del trabajo a realizar. Como desventaja se puede considerar la mayor complejidad de los procesadores y su no especificidad en un tipo de problemas.

La Nvidia GeForce 8800 fue presentada en el 2006, dando origen al nuevo modelo de GPU. La G80, base de GeForce 8800, introdujo muchas innovaciones, las cuales fueron claves para la programación de GPU. Entre las nuevas características de la G80 se encuentran:

- Reemplazó los pipelines separados de píxeles y vértices por un único y unificado procesador, en el cual se puede ejecutar programas de gráficos (geometría, vértice y pixel) y programas de computación en general.
- Reemplazó la administración manual de registros de vector por parte de los programadores por un procesador escalar de threads.
- Presentó el modelo de ejecución Simple Instrucción-Múltiples Threads (SIMT) para que múltiples threads independientes ejecuten concurrentemente una simple instrucción.
- Introdujo la memoria compartida y la sincronización por barreras para la comunicación entre threads.
- Brindó la posibilidad de contar con soporte para lenguaje C, permitiendo a los programadores utilizar el poder de la GPU, sin tener que aprender un nuevo lenguaje de programación.

La arquitectura G80 es una arquitectura totalmente unificada, no existe diferencia a nivel de hardware entre las distintas etapas del pipeline gráfico, está totalmente orientado a la ejecución de threads considerando el balance de carga. Opera de forma integral con una precisión de 32 bits para datos de punto flotante, ajustándose al estándar IEEE 754. La figura A.1 muestra el esquema completo de la arquitectura G80.

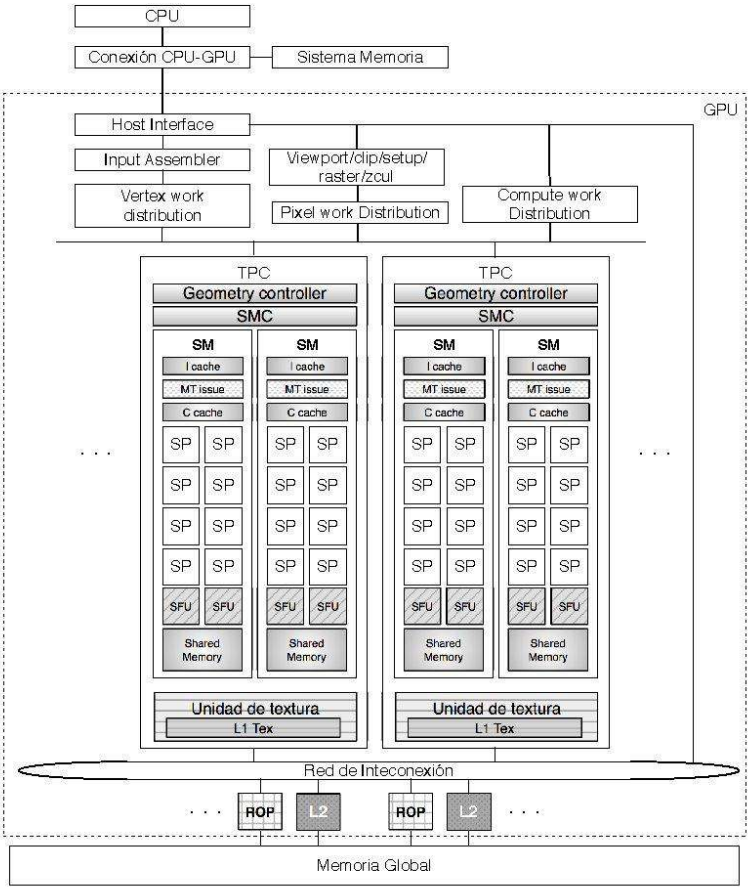


Figura A.1: Arquitectura de G80

En el diagrama se muestra la arquitectura de una GPU Geforce 8800, la cual cuenta con 128 procesadores de streaming o escalares (SP), organizados en 16 multiprocesadores de streaming (SM) en ocho unidades de procesamiento independiente denominadas cluster procesador/texture (Texture/Processor Cluster, TPC). El trabajo comienza desde la CPU, quien le indica el trabajo a la GPU a través del bus PCI-Express.

Como se puede observar, la GPU tiene varios componentes dispuestos en una jerarquía. En el más alto nivel se observa el arreglo de procesadores de streaming (SPA), formados por varios TPC. El sistema de memoria está compuesto por un control de DRAM externa o memoria global y procesadores de operaciones

raster de función fija (ROP), los cuales se encargan de realizar operaciones específicas para color y profundidad directamente sobre la memoria. A través de una red de interconexión se accede desde el SPA a los ROP, se resuelven las lecturas de la memoria de textura desde el SPA a la memoria global, y las lecturas sobre la memoria global a la caché de nivel 2 y al SPA.

El trabajo dentro de la GPU se desarrolla de la siguiente manera, en primer lugar los datos recibidos desde el host son pre-procesados en hardware específico a fin de organizarlos para aprovechar la máxima capacidad de cálculo del sistema y evitar la existencia de unidades ociosas. Una vez hecha esta organización, se transfiere el control de la ejecución a un controlador global de threads, quien decide que thread se ejecutará en cada instante y dónde. Además existe un planificador de threads para determinar su ejecución dentro de las unidades de procesamiento.

El Input Assembler recolecta el trabajo de vertex que ingresa y lo pasa al distribuidor de trabajo de vertex (Vertex work distribution) quien distribuye los paquetes de trabajo a los TCP en el SPA. Los TCP ejecutan programas shader de vertex y programas shader de geometría. Los datos de salida son escritos en buffer on-chip, estos buffers luego pasan sus resultados a la unidad responsable de la rasterización (viewport/ clip/ setup/ raster/ zcull block). La unidad de distribución de trabajo de pixel distribuye fragmentos de pixeles a los TPC apropiados para el procesamiento. Los fragmentos de pixeles sombreados son enviados a través de la red de interconexión para el procesamiento del color y la profundidad a las ROP. La unidad de distribución de trabajo de cómputo (Compute work distribution) envía los threads de cómputo general a los distintos TPC del SPA para su ejecución. Todas las unidades que integran la arquitectura de la G80, incluido los múltiples relojes que las rigen, proveen independencia y optimización de la performance.

Los distintos TPC están organizados como un arreglo de Multiprocesadores Streaming (SM). Por cada TPC hay dos SM. Todos los TPC acceden a la memoria global. Existe un planificador de threads asociado a cada cluster, así como memoria cache Ll y unidades de acceso y filtrado de texturas propias (Unidades de Textura). Cada grupo de l6 SP dentro de un mismo cluster comparten tanto unidades de acceso a texturas como la memoria caché Ll. La figura A.2 muestra un esquema de la estructura de cada uno de los clusters que forman la arquitectura G80 de Nvidia.

Cada SM está formado por ocho Streaming o Scalar Processors (SP), los cuales comparten la lógica de control y la caché de instrucciones. Además de los SP, un SM posee 2 SFUs (Special Function Units), una pequeña caché de instrucciones, una unidad MT (MT issues), responsables de enviar instrucciones a todos los SP y SFUs en el grupo, una caché de sólo lectura de datos y una memoria shared o compartida, generalmente de l6KB.

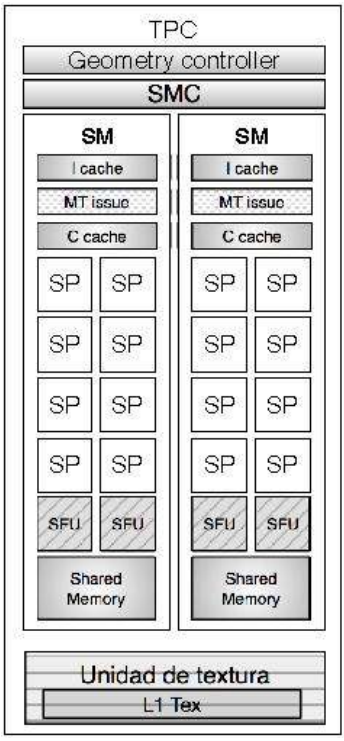


Figura A.2: Arquitectura de un TPC

Las SFU llevan a cabo las funciones de punto flotante tal como la raíz cuadrada o funciones transcendentales como seno, coseno, entre otras. Cada SFU ejecuta una instrucción por thread por ciclo de reloj. Si las SFU están ocupadas, la unidad de planificación ejecuta otras sentencias para evitar el tiempo ocioso. La figura A.3 muestra la estructura de un SM.

Respecto a los SP, ellos son responsables de las operaciones matemáticas o de direccionamiento de datos en memoria y posterior transferencia de los mismos, trabajan sobre datos escalares. Cada uno cuenta con una unidad MAD (Multiply-Add)

y una unidad de multiplicación adicional. En la figura A.4 se muestra la estructura típica de un SP.

Por las características de la arquitectura G80: 8 grupos de 16 SP, es posible realizar una abstracción de la arquitectura y representarla como una configuración MIMD de 8 nodos de computación, cada uno de los cuales formado por 16 unidades o clusters de procesamiento SIMD (SP). Los TPC pueden procesar threads en forma independiente y en un mismo ciclo de reloj.

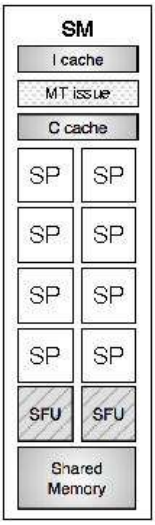


Figura A.3: Estructura de un SM

La comunicación entre la CPU y la GPU es un aspecto importante en el desarrollo de toda aplicación debido al costo que implica. Actualmente la comunicación entre ambos sistemas se realiza a través de un bus PCI-Express, constituyendo en muchos casos un cuello de botella de las prestaciones de la aplicación.

