Resolviendo consultas kNN mediante algoritmos exhaustivos en GPU

Fabricio Millaguir
Departamento de Ingeniería de Sistemas,
Universidad de La Frontera, Chile.
Email: f.millaguir01@ufromail.cl

Ricardo J. Barrientos
Centro de Modelación y Computación Científica (CMCC),
Universidad de La Frontera, Chile.
Email: ricardo.barrientos@ufrontera.cl

***Resumen*—Durante los últimos años, el uso y aplicación de GPUs (Graphics Processing Units) ha sido ampliamente abordado, y actualmente la GPU es una de las mejores alternativas para implementar procesamiento masivo (sobre todo de instrucciones en punto flotante) en paralelo. En el presente trabajo se propone un algoritmo exhaustivo en GPU, basado en el *Selection Sort* para resolver consultas de tipo kNN (k nearest neighbor). Se explotan los distintos niveles de paralelismo que la GPU ofrece, y además se hace un uso eficiente del ancho de banda de la memoria de la GPU al implementar un buen patrón de acceso a memoria. Los resultados obtenidos son comparados con alternativas multi-core.**

***Index Terms*—kNN, Búsqueda exhaustiva, GPU, Computación Paralela.**

I. Introducción

La solución eficiente a consultas kNN (k nearest neighbors) es relevante en una gran cantidad de áreas, tales como, búsqueda por similitud, recuperación de la información, minería de datos, estadística, reconocimiento de patrones, la Web. En particular en el presente trabajo, utilizamos el caso de búsqueda por similitud, utilizando una base de datos de vectores, que representan descriptores de imágenes y como archivo de consulta usamos consultas reales procesadas por un motor de búsqueda por similitud recientemente publicado.

En los últimos años ha sido ampliamente estudiado el uso y aplicación de GPUs [1], [2] (Graphics Processing Units) para la aceleración de procesos. NVIDIA es el principal fabricante de GPUs, y también el que ha diseñado CUDA [3], [4], que es la herramienta que permite la programación sobre sus GPUs y su arquitectura. El presente trabajo propone un algoritmo exhaustivo sobre GPU para resolver consultas kNN.

Es usual que se resuelvan consultas kNN en búsqueda por similitud utilizando algoritmos de indexación en espacios métricos [5], [6], sin embargo, en este trabajo se propone un algoritmo exhaustivo, por lo tanto, nuestra solución puede ser aplicada a un número mayor de bases de datos, y de distinto tipo.

Nuestra solución se basa en el algoritmo de ordenamiento *Selection Sort*, el que fue adaptado para ser utilizado en GPU. El mapeo de los datos a GPU fue convenientemente planeado para minimizar la cantidad de accesos a la memoria de la GPU, utilizando su propiedad de fusionado de operaciones de lectura y escritura bajo ciertas condiciones.

II. Conocimientos Básicos y Trabajo Relacionado

A continuación en la Sección II-A se describe a la GPU y su funcionamiento. En la Sección II-B se describe el trabajo relacionado relevante para el presente trabajo.

II-A. GPU (Graphic Process Unit)

A continuación se presenta una vista general de la arquitectura de una GPU de NVIDIA y el modelo de programación ofrecido por CUDA [3], con la finalidad de exponer las dificultades que presenta para producir un código eficiente.

La GPU es un coprocesador de alto rendimiento, adecuado para acelerar procesos secuenciales. Un programa que se ejecuta en la CPU debe manejar explícitamente la transferencia de datos desde memoria de la CPU a la memoria de la GPU y viceversa, y también qué funciones son las que se ejecutarán en la GPU.

