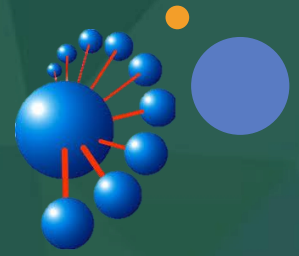


Paralelizar Algoritmo Floyd-Warshall

Nombre: Dazhi Feng Z.

Profesora: Cecilia Hernández R.

Introducción a la Computación Paralela (2023-1)

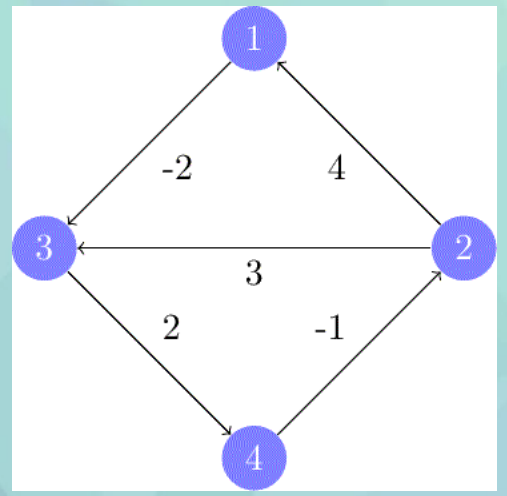
Introducción

El algoritmo Floyd-Warshall se utiliza para encontrar los caminos más cortos entre todos los pares de vértices en un grafo.

Utilizado en diversas aplicaciones, como en redes de comunicación, sistemas de navegación, optimización de rutas logísticas y análisis de redes sociales.

El algoritmo es secuencial y no aprovecha las capacidades de los sistemas multiprocesador o paralelos.

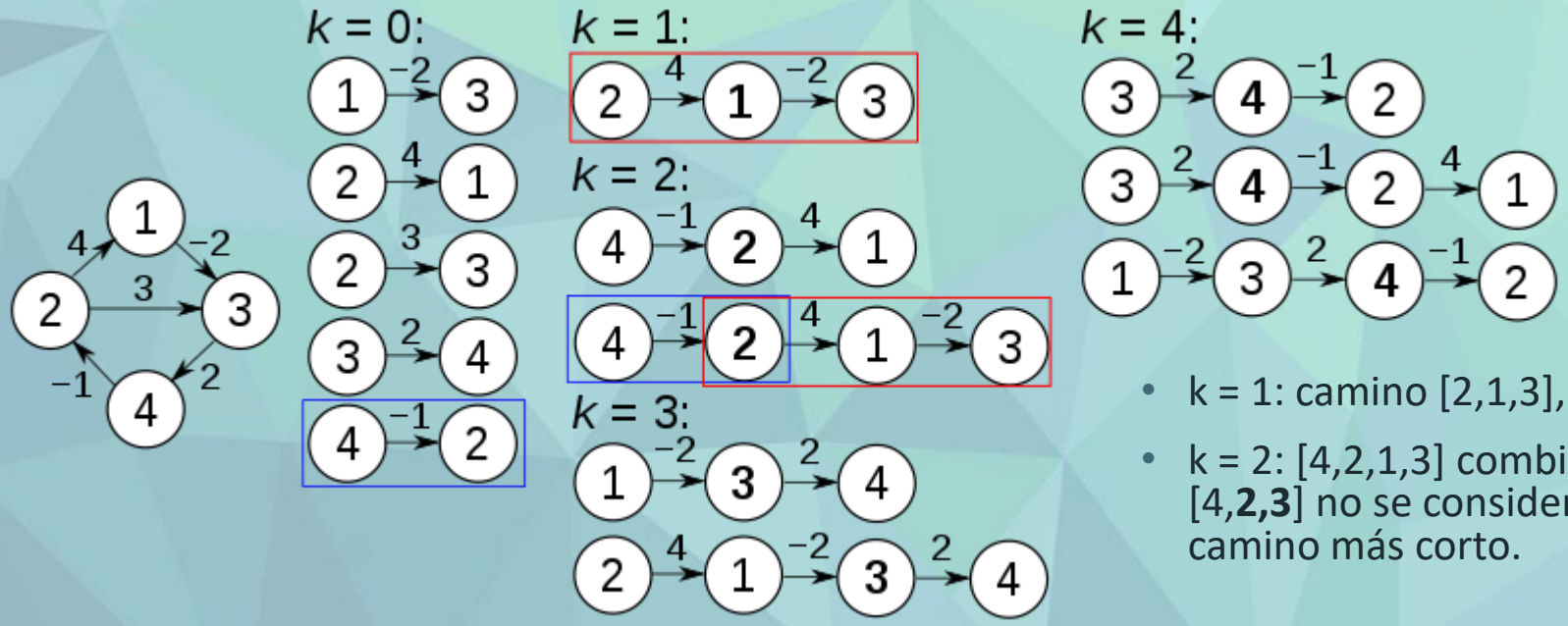
Dijkstra: Camino más corto desde **un** vértice a todos los demás vértices en un grafo.
Bellman-Ford: Camino más corto desde **un** vértice a todos los demás vértices en un grafo, **incluye aristas negativas.**
Floyd-Warshall: Caminos más cortos entre **todos** los pares de vértices en un grafo, **incluye aristas negativas.**