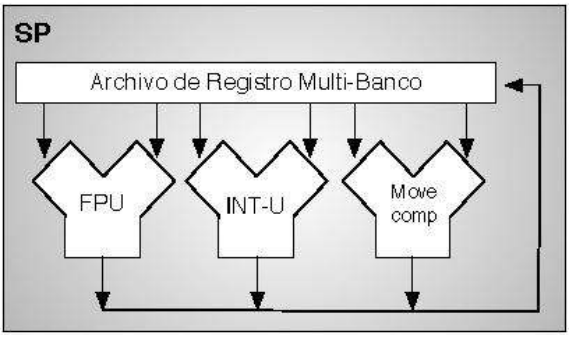


Figura A.4: Arquitectura de SP

A.2. Arquitectura GT200

En junio de 2008, Nvidia presenta una revisión de la arquitectura G80, la GT200 tanto en las GeForce, la Quadro y la Tesla [LNOM08]. Se aumentó el número de SP; la capacidad de registros del procesador se duplicó, permitiendo un mayor número de threads en ejecución en el chip en un momento determinado; se adicionó hardware para mejorar el acceso eficiente a la memoria; y se incluyó soporte para operaciones de punto flotante de doble precisión a fin de satisfacer las demandas de los científicos y de las aplicaciones de computación de alto desempeño.

El chip G80 soporta hasta 768 threads por SM, la serie GT200 soportan 1024 threads por SM, esto significa por ejemplo en la GeForce GTX 260 pueden 24.576 threads ser atendidos en paralelo (24 SM) y en la GeForce GTX 295 tiene 60 SM, lo cual implica 61.440 threads simultáneos (30.720 por placa). Esto conlleva a que el nivel de paralelismo desde el hardware de la GPU se incrementó rápidamente, lo importante entonces es esforzarse por lograr dichos niveles de paralelismo en el desarrollo de aplicaciones. En la figura A.5 puede verse la estructura del TPC para esta serie de arquitecturas.

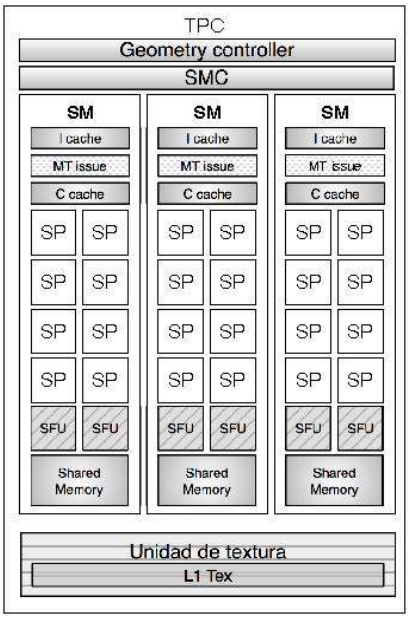


Figura A.5: Estructura de TPC para la GT200

Como pudo apreciarse, el número de SM por TCP en esta generación se incrementó en un 50 %. El incremento del número de SP no es la única mejora respecto a las G80, en la tabla A.1 se resumen las Características fundamentales de las arquitecturas G80 y GT200.

Cuadro A.1: G80 vs. GT200

Característica	G80	GT200
SP por SM	16	24
Unidades de direcciones de textura por TPC	4	8
Unidades de filtrado de textura por TPC	8	8
Total de SP	128	240
Total de unidades de direcciones de textura	32	80
Total de unidades de filtrado de textura	64	80

Respecto a la cantidad de memoria shared por SM se mantuvo en 16KB, no así la cantidad de memoria de registros, la cual se duplicó, y la cantidad de warps residentes en cada SM, fue incrementado en un 33 %.

Resumiendo, la diferencia entre una GPU serie G80 y una GT200 se basa en la ampliación de los recursos, área del chip, no en cambios de estructura como ocurre con la siguiente arquitectura.

A.3. Arquitectura Fermi (GF100)

En el diseño de cada nueva generación de GPU, la filosofía aplicada por Nvidia es mejorar tanto el rendimiento de las aplicaciones existentes como la programación de las GPUs. Aunque obtener mejor performance para aplicaciones existentes tiene sus ventajas, lo que ha transformado a las GPUs en un procesador paralelo de gran aceptación son las facilidades de su programación. La GF100 es la sucesora de la arquitectura GT200 (NVIDIA, 2010).

La primera GPU basada en la arquitectura Fermi cuenta con hasta 512 cores (SP). Cada core resuelve una instrucción de entero o punto flotante por ciclo de reloj. Los 512 SP están organizados en 16 SM de 32 SP cada uno. La GPU cuenta con seis particiones de memoria de 64 bits, para una interfaz de memoria de 384 bits, la cual soporta hasta un total de memoria DRAM de 6 GB de memoria GDDR5. Los SM comparten una caché de nivel 2 (L2). Además cada SM tiene un planificador, un despachador (dispatch), registros y memoria caché L1.

La GF100 se basa en un arreglo de Cluster de Procesamiento Gráfico (Graphics Processing Clusters, GPC), multiprocesadores de streaming (SM) y los controladores de memoria. Una GF100 tiene 4 GPC, l6 SM y 6 controladores de memoria. Estas cantidades dependen del modelo de la arquitectura. Los controladores de memoria atienden a las unidades ROP (Raster Operation Processor) para operaciones específicas como blending de píxeles y operaciones atómicas sobre la memoria global. Los 48 controladores son divididos en 6 grupos de ocho. La memoria caché L2, los controladores de memoria y los grupos de ROP están íntimamente relacionados, la ampliación o modificación de uno influye a los demás. En la figura A.6 se muestra un diagrama de la arquitectura GF100.

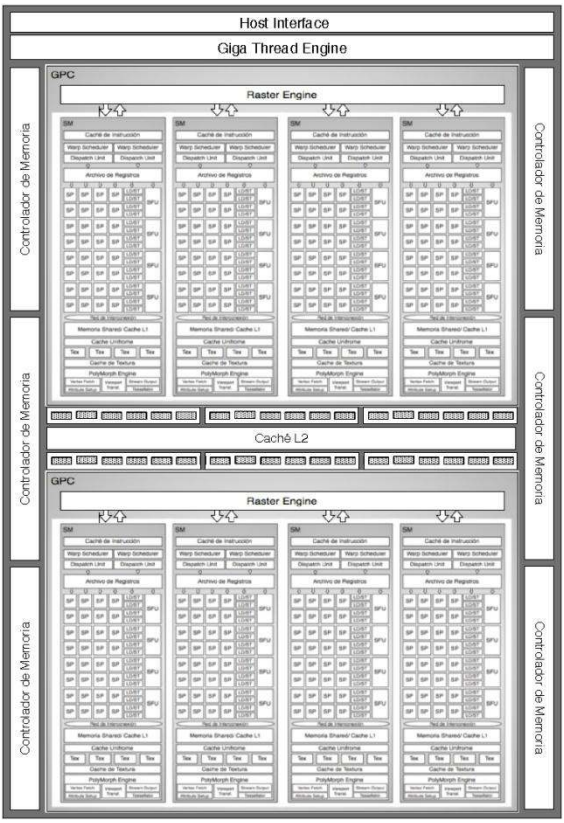


Figura A.6: Esquema de la arquitectura de un GF100

Un GPC contiene una Raster Engine (RE) y hasta cuatro SM (Ver Figura A.7). Es el más alto nivel de bloques de hardware. Sus dos principales innovaciones son la incorporación de la unidad de:

- Raster Engine, escalable para la inicialización de triángulos (triangle setup), rasterización y Z-cull (recorte de la tercer coordenada, z, de los pixeles o triángulos, pasa de 3D a 2D);
- PolyMorph Engine (PME) para la búsqueda de los atributos de vertex (vertex attribute fetch) y la tessellation (división poligonal).

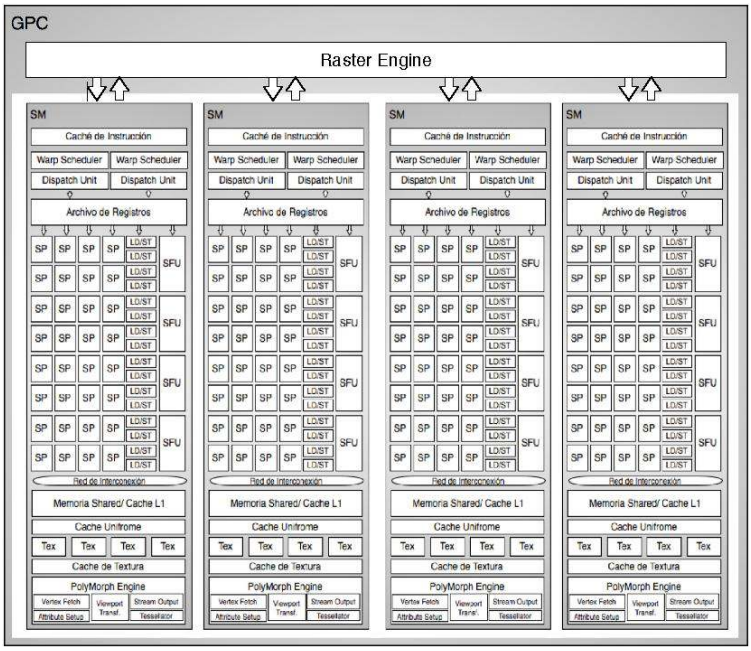


Figura A.7: Esquema de la arquitectura de un GPC.

El RE se encuentra en el GPC y el PME en el SM.

El GPC encapsula todas las unidades claves de procesamiento de gráficos, a excepción de las funciones ROP, un GPC puede ser considerado como una GPU autónoma, una GF100 cuenta con cuatro GPC.

En las anteriores GPU, los SM y las unidades de textura se agrupan en bloques de hardware, los TPC. En la GF100, cada SM tiene cuatro unidades de textura, eliminando la necesidad de tener TCP.

Los SM presentan varias innovaciones, las cuales lo hacen no sólo más potente sino también más eficiente y programable. En la figura A.8 se muestra un diagrama de la arquitectura de un SM.

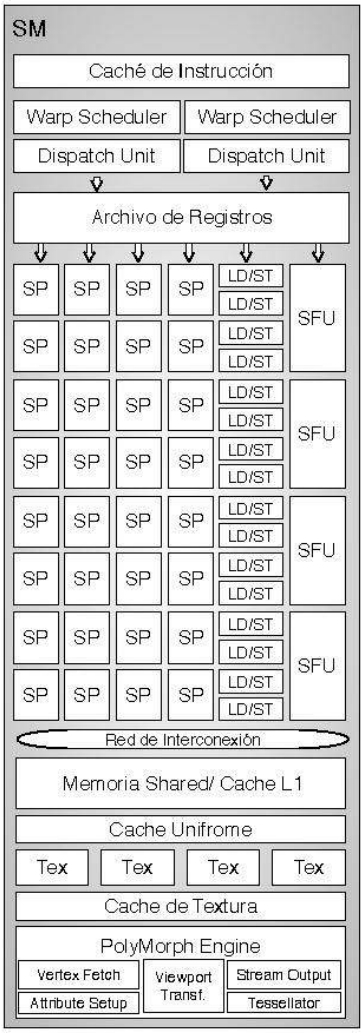


Figura A.8: Arquitectura de un SM de GF100

Cada SM tiene 32 SP, es decir cuatro veces más que sus antecesores. Además cuenta con l6 unidades load/store, las cuales permiten calcular direcciones de origen o destino a l6 threads por reloj. Estas unidades leen o escriben los datos de cada dirección en la caché o la DRAM. Las SFU tienen las mismas características que en las de las anteriores generaciones de GPU.

