Programación masivamente paralela en procesadores gráficos (GPUs)

Informe - Práctico 4

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 4 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 dos ejercicios.

2. Ejercicio 1

El primer ejercicio de este laboratorio se centra en uso de memoria compartida de la GPU (shared memory). Se retoma el problema de la transposición de matrices, agregando el uso y acceso a memoria compartida cuando los accesos no sean coalesced.

La primer parte consiste en construir un kernel que reciba como entrada una matriz de enteros almacenada en la memoria global y devuelva su matriz transpuesta. Esto coincide con el ejercicio 1 del Práctico 3, pero en esta consigna se debe reservar un espacio en memoria compartida de tamaño igual al del tile transpuesto y realizar la transposición directamente en el tile, para luego escribir en la matriz transpuesta leyendo desde el tile en memoria compartida.

2.1. Propuesta de solución

El punto de la implementación es realizar en memoria compartida aquellas lecturas/escrituras que serían no coalesced en sus respectivas matrices en memoria global. Se describe el algoritmo a continuación:

En cada bloque se lee la matriz `d_m` de manera coalesced, guardando el dato obtenido en la posición transpuesta de `tile`. Una vez finalizada esa parte, se debe escribir el bloque transupesto en `d_MTrans` de manera coalesced, es decir que threads contiguos deben escribir en espacios de memoria contigua.

A esto se le agrega el desafío de que la dimensión del tramo a escribir es inverso a las dimensiones del bloque. Es decir que si el tamaño de bloque utilizado es `(x,y)` el bloque a escribir tendrá dimensiones `(y,x)`. Para lograr escritura coalesced en este contexto es necesario escribir por filas de la matriz transpuesta.

Para lograrlo, utilizamos la variable `num_pos` que identifica a un thread por el lugar que ocupa en el bloque si este se ve como unidimensional (ver Figura 1). Utilizando `num_pos` se fuerza una recorrida coalesced del bloque transpuesto en la matriz transpuesta.

Entonces, para que la escritura sea coalesced, se impone la condición de que la cantidad de filas del bloque debe ser múltiplo de cuatro (`blockDimX.y mod 4 = 0`).

La implementación completa puede encontrarse en el archivo `ej1.cu`.

Figura 1: Ejemplo de identificación de hilos en un bloque 4×3

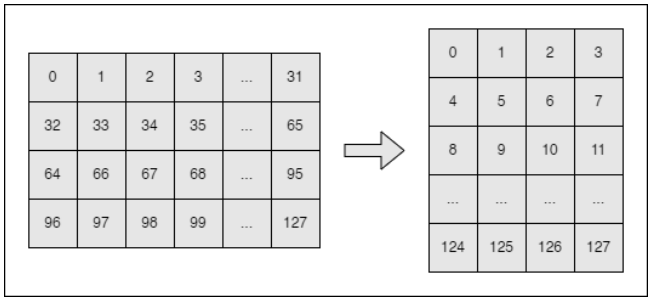

Figura 2: Ejemplo de transposición de un bloque de 32×4, el número en cada cuadarado indica el `num_pos` del thread asignado a realizar la transposición del elemento en dicha casilla, teniendo en cuenta esta estrategia y las restricciones definidas se logra la escritura totalmente coalesced.

2.2. Resultados y análisis

La Tabla 1 muestra el tiempo de ejecución primero para matrices de tamaño $n \times n$ (Para comparar, se pueden ver los resultados del práctico 3 en 4.1)

n	Tiempo (ns)
512	11.385
1024	40.403
2048	149.257
4096	586.585
8192	2.323.947

Cuadro 1: Tabla de datos de transposeKernel utilizando tile en shared (bloques de 32×32).

Los resultados muestran que el tiempo de ejecución utilizando un tile en memoria compartida es, en promedio, alrededor de 516.440 ns más lento que la transposción del Práctico 3 (caso 8192×8192, bloque 32×32), siendo un 28.5 % más lento.

