# Informe - Laboratorio

Mateo Quiller, Leonardo Pombo, Joaquín Mezquita

Grupo 8

30 de noviembre de 2025

# Índice

**1. Introducción** **2**

**2. Diseño de la implementación** **2**
2.1. Estrategia general: . . . . . 2
2.2. Paralelización del algoritmo AlphaTensor usando matrices: . . . . . 2
2.3. Ejecución de algoritmo de Alphatensor en paralelo: . . . . . 3
2.4. Funciones clave: . . . . . 3
2.5. Sincronizaciones y accesos atómicos: . . . . . 3
2.6. Uso de memoria compartida: . . . . . 4

**3. Comparación con cuBLAS** **4**
3.1. Tablas comparativas . . . . . 4
3.2. Parámetros . . . . . 4

**4. Discusión de resultados** **5**
4.1. Impacto de las optimizaciones . . . . . 5
4.2. Comparación con cuBLAS . . . . . 5

**5. Conclusiones y mejoras a futuro** **5**
5.1. Conclusiones . . . . . 5
5.2. Trabajos futuros . . . . . 5

**6. Anexo** **7**
6.1. Análisis de conflictos de bancos . . . . . 7

# 1. Introducción

Como ya sabemos la multiplicación de matrices es una operación fundamental en numerosos campos como gráficos por computadora, aprendizaje automático y simulaciones científicas. Las GPUs ofrecen un alto grado de paralelismo que permite acelerar significativamente este tipo de cálculo. En este informe presentamos una implementación híbrida que combina el método convencional con el algoritmo descubierto por AlphaTensor de DeepMind para bloques de tamaño $4 \times 5$ por $5 \times 5$, con el objetivo de mejorar el rendimiento en matrices de gran tamaño. El objetivo del informe es explicar como fue implementado el algoritmo descubierto por AlphaTensor y como fue optimizado para su ejecución en GPU para resolver el producto de dos matrices A y B de dimensiones $dimA = 4n \times 5m$ y $dimB = 5m \times 5m$ .

# 2. Diseño de la implementación

## 2.1. Estrategia general:

La estrategia que seguimos es la que en la letra se presenta como alternativa **2**, dividimos la la matriz **A** en tiles de tamaño $4 \times 5$ y la matriz **B** en tiles de tamaño $5 \times 5$. Cada tile de salida $C$ (de tamaño $4 \times 5$) se calcula como la suma de productos de una fila de tiles de **A** por una columna de tiles de **B**. Para poder tener lecturas coalesced en **B**, primero la transponemos y en vez de realizar productos fila por columna hacemos fila por fila. A nivel de kernel, cada bloque de hilos procesa múltiples subproductos en paralelo: Esto lo hicimos cargando franjas (stripes) de **A** y **B** en memoria compartida y luego, aplicamos el algoritmo AlphaTensor para multiplicar tiles de $4 \times 5$ de A por $5 \times 5$ de B.

## 2.2. Paralelización del algoritmo AlphaTensor usando matrices:

En un principio, definimos matrices constantes para ayudarnos en la paralelización del algoritmo AlphaTensor con 2 enfoques similares. En principio se definieron 3 pares de matrices para a, b y c donde cada par definía por un lado el indice a leer de memoria y por otro el signo para indicar si ese valor se debía sumar o restar. Luego se optó por utilizar la versión provista en el código de ejemplo ya que resultaba más eficiente, pues evitamos multiplicar por -1 o 1 en cada calculo de h quedando así las siguientes matrices `hA_mas[76][6]`, `hA_menos[76][6]`, `hB_mas[76][7]` y `hB_menos[76][7]` tal y como fueron provistos por los profesores a excepción de hb_mas y hb_menos en los cuales modificamos para mejorar los accesos a memoria global. En el código provisto, se espera que las matrices **A** y **B** estén almacenadas linealizadas (es decir en un arreglo unidimensional) en formato row-major. Esto permite que al realizar la copia de memoria global a memoria compartida, las lecturas a la matriz **A** sean **coalesced**, pero ocurre todo lo contrario con la matriz **B**, ya que se debe leer por columnas, utilizando de cada lectura únicamente 4 bytes. Por este motivo, se decidió previamente transponer la matriz **B** para que posteriormente las lecturas a esta matriz también sean por filas y así puedan ser **coalesced**, por este motivo, las constantes hb_mas y hb_menos fueron adaptadas para trabajar con una matriz **B** transpuesta.