Por su parte un SP tiene una unidad aritmética/lógica de enteros (ALU) y una unidad de punto flotante (FPU) con una

aritmética de punto flotante según el estándar IEEE 754-2008, proporcionando instrucciones fusionadas de multiplicar-sumar (FMA), tanto para aritmética de precisión simple como doble. Este tipo de instrucciones mejora las instrucciones de multiplicar-sumar (MAD) haciendo la multiplicación y la suma con un simple paso de redondeo final, sin ninguna pérdida de precisión en la suma, la FMA es más precisa que la realización de las operaciones por separado. En esta arquitectura se planteó un nuevo diseño de ALU para números enteros, soporta una precisión de 32 bits para todas las instrucciones. La ALU es también optimizada para soportar operaciones de 64-bit. Varias instrucciones específicas son soportadas, incluyendo operaciones: booleanas, shift, move, compare, convert, extracción del bit-field, inserción de bit-reverse y el recuento de la población.

Otra de las características importantes de las GF100 es la aritmética de doble precisión. Esta arquitectura fue específicamente diseñada para ofrecer una performance sin precedentes en doble precisión, hasta l6 operaciones multiplicar-sumar pueden ser realizadas por SM en un ciclo de reloj. Esta mejora es muy importante respecto a la arquitectura GT200. En la figura A.9 se muestra la estructura de cada SP.

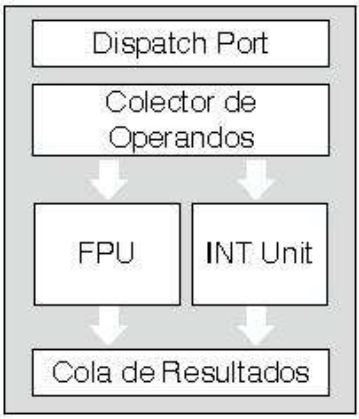


Figura A.9: Arquitectura SP de GF100.

Respecto a la administración de los threads, estos son administrados a los SM en grupos de 32, un warp, como lo hacían sus antecesoras. Cada SM tiene dos administradores de warp y dos unidades de dispatch de instrucciones, permitiendo a dos warps ejecutar concurrentemente. El Planificador dual de warp

selecciona a dos warp y da una instrucción de cada uno a un grupo de 16 SP, 16 unidades load/store o a 4 SFU.

La memoria compartida es una facilidad que permite optimizar la performance de las aplicaciones. En las arquitecturas anteriores, por cada SM se tiene 16KB de memoria compartida, en la GF100 se la extendió a 64KB, la cual se puede configurar hasta 48KB de memoria compartida y 16KB de caché L1 o a la inversa, 16KB de memoria compartida y 48KB de caché L1. Esto triplica la capacidad de la memoria compartida para las aplicaciones que la utilizan, pero para aquellas que no la utilizan se benefician automáticamente de la caché L1.

En la figura A.6 se muestran dos componentes no descriptas aún, la interfaz con el host y la unidad GigaThread. Los comandos de la CPU son leídos por la GPU a través de la interfaz de host y pasados a la unidad GigaThread quien obtiene los datos especificados desde la memoria global y los copia al framebuffer. La GigaThread crea y envía bloques de threads a varios SM. Cada SM, en turno, administra los warps a los SP y otras unidades de ejecución. También es su responsabilidad la redistribución del trabajo de los SM cuando se produce una sobrecarga en el pipeline gráfico, por ejemplo después de las etapas de tessellation y rasterization.

Al igual que las anteriores versiones esta GPU se conecta al host a través de PCI-Express.

A.4. Arquitectura Kepler (GK110)

Con la arquitectura Fermi se pudo acelerar aplicaciones en áreas como sismografía, bioquímica, climatografía, procesamiento de señales, ingeniería, dinámica de fluidos y análisis de datos. La arquitectura Kepler (GK110) incrementa considerablemente el paralelismo en la placa, permitiendo resolver y acelerar las soluciones de problemas más complejos en áreas como medicina, ingeniería y finanzas, además de mejorar los ya existentes.

La arquitectura Kepler ofrece una mayor potencia de procesamiento que sus antecesoras, brindando nuevos métodos para optimizar y aumentar la carga de trabajo de ejecución en paralelo en la GPU. Está compuesto por 7.1 billones de transistores, incluye diversas características nuevas enfocadas a incrementar el desempeño paralelo (más de 1 TFlop de rendimiento de doble precisión para la multiplicación de matrices

obteniendo el 80% de eficiencia frente al 60-65% logrado por la arquitectura Fermi), y ofrece una mejora respecto al consumo energético (incrementa en 3x el rendimiento por watt respecto a la Fermi). Las nuevas características incluidas en esta arquitectura son:

- *Paralelismo Dinámico:* Esta propiedad permite generar un nuevo trabajo para sí mismo, sincronizar los resultados y controlar la administración del trabajo mediante una vía dedicada, sin la participación de la CPU. Esta nueva característica posibilita adaptar la cantidad y forma del paralelismo durante la ejecución de un programa, permitiendo desarrollos con variados tipos de trabajo en paralelo y un uso eficiente de la GPU. Esta capacidad nos habilita a crear soluciones en la GPU de otro tipo de problemas, no necesariamente sólo con paralelismo de datos.
- *Hyper-Q*: A través de esta propiedad, una CPU con múltiples núcleos de CPU, cada uno puede enviar trabajo en forma simultánea a una sola GPU. Esto permite incrementar drásticamente la utilización de la GPU y la CPU reduciendo significativamente los tiempos de inactividad. Para ello se aumenta el número total de colas de trabajo entre el host y la GPU permitiendo 32 conexiones simultáneas (contra una única conexión disponible en Fermi). De esta manera múltiples procesos MPI poseen conexiones separadas para múltiples threads dentro de un proceso.
- *Unidad de Administración de Grid:* El paralelismo dinámico implica contar con un sistema de control de asignación y administración de grid avanzado y flexible. La Unidad de Administración de Grid (Grid Management Unit, GMU) se encarga de administrar y priorizar los grid a ejecutar en la GPU, por ejemplo la GMU puede pausar la planificación de nuevos grid, ponerlos en la cola y suspenderlos hasta que estén listos para ejecutar. La GMU asegura una buena administración tanto para los trabajos generados por la CPU o por la GPU.
- *NVIDIA GPUDirect:* Esta capacidad permite a la GPU en un equipo o a varias GPUs en diferentes servidores en una red intercambiar datos directamente sin necesidad de pasar por la CPU o el sistema. La característica RDMA en GPUDirect permite a dispositivos como los SSD, tarjetas de red y adaptadores de IB, acceder directamente

a la memoria en múltiples GPUs dentro del mismo sistema. De esta manera se reduce la latencia de MPI en el envío y recepción de mensajes desde/para la memoria de la GPU. Esta arquitectura es compatible también con otros GPUDirect típicos incluyendo Peer-to-Peer y GPUDirect para Video.

La GK110 tiene una arquitectura, la cual supera no solamente la potencia de cálculo de Fermi, sino que lo hace en forma eficiente, consumiendo menos energía y produciendo una menor cantidad de calor. Una Kepler GK110 está formada por quince SMX y seis controladores de memoria de 64-bit, en la figura A.10 se esquematiza la arquitectura.

Sus principales características son:

- Una nueva arquitectura de procesador: SMX.
- Un subsistema mejorado de memoria: ofrece memorias cachés adicionales, mayor ancho de banda en cada nivel de la jerarquía de memoria y una DRAM totalmente rediseñada y más rápida.
- Provee el soporte para las nuevas capacidades de cómputo 3.0 y 3.5.

La tabla A.2 compara las capacidades de cómputo entre las arquitecturas Fermi y Kepler:

Cuadro A.2: Fermi vs. Kepler

Arquitectura	Fermi		Kepler	
	GF100	GF104	GK104	GK110
Capacidad de Cómputo	2.0	2.1	3.0	3.5
Max Warps por SMX	48	48	64	64
Max Threads por SMX	1536	1536	2048	2048
Max Bloques de Threads por SMX	8	8	16	16
Nro de Registros por SMX	32KB	32KB	64KB	64KB
Max Registers por Thread	63	63	63	255
Tam. de Conf. de Memoria Shared	16 o 48KB	16 o 48KB	16, 32 o 48KB	16, 32 o 48KB
Max Dim. de Grid en X, Y o Z	$2^{16}-1$	$2^{16}-1$	$2^{32}-1$	$2^{32}-1$
Hyper-Q	No	No	No	Yes
Paralelismo Dinámico	No	No	No	Yes

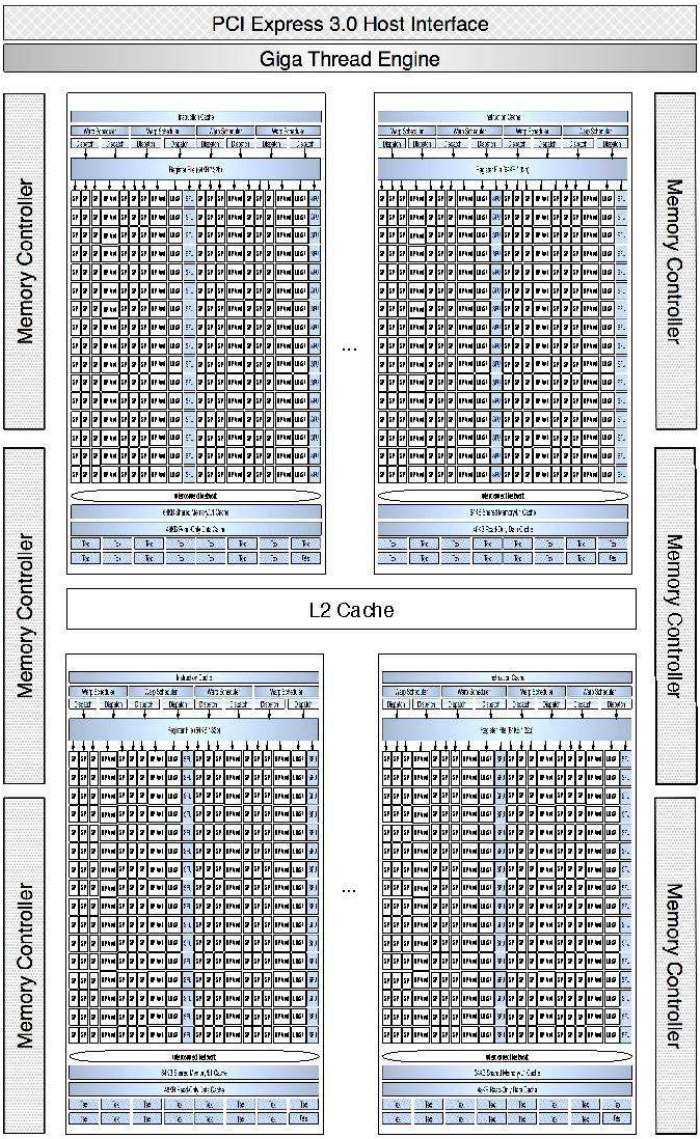


Figura A.10: Diagrama de la Arquitectura Kepler GK110

Se mantiene el número de threads por warp en 32 y el máximo número de threads por bloques en 1024.

