Programación masivamente paralela en procesadores gráficos (GPUs)

Informe - Práctico 3

Grupo 26

Nombre	**CI**
Agustín Matías Martínez Acuña	5.074.743-0
Natalie Valentina Alaniz Ferreira	5.209.018-4

Índice

1. Introducción

Este es el informe del Práctico 3 del Grupo 26 del curso de Programación Masivamente Paralela en Procesadores Gráficos (GPUs) de la Facultad de Ingeniería de la Universidad de la República. El mismo consta de tres ejercicios. El análisis se enfoca en comprender como las características de acceso a la memoria y la configuración de la grilla afectan el rendimiento de la GPU.

Un concepto central es el *acceso coalesced*, que busca optimizar el rendimiento al minimizar las transacciones de memoria necesarias agrupando los accesos a memoria de múltiples hilos dentro de un mismo *warp*.

Recordemos que *los accesos a memoria simultáneos de un warp son agrupados en la menor cantidad de transacciones de 32B necesarias para satisfacer todos los accesos.* Como en este contexto se utilizan integers, se corresponde a la lectura de 8 de ellos.

Se asumen que toadas las matrices manipuladas tienen dimensiones que son múltiplos del tamaño de bloque. Además para la medición del rendimiento, se promedian los tiempos de ejecución de diez corridas diferentes y reportando tanto la media como la desviación estándar. Estas mediciones se realizaron utilizando la herramienta `Nsight Systems`.

2. Ejercicio 1

El primer ejercicio de este laboratorio se centra en el desarrollo de un kernel para realizar la transposición de una matriz de enteros, alojada en memoria global de la GPU.

2.1. Descripción del Problema

El problema consiste en construir un kernel que reciba como entrada una matriz de enteros almacenada en la memoria global y devuelva su matriz transpuesta, también en memoria global. Para esto, se deben reservar dos espacios distintos de memoria: uno para la matriz original y otro para la matriz transpuesta.

El diseño de la grilla y de los bloques es bidimensional, lo que simplifica el acceso a los elementos de la matriz utilizando directamente las variables `threadIdx`, `blockIdx` y `blockDim`.

2.2. Análisis

Seleccionando un bloque de tamaño 32×32 tenemos que cada warp se corresponde a una fila , como se muestra en la figura 1. Esto implica que las lecturas de los datos de la matriz a trasponer se da de manera coalesced.

Se recuerda que el almacenamiento de la matriz en memoria es contiguo por filas, esto significa que los elementos de una misma fila se ubican consecutivamente en la memoria, permitiendo el acceso coalesced cuando los threads en un warp leen estos elementos.

Al escribir la fila transpuesta en la matriz resultado, el proceso se lleva a cabo por columnas, ya que se lee una fila i, y se escribe en la columna $j = i$ de la matriz transpuesta. Este método de escritura con bloques 32×32 no es óptimo para el acceso a memoria coalesced. En un acceso coalesced ideal, los datos que son accedidos por los hilos de un warp son adyacentes en memoria lo que maximiza la eficiencia del ancho de banda de memoria y minimiza el número de transacciones requeridas. En la ejecución, cuando se escribe una columna j en la matriz transpuesta M^t, el primer acceso de escritura ocurre en $M^t_{0,j}$. En este acceso, el warp correspondiente intenta escribir datos verticalmente en la columna, lo que no acompaña el patrón de acceso horizontal de las operaciones coalesced. Cuando un hilo del warp escribe un dato, se cargan 32 bytes de memoria, que corresponden a 8 enteros de 4 bytes cada uno. Sin embargo, aunque se cargan 32 bytes con cada acceso de escritura, el dato que cada hilo necesita es solo uno de esos ocho enteros cargados. Esto significa que de todo el bloque de datos cargados para el warp.