Una posible explicación puede ser los conflictos de bancos que se generan en la ejecución del nuevo kernel: un conflicto de banco ocurre cuando dos o más hilos en un warp intentan acceder a diferentes direcciones dentro del mismo banco de memoria al mismo tiempo. Cuando esto ocurre, las accesos se serializan, reduciendo la performance.

En CUDA, normalmente se utilizan 32 bancos. El criterio para seleccionar el banco de memoria donde debe guardarse una `direccion` es `direccion mod 32`.

Teniendo esto en cuenta observamos que un conflicto de banco se genera si dos o más threads acceden a direcciones d_1 y d_2 tal que $d_1 \equiv d_2 (mod\ 32)$.

Esto sucede por la condición de la letra que fuerza que la cantidad de columnas del bloque sea 32, utilizar otro tamaño de bloque podría ser razonable para evitar los conflictos, sin embargo veremos otra alternativa en la siguiente sección.

La figura 3 muestra el tile con su dirección relativa (ver descripción de la figura)

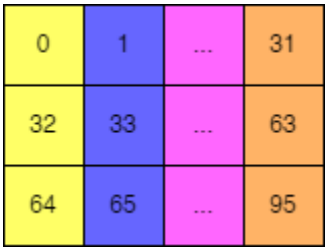

Figura 3: Tile de 32×3 con su índice relativo, se observa que la coincidencia en color implica la existencia de conflictos de bancos al acceder a esas direcciones al mismo tiempo.

2.3. Agregando una columna dummy en el tile

Considerando el análisis de la parte anterior, exploramos la idea de agregar una columna dummy para shiftear los datos cargados en el tile y por ende el conflicto de banco, tal como muestra la figura 4. Es importante destacar que **en la columna dummy no habrían datos**.

Figura 4: Tile con columna dummy (33×3), la coincidencia de color implica conflicto de bancos al acceder al mismo tiempo, se observa que se dispersa la posibilidad de generar el conflicto.

Los resultados se encuentran en la Tabla 2 con bloques de 32×32. Se observa una mejora sustantiva del tiempo de ejecución respecto a no utilizar una columna dummy, esto indica que efectivamente se resolvieron los conflictos de bancos analizados anteriormente. Sin embargo, los tiempos quedan a la par con el Kernel del práctico 3.

n	Tiempo (ns)
512	9.129
1024	32.569
2048	120.895
4096	471.542
8192	1.871.384

Cuadro 2: Tabla de datos de transposeKernelDummy utilizando tile en shared (bloques de 32×32).

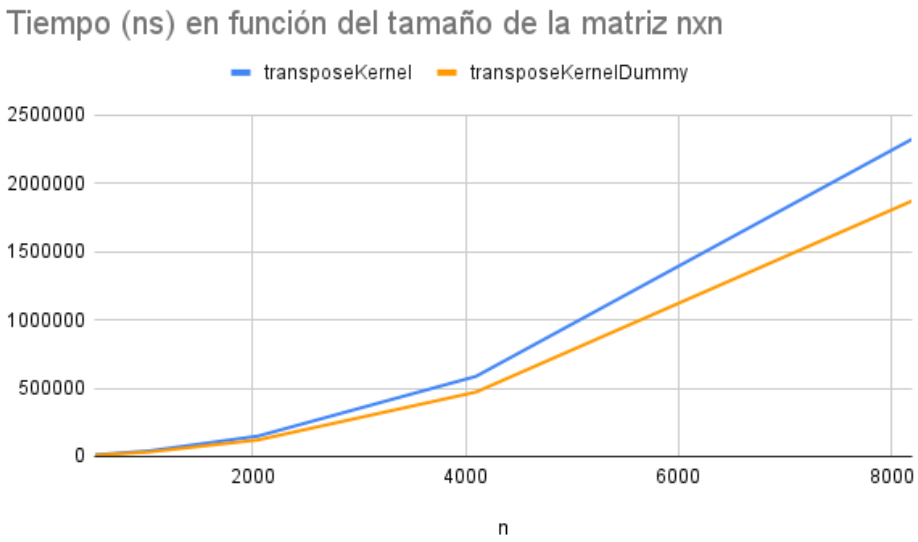

Figura 5: Enter Caption

3. Ejercicio 3

El objetivo del ejercicio es utilizar la memoria compartida de la GPU para la privatización, implementando técnicas de reducción.

