Práctico 4

Nicolás Alejandro Núñez Hosco
Facundo Diaz
Grupo 17
Profesores: Martín Pedemonte, Pablo Ezzatti y Ernesto Dufrechou

GPU

Junio 2024

Índice

1. Introducción

El trabajo se divide en dos ejercicios. En el primer ejercicio, se aborda la transposición de matrices utilizando memoria compartida para mejorar el acceso coaleced a la memoria global y evitar escrituras no coalesed. Se implementan varias optimizaciones, incluyendo la adición de una columna dummy para evitar conflictos de banco, lo que mejora significativamente el rendimiento en comparación con el kernel básico.

El segundo ejercicio se enfoca en el cálculo de histogramas. Se implementan tres variantes de kernels: un kernel adaptado del práctico 2, un kernel que utiliza operaciones atómicas (atomicAdd) en memoria compartida, y un kernel que realiza una reducción global sobre una matriz de histogramas locales. Cada variante se evalúa en términos de tiempo promedio de ejecución y desviación estándar, demostrando mejoras sustanciales en rendimiento y consistencia.

2. Ejercicio 1

Se implementa una función de transposición que hace uso de la memoria compartida para evitar la escritura no coalesed. Cada bloque del kernel comienza por leer de forma coalesed su sub-matriz correspondiente, con el objetivo de almacenar a la misma en memoria compartida (al escribir en memoria compartida se hace de forma que la matriz resultante en shared sea la transposición de la sub-matriz original). Luego se sincronizan los hilos, asegurando así de que todo el bloque haya finalizado la escritura antes de trasladar la sección transpuesta nuevamente a memoria global de forma coalesed.

En el práctico anterior, el mejor resultado obtenido fue

Filas	**Columnas**	**Tamaño de bloque**	Promedio	Desviación
1024	1024	32x32	29.065,5ns	263,9ns

Luego de imeplementar el codigo, para el mismo tamaño de matriz obtenemos

Filas	**Columnas**	**Tamaño de bloque**	Promedio	Desviación
1024	1024	32x32	36.287,5ns	716,9ns

Se observa que el rendimiento obtenido es inferior al del kernel que no utiliza memoria compartida, esto es consecuencia de los conflictos de bancos que tienen lugar en el nuevo algoritmo al escribir en memoria compartida.

Para este algoritmo, se utilizan bloques con blockDim.x = 32 y, como palabras de 32 bits (4 bytes) contiguas en memoria compartida están en bancos contiguos, cada 32 elementos dentro del tile se repetirá el mismo banco de memoria (esto implica que elementos almacenados en la misma columna de una matriz de 32 elementos por fila corresponderán al mismo banco de memoria), por lo tanto en el momento de copiar el tile a memoria compartida todos los hilos de un mismo warp escribirán dentro del mismo banco (esto es consecuencia de que cada warp transpone un segmento de fila, por lo que debe escribir un segmento de columna de la transpuesta), por lo que las escrituras en memoria compartida de cada warp se serializarán, requiriendo 32 ciclos en lugar de 1. Se representa este fenómeno con el siguiente diagrama.

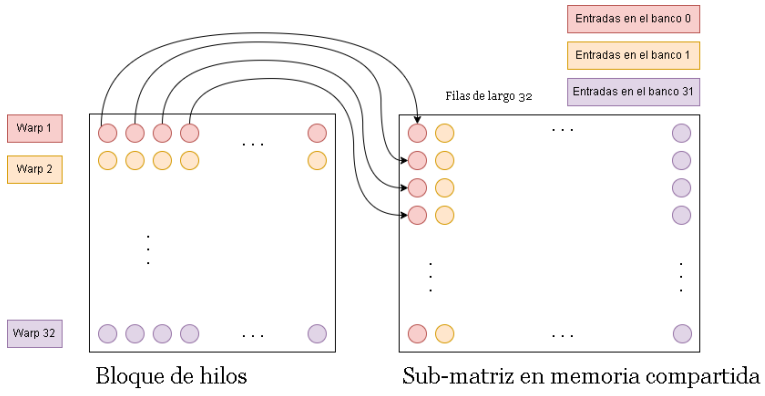

Figura 1: Ejemplo del patrón de escritura en memoria compartida

Al agregar una columna 'dummy' a la matriz de memoria compartida, se agrega un desplazamiento en el banco de memoria sobre el cual cada hilo de un mismo warp escribe, por ejemplo, el hilo con $id_x = 0$ e $id_y = 0$ escribirá en el banco 0, y el hilo con $id_x = 1$ e $id_y = 0$ escribirá en el banco 1 (en lugar del banco 0 si no fuese agregada la columna dummy), con lo que se consigue evitar los conflictos de banco. Se representa este cambio con el siguiente diagrama.