Estos datos, eran en principio cargados en **memoria constante** para calcular rápidamente los **Hs** para cada sub-bloque sin generar una divergencia tan grande como lo haría un **switch** o un **if** según el id de hilo, y finalmente luego de pruebas se optó por mover las matrices a memoria global y hacer uso de la cache **read-only** introducida a partir de la arquitectura **Kepler**. Estas lecturas se lograron gracias al uso de la función `__ldg`, ya que los accesos a memoria constante no eran uniformes y serial izaban perdiendo así el beneficio del broadcast y empeorando los tiempos en comparación con la memoria global cacheada la cual no tiene esta restricción. Para garantizar que las estructuras entren en esta cache optamos por definirlas como **int8_t** y de esta forma se ocupara el menor espacio posible.

## 2.3. Ejecución de algoritmo de Alphatensor en paralelo:

Para aprovechar mejor los recursos de la GPU y explotar los beneficios de las lecturas coalesced a la memoria global, decidimos que el procesamiento de los productos de matrices de 5x4 y 5x5 utilizando el algoritmo de Alphatensor se realicen en paralelo, para esto cada sub-bloque de la matriz **C** ejecuta un bloque CUDA de 456 hilos capaz de procesar 6 sub-bloques de **A** x 6 sub-bloques de **B** en paralelo, este número (456) es el resultado de calcular cuántos números de punto flotante pueden obtenerse en una lectura coalesced, considerando que un punto flotante ocupa 4 bytes y que una lectura a memoria global trae 128 bytes, tenemos que una sola lectura trae 32 posiciones. Cada sub-bloque es de 5 columnas, por lo tanto una sola lectura trae la primera fila de 6.4 sub-bloques, al tomar la parte entera de esto, tenemos que el valor deseado para aprovechar las coalescencia es de 6 sub-bloques, multiplicando esto por 76 que es la cantidad de hilos requeridas para aplicar el algoritmo de Alphatensor, obtenemos el numero 456.

## 2.4. Funciones clave:

- `calculo_sub_bloque`: Función `__device__` que recibe punteros a sub-bloques de $A$ y $B$, junto con `c_partial`. Usa memoria compartida `h_results[456]` (988 $= 76 \times 6$) para almacenar productos parciales. Cada hilo ($id < 76$) suma entradas de **A** usando `hA_mas` y `hA_menos` para obtener `parc_a`, y de **B** usando `hB_mas` y `hB_menos` para `parc_b`, almacenando luego `parc_a*parc_b`.Tras una barrera `__syncthreads()`, los primeros 20 hilos acumulan sus resultados en `c_partial[id]` mediante `atomicAdd`.
- `transponer_matriz`: Kernel `__global__` que reordena la matriz $B$ en memoria, intercambiando índices $(x, y)$ por $(y, x)$ para mejorar la coalescencia en lecturas posteriores.

## 2.5. Sincronizaciones y accesos atómicos:

Se utiliza `__syncthreads()` en `calculo_sub_bloque` tras llenar `h_results` para garantizar la consistencia antes de la reducción. Los acumulados intermedios se suman con `atomicAdd` en `c_partial`, y al concluir todas las iteraciones, otros `atomicAdd` escriben cada elemento de $C$ en `c_global` de forma segura.

### 2.6. Uso de memoria compartida:

- `h_results[456]`: buffer para 76 hilos y hasta 6 pares de sub-bloques de **A** y **B**.
- `c_partial[20]`: almacena los resultados parciales de cada bloque antes de escribir en global.
- `a_stripe[224]` y `b_stripe[224]`: buffers de sub-bloques de **A** y **B**, reduciendo accesos globales y mejorando localidad. Se utilizó un tamaño de 37 posiciones por subbloque para evitar conflictos de banco. El análisis realizado para llegar a este número se puede ver en el anexo 6.1

## 3. Comparación con cuBLAS

CuBLAS es la biblioteca de NVIDIA estándar utilizada entre otras cosas para la multiplicación de matrices en GPU. En esta sección se presenta la comparación del rendimiento de nuestra implementación cuyo diseño fue descrito anteriormente frente a cuBLAS en promedio de 10 ejecuciones.