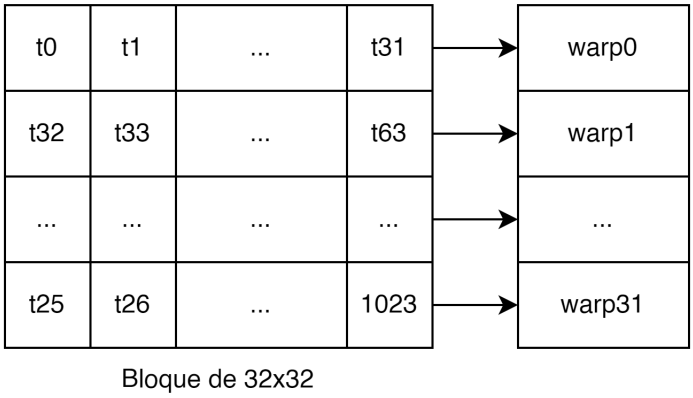

Figura 1: Bloque 32x32

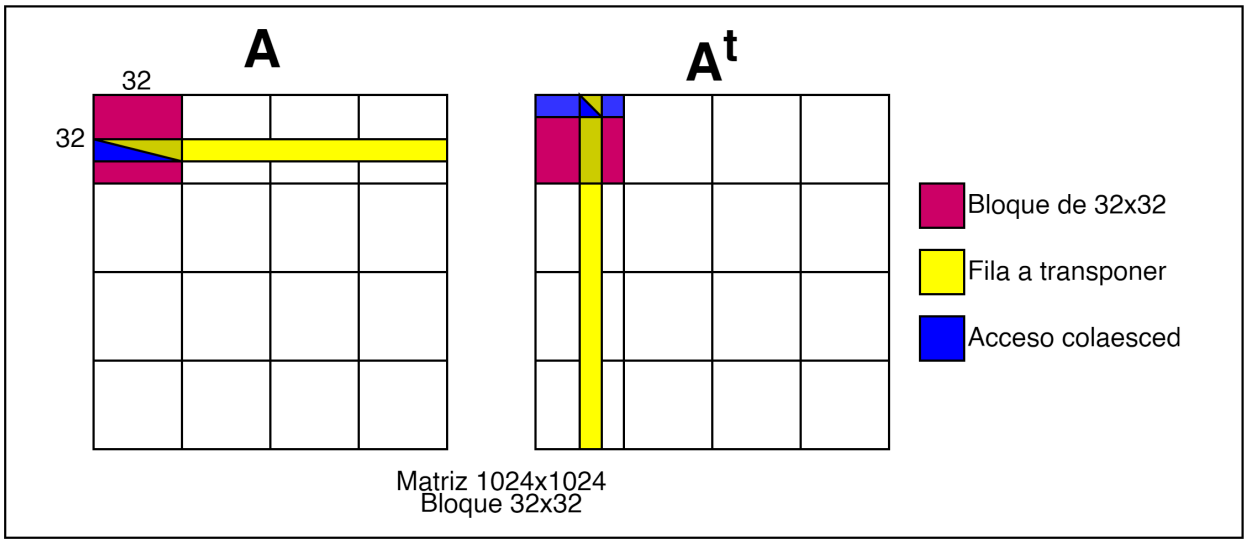

Figura 2: Diagrama que muestra la transposición de una fila en el contexto de un bloque de 32×32, se observa que en la lectura el acceso es full-coalesced, mientras que en la escritura se desaprovecha.

Esto resulta en un throughput efectivo de $\frac{4}{32} = \frac{1}{8}$ en escritura. La figura 2 ilustra este razonamiento, mostrando cómo la organización de los accesos en la transposición afecta la eficiencia del uso del ancho de banda de memoria.

Si consideramos solo acceso a memoria, tenemos de un total de un total de 64 bytes, se aprovechan $32 + 4 = 36$, dando un ratio de $\frac{36}{64} = 56\,\%$.

Este razonamiento aplica para el resto de los elementos de la columna a escribir, por lo que el acceso coalesced en la escritura se pierde en su totalidad para este tamaño de bloque.

En la siguiente sección, se buscará mejorar ese tipo de acceso experimentando con distintos tamaños de

bloques.

2.3. Favoreciendo el acceso coalesced en escritura

En la sección anterior se detalló como el acceso a memoria no se optimiza por la disposición de los datos durante la escritura transpuesta. A partir de ello se sugiere una propuesta de optimización mediante el uso de bloques de dimensión 32×1.