Se estará trabajando con una matriz de 3840 x 2160 que contiene números entre 0 y 255, el objetivo del ejercicio es generar un histograma de la matriz, donde cada entrada indique la cantidad de veces que aparece el número correspondiente a su posición.

En la primera parte, se propone que cada bloque de threads mantenga un histograma local en la memoria compartida (de tamaño 256). Al finalizar el recorrido de la matriz, asegurándonos de que todos los kernels se sincronicen, los datos del histograma local se suman en un histograma global utilizando atomicAdd.

El tamaño de bloque elegido para esta parte es de (256, 4). Esta selección se debe a que al momento de impactar los histogramas locales en el histograma global, nos interesa que no queden hilos ociosos en el bloque, por lo que el tamaño propuesto de bloque en la dimensión X es del largo del histograma, es decir, 256.

Esta variante tiene varias desventajas. Utiliza atomicAdd tanto en la suma en la memoria compartida, como en memoria global, agregando un costo en memoria compartida, y uno mucho mayor en global. Las operaciones atómicas en memoria global tienen un costo mayor ya que la latencia de acceso a la memoria global es mucho mayor que a la memoria compartida, además en la memoria compartida la cantidad de hilos retenidos es menor que en memoria global.

En la segunda parte se desarrolla una variante donde se ejecuten varios kernel. En el primer kernel cada bloque almacena su histograma local en una fila de una matriz de histogramas. Para este primer kernel el tamaño de bloque utilizado es el mismo que optimiza la parte anterior por tener comportamientos muy similares. Posteriormente, otro kernel realiza la reducción de estos histogramas para obtener un histograma global.

