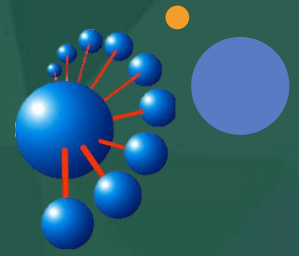


Paralelizar Algoritmo Floyd-Warshall

Nombre: Dazhi Feng Z.

Profesora: Cecilia Hernández R.

Introducción a la Computación Paralela (2023-1)

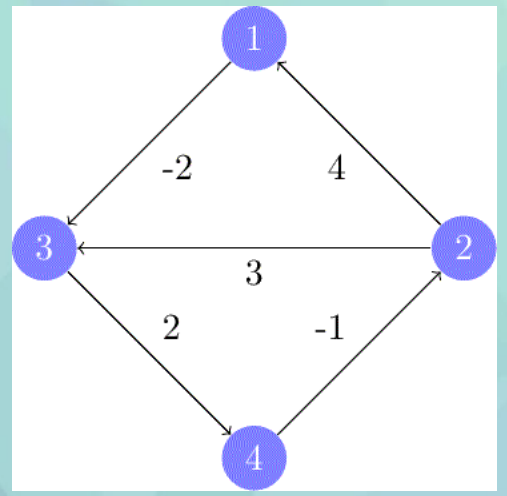
Introducción

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.

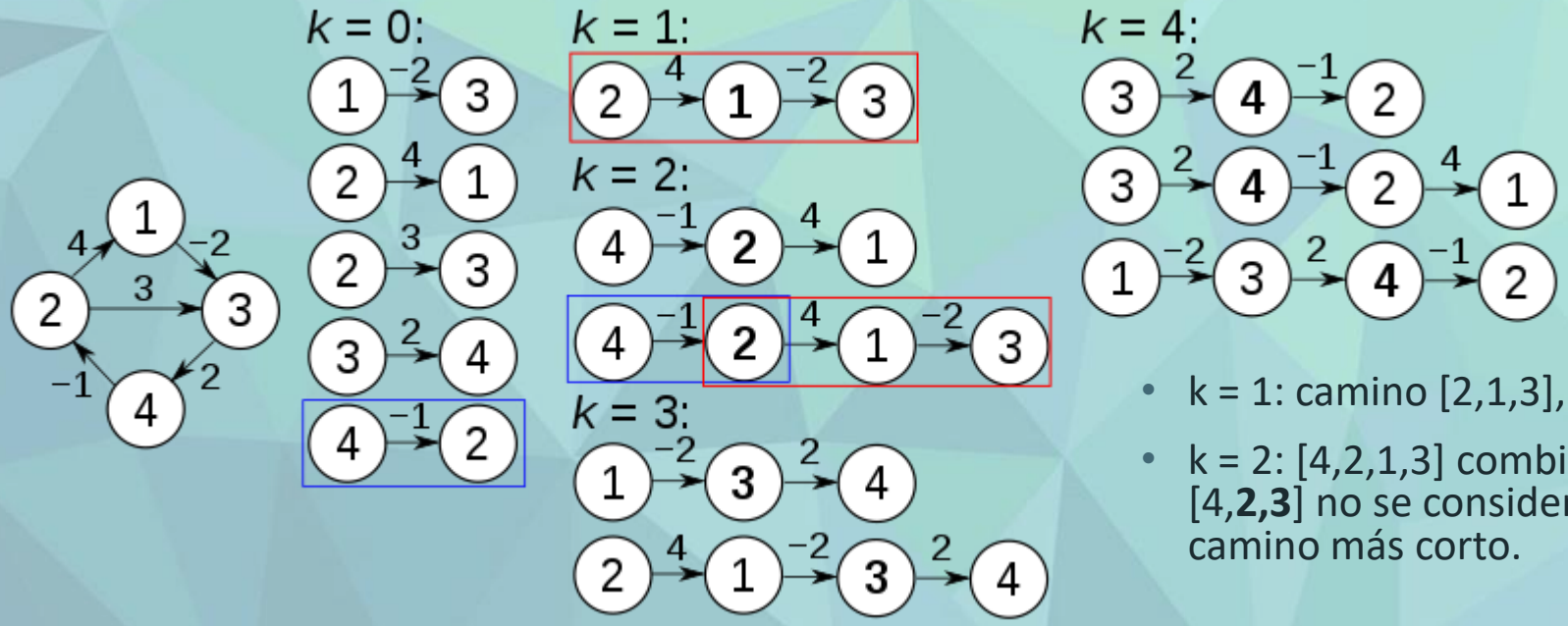
Dijkstra: Camino más corto desde **un** vértice a todos los demás vértices en un grafo.
Bellman-Ford: Camino más corto desde **un** vértice a todos los demás vértices en un grafo, **incluye aristas negativas.**
Floyd-Warshall: Caminos más cortos entre **todos** los pares de vértices en un grafo, **incluye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