En el caso del nuevo Multiprocesador de Streaming, denominado SMX, se introducen varias innovaciones, las cuales lo hacen no sólo el multiprocesador más poderoso construido por NVIDIA, sino también el más fácil de programar y eficiente. Un SMX tiene 192 CUDA-cores de simple presición, 64 unidades de

doble-presición, 32 unidades de función especial (SFU) y 32 unidades load/store (LD/ST), la figura A.11 muestra la arquitectura de un SMX.

Cada CUDA-core tiene sus unidades aritméticas y lógicas de punto flotante y enteros dispuestas en un pipeline, además conserva las características de la aritmética de simple y doble presición introducida por Fermi, incluyendo la operación fusionada de suma-multiplicación (FMA).

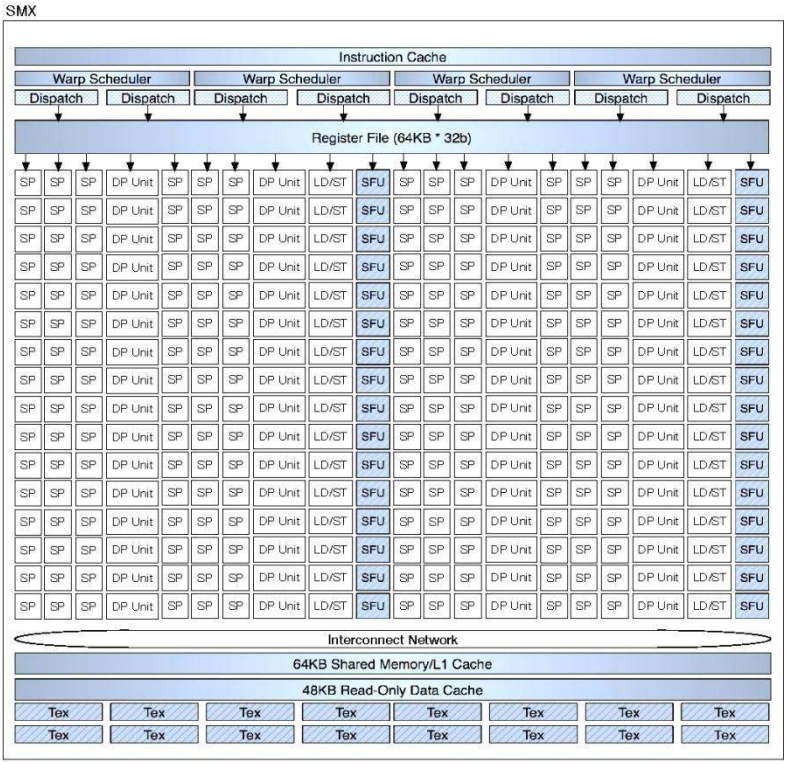


Figura A.11: Arquitectura de SMX de la GK110.

Cada SMX posee 4 schedulers de warps y 8 unidades instruction dispatch, esto permite que 4 warps sean seleccionados por cada uno de los scheduler y 2 instrucciones independientes por warps sean ejecutadas en cada ciclo. A diferencia de Fermi, las instrucciones pueden tener distinta naturaleza, es decir pueden ejecutarse una de doble precisión con otra cualquiera.

Durante el 2014, NVIDIA planifica sacar al mercado la arquitectura *Maxwell* y en 2017 la *Volta*.

APÉNDICE B

Otros Desarrollos en GPU

Además del algoritmo Top-K, durante el desarrollo de esta tesis, se implementaron dos propuestas distintas, una para resolver consultas *all- k*-NN de manera aproximada y la otra para resolver todas las consultas en base a un conjunto de objetos elegidos en forma aleatoria, la solución también es aproximada. La primer propuesta evita cálculos de distancia sin la construcción de un índice, y la segunda es una versión distinta de algoritmo *Top-K*.

Recordemos las nomenclaturas usadas. Dado un espacio métrico (U, d), donde U es el universo de los objetos y d la función de distancia tal que $d: U \times U \mapsto \mathbb{R}^+$. Una base de datos se define como $BD \subseteq U$ con tamaño $n = |BD|$.

A continuación se exponen ambas propuestas.

B.1. Propuesta *all-k*-NN aproximado

El algoritmo propuesto permite reducir el número de evaluaciones de distancia utilizando las distancias que se van calculando para evitar calcular otras, de esta manera se reduce el número de cálculos de distancia.

Esta versión, sin armar índice, calcula los *all-k*-NN de manera aproximada de forma tal de realizar menos cálculos de distancia y sea fácilmente paralelizable. La propuesta secuencial es la siguiente:

1. Seleccionar un pequeño porcentaje p de elementos de BD, por ejemplo $N = \frac{p \times n}{100}$ con $p = 1\%$. Sea X_p el conjunto de N elementos seleccionados en BD ($X_p \subset BD, con\ |X_p| = N \ll |BD| = n$).
2. Calcular los K-NN en BD (siendo $K \gg k$, es decir K es elegido como un valor mayor que *k*), para cada uno de los

N elementos de los X_p. Esto dejaría disponible la siguiente información:

$$\{(x, K-NN(x)) | x \in X_p \wedge K-NN(x) \subset BD\}$$

En este punto ya se pueden informar los k-vecinos más cercanos a cada uno de los elementos de X_p y se realizan $N \times (n-1)$ cálculos de distancia para elegir los K elementos más cercanos a cada uno de los N elementos, porque cada elemento realiza n-1 cálculos de distancia para encontrar sus K elementos más cercanos.

3. Para cada uno de los elementos $x \in BD - X_p$ elegir el elemento más cercano a x en Xp, es decir $o = min_{\forall x_i \in X_p} \{d(x, x_i)\}$, y seleccionar los k-NN a x en el conjunto $K-NN(o)$. Se calculan $K \times (N-n)$ cálculos de distancia.8N-n

De esta manera con $N \times n + K \times (N-n) \in (n)$ evaluaciones de distancia se pueden calcular los k-NN aproximados en BD para todo elemento de BD, si se selecciona un valor de p chico y K constante $K \gg k$, a medida que p sea más *p*grande cada elemento en $x \in BD - X_p$ podrá elegir mejor elemento $o \in X_p$, más cercano a x, por tener un conjunto más grande desde donde elegir el elemento o más cercano, y por consiguiente los K elementos más cercanos a o con mayor probabilidad contendrán todos los k elementos más cercanos a x (si realmente x está muy cerca de o los K-NN(o) vecinos más cercanos serían también cercanos a x y por consiguiente más posiblemente $k-NN(x) \subseteq K-NN(o)$). Por otra parte, en la medida que K se acerque a n, cualquiera sea el o que se elija, la probabilidad de que el conjunto de los K-NN(o) contenga los k-vecinos más cercanos de x en X aumentará (cada conjunto contendrá casi a todo X). Sin embargo, al aumentar p o al aumentar K aumenta el número de cálculos de distancia a realizar. En el caso extremo que p = 100, N = n y eligiendo K = n se realizarían n^2 cálculos de distancia.

B.1.0.1. Desarrollo en GPU

El algoritmo presentado antes puede ser implementado usando técnicas de computación de alto desempeño en GPU. Para ello, al inicio del procesamiento se transfiere la base de datos a la memoria global de la GPU, luego de ejecutar el proceso *all-k*-NN completo, al finalizar se transfiere a la CPU la lista con los *k*-NN

de cada objeto de la BD. El pseudo-código 5, muestra los pasos para su implementación en la GPU.

Algorithm 5 Pseudo-código para el cálculo de los *all-k*-NN

1: Copiar a la GPU la base de datos BD
2: Elegir los x_i tal que $x_i \in X_p$ y $X_p \subset BD$
3: **while** $x_i \in X_p$ **do**
4: Calcular las distancias entre x_i y todos los elemenntos de la BD
5: Ordenar las distancias mediante *Quicksort*
6: Guardar los K_{x_i} tal que $K >> k$ (Obtiene también los k-NN de x_i)
7: **end while**
8: **while** $o_j \in$BD$-X_p$ **do**
9: Determinar x_i talque $o = \min_{\forall x_i \in X_p}\{d(o_i, x_i)\}$
10: Calcular las distancias entre o_i y los objetos de K_{x_i}
11: Ordenar las distancias obtenidas
12: Obtener los k-NN en forma aproximada
13: **end while**
14: Transferir los *all-k*-NN a la CPU

Para el cálculo de los *all-k*-NN, el proceso se divide en las siguientes fases:

1. *Cálculo de Distancias:* Se calcula el vector de distancias a todos los objetos de la BD, es decir para cada $oi\ y\ xi \in BD$, se computa la distancia d (x_i, o_i). Este cálculo se realiza mediante un kernel en GPU donde cada bloque tiene 256 threads y existen x bloques ($x = \frac{|BD|}{256}$) Las distancias son almacenadas en la memoria global de la GPU.
   En esta etapa, además se calcula la distancia mínima entre un objeto o_i, con el conjunto de los objetos de X_p. El resultado se guarda en la memoria global. El máximo tamaño de esta estructura es $n - |X_p|\ y\ |X_p|$ es un porcentaje de BD.
   Las distancias son almacenadas en la memoria global de la GPU.
2. *Ordenar Objetos* $x_i \in X_p$*:* Después de haber calculado las distancias entre los objetos de X_p y los objetos de la

BD se procede a ordenarlos para obtener los *K*-NN, como se dijo anteriormente, al ser $K \gg k$ también se obtienen los *k*-NN de todos los x_i.

