Programación en GPUs con CUDA
Laboratorio 2 — Optimización de Memoria en CUDA
Transposición de matrices

1. Objetivos del laboratorio

En este laboratorio estudiaremos en profundidad cómo el patrón de acceso a memoria afecta al rendimiento en GPU.

El objetivo no es simplemente implementar un *transpose*, sino comprender:

- Cómo acceden los warps a memoria global.
- Qué significa realmente que un acceso sea **coalesced**.
- Cómo la **shared memory** puede reorganizar accesos.
- Qué son los **conflictos de bancos** y cómo evitarlos.
- Cómo evoluciona el rendimiento al modificar el patrón de acceso.

Este laboratorio está estructurado como una progresión de optimizaciones. Se pide demostrar la corrección de cada una de estas optimizaciones a nivel numérico, así como añadir el código necesario que permita temporizar las distintas fases de las implementaciones. Los resultados se reportarán en términos de ancho de banda (bytes por segundo procesados).

2. El problema: Transposición de una matriz

Dada una matriz A de tamaño $N \times N$, queremos calcular su transpuesta:

$$B(j, i) = A(i, j)$$

En memoria lineal (row-major), la posición (i, j) se almacena como:

```
A[i*n + j]
```

Por tanto, el transpose se implementa como:

```
out[j*n + i] = in[i*n + j]
```

Observaciones importantes:

- Se realizan N^2 lecturas.
- Se realizan N^2 escrituras.
- El número de operaciones aritméticas es mínimo.

Conclusión: estamos ante un problema **memory-bound**.
El rendimiento dependerá casi exclusivamente de cómo accedamos a memoria.

3. Conceptos fundamentales

3.1. Warps y coalescing

Un **warp** contiene 32 hilos que ejecutan en lockstep.
Cuando un warp accede a memoria global:

- Si los 32 hilos acceden a posiciones contiguas → 1 transacción grande.
- Si acceden con stride grande → múltiples transacciones.

Caso ideal (coalesced):

```
Thread 0 -> addr 0
Thread 1 -> addr 4
Thread 2 -> addr 8
...
Thread 31 -> addr 124
```

Caso con stride N:

```
Thread 0 -> addr 0
Thread 1 -> addr N*4
Thread 2 -> addr 2N*4
...
```

En este caso:

- Cada hilo cae en una línea de caché distinta.
- El hardware no puede agrupar accesos.
- El rendimiento cae drásticamente.

Pregunta conceptual:
¿Por qué el transpose es un ejemplo clásico donde uno de los accesos (lectura o escritura) tiende a ser no coalesced?

3.2. Shared Memory y Tiling

La memoria compartida:

- Es mucho más rápida que la global.
- Es visible solo dentro de un bloque.
- Permite reorganizar datos.

Idea clave:

1. Leer desde global de forma coalesced.
2. Reordenar en shared.
3. Escribir a global de forma coalesced.

Esto se implementa mediante **tiling**.

3.3. Bancos de memoria compartida

La shared memory está dividida en 32 bancos.
Cada acceso se asigna a un banco:

$$\text{bank} = \text{address} \mod 32$$

Si varios hilos de un warp acceden al mismo banco:

- Se produce un **bank conflict**.
- Los accesos se serializan.

Esto puede arruinar una optimización aparentemente correcta.

4. Ejercicio 1 — Versión v0 (Naive 1D)

4.1. Objetivo

Implementar un kernel donde:

- Se use grid 1D.
- Cada hilo procese una columna completa.
- Se utilice únicamente memoria global.

4.2. Esquema del kernel (a completar)

```
__global__ void transpose_v0(float* in, float* out, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < n)
    {
        for (int j = 0; j < n; j++)
        {
            // COMPLETAR
            // out[ ? ] = in[ ? ];
        }
    }
}
```

4.3. Preguntas

1. ¿Qué patrón de acceso tienen las lecturas?
2. ¿Qué patrón de acceso tienen las escrituras?
3. ¿Cuál es el cuello de botella?

5. Ejercicio 2 — Versión v1 (Grid 2D)

5.1. Objetivo

Modificar la implementación para:

- Usar grid 2D.
- Un hilo por elemento.

5.2. Esquema del kernel

```
__global__ void transpose_v1(float* in, float* out, int n)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (row < n && col < n)
    {
        // COMPLETAR
    }
}
```

5.3. Análisis

1. ¿Aumenta el paralelismo?
2. ¿Se arregla el problema de coalescing?
3. ¿Qué acceso sigue siendo no ideal?

6. Ejercicio 3 — Versión v2 (Tiling con Shared)

6.1. Objetivo

Implementar un transpose por bloques:

- TILE_DIM = 32
- BLOCK_ROWS = 8

6.2. Esquema del kernel

```
__shared__ float tile[TILE_DIM][TILE_DIM];

__global__ void transpose_v2(float* in, float* out, int n)
{
    int x = blockIdx.x * TILE_DIM + threadIdx.x;
    int y = blockIdx.y * TILE_DIM + threadIdx.y;

    // 1. Cargar tile desde global a shared
    // COMPLETAR

    __syncthreads();

    // 2. Escribir tile transpuesto a global
    // COMPLETAR
}
```

6.3. Reflexión

1. ¿Por qué ahora ambos accesos globales pueden ser coalesced?
2. ¿Dónde puede aparecer un problema oculto?

7. Ejercicio 4 — Versión v3 (Eliminación de Bank Conflicts)

Modificar la declaración:

```
__shared__ float tile[TILE_DIM][TILE_DIM+1];
```

7.1. Pregunta

¿Por qué añadir una columna extra puede mejorar el rendimiento si no cambia el número total de elementos procesados?

Explicar en términos de bancos.

8. Ejercicio 5 — Separación de transferencias

Modificar el código para medir por separado:

- Tiempo H2D.
- Tiempo kernel.
- Tiempo D2H.

8.1. Reflexión

1. ¿El tiempo total está dominado por el kernel o por las transferencias?
2. ¿Qué ocurre si el kernel se ejecuta múltiples veces sin copiar datos?

9. Análisis final

Para varios tamaños de matriz:

- Comparar rendimiento CPU vs GPU.
- Comparar v0, v1, v2 y v3.
- Analizar la evolución del rendimiento.

Discutir:

- Impacto del coalescing.
- Impacto del uso de shared.
- Impacto del padding.