La Figura 1 muestra un esquema del hardware de una GPU de NVIDIA. Los núcleos de la GPU están organizados en *multiprocesadores*. Cada núcleo es un procesador escalar que comparte una única unidad de instrucción con el resto de los núcleos del multiprocesador. Todos los núcleos de un multiprocesador ejecutan la misma instrucción a la vez en cada ciclo de reloj, pero sobre diferentes datos. Cada multiprocesador puede mantener cientos de hilos en ejecución. Estos hilos son organizados en grupos denominados *warps*.[1] En cada ciclo, el planificador en hardware de cada multiprocesador selecciona el siguiente warp a ejecutar, es decir, la unidad mínima de

[1] Actualmente, son 32 los hilos por warp.

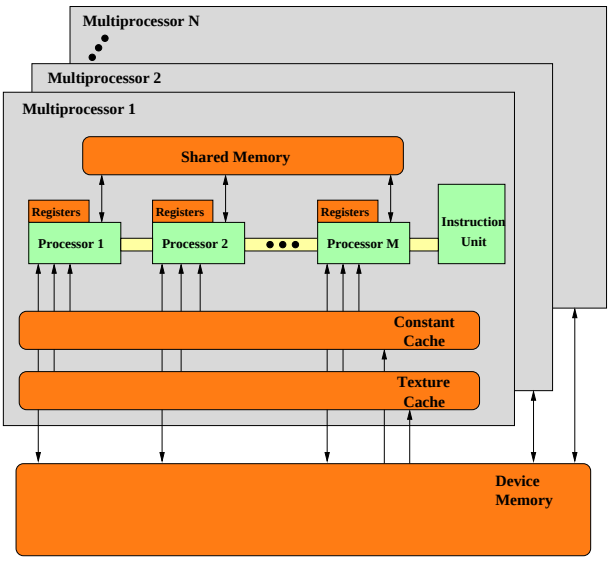


Figura 1. Esquema de una GPU.

ejecución no es un hilo, sino que un warp. Este modelo de ejecución se denomina SIMT (Single Instruction Multiple Thread) por NVIDIA. Si un hilo en un warp debe ejecutar una instrucción distinta del resto, entonces sólo dicho hilo estará usando la unidad de instrucción sobre un núcleo, y el resto deberá esperar los ciclos de reloj necesarios a que la instrucción (o instrucciones) del hilo termine, decrementando el rendimiento. Por lo anterior, es aconsejable (según [7]) que todos los hilos de un warp siempre ejecuten el mismo conjunto de instrucciones.

Con respecto a la jerarquía de memoria, todos los multiprocesadores pueden acceder a la misma memoria DRAM instalada en la misma GPU (*device memory*) a través de un bus con un gran ancho de banda. Esta memoria tiene la propiedad de poder fusionar varios accesos a memoria simultáneos a posiciones contiguas de memoria en sólo una transacción de memoria. Además, cada multiprocesador contiene una memoria SRAM más pequeña (*shared memory* en la Figura 1), la que puede ser manipulada por software, y una caché L1 controlada por hardware. El usuario puede decidir (con ciertas restricciones) la cantidad de caché y tamaño de memoria necesarios. Esta característica de los últimos modelos de GPU incorporan también una caché L2, a la que tienen acceso todos los multiprocesadores. También, cada multiprocesador tiene acceso a una *memoria de texturas*, físicamente contenida dentro de *device memory*, la que posee su propia caché (*texture cache* en la Figura 1). Finalmente, también existe una memoria de tamaño pequeño de sólo lectura denominada *memoria de constantes*, la que también posee una caché (*constant cache* en la Figura 1).

Si bien los hilos están agrupados en warps, los warps están agrupados en bloques denominados *CUDA Blocks*. Todos los warps en un CUDA Block son todos procesados siempre en el mismo multiprocesador, y también sus hilos pueden cooperar entre ellos mediante (1) la memoria compartida del multiprocesador, y (2) sincronizándose mediante funciones de barreras. Pero, hilos de distintos CUDA Blocks no pueden sincronizarse mediante barreras, pero sí comparten *device memory*. Bajo ciertas restricciones, el programador puede indicar cuantos CUDA Blocks y cuantos hilos por bloque se utilizarán.