El ordemamiento se realiza aplicando el algoritmo *Quicksort*, el cual es implementado teniendo en cuenta la naturaleza altamente paralela de la GPU y las capacidades de CUDA 1.2 o superior. El *Quicksort* se lleva a cabo en dos etapas: *Ordenamiento Local* y *Unión de Listas Ordenadas*, ambos encargados de:

a. Ordenamiento Local: Esta fase recibe como entrada el vector de distancias y su salida son L sublistas ordenadas, una por cada bloque. Cada sublista es ordenada por un bloque de t threads según el algoritmo Quicksort iterativo. El número de threads por bloque T, es fijo y se determina en relación con los recursos requeridos por el bloque.

   Cada bloque elige un pivote local, el cual pertenece a la lista, y divide la secuencia de datos en dos sublistas: una con los elementos menores al pivote y otra con los elementos mayores o iguales al pivote. El pivote es la mediana de los tres elementos de la subsecuencia de datos: el primero, medio y último elemento de la lista [Sin69]. Cada bloque trabaja independiente a los otros, eliminando la necesidad de sincronización entre threads de diferentes bloques. En base al pivote seleccionado, todos los elementos menores que el pivote se mueven a una posición a la izquierda del pivote, y los mayores o iguales se desplazan a la derecha del pivote. La tarea se hace mediante el uso de memoria compartida y cada thread puede determinar la posición donde ubicar su elemento en la estructura compartida usando las funciones atómicas de CUDA.

   Una vez obtenidas ambas subsecuencias, una de ellas se almacena en una pila (residente en memoria compartida) continuando el proceso sobre la otra subsecuencia. Cuando una subsecuencia se ordenó se extrae de la pila la subsecuencia almacenada en el tope y se procede a ordenarla. Este proceso se repite iterativamente hasta que no existan subsecuencias a ordenar en la pila.

   Cuando el número de elementos de la secuencia es inferior a ocho, se ordena en forma secuencial, la

sobrecarga del proceso es muy grande en comparación con el tamaño de la secuencia a ordenar.

Al final de la etapa, cada uno de los bloques copia su subsecuencia ordenada a la memoria global. La salida son L sublistas ordenadas.

1. Unión de Listas Ordenadas: Esta fase recibe como entrada las L sublistas ordenadas y la salida es una única secuencia ordenada. Esta fase se realiza aplicando una reducción por unión de las listas ordenadas. Un bloque combina dos listas a la vez, en consecuencia, $\log_2(L)$ iteraciones son necesarias para encontrar el resultado final. Esta etapa requiere $\left\lceil \frac{L}{2} \right\rceil$ bloques con treinta y dos threads por cada uno y una estructura auxiliar en memoria global.

En ambas etapas, diferentes técnicas han sido aplicadas para optimizar el desempeño, las cuales son el uso de memoria compartida, copias anticipadas y acceso coalescente a la memoria global.

3. *Obtener los* $K_{x_i} - NN$*:* A partir de las distancias ordenadas de los elementos del conjunto X_p, se guardan en un vector los *K*-NN de cada x_i y en otra lista los *k*-NN de los objetos $x_i \in X_p$, los cuales ya son parte del resultado. Como se han obtenido los *k*-NN de los x_i, sólo falta obtener los *k*-NN de los objetos o_i, tal que $o_i \in (BD - X_p)$, para finalizar el proceso.

4. *Obtener los k-NN de todos los* o_i*:* En esta etapa se procede a buscar los *k*-NN de todos los o_i, $o_i \in (X - X_p)$. Se define un bloque de 256 threads por cada o_i. Si el tamaño de $|BD - X_p|$ supera la cantidad de bloques máxima aceptada por la GPU, entonces un bloque tratará más de un objeto.

En cada bloque, se obtiene el elemento x_i cuya distancia al objeto o_i es mínima. Esto da el conjunto de los *K*-NN del objeto x_i donde se encontrarán los *k*-NN de o_i.

Cada thread del bloque calcula la distancia entre el objeto o_i y los elementos del conjunto $K_{x_i} - NN$. Estas distancias son almacenadas en un vector, definido en memoria compartida. También se almacenan, en esta memoria, los índices de los objetos $K_{x_i} - NN$.

Después de obtener el vector de distancias y los índices de o_i se ordenan las distancias en orden ascendente

aplicando *Quicksort*. De esta manera se obtiene una lista con los elementos cuya distancia a o_i es menor, los *k*-NN aproximados de cada o_i.

Una vez finalizado el proceso, la lista conteniendo los all-k-NN de todos los o_i ($o_i \in X$) se transfiere a la CPU.

Esta propuesta es un algoritmo aproximado (relaja la condición de calidad o exactitud de la respuesta en función de obtener menor tiempo, no necesariamente los resultados son los verdaderos). Si bien es un buen método para obtener los all-k-NN cuando se implementó' en la GPU se tuvieron que realizar algunas restricciones, una de ellas es fijar el tamaño del conjunto X_p y de K para un mejor uso de la GPU. Tamaños variables implican que deberíamos almacenarlos en la memoria global de la GPU, y consecuentemente, se incrementará el tiempo, el acceso a la memoria global tiene un costo cien veces mayor que acceder a la memoria shared. Otra dificultad detectada es la necesidad de realizar un ordenamiento de las distancias calculadas. Si bien el algoritmo de ordenamiento implementado es el Quicksort (menor costo que otros ordenamientos), en el contexto de toda la aplicación esto incrementa su costo. Otro aspecto a tener en cuenta es la determinación de la mínima distancia entre el objeto $o_i \in (BD - X_p)$ y los objetos $x_i \in X_p$ no teniendo independencia en los datos.

Esta solución obtuvo baja aceleración producto de la cantidad de estructuras de datos en la memoria global necesarias, el overhead agregado por el algoritmo de ordenamiento y la dependencia de datos.

B.2. Propuesta Men-M

El algoritmo propuesto es una variante del algoritmo de fuerza bruta. Éste consiste en elegir de manera aleatoria |*M*| objetos $M \subseteq BD$, donde |*M*| es relativamente pequeño, por ejemplo 0,01% del tamaño de la BD. Los objetos de BD cuya distancia es menor que todos los *M* objetos elegidos, contendrán a los *k*-NN. Los *M* objetos son elegidos aleatoriamente.

Las consultas resueltas en esta propuesta son: búsqueda por rango, *k*-NN, all-*k*-NN y Join-*k*-NN. La búsqueda por rango se resuelve de la misma manera que la explicada en 6.3. La dificultad está en dar solución a las consultas *k*-NN, all-*k*-NN y

join-*k*-NN. Como se explicó anteriormente, los objetos cuya distancia es menor a todas las distancias de los objetos de *M* forman parte del resultado de la consulta. Puede suceder que el número de objetos resultantes sea menor a *k*, en ese caso se deben considerar todos los objetos, cuya distancia sea menor a todas las

M-1 distancias. Este proceso se repite iterativamente hasta que el resultado final sea mayor o igual a *k*. Si el conjunto resultante es mayor que *k*, se deben ordenar las distancias y de esta manera se obtienen los *k* elementos menores a la consulta.

B.2.0.2. Desarrollo en GPU

Al inicio de la computación, se deben guardar en la GPU los datos de entrada del algoritmo, para ello se transfiere a la memoria global los objetos de la BD, el conjunto *M* y el de consultas *Q*. Cada objeto de *M* es seleccionado en forma aleatoria. Al finalizar el proceso de resolución de la consulta, el resultado se transfiere a la CPU. El pseudo-código 6 muestra las características de la implementación en GPU del algoritmo propuesto.

Algorithm 6 Pseudo-código los *Men-M*

1: Elegir los m_i tal que $m_i \in M$ y $M \subset$ BD
2: Copiar a la GPU la base de datos BD y los conjuntos Q y M
3: **while** q_j tal que $q_j \in Q$, donde Q es el conjunto de obj. de consulta **do**
4: Calcular las distancias entre $d(q_i, o_i) \forall x_i \in$ BD
5: Obtener $d(q_j, m_i)$, donde $q_i \in Q$ y $m_i \in M$
6: **if** $d(q_j, o_i) < d(q_j, m_i) \forall m_i \in M, M-1, \ldots$ **then**
7: $k\text{-NN}(q_i) = k\text{-NN}(q_i) + \{o_i\}$
8: **end if**
9: **end while**
10: Transferir los resultados a la CPU

El proceso se divide en las siguientes fases:

1. *Calcular Distancias*: Se calcula el vector de distancias entre un objeto de consulta q_j y todos los

elementos de la base de datos o_i. Esta etapa se ejecuta en un kernel donde la cantidad de bloques, de 256 threads cada uno, se define según el tamaño de la base de datos $|\#B| = \frac{|X|}{256}$ El cómputo de las distancias se almacena en memoria global de la GPU.

2. *Calcular k-NN*: Cualquiera de las consultas (*k*-NN, *all-k*-NN y *join-k*-NN) las resolvemos mediante *k*-NN.
    a. Se obtienen las distancias entre los elementos de M y q_j. Estas distancias se almacenan en un vector en memoria global.
    b. Se determinan cuáles son los objetos menores al conjunto M. Para ello, cada thread compara su $d\ (q_j\ ,\ o_i)$ con todas las $d\ (q_j\ ,\ m_i)$, de esta manera se puede conocer cuáles son los o_i con menor distancia que la distancia de la consulta a todos los objetos en M, a todos los de M-1 y así sucesivamente, hasta completar los *k* elementos. El resultado se almacena en memoria global.
    c. En base al resultado obtenido en la fase anterior, se reportan los *k* objetos ordenados, primero aquellos menores a los M, luego a los M-1 y así siguiendo.

    Si la cardinalidad del conjunto obtenido como respuesta es mayor a *k* se ordenan las distancias y se devuelven los *k*-NN de la consulta.

