Informe - Practico 3

Mateo Quiller, Leonardo Pombo, Joaquín Mezquita

Grupo 8

30 de noviembre de 2025

Índice

1. Ejercicio 1 - Memoria Global

1.1. Análisis del patrón de acceso a memoria global

En el kernel de transposición con bloques de 32×32 hilos, cada hilo realiza:

```
int id       = y * COLUMNS + x; // calculo del indice para lectura de d_matrix
int idTrans  = x * ROWS   + y;  // calculo del indice para escritura en
    transposed_matrix
transposed_matrix[id_transposed] = d_matrix[id]; // leo de d_matrix y escribo
    a transposed
```

- **Lecturas (`d_matrix[id]`):** Para un warp con hilos contiguos en el eje x, las direcciones
$$y \times \text{COLUMNS} + x$$
son estrictamente consecutivas en memoria. Luego como cada bloque lee un int y cada int ocupa 4 bytes, al ser las transacciones de 128 B coincide exactamente con 4 Bytes por hilo * 32 hilos = 128 B por warp. Por lo tanto cada warp se agrupa en una única transacción de 128 B, dando accesos *coalesced*.

- **Escrituras (`transposed_matrix[idTrans]`):** Cada hilo del mismo warp escribe en
$$x \times \text{ROWS} + y,$$
que para valores de x consecutivos no son direcciones contiguas, por lo que cada escritura genera su propia transacción a memoria global, resultando en 32 accesos *no–coalesced* por warp.

1.2. Modificación del tamaño de bloque para reducir accesos no–coalesced

Para reducir los accesos no–coalesced, analizamos el código y notamos que estos se dan en la escritura de la matriz transpuesta, por lo que decidimos cambiar el tamaño de bloque para que los warp contengan variaciones en los valores de threadIdx.y. Se probaron distintas combinaciones, como 16×32, 8×16, 16×16, 1×32 y viendo los tiempos de ejecución vimos que 1×32 logro un 100 % de escrituras coalesced, pero un % bajo de lecturas en consecuencia. Con el fin de mejorar las lecturas luego notamos que 2×32 obtenía un mejor rendimiento:

- Al usar bloques de 2×32, cada warp sigue conteniendo 32 hilos, pero solo hay un valor de `threadIdx.x` (0 o 1) y 32 de `threadIdx.y` (0,1...31).

- En las escrituras a `transposed_matrix` esto reduce las transacciones de memoria global de 32 (en 32×32) a solo 2 transacciones de 128 B (una por cada valor de `threadIdx.x`), minimizando el overhead de accesos no–coalesced.

- De esta forma, las lecturas mantienen algunos accesos contiguos, aprovechando únicamente el 25 % del ancho de banda disponible; en cambio, las escrituras se agrupan en transacciones completas (coalesced), utilizando asi el 100 % del ancho de banda.

Al ejecutar el kernel con ambos tamaños de bloque, el originl y matrices de 1024×1024 se midieron los siguientes tiempos:

Tamaño de bloque	Tiempo promedio (ns)	Desviación estándar (ns)
32×32	96.482,0	518,7
2×32	38.413,7	618,0
1×32	67.655,8	464,3

Lo que evidencia la ventaja de la configuración 2×32 al optimizar las escrituras coalesced por sobre las lecturas coalesced.

2. Ejercicio 2 - Memoria Compartida

2.1. Uso de memoria compartida

Para evitar los accesos no–coalesced en la escritura de la matriz transpuesta usando bloques de 32×32 se introdujo un *tile* en memoria compartida de tamaño igual al bloque. El kernel queda así:

Listing 1: Tile en shared sin padding

```
__shared__ int tile[BLOCK_SIZE_X * BLOCK_SIZE_Y];

int idIn = y * COLUMNS + x;
int idT  = BLOCK_SIZE_Y * threadIdx.y + threadIdx.x;
tile[idT] = d_matrix[idIn];

__syncthreads();

int idOut = y_transposed * ROWS + x_transposed;
int idTt = BLOCK_SIZE_X * threadIdx.x + threadIdx.y;
transposed_matrix[idOut] = tile[idTt];
```

