GPGPU - Laboratorio 4

Ejercicio 1

Este ejercicio pretende ilustrar el uso de la memoria compartida de la GPU para evitar el acceso no coalesced a memoria global.

Dada una matriz $A_{ij} \in R^{n \times n}$ su transpuesta es una matriz $A^T \in R^{n \times n} : A^T_{ij} = A_{ji}$, $i, j = 0, ..., n$. En nuestro caso usaremos la matriz correspondiente a los pixels de la imagen del práctico anterior. El resultado de transponer dicha matriz es el resultado de rotar imagen 90 grados en sentido horario y luego reflejarla horizontalmente (o rotar 90 grados antihorario y reflejar verticalmente) como se muestra en la Figura 1.

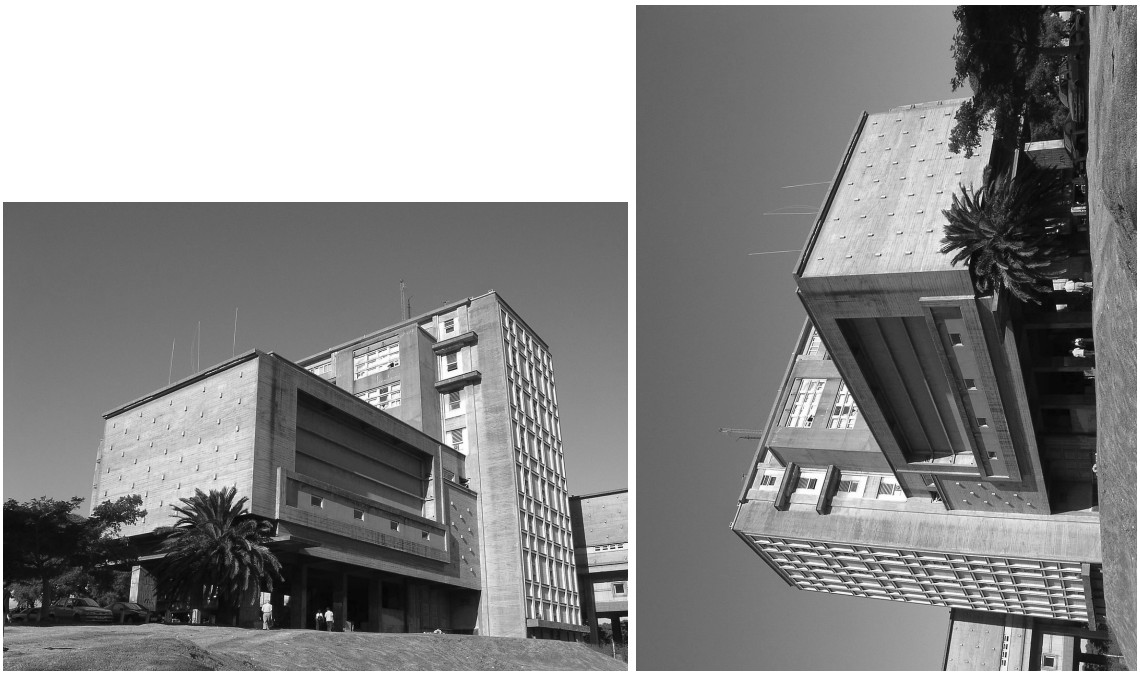

Figura 1: Imagen original y su transpuesta

Una forma sencilla de resolver el problema consiste en leer una fila de la matriz de la memoria, para luego escribir dicha fila como una columna de la matriz transpuesta. En el contexto de la GPU, pueden lanzarse tantos threads como elementos haya en la matriz, y cada thread puede encargarse de leer un elemento de la matriz y escribirlo en la posición correspondiente de la matriz resultado.

- **a-i) Construir un programa que obtenga la matriz transpuesta de una matriz de entrada almacenada en la memoria de la GPU utilizando únicamente la memoria global del dispositivo.**
- **a-ii) Utilizando `nvprof` registre el tiempo del kernel y las métricas `gld_efficiency` y `gst_efficiency`. explicando el resultado.**

Mejorando el acceso a memoria

Para mejorar el patrón de acceso a la memoria podemos utilizar la memoria compartida. Dividiremos conceptualmente la matriz en *tiles* bi-dimensionales (cuadrados) de tamaño igual al tamaño de bloque elegido en la configuración de la grilla y luego se realiza la operación en tres pasos, como se muestra en la Figura 2:

1. Cada bloque de threads carga un *tile* de la matriz a memoria compartida leyendo sus elementos por fila de forma coalesced.
2. Los threads de cada warp (`threadIdx.x` contiguo) leerán una columna del *tile* almacenado en memoria compartida.
3. Luego los threads de cada warp (`threadIdx.x` contiguo) realizan la escritura de dicha columna como (parte de) una fila de la matriz de salida en memoria global de forma coalesced.