Algoritmo

```
let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)
for each edge (u, v) do
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

Utiliza una matriz para almacenar las distancias mínimas entre los pares de nodos del grafo. El algoritmo realiza una serie de iteraciones para actualizar la matriz y encontrar las distancias mínimas.



- k = 1: camino [2,1,3], reemplaza el camino [2,3]
- k = 2: [4,2,1,3] combinación de [4,2] y [2,1,3]. [4,**2,3**] no se considera, porque [**2,1,3**] es el camino más corto.

Paper

CUDA Analysis of Parallelization in Large Graph Algorithms

Munesh Singh Chauhan
Information Technology Department
College of Applied Sciences
Ibri, Oman
munesh.ibr@cas.edu.om

***Abstract*— Large graph problems are a commonly occurring problems that are found in real life situations. Finding the shortest route to one's destination through a maze of roads in a large metropolitan city is one such type. Similar problems are encountered in DNA sequencing, data mining, and search engines to name the few. Large graphs are associated with large number of vertices (V>>1000) and have large if not equivalent edges. Conventional Graph Algorithms though relevant are not suitable for such large graph based conditions because of the sheer increase in the computations. Most of the graph searching algorithms have time complexity in the range of *O(nlogn)* to $O(n^2)$. In order to optimize these computations and speed up the task, parallel graph algorithms are required. Parallelism is often implemented in the form of cache coherence and Divide & Conquer strategies. Clustering is another way of dealing with large graph problems.**

prime candidates for contemporary research initiatives. Graph problems in general and large graph problems in particular have irregular and random memory accesses. This pose a serious challenge for parallelization as most parallel frameworks rely on contiguous memory access which helps amortize the cost of single memory transaction.

The following sections explore the possibilities of overcoming the previously mentioned bottlenecks and iron out the issues that impede fast processing. Two main graph algorithms are studied and are parallelized keeping in view their vertex sets. The first algorithm called Floyd Warshall Algorithms belongs to the category of All Pairs Shortest Path First algorithms and the second one, Prim's Algorithm, belongs to the class of greedy algorithms for finding Minimum

I. Introducción

Grafos grandes son adecuados para la computación paralela utilizando GPUs.

El rendimiento de una GPU se correlaciona con el **ancho de banda de la memoria** y los **accesos a la memoria**.

El acceso a un registro es más rápido comparado con el acceso a la memoria asignada al host.

Por lo tanto, estos accesos a memoria asignada deben minimizarse.

Los problemas de grafos grandes tienen accesos a memoria **irregulares y aleatorios**.

Es un desafío para la paralelización, ya que la mayoría de los marcos de trabajo paralelos se basan en accesos a memoria **contiguos**.

II. Patrones de acceso a memoria

warp: grupo de hilos que se ejecuta de manera concurrente en CUDA.

Unidad básica de paralelismo.

Los hilos de un warp se **detienen** cuando los accesos a la memoria son **aleatorios** y no entran en una única transacción de banco de memoria.

Solución: hacer los accesos irregulares en la memoria compartida (similar a una caché L1) para un acceso rápido.

Caso ideal para paralelismo de accesos a memoria irregulares: tener grandes bloques de memoria compartida.

Pero, esta memoria es costosa y, actualmente, la arquitectura Fermi proporciona 48 KB de memoria compartida.

III. Algoritmo de Floyd-Warshall