Una primer propuesta es utilizar bloques de dimensión 32×1, de esta manera habría una correspondencia $1 \longleftrightarrow 1$ entre bloque y warp. Y, en particular, cada warp se correspondería a una columna de la matriz. Por lo que el acceso coalesced **en escritura** sería óptimo.

Además, se plantea explorar el efecto de aumentar progresivamente el ancho de los bloques, comenzando desde una columna hasta alcanzar 32 columnas. Específicamente, se experimentará con dimensiones $32 \times Y \& 16 \times Y, (\forall Y \in \{1, 2, 4, 8, 16\})$

Considerando que el número de lecturas y escrituras es el mismo, y que generalmente se asume que las escrituras son más costosas que las lecturas en términos de rendimiento, se formula la siguiente hipótesis: *favorecer el acceso coalesced en escritura debería mejorar el tiempo de ejecución promedio para este caso partícular.*

2.4. Pruebas y resultados

Las pruebas se realizan en matrices cuadradas $n \times n$, con $n \in \{1024, 2048, 8192\}$. También se realiza con bloques de $16 \times blockY$, siguiendo la línea argumental planteada en la sección anterior.

A continuación se presentan únicamente con tamaños 8192 por simplicidad, los resultados para los otros tamaños fueron consistentes.

m	n	blockX	blockY	Avg (ns)	StdDev (ns)
8192	8192	32	1	5 069 815	10 710
			2	3 947 441	4899
			4	3 230 174	2367
			8	3 123 212	1913
			16	1 957 881	1134
			32	1 807 507	2001
		16	1	9 042 373	2515
			2	4 773 479	2940
			4	3 252 598	2116
			8	3 106 314	1921
			16	1 928 869	1557
			32	1 569 458	1824

2.5. Conclusiones

Contrario a lo esperado, los resultados obtenidos desmienten la hipótesis 2.3 no se cumple. La estrategia a emplear bloques con dimensiones $x \times y$ con $x > y$ (favoreciendo la escritura) no solo no ha resultado en una mejora del tiempo de ejecución, sino que se ha observado que empeora.

Esto sugiere que en este contexto específico optimizar agresivamente la escritura coalesced en detrimento de un acceso equilibrado entre la lectura y escritura puede no ser beneficioso. La observación apunta a que la gestión de memoria sea más equilibrada, considerando características de escritura y lectura para mejorar el rendimiento global del kernel.

3. Ejercicio 2

Se construye un kernel donde se realiza una correspondencia $1 \longleftarrow 1$ entre hilo y entrada de la matriz. De esta manera cada hilo identificará su entrada correpsondiente `(x,y)` y escribirá en esa posición de la matriz resultado de la operación `(M[x + y * n] + M[(x+4) + y * n]`.

Luego, se busca encontrar un tamaño de bloque adecuado para aprovechar accesos de memoria coalesced.

Tenemos que cada hilo realiza tres accesos a memoria:

- Leer el dato en la posición `(x,y)` de la matriz de entrada.
- Leer el dato `(x+4,y)` en la matriz de entrada.
- Escribir el resultado en la posición `(x,y)` de la matriz de salida.

Se desglosa la cantidad de transacciones por operación:

- Para el primer acceso `(x,y)` en la matriz de entrada, se deben leer 32 integers (128 bytes). Por lo que es necesario cuatro transacciones de 32 bytes para resolverlo.
- Para el acceso de las entradas `(x+4,y)`, se da el caso que los accesos de "más a la derecha"del warp, se dan fuera de los segmentos cuatro segmentos definidos en el item anterior, por lo que es necesario utilizar una transacción extra.
- Para la escritura en la matriz resultado en la posición `(x,y)`, el razonamiento es análogo a la lectura en `(x,y)`, por lo que se necesitan cuatro transacciones.

En total, se realizan $4 + 4 + 5 = 13$ transacciones por warp.

Una primera implementación del kernel que resuelve este problema puede encontrarse en `ej2.cu`.

El tiempo promedio de esta implementación es de 1.359.867 ns y una desviación estándar de 1.434 ns para bloques de tamaño

4. Ejercicio 3

Esta sección describe la implementación de un kernel que opera sobre una matriz A y un vector v para calcular el producto $x = Av$. Dadas las dimensiones de A, que son $dim(A) = 10240 \times 256$ (implicando una columna cada 40 filas). Sabiendo que $dim(v) = 256 \times 1$ se establecen las dimensiones de $dim(Av) = 10240 \times 1$. Esta información permite elegir tamaños de bloques particulares para el caso.