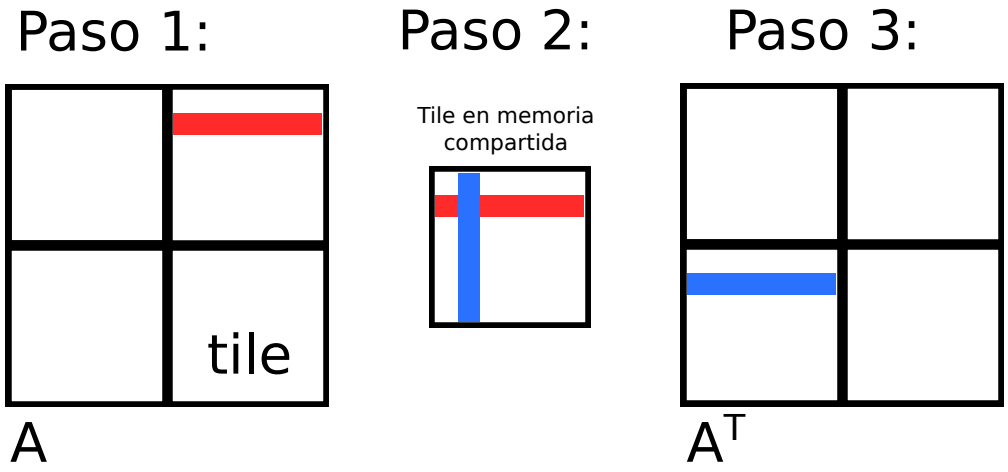

Figura 2: Luego de completado el paso 1, la memoria compartida de cada bloque se encuentra cargada con el *tile* correspondiente, el cual fue leído por fila de forma coalesced. En el paso 3 se escribe el tile correspondiente en la matriz transpuesta leyéndolo de la memoria compartida por columnas y escribiéndolo en memoria global por filas de forma coalesced.

- **b-i) Modificar el código de la parte a) para utilizar la memoria compartida siguiendo el procedimiento que fue explicado anteriormente.**
- **b-ii) Utilizando `nvprof` registre el tiempo del kernel y las métricas `gld_efficiency` y `gst_efficiency`. Compare con los resultados obtenidos en la parte anterior.**

Solución de conflictos de bancos

Para maximizar la eficiencia en el acceso a la memoria compartida, la misma se divide en 32 bancos con un ancho de 32 bits. Las palabras de 32 bits de un arreglo almacenado en memoria compartida se distribuirán secuencialmente entre los bancos, de manera que si los 32 threads de un warp acceden a palabras contiguas, las mismas pueden accederse en paralelo[1]. Sin embargo, si dos threads de un mismo warp acceden a palabras distintas correspondientes a un mismo banco, ocurre un *conflicto de bancos* y dicho acceso se serializa.

- **c-i) Analice el patrón de lectura y escritura en la memoria compartida y determine si (y cuando) ocurren conflictos de bancos.**

[1] Más información en https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory-5-x

- **c-ii) Utilizando `nvprof` registre la métrica `shared_efficiency` para el kernel de la parte anterior utilizando bloques de tamaño 16×16 y 32×32, explicando los resultados.**
- **c-iii) Solucione los conflictos de bancos mediante la técnica vista en teórico. ¿Qué efecto tiene esta modificación sobre el patrón de lectura/escritura?**

Ejercicio 2

Realizaremos dos optimizaciones al filtro de imágenes desarrollado en el práctico anterior, basadas en reducir la cantidad de accesos a memoria global.

Para utilizar la memoria compartida se deberá dividir la imagen en bloques rectangulares. Dadas las características del filtro, aquellos threads que operen sobre el borde de cada bloque deberán acceder a pixels (que están dentro de su ventana) pertenecientes a otro bloque de la imagen. Deberán cargarse en memoria compartida los pixels correspondientes a cada thread y además aquellos pixels que pertenecen a la ventana de los threads del borde, tal como muestra la Figura 3.