El resultado se transfiere a la CPU. Es importante destacar que este proceso se realiza iterativamente por cada una de las consultas.

Por sus características, el resultado de esta solución es altamente dependiente de la elección del conjunto M, es importante el tamaño de M. Si la elección de los M elementos es mala, influirá directamente en el resultado. Además en caso de no completar los *k* con distancia menor a todos los M, se deben reportar los menores a M-1. En este último caso, es necesario aplicar un algoritmo de ordenamiento sobre el resultado, éste se debe realizar según los M (primero los menores a todos los M, después los correspondiente a M-1 y así siguiendo).

APÉNDICE C

Ventajas y Desventajas del Estado del Arte

De la bibliografía existente relacionada a la resolución de consultas en espacios métricos, más precisamente en la búsqueda de los k-NN en una base de datos, y la GPU, podemos establecer para cada una de ellas las siguientes ventajas y desventajas:

1. En [BGTP10, Bar11] se presentan tres versiones distintas del algoritmo fuerza bruta para obtener los k-NN de la consulta. Previamente se ejecutó un kernel en el cual cada thread calcula la distancia entre el objeto de consulta y el objeto de la BD que le corresponde. El resultado es un vector de distancia de longitud *n* almacenado en memoria global, el cual es el dato de entrada del kernel que obtiene los *k*-NN. Las desventajas de las tres versiones son:
    a. Versión 1 *Ordering Reduction*: Implementa el quicksort a través de dos lanzamientos de kernels. En el primero el vector de distancias se particiona según un pivote y se determina la posición de los datos en la memoria global mediante el uso de la función atómica global *AtomicAdd*. Este kernel se invoca P-1 veces, cada vez que un pivote debe ser seleccionado, en consecuencia P-1 sincronizaciones por barrera y comunicaciones entre CPU y GPU son hechas. Al finalizar este kernel se obtienen P particiones de distintos tamaños, lo cual produce un desbalance de carga en el sistema imposible de evitar por que el tamaño de las particiones está sujeto a la selección adecuada del pivote. Al finalizar el kernel de partición se ejecuta el kernel de ordenamiento en el cual cada bloque de threads se encarga de ordenar la partición que le corresponde usando quicksort.

b. Versión 2 *Heap Reduction*: En este caso se utiliza un único bloque de *t* threads para resolver la consulta, desperdiciando las características de la GPU. Cada thread es sobrecargado con los cálculos de los *k*-NN. Primero se crea una matriz de heaps de $t \times k$, la cual dependiendo del valor de *k* y de *t* se va a almacenar en memoria compartida o en memoria global. Luego 32 threads trabajan obteniendo una matriz de heaps de $32 \times k$ y por último un único thread obtiene los *k*-NN de la consulta a partir de los 32 heaps de *k* elementos anteriores. Todos estos pasos están divididos por sincronización por barrera en el bloque.
c. Versión 3 *Bulk Heap Reduction*: Realiza el mismo procedimiento que la versión anterior sólo que cada bloque de threads resolverá una consulta, es decir que *M* bloques resolverán en paralelo *M* consultas. Los problemas de la versión anterior siguen persistiendo. Los requerimientos de memoria se ven afectados para poder soportar los *t* heaps de tamaño *k* de las múltiples consultas.

Cuando muestran los resultados, especifican las bases de datos y los valores de *k* utilizados pero no los valores de aceleración obtenidos, sólo menciona que la versión 3 obtuvo mejor desempeño que el resto de las versiones, el cual decrementó suavemente a medida que crecía el valor de *k*.

También analizaron la escabilidad de los algoritmos con respecto al tamaño de la BD, estableciendo que el algoritmo *Bulk Heap Reduction* tiende a ser constante aunque el tamaño de la BD aumente. Esta afirmación se basa en que la aceleración aumenta levemente con respecto al algoritmo *Ordering Reduction*.

2. En [GDB08, GDNB10] se calculan todas las consultas en Q, obteniendo una matriz de distancias de $n \times m$ almacenada en memoria global (*n* es el número de elementos en la BD y *m* el número de consultas realizadas), la BD está almacenada en la memoria de textura para que los accesos sean a alta velocidad. El tamaño de la BD y el número de consultas están restringidos a los requerimientos de la memoria global de la GPU utilizada. En [GDNB10] se intentó optimizar este problema de espacio usando la librería de CUDA, CUBLAS, almacenando sólo la matriz triangular.

   Para la etapa de ordenamiento, cada thread ordena de forma secuencial una fila de la matriz, es decir una consulta, usando

el algoritmo *Comb Sort* o *Insertion Sort*. Ambos algoritmos tienen una complejidad de $O(n \times \log(n))$ (complejidad del algoritmo en secuencial, la cual se mantiene porque cada thread aplica el algoritmo de esa forma). Por lo tanto la complejidad final de la etapa de ordenamiento es de $O(m \times n \times \log(n))$. Por lo tanto, se puede observar claramente la sobrecarga de trabajo que realiza cada thread.

Los resultados se mostraron a través de la forma de trabajo de ambos algoritmos (el implementado con *Comb Sort* y el con *Insertion Sort*) con tamaños de la BD y de consultas relativamente pequeños (la BD más grande tenía aproximadamente 5000 elementos) y para una dimensión de 64. Los valores de k considerados son hasta 20. Primero comparan ambos algoritmos entre sí, mostrando que el algoritmo *Insertion Sort* tiene mejor desempeño.

Una vez seleccionado el mejor algoritmo utilizando CUDA: *Insertion Sort*, se lo compara con dos algoritmos secuenciales, uno implementado en C y el otro en Matlab[GI06]. Obtiene muy buenos resultados el algoritmo de GPU con respecto a los secuenciales se logra una aceleración entre 200 y 400 para la BD más grande considerada (5000 elementos aproximadamente). En [GDNB10] al utilizar la librería CUBLAS se obtuvo una aceleración 25 veces más que el algoritmo implementado en CUDA

3. En [KH10a] se calcula la matriz de distancias, por lo tanto el tamaño de los datos está restringido a los requerimientos de la memoria global. Cada bloque de threads resuelve una fila de la matriz de distancias y obtiene los *k*-NN de la consulta que le corresponde mediante la administración de un heap. Cada thread realiza mucho trabajo desaprovechando las características de la GPU.

   El problema fue presentado para BDs de distintos tamaños, entre 10000 y 80000 objetos cada uno cuya dimensión de los elementos es de 256. Los autores compararon el algoritmo con el respectivo secuencial obteniendo un speedup de 130 medido en segundos.
4. En [KZ09] se calculan todas las distancias, formando una matriz de $n \times m$, cada bloque de threads se hace cargo de una baldosa de datos, luego aplica el algoritmo de ordenamiento *Radix-sort*. Este algoritmo realiza muchas interacciones con accesos a la memoria global implicando alta latencia de escritura.

5. En [LWLJ09, LLWJ09, LLWJ10] Primero se tiene un kernel que calcula las distancias entre un objeto de consulta y los elementos de la BD para almacenar las distancias en memoria global de la GPU. Luego de haber calculado los *k*-NN de cada bloque, los cuales se almacenan en *B* colas, una por cada bloque, un único thread calcula los *k* vecinos más cercanos en las *B* colas y lo realiza en forma iterativa buscando primero el elemento cuya distancia es mínima en el conjunto de las *B* colas, lo elimina de dicha cola, lo cual el segundo elemento ocupa el lugar del objeto removido y que ahora forma parte del resultado. Al trabajar un único thread se desperdicia los recursos de la GPU y al tener threads ociosos puede provocar disminución en la performance del algoritmo, viéndose afectado el tiempo de ejecución.
   Este autor propone también la solución de múltiples consultas siguiendo el método anterior, solo que cada thread trabajará con *r* elementos en vez de uno. Esto provoca que los threads se sobrecarguen y no se aprovechen las características de la GPU al usar tantos bloques como threads sean posibles para resolver el problema en forma paralela.
   El algoritmo implementado se compara con dos algoritmos de fuerza bruta implementado en secuencial, uno utiliza *quicksort* y el otro *insertion sort* en la etapa de obtención de los *k*-NN. La base de datos que usan, es una BD sintética cuyos tamaños oscilan entre 8Kb y 1MB de objetos.
   En el paper los autores analizan la escabilidad del algoritmo en dos aspectos: uno al aumentar la dimensión de los datos desde 1 hasta 80 y el otro al aumentar el tamaño de la BD desde 8Kb hasta 1048Kb fijando la dimensión de los objetos en 7 concluyendo en ambos casos que el tiempo del algoritmo se incrementa en forma lineal. También se analiza el speedup, el cual aumenta con el tamaño de los datos en forma lineal. El mejor speedup se obtiene cuando se tiene una BD de tamaño 512Kb, es alrededor de 21 comparándolo con el *quicksort* secuencial y de 40 si se considera el *insertion sort* secuencial. Por los tanto los valores de speedup no son significativos con respecto a otros autores.
6. En [QMN09] se usa una estructura de árbol para resolver los *k*-NN, denominado kd-tree. Para la búsqueda de los *k*-NN se usan colas por prioridad para simular la recursión, el problema de implementarlo en GPU es que esta estructura debe ser de tamaño fijo, entonces futuras inserciones en el árbol son ignoradas, llegado a este punto el árbol queda

desbalanceado ocasionando de esta manera mucho backtracking para encontrar los *k*-NN. Cuando el árbol está desbalanceado no es adecuado para ejecutar en GPU.

7. En [BGT+ll, Barll] se presentan dos algoritmos basados en índices. Una de las desventajas en ambos algoritmos es el tamaño de las estructuras mantenidas en memoria global. Además utiliza un único bloque de threads para resolver una consulta, por lo tanto los threads deben realizar mucho trabajo y de esta forma no se aprovechan las características de la GPU y usar un mayor número de bloques de threads para resolver el problema. Además en [BGTP10] en la etapa partición de la implementación se hace uso de funciones atómicas globales.
   Ambos algoritmos se basan en el método *Heap Reduction* explicado en [Barll] por lo tanto las deventajas de este método influye en los índices.
   Otro problema que se presenta en ambos índices es que los kernels deben ejecutarse a veces en más de una iteración lo que disminuye el rendimiento del sistema debido a las sincronizaciones por barriers y a la comunicación CPU-GPU. Por todas estas razones estos algoritmos presentan speedups pobres.