Según NVIDIA, el factor más significante que afecta el rendimiento es el uso del ancho de banda. Aunque una GPU toma ventaja del *multithreading* para ocultar latencias en los accesos a memoria, el tener cientos de hilos simultáneamente accediendo a la memoria global introduce una gran presión al ancho de banda del bus de memoria. Por lo tanto, el mejorar los patrones de acceso a memoria es importante para permitir el fusionado de operaciones de lectura y escritura a *device memory*.

En resumen, algunas de las conclusiones para lidiar con este tipo de coprocesador son las siguientes. Primero, el programador necesita dividir el código entre lo que se procesará por la CPU y por la GPU. El código que se ejecutará en GPU debe ser mapeado en el modelo de CUDA de warps y bloques. Aquí se tiene dos niveles de paralelismo, al tener hilos independientes (los asignados a diferentes bloques) e hilos colaborativos (los asignados al mismo bloque). Se debe minimizar la divergencia en la secuencia de instrucciones de hilos del mismo warp, y acceder a direcciones contiguas de memoria por parte de hilos del mismo warp. El programador es el encargado de manejar la transferencia de datos entre las distintas memorias de la GPU,[2], debiendo minimizar los accesos a *device memory*.

II-B. Trabajo Relacionado

Los trabajos realcionados en GPU para resolver consultas por rango utilizan algoritmos de indexación para descartar elementos, y ellos son [8], [9].

En [9] los autores proponen soluciones basándose en un índice basado en una tabla de pivotes. En [8] se utilizan los índices *SSS-Index* [10] (basado en pivotes) y *List of Clusters* [11] (basado en clustering). Dichos trabajos intentan descartar elementos, y evitar comparar todos los elementos de la base de datos con la consulta.

Para poder realizar el descarte de elementos, los trabajos antes mencionados restringen su ámbito de aplicación a espacios métricos. Un *espacio métrico* [12] (X, d) está compuesto de una colección de datos $\mathbb{X}$ y una *función de*

[2]También se debe tomar en cuenta las distintas cachés, que son controladas por hardware

distancia $d : \mathbb{X} \times \mathbb{X} \rightarrow \mathbb{R}^+$ definida sobre los datos. La distancia determina la similitud entre dos objetos dados y debe mantener las siguientes propiedades:

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

Sin embargo, en el presente trabajo se proponen algoritmos exhaustivos, lo que implica que no estamos restringidos a utilizar espacios métricos, por lo tanto, las soluciones pueden ser aplicadas a un número mayor de bases de datos, y de distinto tipo.

III. Algoritmos Exhaustivos sobre GPU

A continuación se describe nuestra propuesta de implementación y mapeo para el algoritmo *Selection Sort*.

III-A. Selection Sort

Lo primero que se tomó en cuenta es la distribución espacial de los datos para beneficiarse de la propiedad de fusionado de operaciones de lectura y escritura de la GPU. Por lo anterior, los datos de la base de datos se organizaron en columnas en una matriz BD (de tamaño $Dim \times SIZE_{DB}$, Dim: dimensión, $SIZE_{DB}$: número total de elementos), donde cada columna de la matriz BD almacena un elemento de la base de datos.

Intentando explotar dos niveles de paralelismo (de grano grueso y fino), se distribuyen las consultas entre los CUDA Blocks (grano grueso), y cada consulta se resuelve con todos los hilos de un CUDA Block (grano fino). Se utilizaron 1024 hilos por CUDA Block, número encontrado (empíricamente) adecuado.

Si bien a continuación señalamos que ordenamos distancias, en realidad, ordenamos estructuras que además de la distancia también se almacena el índice del elemento. Por lo anterior, en nuestros algoritmos no usamos la función `atomicMin()` para encontrar la menor de las distancias porque no se desea encontrar solamente la menor de ellas, sino que además el elemento de la base de datos que posee dicha distancia.