A continuación adjuntamos una tabla comparativa con sus respectivos parámetros de prueba (recordamos que B es cuadrada de dimensiones colA x colA):

### 3.1. Tablas comparativas

| Dimensiones de matriz A | Tiempo CUBLAS | Tiempo de nuestra Imp. | $\|\Delta\|_2$ |
|---|---|---|---|
| $80 \times 100$ | 0.792ms | 0.211ms | 0 |
| $400 \times 500$ | 2.567ms | 30.29ms | 0 |
| $800 \times 1000$ | 1.163ms | 113.8ms | 0 |
| $2000 \times 2500$ | 5.282ms | 1751ms | 0 |
| $4000 \times 5000$ | 34.958ms | 13.91s | 0 |

### 3.2. Parámetros

- Error numérico: La norma $\|\Delta\|_2$ es el error $\|\mathbf{C}_\alpha - \mathbf{C}_{\text{cuBLAS}}\|_2$, que verifica la exactitud frente al resultado de cuBLAS fue calculado usando cublasSaxpy y luego cublasSnrm2.
- Matriz de pruebas: La matriz de pruebas utilizada para la medición de tiempos fue la matriz de 1 en todas las entradas para A y 5 en las entradas de B. Otra matriz de prueba que intentamos usar fue la matriz que para cada entrada (i,j) tenia el valor i pero presentaba un gran error numérico en cuBLAS para tamaños de matriz muy grandes debido a la precisión utilizada por los puntos flotante.

# 4. Discusión de resultados

## 4.1. Impacto de las optimizaciones

Nuestra implementación incorpora varias mejoras clave que contribuyen de manera significativa al aumento de rendimiento respecto a una versión nada optimizada:

- *Memoria compartida:* reduce la latencia de acceso a datos al almacenar sub-bloques en la jerarquía más rápida de la GPU, disminuyendo hasta un 60 % los accesos a memoria global.
- *Transposición de bloques:* garantiza accesos coalescentes al reorganizar datos en memoria compartida, evitando stalls por accesos dispersos.
- *Accesos coalescentes:* alinean lecturas y escrituras de hilos vecinos, maximizando el ancho de banda efectivo.

## 4.2. Comparación con cuBLAS

Si bien cuBLAS ofrece el mejor rendimiento absoluto gracias a el uso de kernels altamente tuneados por NVIDIA, presenta algunas limitaciones:

- **Overhead por conversión:** trabajar con datos en formato row-major requiere transformaciones previas.
- **Código no visible:** Al ser parte de una biblioteca privada su uso esta limitado por su interfaz lo cual limita el control fino sobre los kernels y el flujo de datos.

Nuestra implementación, al operar en formato row-major nativo, evita estos costes de conversión y ofrece una integración directa con aplicaciones que utilizan este estándar, logrando un rendimiento competitivo sin depender de bibliotecas externas.

# 5. Conclusiones y mejoras a futuro

## 5.1. Conclusiones

Nuestra implementación demuestra un rendimiento significativamente inferior a cuBLAS, sobretodo en matrices de gran tamaño. Mantiene un error numérico muy chico lo cual garantiza precisión computacional.

## 5.2. Trabajos futuros

Con el objetivo de seguir mejorando el rendimiento y la flexibilidad de nuestra implementación, se proponen las siguientes mejoras:

- **Reducción usando `shuffle`:** emplear primitivas de warp-level (por ejemplo, `__shfl_down_sync`) para acumular sub-resultados entre hilos de un warp, eliminando la necesidad de atomics adds y reduciendo la contención.