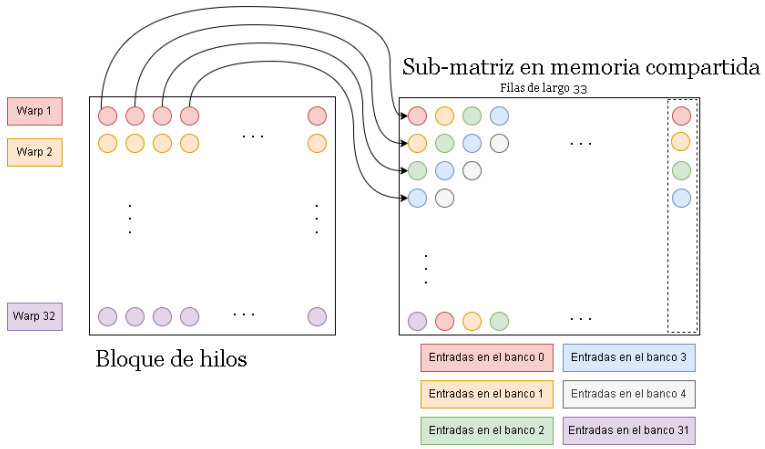

Figura 2: Ejemplo del patrón de escritura en memoria compartida al agregar una columna dummy (indicada por la linea puntuada)

Con este algoritmo se obtienen los siguientes resultados:

Filas	**Columnas**	**Tamaño de bloque**	Promedio	Desviación
1024	1024	32x32	26.992 ns	421,6 ns

Se observa que esta implementación consigue el mejor tiempo entre todos los kernels implementados, puesto que todos los acceso en memoria global son coalesed, y se evitan los conflictos de banco.

3. Ejercicio 2

En primer lugar se adapta el kernel implementado en el ejercicio 2 del práctico 2, este kernel sera referenciado como KernelAdaptado. Esta implementación tiene dos limitaciones. En primer lugar, no hace uso completo de las capacidades de paralelización disponibles y, por otro lado, no utiliza memoria compartida, por lo que todas las actualizaciones del histograma impactan directamente en memoria global, haciendo necesario el uso de atomicAdd, y causando que muchas escrituras sean serializadas.

Luego, se desarrolla una variante del kernel en el cual se divide la matriz en bloques de 16×32, se asigna un hilo por cada entrada de cada bloque, y se crea un array de 256 entradas en memoria compartida. Cada hilo lee su posición correspondiente y suma 1 en la entrada del valor leído en la memoria compartida, usando —atomicAdd— para evitar condiciones de

carrera. Antes de finalizar el kernel, el thread con `threadIdx.x = threadIdx.y = 0` impacta su copia del histograma local en el histograma global, nuevamente utilizando `AtomicAdd`.

Por último, se desarrolla otra implementación en la cual cada bloque, luego de computar su histograma local, lo adjunta dentro de una matriz global. Para realizar el calculo del vector final de ocurrencias se asigna la reducción de cada columna a distintos bloques de la grid. Como cada tamaño de columna supera el limite posible para aplicar reduce en una sola ejecución del kernel, se guardan los resultados parciales dentro de la propia matriz, para luego ser ejecutado nuevamente, y así obtener el resultado final.

A continuación, se muestran los resultados obtenidos para cada implementación.

Kernel	**Tamaño de bloque**	**Promedio**	**Desviación**
Kernel adaptado	16x32	7.101.117,8ns	181.114,1
Kernel con atomicAdd	16x32	2.089.260,2ns	53.683,0ns
Kernel con reduce	16x32	426.635,6ns	124.611.9ns

Nota: El tiempo promedio del kernel con reduce se calcula sumando el tiempo promedio del kernel que crea la matriz de histogramas y el kernel que aplica el reduce sobre esta, puesto que ambos kernels se ejecutan secuencialmente (una ejecución de count_kernel_matrix y dos ejecuciones de reduce_matrix), y la desviación estándar se calcula como la desviación combinada entre ambos kernels.

Se observa que la variante desarrollada reduce el tiempo promedio de ejecución en un factor de 3.4 en comparación con el kernel adaptado. La menor desviación estándar también indica una mayor consistencia en los tiempos de ejecución.

La utilización de reduce resulta en la implementación más rápida, con un tiempo promedio significativamente menor que los otros dos kernels. La mejora es considerable, siendo aproximadamente 16.6 veces más rápida que el kernel adaptado y casi 4.9 veces más rápida que el kernel con Atomic Add. Aunque la desviación estándar es mayor en comparación con el kernel con Atomic Add, el tiempo de ejecución sigue siendo muy favorable.