La ejecución del algoritmo en GPU cuenta con los siguientes pasos:

1) El primer paso del Selection Sort en GPU distribuye de manera circular los elementos entre los hilos de un CUDA Block, y cada hilo calcula la distancia entre sus elementos asignados y la consulta correspondiente (asignada al CUDA Block). Las distancias son almacenadas en memoria global de la GPU.

2) Las distancias son distribuidas (circularmente) entre los hilos del CUDA Block, de tal forma que al terminar de recorrer el arreglo de distancias, cada hilo tendrá su elemento más cercano a la consulta. Las distancias son almacenadas en memoria compartida de la GPU esta vez.

3) Las distancias previamente almacenadas en memoria compartida son distribuidas (circularmente) entre los hilos del primer warp del CUDA Block. Al igual que el paso previo, al finalizar este paso cada hilo tendrá su elemento más cercano a la consulta, que es almacenado en memoria compartida nuevamente.

4) El primer hilo del primer warp del CUDA Block es el encargado de recorrer las distancias almacenadas en el paso previo, y encontrar finalmente el elemento más cercano a la consulta.

Los pasos previos son capaces de encontrar sólo un elemento, el más cercano a la consulta, por lo tanto para encontrar k elementos, los mismos pasos son aplicados k veces. Pero sólo la primera vez las distancias de los elementos de la base de datos contra la consulta es calculada. Dichas distancias son almacenadas en memoria global de la GPU, y de allí son accedidas en cada nueva iteración.

Con la finalidad de maximizar el uso de la propiedad de fusionado de operaciones de lectura y escritura, los elementos en memoria global y compartida de la GPU siempre son distribuidos circularmente.

La reducción ejecutada en los pasos 3 y 4 está basada en los algoritmos presentados en [8], en donde se muestra lo eficiente de realizar una reducción utilizando los threads del mismo warp, por ser la unidad mínima de ejecución y además estar todos sus hilos sincronizados al momento de ejecutar instrucciones.

III-B. Algoritmo multi-core

También se implementó un algoritmo multi-core exhaustivo para tomarlo como referencia. Las consultas se distribuyen entre los hilos, y cada hilo accede a todos los elementos de la base de datos. Este algoritmo utiliza como estructura auxiliar un heap por hilo, es decir, cada hilo almacena temporalmente los k elementos más cercanos a la consulta encontrados hasta el momento en su heap de tamaño k. Una vez que el hilo ha visitado todos los elementos de la base de datos, entonces los elementos almacenados en su heap serán los k elementos resultados para su consulta.

IV. Resultados Experimentales

La GPU utilizada fue una NVIDIA Tesla K20c, basada en arquitectura Kepler. Cuenta con 13 multiprocesadores y 192 núcleos por multiprocesador, lo que hace un total de 2496 núcleos. Tiene 49KB de memoria compartida y 5GB de *device memory*. La CPU host es un Intel Xeon E5530, 2.40GHz, arquitectura Nehalem, de 4 núcleos hyperthreading, y 50GB de memoria.

Las versiones multi-core fueron ejecutadas en un servidor equipado con 2xIntel Xeon E5-2670, 2.60GHz con 16 núcleos en total, arquitectura Sandy Bridge, 32

GB de memoria, y compiladas con icc versión 14.0.3, incluyendo parámetros de optimización.