Este análisis favoreció el desarrollo de *Top-K*, permitiendo incluir todos aquellos aspectos tendientes a favorecer el desempeño de la solución y evitar los que lo afectarían.

Bibliografía

[AHH+01] E. Allamanche, J. Herre, 0. Hellmuth, B. Froba, T. Kast- ner, and M. Cremer. Content-based identification of audio material using mpeg-7 low level description. In *Proceedings of the 2nd Annual International Symposium on Music Information Retrieval*, pages 197-204, Bloomington (Indiana), USA, 0ctober 2001.

[Bar11] R.J. Barrientos. Búsqueda por similitud en espacios métricos sobre plataformas multi-core (cpu y gpu). Master's thesis, Departamento de Ciencias de la Computación, Universidad de Chile, Jun 2011.

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: Stream computing on graphics hardware. *ACM TRANSACTIONS ON GRAPHICS*, 23:777-786, 2004.

[BFPR06] N. Brisaboa, A. Farina, 0. Pedreira, and N. Reyes. Similarity search using sparse pivots for efficient multimedia information retrieval. *Multimedia, International Symposium on*, 0:881-888, 2006.

[BGT+11] R.J. Barrientos, J.I. Gómez, C. Tenllado, M. Prieto, and M. Marin. knn query processing in metric spaces using gpus. In 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011), volume 6852 of Lecture Notes in Computer Science, pages 380-392, Bordeaux, France, September 2011. Springer.

[BGTP10] R.J. Barrientos, J.I. Gómez, C. Tenllado, and M. Prieto. Heap-based k-nearest neighbor search on gpus. In XXI Jornadas de Paralelismo (JP 2010), pages 559-566, Septiembre 2010.

[BGvN46] A.W. Burks, H. Goldstine, and J. von

Neumann. Preliminary discussion of the logical design of an electronic computing instrument. Technical report, Institute for Advanced Study, Princeton, New Jersey, June 1946.

[BMG04] E. Batlle, J. Masip, and E. Guaus. Amadeus: a scalable hmm-based audio information retrieval system. In *First International Symposium on Control, Communications and Signal Processing*, pages 731-734, 2004.

[BMGC04] E. Batlle, J. Masip, E. Guaus, and P. Cano. Scalability issues in an hmm-based audio fingerprinting. In *IEEE International Conference on Multimedia and Expo, ICME'04*, volume 1, pages 735-738, 2004.

[BPS+08] N. Brisaboa, 0. Pedreira, D. Seco, R. Solar, and R. Uribe. Clustering-based similarity search in metric spaces with sparse spatial centers. In *SOFSEM*, pages 186-197, 2008.

[Bri95] S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 574-584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[Buc07] I. Buck. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[BV00] J.F. Bercher and C. Vignat. Estimating the entropy of a signal with applications. *Signal Processing, IEEE Transactions on*, 48(6):1687-1694, 2000.

[BYCMW94] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In M. Crochemore and D. Gusfield, editors, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, number 807, pages 198-212, Asilomar, CA, 1994. Springer-Verlag, Berlin.

[BYR10] V. Britanak, P.C. Yip, and K.R. Rao. *Discrete Cosine and Sine Transforms: General Properties, Fast Algorithms and Integer Approximations*. Elsevier Science, 2010.

[CBKH05] P. Cano, E. Batlle, T. Kalker, and J. Haitsma. A review of audio fingerprinting. *J. VLSI Signal Process. Syst.*, 41(3):271-284, november 2005.

[CI07] A. Camarena-Ibarrola. *Análisis Digital de la Señal de Voz PhD thesis Borrador*. PhD thesis, Universidad Michoacana de San Nicolás de Hidalgo, México, Agosto 2007.

[CIC06] A. Camarena-Ibarrola and E. Chávez. A robust entropy- based audio-fingerprint. In *Multimedia and Expo, 2006 IEEE International Conference on*, pages 1729-1732, 2006.

[CIC10] A. Camarena-Ibarrola and E. Chávez. Real time tracking of musical performances. In *Proceedings of the 9th Mexican international conference on Artificial intelligence conference on Advances in soft computing: Part IFI*, MICAI'10, pages 138-148, Berlin, Heidelberg, 2010. Springer-Verlag.

[CLnRR11] E. Chávez, V. Ludueña, N. Reyes, and P. Roggero. Reaching near neighbors with far and random proxies. *Proceedings of the 8th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE 2011)*, pages 574-581, 2011.

[CLnRR14] E. Chávez, V. Ludueña, N. Reyes, and P. Roggero. Faster proximity searching with the distal sat. Technical report, Universidad Nacional de San Luis, 2014. Manuscrito a ser enviado a revista.

[CMBY99] E. Chávez, J. L. Marroquín, and R. Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, SPIRE '99, Washington, DC, USA, 1999. IEEE Computer Society.

[CMN01] E. Chávez, J.L. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching.

Multimedia Tools Appl., 14(2):113-135, 2001.

[CN00] E. Chavez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proceedings of the Seventh International Symposium on String Processing Information Retrieval*, SPIRE '00, Washington, DC, USA, 2000. IEEE Computer Society.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Proximity searching in metric spaces. *ACM Com puting Surveys*, 33(3):273-321, 2001.

[Coo13] S. Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU Computing Series. Morgan Kaufmann, 2013.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426-435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[CT65] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematical Computing*, 19:297-301, 1965.

[CT08] D. Cederman and P. Tsigas. A practical quicksort algorithm for graphics processors. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA'08, pages 246-258, Berlin, Heidelberg, 2008. Springer-Verlag.

[DL42] G.C. Danielson and C. Lanczos. Some improvements in practical fourier analysis and their application to x-ray scattering from liquids. *J. Franklin Institute*, 233:365-380, 435-452, 1942.

[DN88] F. Dehne and H. Nolteméer. Voronoi trees and clustering problems. In Gabriel Ferraté, Theo Pavlidis, Alberto Sanfeliu, and Horst Bunke, editors, *Syntactic and Structural*

Pattern Recognition, volume 45 of *NATO ASI Series*, pages 185-194. Springer Berlin Heidelberg, 1988.

[Dyk99] P.P.G. Dyke. *An Introduction to Laplace Transjorms and Fourier Series*. Springer Undergraduate Mathematics Series. Springer London, 1999.

[Far11] R. Farber. *CUDA Application Design and Development*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[FHvD+13] J.D. Foley, J.F. Hughes, A. van Dam, S.K. Feiner, M. McGuire, and D.F. Sklar. *Computer Graphics: Principles and Practice*. The systems programming series. Addison-Wesley, 2013.

[FJ98] M. Frigo and S.G. Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings oj the 1998 IEEE International Conference on*, volume 3, pages 1381-1384, May 1998.

[Fly95] M.J. Flynn. *Computer Architecture: Pipelined and Parallel Processor Design*. Computer Science Series. Jones and Bartlett, 1995.

[FR96] M. J. Flynn and K.W. Rudd. Parallel architectures. *ACM Comput. Surv*, 28(1):67-70, mar 1996.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. *SIGMOD Rec.*, 23(2):419-429, may 1994.

[Fuj96] H. Fujisaki. *Recent Research Towards Advanced Man- Machine Interface Through Spoken Language*. Elsevier Science, 1996.

[GBC01] V. Gupta, G. Boulianne, and P. Cardnal. Content based audio copy detection, April 2001. Patent Application number: 20110082877, IPC8 Class: AG06F1730FI, USPC Class: 707769.

[GDB08] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on*

Computer Vision and Pattern Recognition Workshops, volume 0, pages 1-6, 2008.

[GDNB10] V. Garcia, E. Debreuve, F. Nielsen, and M. Barlaud. K-nearest neighbor search: fast GPU-based implementations and application to high-dimensional feature mat- ching. In *IEEE International Conference on Image Pro- cessing (ICIP)*, pages 3757-3760, Hong Kong, China, September 2010.

[Geh96] W. Gehrke. *FORTRAN 95 Language Guide*. Springer, 1996.

[GI06] A. Gilat and J.A.M. Iglesias. *Matlab: una introducción con ejemplos prácticos*. Reverté, 2006.

[Gra03] A. Grama. *Introduction to Parallel Computing*. Pearson Education. Addison-Wesley, 2003.

[GRH+ 05] N. Govindaraju, N. Raghuvanshi, M. Henson, D. Tuft, and D. Manocha. A cache-efficient sorting algorithm for database and data mining computations using graphics processors. Technical report, 2005.

[HAC+01] 0. Hellmuth, E. Allamanche, M. Cremer, T. Kastner, C. Neubauer, S. Schmidt, and F. Siebenhaar. Content- based broadcast monitoring using mpeg-7 audio fingerprints. *In International Symposium on Music Information Retrieval, ISMIR*, 2001.

[Hen07] J. Hensley. AMD CTM overview. In *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*, New York, NY, USA, 2007. ACM.

[HK02] J. Haitsma and T. Kalker. A highly robust audio fingerprinting system. In *In International Symposium on Music Information Retrieval, ISMIR*, 2002.

[HKM08] W. Hwu, K. Keutzer, and T.G. Mattson. The concurrency challenge. *IEEE Des. Test*, 25(4):312-320, jul 2008.

[Hoa61] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321-322, jul 1961.

[Hwu11] W. Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[JNW07] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[KH10a] K. Kato and T. Hosino. Solving k-nearest neighbor problem on multiple graphics processors. In *Cluster, Cloud and Grid Computing (CCGRID) - 10th IEEE/ACM International Conference on*, pages 769-773. IEEE, 2010.

[KH10b] D. B. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[KLT13] S.M. Kuo, B.H. Lee, and W. Tian. *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications*. Wiley, 2013.

[KR88] B.W. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, 1988.

[KS03] F. Kurth and R Scherzer. A unified approach to content-based and fault tolerant music recognition. In *In 114th AES Convention*, Amsterdam, NL, 2003.

[Kub06] M. Kuba. On quickselect, partial sorting and multiple quickselect. *Inf. Process. Lett*, 99(5):181-186, sep 2006.

[KZ09] Q. Kuang and L. Zhao. A practical gpu based knn algorithms. In *International Symposium on Computer Science and Computational Technology (ISC-SCT)*, pages 151-155, Huangshan P. R. China, December 2009.

[Leo01] C. Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. A Wiley- Interscience publication. Wiley, 2001.