Recordando el producto matriz-vector, tenemos que si $x = Av$:

$$x_i = \sum_j (A_{i,j} * v_j)$$

.

En la implementación del kernel, cada hilo se asigna a una fila específica de A. Durante la ejecución cada hilo realiza la suma de los productos de los elementos de su fila asignada por los elementos correspondientes de v, y almacena el resultado en la posición correspondiente del vector resultado x como se muestra la figura 4, donde los espacios de memoria coloreados son los accedidos por el hilo `x`. Esto permite acumular localmente el resultado y luego asignarlo a la entrada del vector resultado correspondiente.

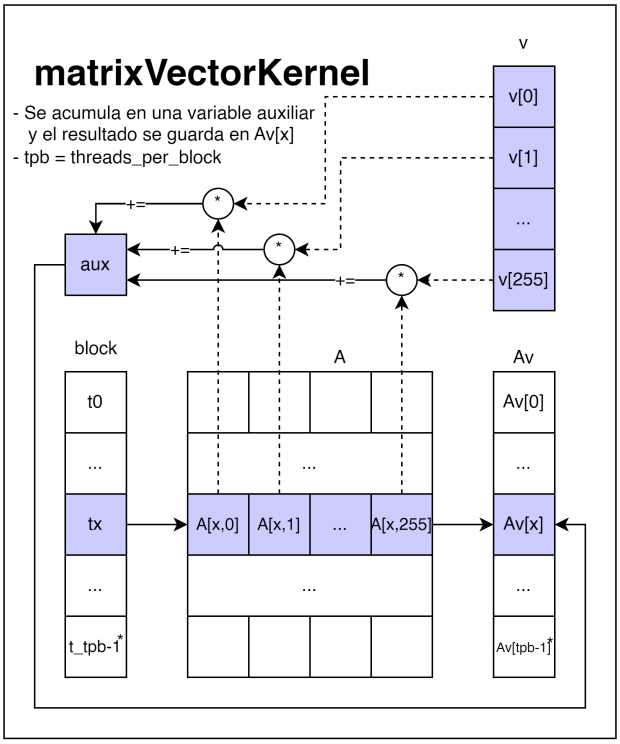

Figura 3: Diagrama de la multiplicación matriz-vector simple.

Por la naturaleza de la implementación, cada thread debe corresponder a una fila completa de la matriz, por lo que el tamaño del mismo será de 256 hilos por bloque.

Los resultados muestran un tiempo promedio 76.233 ns y una desviación estándar de 1.011 ns.

4.1. Propuesta de optimización

Como se mencionó en la sección 4, contar con el tamaño específico de la matriz permite aprovechar las características para avanzar

4.1.1. Experimento 1: Reduce

Para optimizar el cálculo del producto matriz-vector, se aplica una técnica de reduce, solucionando el problema en dos fases. En la primera fase, el primer kernel es responsable de realizar únicamente las multiplicaciones de los elementos de cada fila por la matriz A por el vector v. En este proceso no se incluye la suma de estos productos, resultando en una matriz intermedia que contiene los productos individuales. La ejecución de este kernel asigna a cada hilo una fila específica de A, donde cada hilo multiplica cada elemento de su fila asignada por el elemento correspondiente de v, aprovechando el acceso coalesced para optimizar el rendimiento de las operaciones en memoria.

El segundo kernel toma esa matriz intermedia y realiza la operación de reducción, sumando los elementos de cada fila para formar el vector resultado de x. Esta suma se efectúa utilizando un patrón de reducción de árbol, donde varios hilos efectúan la suma, almacenando el resultado en el vector final x.