```
for k = 1 → n do
for j = 1 → n do
for i = 1 → n do
  A_k[i,j]=min(A_k-1[i,k]+A_k-1[k,j], A_k-1[i,j])
end for
end for
end for
```

Fig. 1. Naïve Floyd Warshall Algorithm

Calcula el camino más corto entre todos los pares de vértices en un grafo.

Tiene aplicaciones en áreas como análisis de redes sociales, topología de redes y secuenciación de ADN.

Para aplicar este algoritmo a grafos grandes, se debe cargar el grafo completo como entrada.

Limita el rendimiento al guardar en caché todos los detalles del grafo.

El algoritmo de Floyd-Warshall naive tiene una complejidad temporal de $O(|V|^3)$.

IV. Algoritmo FW paralelizado

- La paralelización se aplica en el kernel (figura 2), donde a cada loop se le asigna un hilo para lograr ejecución concurrente.
- El tamaño máximo del grafo está relacionado con el tamaño del bloque CUDA, y actualmente es de **1024** hilos por bloque máximo.
- Si el tamaño del grafo supera los 1024 vértices, se debe dividir la matriz en tiles (matrix tiling).
  - El tamaño de los tiles es igual al tamaño de la memoria compartida.

```
__global__ void floydKernel(double *dist) {
        int i = threadIdx.y + blockIdx.y*blockDim.y;
        int j = threadIdx.x + blockIdx.x*blockDim.x;
        for (int k = 0; k<V; k++)
        if (dist[i*blockDim.x + k]
        + dist[k*blockDim.x + j]<dist[i*blockDim.x + j])
        dist[i*blockDim.x + j] = dist[i*blockDim.x + k] +
        dist[k*blockDim.x + j];
    }
```

Fig. 2. Naïve Parallel Floyd Warshall Algorithm

IV. Algoritmo FW paralelizado

- El algoritmo tiene una buena aceleración utilizando múltiples hilos en un solo bloque.
  - Sin embargo, debido a los accesos de memoria desalineados, el aumento de velocidad está limitado.
- Para minimizar estas transferencias costosas, el tamaño de la matriz del grafo debe ser pequeño.
  - De lo contrario se produce una latencia incremental (figura 3).

TABLE I. BENCHMARKING OF FW CUDA KERNEL

Threads Per Block	Graph Matrix Size	Time in ms
16	4 x 4	0.009792
36	6 x 6	0.012416
64	8 x 8	0.014208
100	10 x 10	0.016960
144	12 x 12	0.018464
196	14 x 14	0.020544
