Minimum Vertex Cover en CUDA

César Bragagnini (cesarbrma91@gmail.com)
Angela Mayhua (amayhuaq@gmail.com)

Maestría en Ciencia de la Computación
Universidad Católica San Pablo

I. Introducción

La teoría de grafos y sus problemas asociados son áreas de investigación tradicionales en ciencias de la computación, pero han sido retomados recientemente por su utilidad en minería de datos en sistemas estructurales de gran escala, tales como redes de contenidos web, redes de tráfico de tránsito, datos de las redes sociales, etc.

Los algoritmos sobre grafos también son importantes en aplicaciones de alto performance para minería de datos estructural. Este trabajo propone la utilización de CUDA para la obtención del *Minimum Vertex Cover* de un grafo, el cual es un problema importante en teoría de grafos y es conocido como NP-Hard [5].

II. Minimum Vertex Cover (MVC)

Un *Minimum Vertex Cover (MVC)* es el conjunto más pequeño de vértices que al ser removidos desconectan completamente un grafo [1].

El problema de obtener el MVC para un grafo de entrada $G = (V, E)$ con el conjunto de vértices $V = \{1, 2, ..., n\}$ y el conjunto de aristas $E = \{(v, u)|v, u \in V\}$ está representado como

$$\text{mín} \sum_{v \in V} x_v$$

tal que:

$$x_u + x_v \geq 1, \forall (u, v) \in E$$
$$x_v \in \{0, 1\}$$

III. CUDA y MVC

En este trabajo se revisó el algoritmo propuesto en [5], el cual analiza el estado actual de cada vértice y el de sus vecinos de tal manera que después de un conjunto de iteraciones se logra obtener el conjunto de vértices que pertenecen al MVC. Esta característica de analizar un vértice nos permite el uso de CUDA sin necesidad de realizar mucho cambios en el código, de esta manera se asigna un vértice a un hilo del device para que se encargue del procesamiento y análisis respectivo.

El algoritmo usa básicamente las siguientes estructuras:

- Mvc es un vector booleano, si $Mvc[v]$ es TRUE indica que el vértice v pertenece al MVC y sería FALSE en caso contrario.
- Adj es un vector booleano, si $Adj[v]$ es TRUE indica que todos los vecinos del vértice v pertenecen al MVC y sería FALSE en caso contrario.

A continuación se detalla la secuencia de pasos para obtener el MVC:

Paso 1: Los vectores Mvc y Adj inicializan sus valores con TRUE para todos los vértices.

Paso 2: $Mvc[v]$ será FALSE cuando el grado de v es el menor entre todos sus vértices vecinos.

Paso 3: $Adj[v]$ será FALSE si existe un vecino u tal que $Mvc[u] = false$.

Paso 4: $Mvc[v]$ será FALSE si existe un vecino u tal que $(Mvc[u], Adj[u]) = (true, false)$ y si no existe un vecino w tal que $(Mvc[w], Adj[w]) = (false, true)$. Luego ejecutar el Paso 3 para actualizar el estado de Adj.

Paso 5: Si $(Mvc[v], Adj[v]) = (false, false)$ quiere decir que existe un vecino u tal que $(Mvc[u], Adj[u]) = (false, false)$. Esta situación indica que existe una arista que no esta siendo considerada dentro del VC por lo que el conjunto generado hasta el momento no podría considerarse un VC, en este caso se actualizará $Mvc[v] = true$ cuando $ID[v]$ es el menor en comparación a $ID[u]$. Luego proceder con la actualización de Adj (Paso 3).

El algoritmo se detiene cuando ningún valor sea actualizado en la iteración actual, en otro caso se regresa al Paso 4. Al final del algoritmo se tendrá $(Mvc[v], Adj[v]) = (true, false)$ o $(Mvc[v], Adj[v]) = (false, true)$ para cada vértice $v \in V$.

IV. Pruebas y resultados

IV-A. Entorno de prueba

Se uso una maquina con un procesador Intel Core I7, Memoria RAM de 8GB, Sistema operativo Ubuntu 14.04 de 64 bits y GCC 4.8.4 y con NVCC 6.5.12, con una tarjeta gráfica de GeForce GTX 750. Se implementaron 3 programas, el primero usando una versión serial, segundo usando CUDA con memoria global y el tercero con CUDA usando zero-copy memory [1]

IV-B. Conjunto de datos

Se usaron distintos tipos grafos conexos aleatorios y grafos de red. Algunos dataset fueron obtenidos de un repositorio online [4], los demás fueron generados usando el paquete de generación de grafos de Richard Johnsonbaugh [3] [2].

IV-C. Resultados

En Tabla I se presenta los resultados para los distintos tipos de grafos usados, en las figuras 1, 2 y 3 ; los nodos coloreados de color verde representa el Minimum Vertex Cover del grafo total.