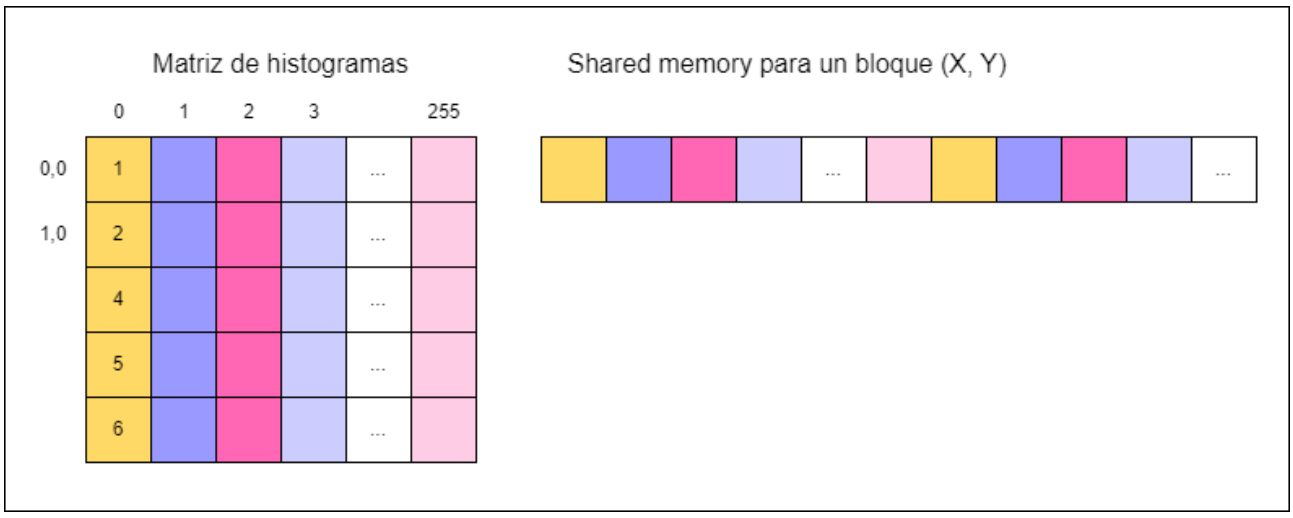

Figura 6: Matriz de histogramas

El kernel encargado de la reducción realiza su tarea en pasos iterativos, utilizando un parámetro de salto dentro de cada bloque. Inicialmente, cada bloque realiza una reducción en la que sólo se utilizan la mitad de los hilos disponibles. Cada hilo suma el valor de la posición actual en la matriz al valor de la matriz ubicado $i = \frac{cant\_hilos\_bloque}{2}$ entradas más adelante, correspondiendo esto a sumar la primera y la segunda mitad del bloque. En la siguiente iteración, el valor de i se reduce a la mitad y sólo participan la mitad de los hilos de la iteración anterior. Este proceso se repite, reduciendo la cantidad de hilos activos, hasta que se forma un histograma local en la primera fila de cada bloque.

Luego, se realiza otra reducción. En esta parte, las únicas filas válidas son las primeras filas de cada bloque. Para ejecutar esta reducción utilizando el mismo kernel, se simula un escenario análogo al de la fase anterior. En lugar de sumar filas consecutivas, las sumas se realizan entre filas separadas por un salto blockDim.y. Esto se logra ajustando el cálculo de la posición para simular filas contiguas. Al terminar con este proceso, sólo una fila cada blockDim.y bloques resulta válida.