256	16 x 16	0.022048
1024	32 x 32	0.055584

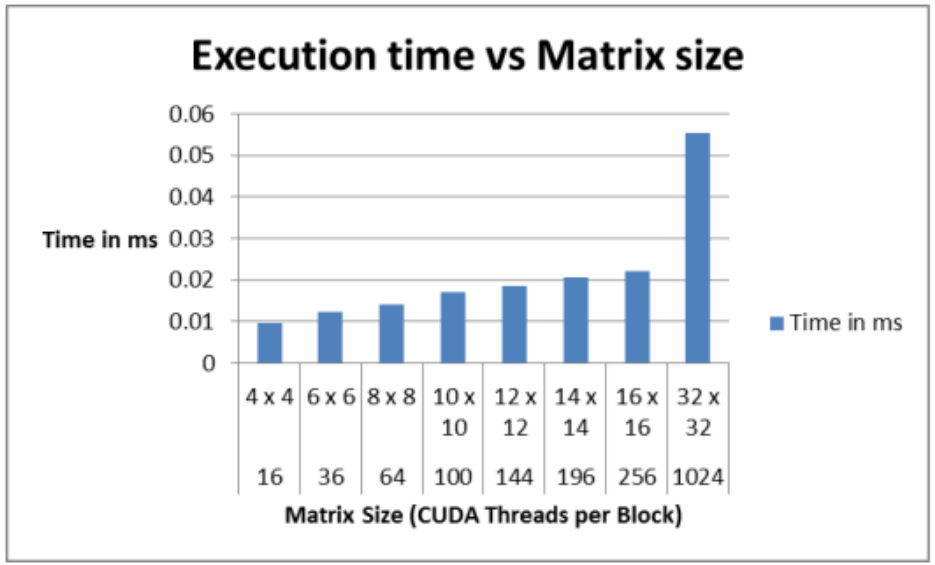


Fig. 3. CUDA Execution times – Floyd Warshall Algorithm

V. Paralelizar FW con memoria compartida

La implementación tiene bottlenecks relacionados con el acceso a la memoria.

Hay latencia debido a que los accesos a la memoria no están alineados en coalescencia (noncoalesced), generando múltiples transacciones de memoria.

Solución: Dejar que un warp (grupo de 32 hilos) acceda a 32 ubicaciones de memoria contiguas pertenecientes al mismo banco de memoria.

Esto permite realizar un acceso coalescido y minimizar las transacciones globales de memoria.

Entonces, se debe colocar los accesos de memoria desalineados en la memoria compartida.

Los accesos desalineados dentro de la memoria compartida tienen una latencia insignificante.

V. Paralelizar FW con memoria compartida

- Estos accesos a memoria deben hacerse en bloques de tamaño **entre 32 y 256 bits**.
  - En lugar de realizar múltiples accesos individuales de menor tamaño a la memoria, realizar accesos más grandes y continuos en bloques que se ajusten al tamaño.
- El autor también muestra en la figura 4 la relación entre el **ancho de banda efectivo** en GB/s y los diferentes **offsets** para *Single Precision* (**32 bits**) en distintas GPUs.

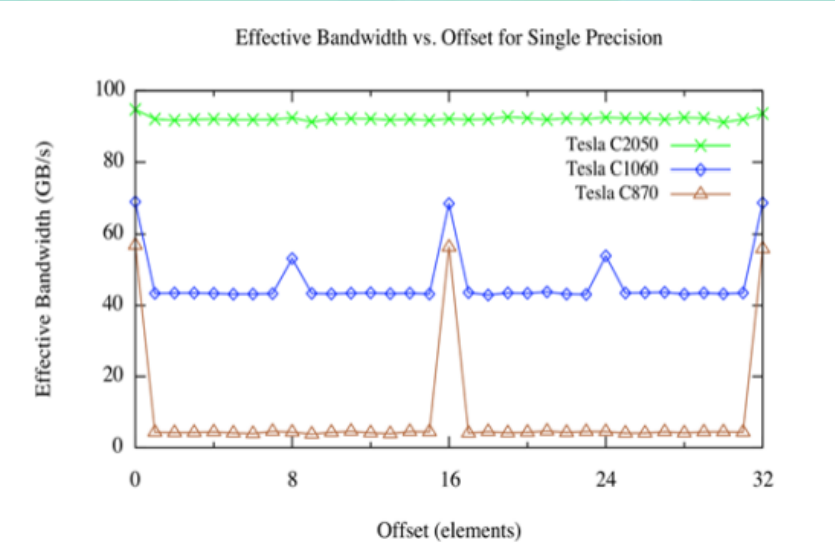


Fig. 4. Effective Bandwidth vs. Offset for Single Precision (Source: Nvidia-parallelforall)

VI. Paralelizar grafos grandes con tiles

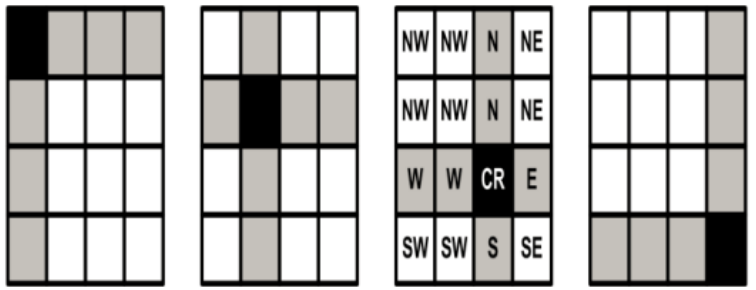


Fig. 8. Tiled Matrix Division

- Se segmenta la matriz del grafo en tiles (bloques) cuadrados de igual tamaño.
  - Los tiles tienen tamaño igual o menor que al tamaño de la memoria compartida (48 KB máx.).
- Se aplica un algoritmo de FW paralelo modificado basado en tiles:
  - Para la iteración *k* del bloque tile, se calcula el camino más corto (SP) de la siguiente manera:
    1. SP para el tile actual (CR).
    2. SP para todos los tiles (W) a la izquierda de CR.
    3. SP para todos los tiles (E) a la derecha de CR.
    4. SP para todos los tiles en la parte superior (N) de CR.