[Ler12] A. Lerch. *An Introduction to Audio Content Analysis: Applications in Signal Processing and Music Informatics*. Wiley, 2012.

[Lev66] V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals.

Soviet Physics Doklady, 10(8):707-710, 1966.

[LLWJ09] S. Liang, Y. Liu, C. Wang, and L. Jian. A cuda-based parallel implementation of k-nearest neighbor algorithm. In *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC'09.*, pages 291-296. IEEE, October 2009.

[LLWJ10] S. Liang, Y. Liu, C. Wang, and L. Jian. Design and evaluation of a parallel k-nearest neighbor algorithm on cuda-enabled gpu. In *Web Society (SWS), 2010 IEEE 2nd Symposium on*, pages 53-60, August 2010.

[LNOM08] E. Lindholm, J. Nickolls, S.F. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39-55, 2008.

[LWLJ09] S. Liang, C. Wang, Y. Liu, and L. Jian. Cuknn: A parallel implementation of k-nearest neighbor on cuda-enabled gpu. In *IEEE Youth Conference on Injormation, Compu ting and Telecommunication, 2009. YC-ICT'09*, pages 415-418. IEEE, September 2009.

[Mar05] H. Mark. Mapping computational concepts to gpus. In Matt Pharr, editor, *GPU Gems 2*, pages 493-508. Addison-Wesley, 2005.

[McC06a] J.J. McConnell. *Computer Graphics: Theory Into Practice*. Jones and Bartlett Publishers, 2006.

[McC06b] M. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *Proceeding of GSPx Multicore Applications Conference*, 2006.

[MOV94] M.L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15(1):9-17, 1994.

[MPC10] N. Miranda, F. Piccoli, and E. Chávez. Finding audio fingerprinter using gpu. In

MECOM 2010 CILAMCE 2010., pages 3127-3141, Buenos Aires, Nov. 2010.

[MPCCI10a] N. Miranda, F. Piccoli, E. Chávez, and A. Camarena-Ibarrola. Fast gpu audio identification. In *XVI Congreso Argentino de Ciencias de la Computación. XVI Congreso Argentino en Ciencias de la Computación (CACIC 2010)*, pages 229-242, Buenos Aires, Oct. 2010.

[MPCCI10b] N. Miranda, F. Piccoli, E. Chávez, and A. Camarena- Ibarrola. Using gpu to speed up the process of audio identification. In IEEE, editor, *10th International In- formation and Telecommunication Technologies Symposium, I2TS'2010*, pages 153-160, Río de Janeiro - Brazil, Dec. 2010.

[MSDS13] H. Meuer, E. Strohmáer, J. Dongarra, and H. Simon. Top500, supercomputers site. http://www.top500.org, November 2013.

[MTP+04] M.D. McCool, S. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787-795, 2004.

[MVSP11] J.I. Martinez, J. Vitola, A. Sanabria, and C. Pedraza. Fast parallel audio fingerprinting implementation in reconfigurable hardware and gpus. In *Programmable Logic (SPL), 2011 VIFI Southern Conference on*, pages 245-250, April 2011.

[MW10] H. Mark and J. Workman. *Chemometrics in Spectroscopy*. Elsevier Science, 2010.

[MZ97] I.L. MacDonald and W. Zucchini. *Hidden Markov and Other Models for Discrete-valued Time Series*. Chap- man & Hall/CRC Monographs on Statistics & Applied Probability. Taylor & Francis, 1997.

[Nav02] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28-46, 2002.

[NN97] S.A. Nene and S.K. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*,

19(9):989-1003, Sep 1997.

[Nol92] H. Nolteméer. Voronoi trees and applications. *In International WorkShop on Discrete Algorithms and Complexity*, pages 69-74, 1992.

[NUP11] G. Navarro and R. Uribe-Paredes. Fully dynamic metric access methods based on hyperplane partitioning. *Inf. Syst.*, 36(4):734-747, 2011.

[NVI12] NVIDIA. Cufft library programming guide, January
2012.

[NVI13a] NVIDIA. Cublas library user guide, July 2013.

[NVI13b] NVIDIA. Nvidia cuda c programming guide. vers. 5.5, July 2013.

[NVZ92] H. Nolteméer, K. Verbarg, and C. Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex schemes of geometric objects. *In Data Structures and Efficient Algorithms, LNCS 594*, pages 186-203, 1992.

[Pac11] P. Pacheco. *An Introduction to Parallel Programming*. An Introduction to Parallel Programming. Elsevier Science, 2011.

[Pea06] Peakstream. The peakstream platform: High productivity software development for multi-core processors. http://www.peakstreaminc.com, 2006.

[PH13] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 5th edition, 2013.

[PR09] R. Paredes and N. Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Discrete Algorithms (JDA)*, 7:18-35, March 2009.

[QMN09] D. Qiu, S. May, and A. Núchter. Gpu-accelerated nearest neighbor search for 3d registration. In *Proceedings of the 7th International Conference on Computer Vision*

Systems: Computer Vision Systems, ICVS '09, pages 194-203, Berlin, Heidelberg, 2009. Springer-Verlag.

[Qui02] M. Quinn. *Parallel Computing: Theory & Practice 21E*. McGraw-Hill Education (India) Pvt Limited, 2002.

[Qui04] M.J. Quinn. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw- Hill Higher Education, 2004.

[RB99] M.D. Root and J.R. Boer. *DirectX complete*. Complete Series. McGraw-Hill, 1999.

[Rey02] N. Reyes. Índices dinámicos para espacios métricos de alta dimensionalidad. Master's thesis, Universidad Nacional de San Luis, Diciembre 2002.

[Sam05] H. Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[San10] E. Sanders, J.and Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition, 2010.

[SG09] D. Shreiner and B.T.K.O.G.L.A.R.B.W. Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Pearson Education, 2009.

[SHG09] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1-10, Washington, DC, USA, 2009. IEEE Computer Society.

[Sin69] R.C. Singleton. Algorithm 347: An efficient algorithm for sorting with minimal storage. *Commun. ACM*, 12(3):185-186, 1969.

[Sin01] K. Sink. *DirectX 8 and Visual Basic Development*. Sams, Indianapolis, IN, USA, 2001.

[SKKC02] S. Shin, O. Kim, J. Kim, and J. Choil. A robust audio watermarking algorithm using pitch scaling. In *Digital Signal Processing, 2002. DSP 2002. 2002 14th International Conference on*, volume 2, pages 701-704, 2002.

[SL05] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54-62, sep 2005.

[SOB11] A. Sanabria, J.V. Oyaga, and C.P. Bonilla. Fast parallel algorithm for audio content retrieval on gpus. In *Computing Congress (CCC), 2011 6th Colombian*, pages 1-4, May 2011.

[SPA06] A. Spanias, T. Painter, and V. Atti. *Audio Signal Processing and Coding*. Wiley, 2006.

[SPV11] A. C. Sanabria, C. Pedraza, and J. Vitola. Algoritmo rápido para la búsqueda de audio por contenido sobre gpus. *Intekhnia*, 6(1):35-43, Enero-Junio 2011.

[Sun01] D. Sundararajan. *The Discrete Fourier Transform: Theory, Algorithms and Applications*. World Scientific, 2001.

[SW49] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[SWH13] G. Sellers, R.S. Wright, and N. Haemel. *OpenGL Super-Bible: Comprehensive Tutorial and Reference*. OpenGL. Pearson Education, 6th edition, 2013.

[TPO06] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. *SIGOPS Oper. Syst. Rev.*, 40(5):325-335, oct 2006.

[Uhl91] J. Uhlmann. Satisfying general proximity I similarity queries with metric trees. *Information Processing Letters*, 40(4):175 - 179, 1991.

[Uri05] R. Uribe. Manipulación de estructuras métricas en memoria secundaria. Master's thesis, Facultad de Ciencias Físicas y Matemáticas, Universidad de Chile, Santiago,

Chile, Abril 2005.

[VA04] P.K. Varshney and M.K. Arora. *Advanced Image Processing Techniques for Remotely Sensed Hyperspectral Data*. U.S. Government Printing Office, 2004.

[VR86] E. Vidal Ruiz. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recogn. Lett.*, 4(3):145-157, 1986.

[Wal10] P. Walsh. *Advanced 3D Game Programming with DirectX 10.0*. Jones & Bartlett Learning, 2010.

[Wan03] A.L. Wang. An industrial-strength audio search algorithm. In *Proceedings of the 4 th International Conference on Music Information Retrieval*, 2003.

[WFH04] N. Wardrip-Fruin and P. Harrigan. *First Person: New Media as Story, Performance, and Game*. MIT Press, 2004.

[WHW93] H.C. Wang, K. Hwang, and J.G. Wu. *Solutions Manual to Accompany: Hwang Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw- Hill Education, 1993.

[Woo12] J.W. Woods. *Multidimensional Signal, Image, and Video Processing and Coding*. Academic Press. Academic Press, 2012.

[ZADB06] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.

[ZF90] E. Zwicker and H. Fastl. *Psychoacoustics: Facts and Models*. Springer Series in Information Sciences, 1990.

Esta red se constituyó formalmente en noviembre de 1996 y actualmente 58 universidades argentinas son miembros activos. Sus objetivos son:
"Coordinar actividades académicas relacionadas con el perfeccionamiento docente, la actualización curricular y la utilización de recursos compartidos en el apoyo al desarrollo de las carreras de Ciencia de la Computación y/o Informática en Argentina".
"Establecer un marco de colaboración para el desarrollo de las actividades de posgrado en Ciencia de la Computación y/o Informática de modo de optimizar la asignación y el aprovechamiento de recursos".

El procesamiento e identificación en un repositorio de varias señales de audio en forma simultánea y en tiempo real, determina la necesidad de aplicar técnicas de computación de alto desempeño en la solución computacional para la determinación de la huella digital y la identificación de cada señal.
Existen situaciones en las que se producen de manera simultánea varias señales, las cuales deben ser procesadas también simultáneamente para ser analizadas contra un repositorio de información previamente almacenado. Una CPU dedicada con dos núcleos puede calcular en tiempo real, sin problema, las características de un par de señales; pero difícilmente podría calcular en tiempo real las características de decenas de señales que coexisten en el espectro.
El objetivo de esta tesis doctoral es diseñar e implementar algoritmos eficientes para la extracción de las características robustas de señales de audio y su recuperación desde un repositorio de señales usando GPU y en tiempo real.

ISBN 978-950-34-1302-9