- **Reutilización de subproductos en memoria compartida:** almacenar resultados intermedios en el calculo del producto de sub bloques de 4x5 x 5x5, disminuyendo accesos redundantes a memoria constant y cálculos duplicados.
- **Exploración de co-procesadores:** adaptar kernels a Tensor Cores o utilizar memcpy asíncrono para solapar transferencia y cómputo.
- **Permitir multiplicar dos matrices de cualquier dimensión:** adaptar el kernels para que no sea necesario que las matrices de input cumplan $dimA = 4n \times 5m$ y $dimB = 5m \times 5m$.

# 6. Anexo

Listing 1: Kernel transponer matriz

```
__global__ void transponer_matriz(VALUE_TYPE *d_matrix, VALUE_TYPE
    *transposed_matrix)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < MATRIX_COLUMNS && y < MATRIX_COLUMNS)
    {
        int id = y * MATRIX_COLUMNS + x;
        int id_transposed = x * MATRIX_COLUMNS + y;
        transposed_matrix[id_transposed] = d_matrix[id];
    }
}
```

## 6.1. Análisis de conflictos de bancos

En la siguiente figura se puede apreciar el análisis realizado para entender cómo la memoria compartida era ocupada por los distintos sub-bloques procesados en paralelo. En las columnas de la derecha se representaron los 32 bancos de la memoria compartida, donde cada banco tiene además información sobre que hilo y warp lo está accediendo. El código de colores además se puede comparar con los bloques de la derecha, donde cada color representa un sub-bloque de los procesados en paralelo. En la columna warp, podemos observar como en ningún momento se producen 2 accesos al mismo banco dentro de un mismo warp. Esto se pudo realizar reservando 37 posiciones de memoria compartida para cada sub-bloque.

| Banco | Warp | Thread | Warp | Thread | Warp | Thread | Warp | Thread | Warp | Thread | Warp | Thread | Warp | Thread |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | -1 | -1 | -1 | -1 | 4 | 141 | 3 | 114 | 2 | 87 | 1 | 60 |
| 2 | 0 | 2 | -1 | -1 | -1 | -1 | 4 | 142 | 3 | 115 | 2 | 88 | 1 | 61 |
| 3 | 0 | 3 | -1 | -1 | -1 | -1 | 4 | 143 | 3 | 116 | 2 | 89 | 1 | 62 |
| 4 | 0 | 4 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 144 | 3 | 117 | 2 | 90 |
| 5 | 0 | 5 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 145 | 3 | 118 | 2 | 91 |
| 6 | 1 | 33 | 0 | 6 | -1 | -1 | -1 | -1 | 4 | 146 | 3 | 119 | 2 | 92 |
| 7 | 1 | 34 | 0 | 7 | -1 | -1 | -1 | -1 | 4 | 147 | 3 | 120 | 2 | 93 |
| 8 | 1 | 35 | 0 | 8 | -1 | -1 | -1 | -1 | 4 | 148 | 3 | 121 | 2 | 94 |
| 9 | 1 | 36 | 0 | 9 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 149 | 3 | 122 |
| 10 | 1 | 37 | 0 | 10 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 150 | 3 | 123 |
| 11 | 2 | 65 | 1 | 38 | 0 | 11 | -1 | -1 | -1 | -1 | 4 | 151 | 3 | 124 |
| 12 | 2 | 66 | 1 | 39 | 0 | 12 | -1 | -1 | -1 | -1 | 4 | 152 | 3 | 125 |
| 13 | 2 | 67 | 1 | 40 | 0 | 13 | -1 | -1 | -1 | -1 | 4 | 153 | 3 | 126 |
| 14 | 2 | 68 | 1 | 41 | 0 | 14 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 154 |
| 15 | 2 | 69 | 1 | 42 | 0 | 15 | -1 | -1 | -1 | -1 | -1 | -1 | 4 | 155 |
| 16 | 3 | 97 | 2 | 70 | 1 | 43 | 0 | 16 | -1 | -1 | -1 | -1 | 4 | 156 |
| 17 | 3 | 98 | 2 | 71 | 1 | 44 | 0 | 17 | -1 | -1 | -1 | -1 | 4 | 157 |
| 18 | 3 | 99 | 2 | 72 | 1 | 45 | 0 | 18 | -1 | -1 | -1 | -1 | 4 | 158 |
| 19 | 3 | 100 | 2 | 73 | 1 | 46 | 0 | 19 | -1 | -1 | -1 | -1 | -1 | -1 |
| 20 | 3 | 101 | 2 | 74 | 1 | 47 | 0 | 20 | -1 | -1 | -1 | -1 | -1 | -1 |
| 21 | 4 | 129 | 3 | 102 | 2 | 75 | 1 | 48 | 0 | 21 | -1 | -1 | -1 | -1 |
| 22 | 4 | 130 | 3 | 103 | 2 | 76 | 1 | 49 | 0 | 22 | -1 | -1 | -1 | -1 |
| 23 | 4 | 131 | 3 | 104 | 2 | 77 | 1 | 50 | 0 | 23 | -1 | -1 | -1 | -1 |
| 24 | 4 | 132 | 3 | 105 | 2 | 78 | 1 | 51 | 0 | 24 | -1 | -1 | -1 | -1 |
| 25 | 4 | 133 | 3 | 106 | 2 | 79 | 1 | 52 | 0 | 25 | -1 | -1 | -1 | -1 |
| 26 | -1 | -1 | 4 | 134 | 3 | 107 | 2 | 80 | 1 | 53 | 0 | 26 | -1 | -1 |
| 27 | -1 | -1 | 4 | 135 | 3 | 108 | 2 | 81 | 1 | 54 | 0 | 27 | -1 | -1 |
| 28 | -1 | -1 | 4 | 136 | 3 | 109 | 2 | 82 | 1 | 55 | 0 | 28 | -1 | -1 |
| 29 | -1 | -1 | 4 | 137 | 3 | 110 | 2 | 83 | 1 | 56 | 0 | 29 | -1 | -1 |
| 30 | -1 | -1 | 4 | 138 | 3 | 111 | 2 | 84 | 1 | 57 | 0 | 30 | -1 | -1 |
| 31 | -1 | -1 | -1 | -1 | 4 | 139 | 3 | 112 | 2 | 85 | 1 | 58 | -1 | -1 |
| 32 | -1 | -1 | -1 | -1 | 4 | 140 | 3 | 113 | 2 | 86 | 1 | 59 | -1 | -1 |