    5. SP para todos los tiles en la parte inferior (S) de CR.
    6. SP para el resto de los tiles en las diagonales (NW, NE, SW y SE).
- Los cálculos tienen acceso a memoria desde la memoria compartida, optimizando el rendimiento.

Conclusión

El uso de algoritmos paralelos CUDA es efectivo para paralelizar problemas de grafos grandes.

Sin embargo, la implementación de CUDA también presenta desafíos, como los cuellos de botella del acceso a la memoria.

Para minimizar esto, se propone el uso de la memoria compartida.

Pero esto no funciona para grafos muy grandes.

Para grafos de gran tamaño se propone segmentar la matriz en tiles y procesar cada tile.

Estos enfoques permiten una ejecución más rápida y eficiente, lo que resulta beneficioso en diversas aplicaciones como análisis de redes sociales, topología de redes, secuenciación de ADN, entre otros.

Opinión

- Se explica bien el problema de paralelizar algoritmos de grafos como Floyd-Warshall, en particular, problemas como accesos a la memoria no alineados, aleatorios y/o irregulares.
  - Mas que explicar cómo paralelizar el algoritmo, se propone cómo resolver estos problemas y las limitaciones que tiene este.
- Pero, en la explicación de los algoritmos, faltó detallar:
  - Se propone el uso de memoria compartida, pero no se muestra el algoritmo específico que lo implementa.
  - Además, el algoritmo propuesto para grafos grandes no tiene una explicación clara sobre por qué usar esos pasos.
- En cuanto al apartado de resultados/análisis experimental, este deja bastante que desear. El documento aborda la limitación del rendimiento debido a los accesos de memoria, pero no presenta una comparación de la mejora de rendimiento del algoritmo cuando se resuelven estos problemas.
  - Además, los resultados mostrados son con datos muy pequeños y no se realiza una comparación con el algoritmo secuencial.

Referencias

- Chauhan, Munesh. (2017). CUDA Analysis of Parallelization in Large Graph Algorithms. https://www.researchgate.net/publication/315489843_CUDA_Analysis_of_Parallelization_in_Large_Graph_Algorithms