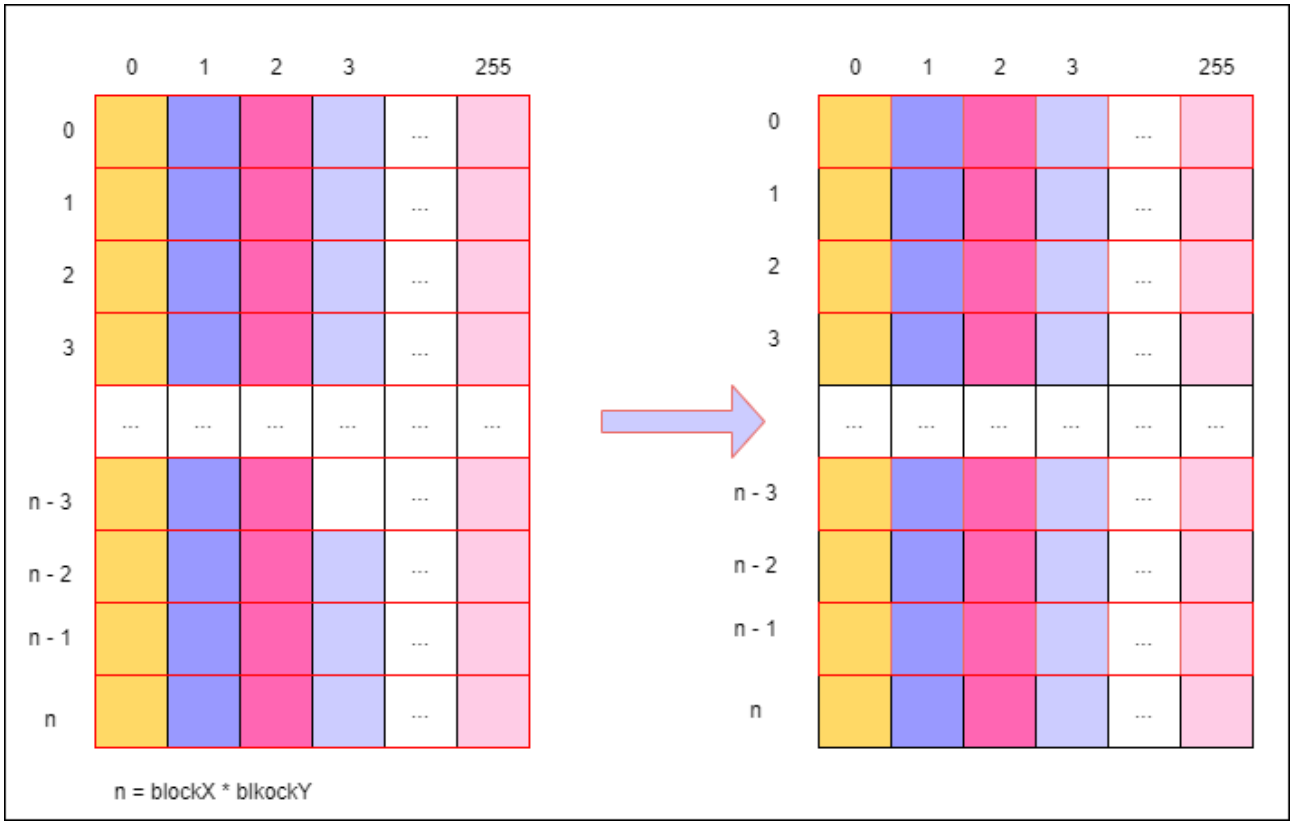

Figura 7: Matriz de histogramas con bloque de tamaño (256, 2): en la primera reducción todas las filas son tomadas en cuenta para operar, en la siguiente reducción es necesario operar sólo con las filas par. Salto inicial de 1, siguiente salto de blockDim.y = 2... Y así sucesivamente hasta que quede una sola fila valida.

En las siguientes iteraciones, el intervalo de salto aumenta de manera exponencial, siendo blockDim.y elevado a la potencia del numero de iteración, hasta que finalmente converge a una única fila válida.

Para asegurarnos de que la cantidad de filas de la matriz es divisible por la dimensión y (blockDim.y) de los bloques, es necesario añadir filas dummy (rellenas con ceros) al final de la matriz. La cantidad de filas agregadas depende directamente del tamaño del bloque, ya que las dimensiones de la matriz son fijas.

Para mantener el acceso a memoria coalesed, y así optimizar el rendimiento, la dimensión de los bloques en X debe ser de al menos 4 enteros. En cuanto a la dimensión Y , es posible maximizar su tamaño hasta un límite de 256 ya que el número máximo de hilos por bloque es de 1024. Aunque inicialmente esto podría parecer beneficioso debido al menor número de llamadas al kernel de reducción, no sucede de esa manera.

El problema surge debido a la implementación del kernel de reducción, donde se ejecuta un bulcle while. En cada iteración del while, se realiza una sincronización de hilos, lo que implica que para bloques más altos, el bucle se ejecutará más veces y, por lo tanto, habrá más sincronizaciones. Es importante elegir una altura de bloque que maximice la eficiencia de la reducción sin introducir un exceso de sincronizaciones de hilos.

Se elige usar bloques de dimensiones (128,8) porque ofrece un equilibro entre el aprovechamiento de bloques medianamente altos como para no hacer tantas llamadas al kernel, pero no lo suficientemente altos como para introducir demasiadas sincronizaciones de hilos. Optimizando así el rendimiento general.

Si bien el análisis indica que el rendimiento de la segunda versión debería ser mejor que el de la primera, esto no sucede. Aunque lo que empeora no es mucho (un 11 %) se ve un rendimiento peor en el programa que debería funcionar mejor, esto puede deberse a que el tamaño de la matriz no es suficiente como para mostrar estas diferencias.