A continuación se explica el propósito de cada línea del kernel:

- `__shared__ int tile[BLOCK_SIZE_X * BLOCK_SIZE_Y];` Reserva un bloque de memoria compartida de tamaño 32×32 para almacenar temporalmente los datos.
- `int idIn = y * COLUMNS + x;` Calcula el índice lineal de lectura en la matriz original en memoria global.
- `int idT = BLOCK_SIZE_Y * threadIdx.y + threadIdx.x;` Calcula el índice lineal dentro del *tile* donde este hilo escribirá.
- `tile[idT] = d_matrix[idIn];` Carga desde memoria global (matriz original) al *tile* en memoria compartida escribiendo el tile transpuesto.
- `__syncthreads();` Sincroniza todos los hilos del bloque para asegurar que el *tile* esté completamente poblado antes de leer.

- `int idOut = y_transposed * ROWS + x_transposed;` Calcula el índice lineal de escritura en la matriz transpuesta en memoria global.
- `int idTt = BLOCK_SIZE_X * threadIdx.x + threadIdx.y;` Calcula el índice lineal dentro del *tile* para leer el dato que ya deberia pertenecer al tile transpuesto.
- `transposed_matrix[idOut] = tile[idTt];` Escribe desde memoria compartida al arreglo transpuesto en memoria global de forma coalesced.

Comparación de desempeño

Versión	Tiempo (ns)	Desviación estándar (ns)
Global-only 32×32 (1A)	36.944,8	520,1
Shared 32×32 sin padding (2A)	69.979,2	367,1

Se observa que la versión 2A mejora respecto a la 1A al eliminar los accesos no–coalesced en memoria global, aunque se mantiene peor que 2B debido a conflictos de bancos en acceso a memoria compartida.

2.2. Conflictos de bancos y padding

En la versión 2A, los hilos acceden al *tile* con el mismo stride, provocando que todos los hilos de un warp soliciten datos del mismo banco en distintos ciclos (conflictos de banco). Para solucionarlo, se añade una columna extra al tile:

Listing 2: Tile con padding para evitar conflictos

```
__shared__ int tile[(BLOCK_SIZE_X + 1) * BLOCK_SIZE_Y];

int idIn = y * COLUMNS + x;
int idT  = threadIdx.y * (BLOCK_SIZE_X + 1) + threadIdx.x;
tile[idT] = d_matrix[idIn];
__syncthreads();

int idOut = y_transposed * ROWS + x_transposed;
int idTt = threadIdx.x * (BLOCK_SIZE_X + 1) + threadIdx.y;
transposed_matrix[idOut] = tile[idTt]; // salteamos el elemento DUMMY que
    solo sirve para evitar conflicto de banco
```

Patrón de accesos en shared

- Sin padding, cada acceso de hilos consecutivos al `tile` salta 32 `int` (128B), colisionando en el mismo banco.
- Con (BLOCK_SIZE_X+1) columnas, el stride entre elementos consecutivos deja de ser múltiplo de 128B, dispersando las direcciones en bancos distintos.

Comparación de desempeño respecto a 2.1 (matrices 1024x1024)

Versión	Tiempo (ns)	Desviación estándar (ns)
Sin Shared 2×32	36.944,8	520,1
Shared 32×32 sin padding	69.979,2	367,1
Shared 32×32 con padding	74.948,7	436,3

Comparación de desempeño respecto a 2.1 (matrices 4096x4096)

Versión	Tiempo (ns)	Desviación estándar (ns)
Sin Shared 2×32	604.678,0	1.599,4
Shared 32×32 sin padding	1.076.112,2	1.148,9
Shared 32×32 con padding	575.036,1	1.342,4

La reducción de conflictos de banco en la versión 2.2 se traduce en una menor latencia y por tanto, en un tiempo de ejecución inferior a la versión 2.1 siempre que la matriz sea lo suficientemente grande, para matrices chicas el overhead de escribir el padding en memoria compartida no supera la ganancia por evitar los conflictos de bancos ni las lecturas no coalesced.