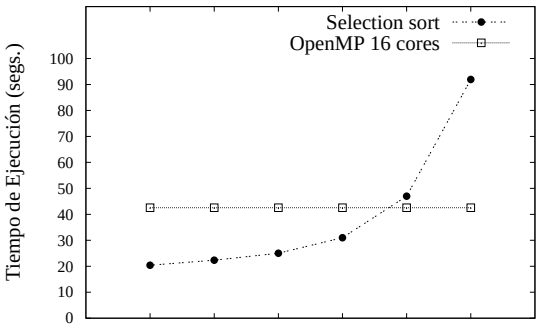
Se usó como bases de datos, la colección *CoPhIR* [13] (Content-based Photo Image Retrieval), la que consiste de metadatos extraídos de Flickr. Es una colección de 106 millones de imágenes, y por cada imagen existen cinco descriptores MPEG-7. Para el propósito del presente trabajo sólo se utilizó por cada imagen el descriptor *Color Structure*, el que se representa como un vector de dimensión 64. Se usó la *distancia euclidiana* como función de distancia. Los k utilizados fueron 1, 2, 4, 8, 16 y 32 por haber sido valores utilizados previamente por trabajos similares ([8], [9]).

Como archivo de consultas, se ha utilizado uno recientemente publicado en [14], el que está compuesto de consultas reales, obtenidas desde el sitio web [15]. El archivo de consultas está compuesto de 30,000 imágenes, representadas por su descriptor *Color Structure* de dimensión 64.

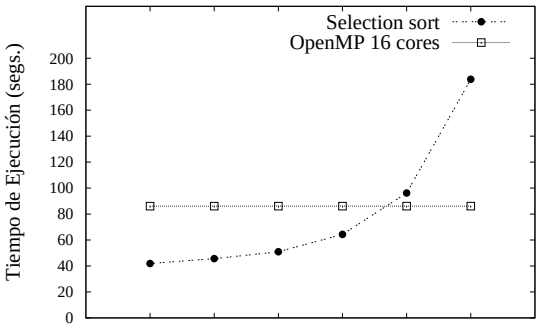
Por restricción en el tamaño de la memoria de la GPU, las consultas en la versión del Selection Sort son procesadas por lotes, hasta resolverlas completamente.

Las Figuras 2(a), 2(b) y 2(c) presentan el tiempo de ejecución de nuestra versión del Selection Sort en GPU (descrita en III-A) y la versión multi-core (descrita en III-B) usada como referencia. Se observa la ventaja obtenida con un valor de k pequeño para el algoritmo Selection Sort sobre el algoritmo multi-core. Esto es debido al buen patrón de acceso a memoria del Selection Sort para encontrar el elemento más cercano a la consulta. Pero, como el algoritmo se debe repetir k veces (para encontrar k resultados), mientras mayor es el k, su rendimiento se decrementa rapidamente. La versión multi-core no sufre cambios en su rendimiento al variar el parámetro k, debido a que lo único que debe cambiar es el tamaño del heap en memoria, y esto es implica un tiempo de ejecución extra despreciable para k pequeños.

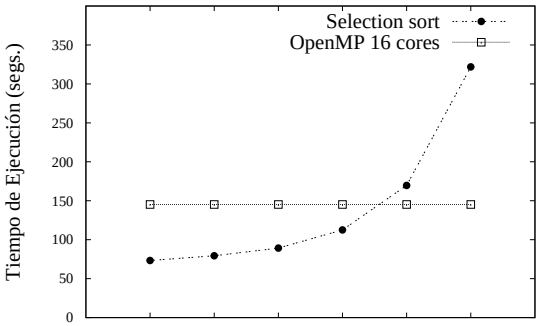
La Figura 3 muestra el speed-up logrado por el Selection Sort en GPU y el algoritmo multi-core, sobre un algoritmo secuencial (que es la misma versión multi-core pero ejecutada con sólo un núcleo). Se observa el alto speed-up alcanzado por nuestro algoritmo de Selection Sort con un k pequeño y el rápido decremento dependiente del tamaño de k.



(a) BD de 500,000 elementos.



(b) BD de 1,000,000 elementos.



(c) BD de 1,700,000 elementos.

Figura 2. Tiempo de ejecución sobre distintas Bases de Datos.