La desviación estándar del kernel de reducción es relativamente alta, ya que cada llamada al kernel de reducción disminuye sustancialmente su tiempo en comparación a la llamada anterior. A continuación, se adjunta una tabla que muestra algunos tamaños de bloque (tamaño de bloque del decrypt kernel fijo en (256, 4)) y sus tiempos de ejecución:

Total Time (ns)	**Instances**	**Avg (ns)**	**StdDev (ns)**	**Block Size**	**Kernel**
214,112	1	214,112.0	0.0	(128, 8)	decrypt_kernel_ej3B
177,726	5	35,545.2	39,955.8	(128, 8)	reduction
236,672	2	118,336.0	5,430.6	(8, 128)	reduction
218,560	1	218,560.0	0.0	(8, 128)	decrypt_kernel_ej3B
215,583	1	215,583.0	0.0	(32, 32)	decrypt_kernel_ej3B
202,272	3	67,424.0	52,896.5	(32, 32)	reduction
218,687	1	218,687.0	0.0	(128, 4)	decrypt_kernel_ej3B
181,792	5	36,358.4	40,619.1	(128, 4)	reduction

Cuadro 3: Performance Metrics

Una posible mejora en la implementación actual del proceso de reducción podría ser ajustando el kernel que realiza la reducción. En lugar de invocar el kernel con bloques que ocupan toda la matriz y terminar utilizando sólo una fracción de ellos, se propone un uso adaptativo. La idea es ajustar dinámicamente el número de bloques utilizados en cada llamada al kernel de reducción a medida que se progresa en las etapas de reducción. Inicialmente se pueden utilizar más bloques para cubrir toda la matriz, pero a medida que las filas válidas se reducen se debería disminuir el número de bloques activos en proporción al número de filas válidas.

4. Anexo: Consideraciones sobre el Ejercicio 1A del Práctico 3

Para tener una comparación del rendimiento de la transposición de matrices se utiliza el código realizado para el práctico 3, en el proceso de EVA.

Durante la realización de esta tarea, notamos que utilizamos la notación de matrices `(fila, columna)` cuando delimitamos las dimensiones de los bloques. Esto lleva a que, cuando se suponía que debíamos generar bloques de 32 filas × 16 columnas, en realidad estábamos haciendo lo contrario (16 filas × 32 columnas). La Figura 8 ilustra la notación CUDA

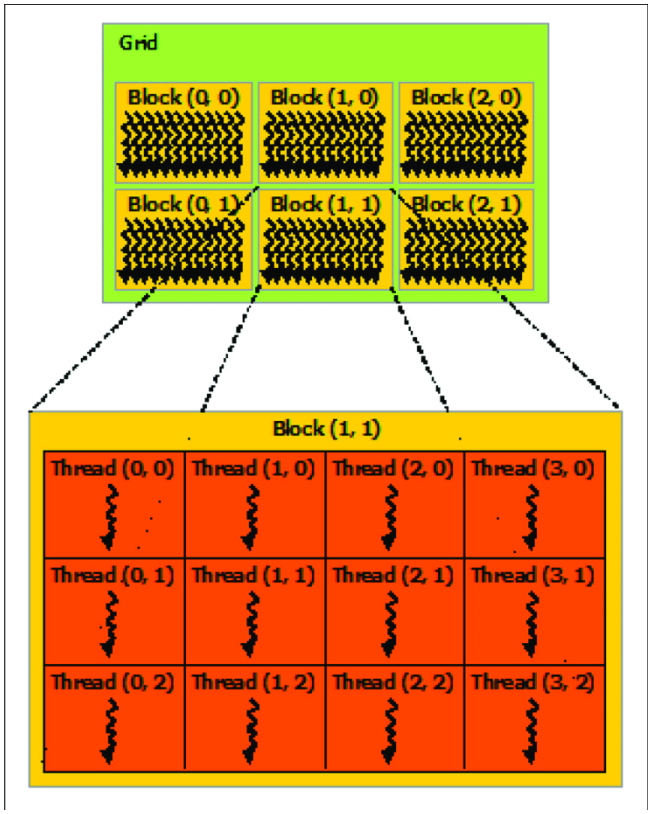

Figura 8: Esquema Grill | Bloques | Threads en Cuda

Para solventar el error y unificar criterios se realizan las siguientes acciones:

- Se pasa de usar la notación de CUDA `(X, Y)` en lugar de la notación de matrices `(fila, columna)`.
- Se corrige el código para que respete la notación.

4.1. Resultados

matrixX	matrixY	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