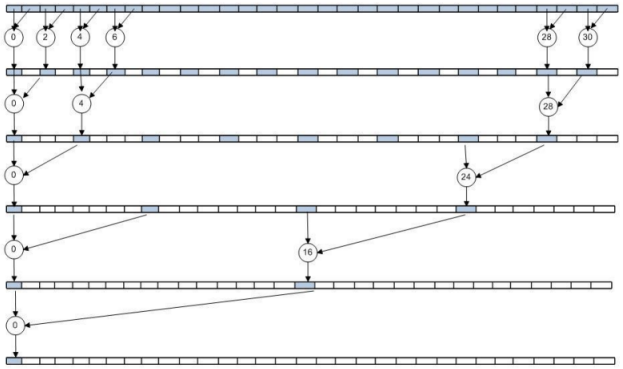

Figura 4: Diagrama de reducción.

Sin embargo, dadas las características de la matriz proporcionada (específicamente la proporción alta de filas con respecto a las columnas), el proceso de reducción de las 256 operaciones realizadas 10240 veces produce un overhead alto. El factor dominante en el rendimiento es el manejo de la cantidad de operaciones de reducción a pequeña escala repetidas muchas veces.

4.1.2. Experimento 2: Paralelización de la multiplicación utilizando memoria compartida

Esta aproximación adopta un enfoque más directo que en el experimento anterior mediante el uso de la memoria compartida. En este método, se reserva un arreglo en memoria compartida de tamaño 256 (correspondiente a la cantidad de columnas de la matriz A). Recordemos que la memoria compartida es accesible únicamente dentro del contexto del bloque al que pertenecen los hilos.

En esta configuración, cada bloque de la GPU se asigna a una fila completa de la matriz A, con el tamaño del bloque definido en 256 x 1. Esto significa que cada hilo dentro del bloque es responsable de procesar un elemento individual de esa fila, o sea, una columna de A.

Los resultados obtenidos muestran una mejora significativa en la eficiencia del acceso a los datos debido al uso de memoria compartida, que permite acceder rápidamente a los datos necesarios sin incurrir en la latencia asociada a la memoria global.

Un problema notable que surgió durante la implementación fue la suma de los resultados intermedios del arreglo local se centraliza en un único hilo, específicamente en el hilo cero de cada bloque. Esta decisión implica una serialización del proceso de suma, lo que reduce los beneficios del paralelismo inicial en la multiplicación. Aunque se realizan las 256 multiplicaciones en paralelo, luego las 256 sumas son realizadas de forma secuencial por un solo hilo.

El experimento culmina con un promedio de 71,935.8 ns con una desviación estándar de 705.5 ns, siendo apenas más rápido que el experimento base.

La serialización en la fase de suma podría explicar por qué a pesar de las ventajas iniciales de paralelización de la multiplicación, el rendimiento general del kernel no mejora tanto como se esperaba. La fase secuencial no es despreciable ya que cada uno de los 256 primeros hilos realiza 256 sumas. Siendo éste el cuello de botella del experimento.

4.1.3. Experimento 3: Uso de atomicAdd para escritura directa en memoria compartida

Dado que el experimento en 4.1.2 no mostró una mejora sustantiva del tiempo promedio de ejecución. En esta segunda versión del experimento, se eliminó el uso de un arreglo de memoria compartida para la suma, y en su lugar se implementó un único entero que funciona como acumulador compartido entre los hilos del bloque.

En esta nueva configuración, se evita el cuello de botella de la serialización al permitir que cada hilo contribuya al acumulador de manera concurrente. Esto se logra mediante el uso de la operación atomicAdd, que garantiza que las actualizaciones al acumulador compartido se realicen de manera atómica, evitando así las condiciones de carrera que podrían surgir de accesos concurrentes a la memoria compartida. A través de esta técnica, cada hilo suma directamente el producto de su operación al acumulador, lo que minimiza los retrasos asociados con la espera por la disponibilidad de recursos de memoria.

Por último, se realiza una sincronización de los hilos para que el `thread(0,0)` pase el resultado del acumulador a la entrada correspondiente de la matriz

Se observa una mejora en el rendimiento de entre un 13 % y un 19 % (66.320 ns promedio, 834.5 ns de desviación estándar) en pruebas realizadas con distintos tamaños de matriz, específicamente para dimensiones de 1024, 2048 y 8192. Este aumento en el rendimiento sugiere que la eliminación del segmento serial y la introducción de operaciones atómicas para gestionar la acumulación han optimizado el proceso de reducción necesario para completar el producto matriz-vector.