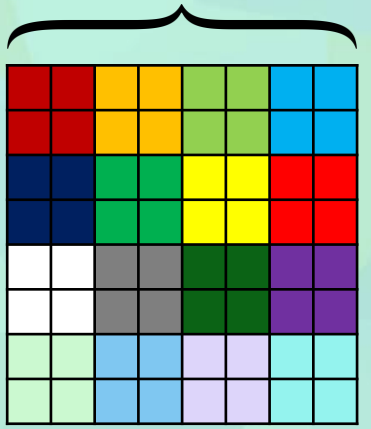
- k = 1: camino [2,1,3], reemplaza el camino [2,3]
- k = 2: [4,2,1,3] combinación de [4,2] y [2,1,3]. [4,**2,3**] no se considera, porque [**2,1,3**] es el camino más corto.

Floyd-Warshall Paralelizado con CUDA

- Dividir la matriz en bloques (submatrices) cuadrados con tamaños iguales.
- Cada bloque se ejecuta en paralelo (si hay suficientes recursos).
- Para cada k:
  - Cada **bloque** ejecuta el kernel con el algoritmo.
  - Los **hilos** de los bloques se ejecutan en paralelo.

```
// Dividir la matriz en bloques de tamaño B
// Grid es el numero de bloques
int B = 32;
dim3 block(B, B);
dim3 grid((V + B - 1) / B, (V + B - 1) / B);

// Para cada k, se recorre todos los bloques del grid, y se ejecuta el kernel FW
for (int k = 0; k < V; k++)
    floydKernel<<<grid, block>>>(thrust::raw_pointer_cast(device.data()), V, k);
```

16 bloques de 2x2



V = 8

B = 2

Grid = 8/2 = **4**

4x4 (16) bloques

Kernel

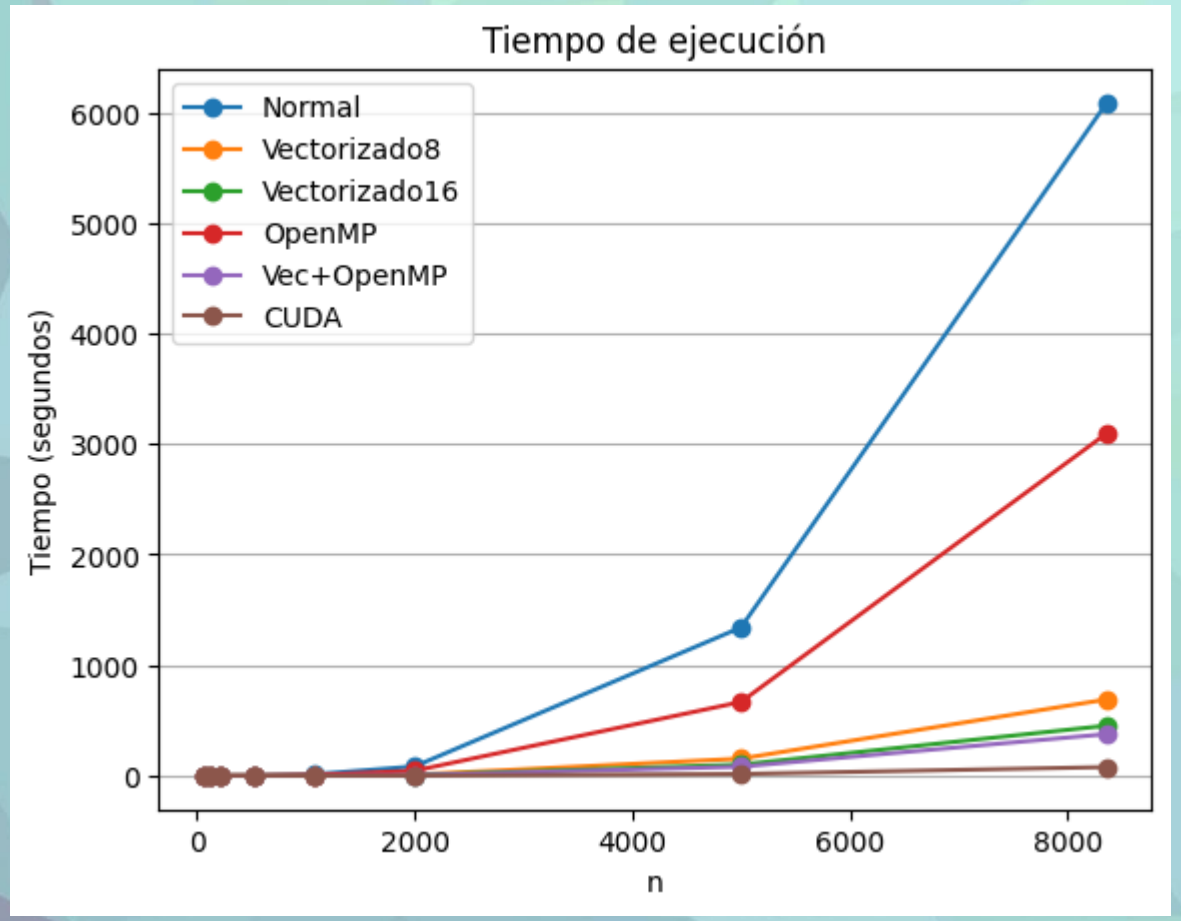
```
__global__ void floydKernel(float *dist, int V, int k)
{
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;

    __shared__ float ik[32], kj[32];

    if (i < V && j < V)
    {
        if (!threadIdx.x)
            ik[threadIdx.y] = dist[i * V + k];
        if (!threadIdx.y)
            kj[threadIdx.x] = dist[k * V + j];
        __syncthreads();

        float ikj = ik[threadIdx.y] + kj[threadIdx.x];
        dist[i * V + j] = fminf(dist[i * V + j], ikj);
    }
}
```