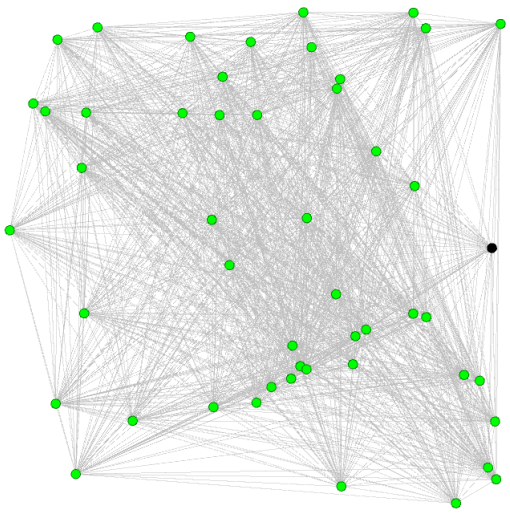

Figura 1. Grafo completo de 50 nodos

IV-D. Rendimiento

En las figuras 4 y 5 se puede observar la información del profiler de NVIDIA después del análisis de la base de datos p2p-Gnutella31.txt, se observan dos segmentos de flujo (el del lado izquierdo corresponde al uso de Global Memory y del lado derecho corresponde al procesamiento usando Zero-Copy Memory). En esta base de datos se realizaron 3 iteraciones para encontrar el mínimo vertex cover es por ese motivo que se observa la ejecucion de cada kernel 3 veces. Para el primer procesamiento se tiene envío de datos del host a la memoria global del device que son usados posteriormente en los kernels y es de acceso más rápido para el GPU. En el segundo flujo no hay presencia de envío de datos porque los datos se encuentran en memoria del host y el GPU puede acceder a esa información, pero como se puede observar cada kernel tiene un mayor tiempo de ejecución en comparación al uso de global memory.

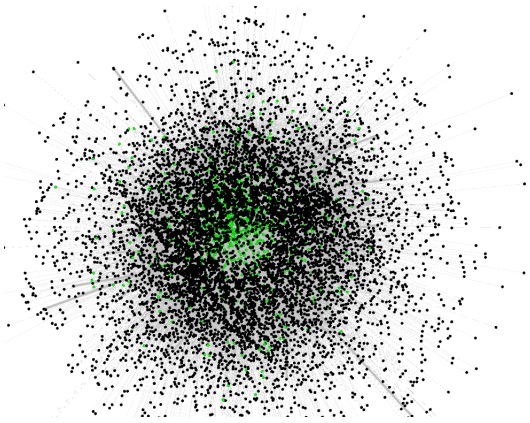

Figura 2. Grafo aleatorio de 10000 nodos y 50000 aristas

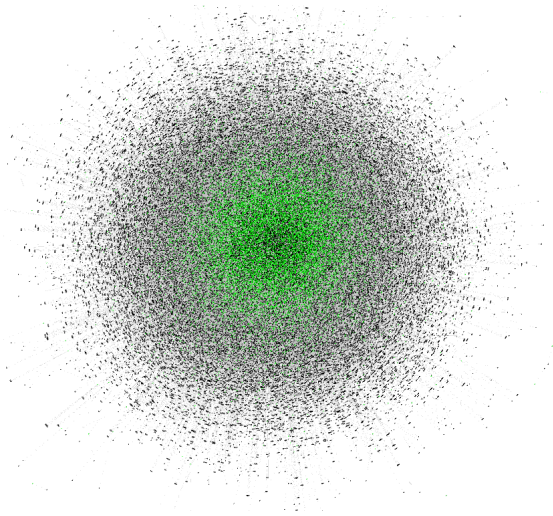

Figura 3. Grafo de 62586 nodos con 147892 aristas

V. Conclusiones

- Cuando se necesita acceder constantemente a los datos de memoria para realizar algún procesamiento en GPU, es recomendable hacer uso de la memoria global del device para que el acceso sea rápido y de esta manera se reduce el tiempo total de ejecución. A pesar que usar la memoria Zero-Copy no gasta tiempo en envío de datos, la acción de acceder constantemente a esa sección de memoria como en este trabajo reduce el performance y aumenta el tiempo de ejecución de

[1] Se puede descargar los codigos en https://github.com/marbramen/WorkingGroupCUDAPP

[2] Se puede obtener el programa de generación de grafos de Richard Johnsonbaugh en [2]

Tabla I. BASES DE DATOS USADAS PARA LAS PRUEBAS Y RESULTADOS

Nombre Grafo	Num Vert.	Num Aris.	CUDA Glob. Mem.	CUDA Zero Mem	Serial
Random Graph	50	1225	0.000900 s	0.009573 s	0.000108 s
Random graph	100	500	0.001294 s	0.008559 s	0.000080 s
Network Graph	1000	12000	0.001631 s	0.016122 s	0.001429 s
Random Graph	100000		0.002431 s	0.028635 s	0.008010 s
Random Graph	15000	100001	0.006896 s	0.052649 s	0.007685 s
Network Graph	20000	600000	0.026852 s	0.154240 s	0.024327 s
Random Graph	35000	300000	0.011732 s	0.077444 s	0.015372 s
p2p-Gnutella31	62586	147892	0.007589 s	0.052361 s	0.046714 s
soc-sign-epinions	131828	841372	0.141178 s	0.580496 s	0.197025 s