| | | | | | Subbloque |
|---|---|---|---|---|---|
| 1 | 33 | 65 | 97 | 129 | |
| 2 | 34 | 66 | 98 | 130 | |
| 3 | 35 | 67 | 99 | 131 | 1 |
| 4 | 36 | 68 | 100 | 132 | |
| 5 | 37 | 69 | 101 | 133 | |
| 6 | 38 | 70 | 102 | 134 | |
| 7 | 39 | 71 | 103 | 135 | |
| 8 | 40 | 72 | 104 | 136 | 2 |
| 9 | 41 | 73 | 105 | 137 | |
| 10 | 42 | 74 | 106 | 138 | |
| 11 | 43 | 75 | 107 | 139 | |
| 12 | 44 | 76 | 108 | 140 | |
| 13 | 45 | 77 | 109 | 141 | 3 |
| 14 | 46 | 78 | 110 | 142 | |
| 15 | 47 | 79 | 111 | 143 | |
| 16 | 48 | 80 | 112 | 144 | |
| 17 | 49 | 81 | 113 | 145 | |
| 18 | 50 | 82 | 114 | 146 | 4 |
| 19 | 51 | 83 | 115 | 147 | |
| 20 | 52 | 84 | 116 | 148 | |
| 21 | 53 | 85 | 117 | 149 | |
| 22 | 54 | 86 | 118 | 150 | |
| 23 | 55 | 87 | 119 | 151 | 5 |
| 24 | 56 | 88 | 120 | 152 | |
| 25 | 57 | 89 | 121 | 153 | |
| 26 | 58 | 90 | 122 | 154 | |
| 27 | 59 | 91 | 123 | 155 | |
| 28 | 60 | 92 | 124 | 156 | 6 |
| 29 | 61 | 93 | 125 | 157 | |
| 30 | 62 | 94 | 126 | 158 | |
| 31 | 63 | 95 | 127 | 159 | |
| 32 | 64 | 96 | 128 | 160 | |

Figura 1: Análisis conflicto de bancos