Secuencial

```
for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                // si dist de: i->k->j < i->j
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
```

- Se obtiene **i** y **j**
- Se crean 2 memorias compartidas.
- La memoria compartida **ik**, almacena la columna **dist[i][k]** del bloque.
- La memoria compartida **kj**, almacena la fila **dist[k][j]** del bloque.
- Se compara **dist[i][j]** con **ik + kj.**

Resultados



- Se puede ver claramente que **CUDA** tiene el mejor desempeño y se demora aproximadamente **1%** del tiempo secuencial.
- GPU utilizada: NVIDIA GeForce GTX 1650 (4 GB)

Tabla 1: Numero de vertices y tiempo en milisegundos [ms]						
n	Normal	Vec8	Vec16	OpenMP	Vec+OpenMP	CUDA
64	4	0	0	3	0	8
128	22	2	1	18	1	13
214	112	12	8	70	7	15
531	1.577	183	111	860	108	36
1.080	13.463	1.482	920	7.179	766	154
2.000	83.954	10.135	6.239	44.833	5.210	862
5.000	1.341.107	154.406	100.318	667.092	81.281	15.459
8.361	6.085.038	689.809	452.683	3.096.147	376.512	77.726

Análisis

- Cualquier algoritmo es más eficiente que el secuencial, especialmente en grafos/matrices grandes.
- Buena escalabilidad, al aumentar el “n”, los algoritmos paralelos siguen siendo más rápidos.
- Al paralelizar el bucle i y bucle j, la complejidad temporal del algoritmo CUDA es de:
  - $\frac{O(n^3)}{B^2}$
  - Con $\boldsymbol{B = n} \implies \frac{O(n^3)}{n^2} \Leftrightarrow \boldsymbol{O(n)}$
  - donde B es el tamaño del bloque y entonces, B^2 es el número de hebras ejecutándose en paralelo.

Demo

1. Ejecutar el programa con el grafo en un archivo.
2. Luego imprimirá los tiempos de ejecución de cada algoritmo.

```
./a.exe 4.mtx -p
Normal: 0 [ms]
CUDA: 0 [ms]
```

3. Si se agrega “-p” al ejecutar el programa, imprimirá la matriz original y resuelta. Después lee infinitamente 2 vértices para imprimir el camino más corto y su costo.

```
dist:
     1   2   3   4
   ---------------
1|   0 INF  -2 INF
2|   4   0   3 INF
3| INF INF   0   2
4| INF  -1 INF   0

dist:
     1   2   3   4
   ---------------
1|   0  -1  -2   0
2|   4   0   2   4
3|   5   1   0   2
4|   3  -1   1   0

Vertices: 1 2
Camino: 1 -> 3 -> 4 -> 2
Costo:    -2 +  2 + -1 = -1

Vertices: 3 1
Camino: 3 -> 4 -> 2 -> 1
Costo:     2 + -1 +  4 = 5
```

Planificación restante

- Implementar FW paralelo divido en bloques con OpenMP.
- Escribir informe.

Referencias

- "Floyd-Warshall Algorithm." Wikipedia, 2023, https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- "Parallel All-Pairs Shortest Path Algorithm." Wikipedia, 2023, https://en.wikipedia.org/wiki/Parallel_all-pairs_shortest_path_algorithm
- Nagavalli, Sasanka. "Using Hardware Parallelism for Faster Search via Dynamic Programming." Robotics Institute, Carnegie Mellon University, May 17, 2013. https://www.andrew.cmu.edu/user/snagaval/16-745/Project/16-745-Project-Report-SasankaNagavalli.pdf
- 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