V. Conclusiones

En el presente trabajo se ha propuesto una alternativa del algortimo *Selection Sort* para ser usado sobre GPU, mostrando su utilidad para resolver consulas kNN (k nearest neighbors), y buen rendimiento por sobre alternativas multi-core.

El algortimo de *Selection Sort* aquí presentado es un algoritmo exhaustivo, por lo tanto, no está limitado a espacios métricos, como suele ocurrir al resolver consultas de este tipo. De allí que nuestras soluciones pueden ser aplicadas a un número mayor de bases de datos, y de distinto ámbito.

En la Sección IV se muestra la eficiencia de nuestra versión del Selection Sort en GPU por sobre sobre un algoritmo multi-core exhaustivo de 16 núcleos. Pero, su rendimiento decrece rapidamente al crecer el parámetro k de la consulta.

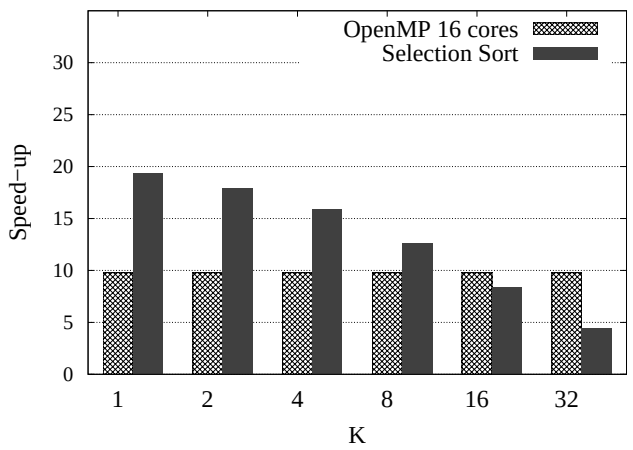


Figura 3. Speed-up sobre el algoritmo secuencial exhaustivo.

Por lo anterior, el algoritmo Selection Sort en GPU descrito en el presente trabajo es una alternativa eficiente para resolver consultas kNN con un k pequeño.

REFERENCIAS

[1] "GPU Computing. *http://www.nvidia.com/object/what-is-gpu-computing.html*."

[2] W. mei Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann, 2012.

[3] "CUDA: Compute Unified Device Architecture. ©2007 NVIDIA Corporation." [Online]. Available: http://developer.nvidia.com/object/cuda.html

[4] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

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

[6] R. J. Barrientos, J. I. Gómez, C. Tenllado, M. P. Matias, and M. Marin, "Range query processing on single and multi GPU environments," *Computers & Electrical Engineering*, vol. 39, no. 8, pp. 2656 – 2668, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790613001560

[7] *CUDA C Best Practices Guide*, 4th ed., NVIDIA Corporation, january 2012.

[8] R. Barrientos, J. 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)*, 2011, pp. 380–392.

[9] M. Kruliš, T. Skopal, J. Lokoč, and C. Beecks, "Combining cpu and gpu architectures for fast similarity search," *Distributed and Parallel Databases*, vol. 30, no. 3-4, pp. 179–207, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10619-012-7092-4

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

[11] E. Chávez and G. Navarro, "A compact space decomposition for effective metric indexing," *Pattern Recognition Letters*, vol. 26, no. 9, pp. 1363–1376, 2005.

[12] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[13] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti, "Cophir: a test collection for content-based image retrieval," *CoRR*, vol. abs/0905.4627, 2009. [Online]. Available: http://cophir.isti.cnr.it

[14] R. Barrientos, J. Gómez, C. Tenllado, M. Prieto, and P. Zezula, "Multi-level clustering on metric spaces using a multi-gpu platform," in *19th International European Conference on Parallel and Distributed Computing (Euro-Par 2013)*, ser. LNCS, vol. 8097. Aachen, Germany: Springer, August 2013, pp. 216–228.

[15] "MUFIN Web Site: Multi-feature Indexing Network. http://mufin.fi.muni.cz/imgsearch/similar."