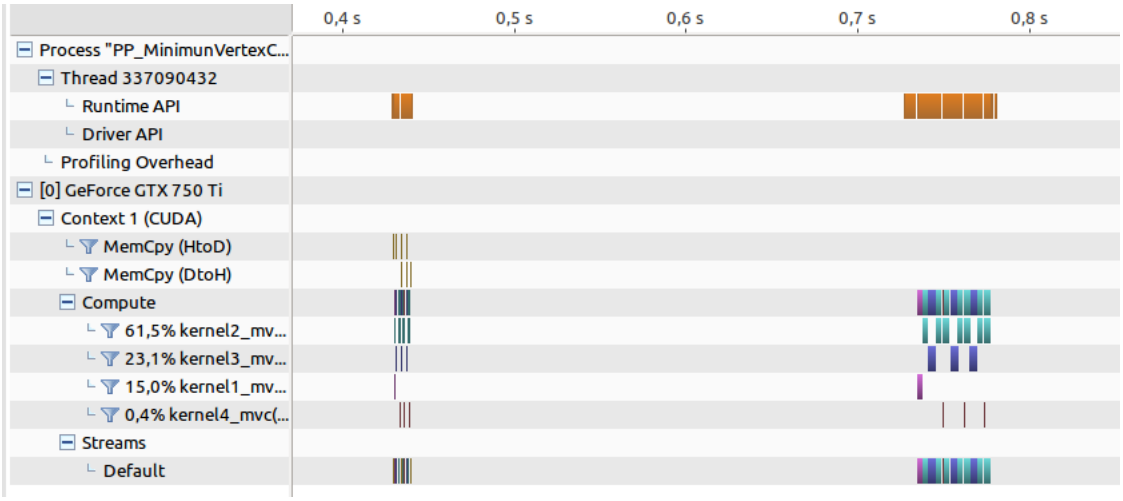

Figura 4. Resultados del profiler de NVIDIA después de analizar un grafo de 62586 nodos y 147892 aristas (usando Global Memory y Zero-Copy Memory)

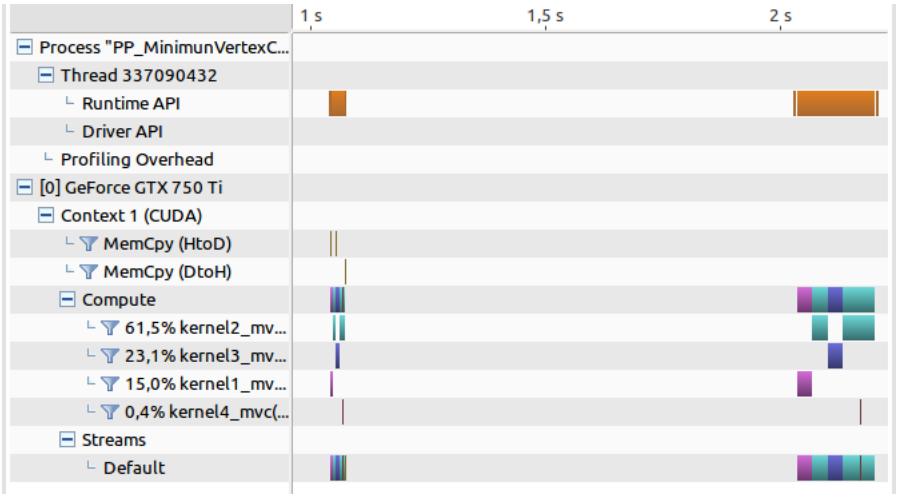

Figura 5. Resultados del profiler de NVIDIA después de analizar un network graph de 20000 nodos y con 600000 aristas (usando Global Memory y Zero-Copy Memory)

los diferentes kernels.

- Es recomendable trabajar con estructuras simples en CUDA, ya que enviar estos tipo de datos en gran medida a la memoria del device, su posteriores accesos constantes perjudicaria el rendimiento.

- No siempre la versión CUDA de un algoritmo va a ser mas rapido que su versión, hay que considerar que el acceso y operaciones con gran cantidad de datos, puede perjudicar el performance.

REFERENCIAS

[1] Mariana O. Da Silva, Gustavo A. Gimenez-Lugo, and Murilo V. G. Da Silva. Vertex cover in complex networks. *International Journal of Modern Physics C*, 24(11), 2013.

[2] Semantic Designs. A Graph Generation Package Richard Johnsonbaugh and Martin Kalin Department of Computer Science and Information Systems. http://condor.depaul.edu/rjohnson/source/graph_ge.c.

[3] Richard Johnsonbaugh and Martin Kalin. A graph generation software package. *SIGCSE Bull.*, 23(1):151–154,

March 1991.
[4] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.
[5] Kouta Toume, Daiki Kinjo, and Morikazu Nakamura. A gpu algorithm for minimum vertex cover problems. *International Conference of Computational Methods in Sciences and Engineering*, page 724, 2014.