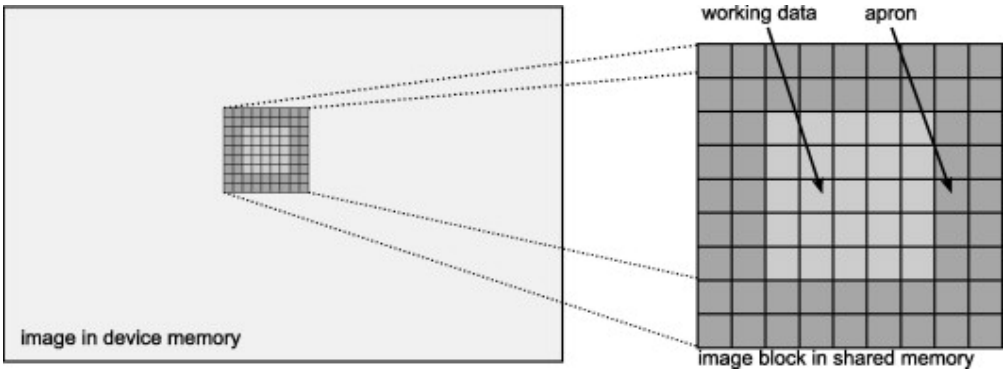

Figura 3: Datos cargados en memoria compartida

La carga a memoria compartida debe llevarse a cabo con el mayor grado de paralelismo posible pero utilizando únicamente la cantidad de threads necesaria para realizar el cómputo, es decir, algunos hilos deberán cargar más de un valor debido a los "bordes". Esta carga puede realizarse iterativamente como se muestra en la Figura 4.

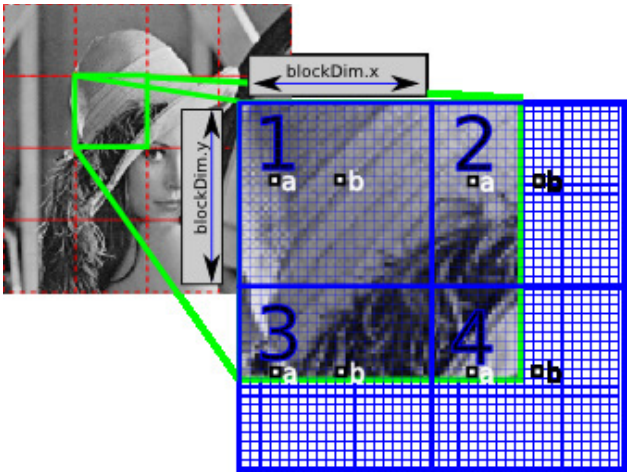

Figura 4: Carga iterativa de los datos. Aquí el thread de la grilla marcado con la letra a carga 4 elementos de la zona de la imagen. El thread marcado con b carga únicamente 2, quedando inactivo en las iteraciones 2 y 4.

- **a-i) Construya una versión del filtro de imágenes del laboratorio anterior que utilice la memoria compartida utilizando el método descrito para cargar los datos.**

- **a-ii) Construya otra versión donde cada hilo cargue un único dato a memoria compartida y el resto de los datos sean accedidos directamente desde memoria global (activar la caché L1 para accesos a global usando `nvcc -Xptxas -dlcm=ca ...`).**
- **a-iii) Compare el tiempo de ejecución de las 2 versiones entre sí y con el del kernel del práctico anterior.**

Mejorando el acceso a constantes

CUDA dispone de dos mecanismos para mejorar el acceso a datos que no son modificados durante la ejecución del kernel. La GPU dispone de una memoria especial (llamada memoria constante) de 64KB, la cual es cacheada de forma que el acceso es óptimo cuando todos los threads de un warp acceden al mismo elemento. El segundo mecanismo es utilizar `const __restrict__` para facilitar al compilador la tarea de optimizar los accesos a memoria a través del uso de la caché L1.

- **b-i) Utilice la memoria compartida para almacenar la máscara. Utilice memoria compartida estática.**
- **b-ii) Construya una versión similar a la b-i) pero utilizando memoria compartida dinámica.**
- **b-iii) Utilice `const __restrict__`[2] para el parámetro del kernel correspondiente a la máscara (sin usar memoria compartida para la máscara).**
- **b-iv) Utilice `__constant__` y `cudaMemcpyToSymbol`[3] para que la máscara resida en memoria constante, modificando el kernel adecuadamente.**
- **b-v) Estudie el desempeño de las versiones anteriores comparándolas entre sí y con la mejor versión entre a-i) y a-ii).**

[2] https://devblogs.nvidia.com/cuda-pro-tip-optimize-pointer-aliasing/ o Cuda C Programming Guide (en eva del curso) página 92.

[3] Ver Cuda C Programming Guide (en eva del curso) página 23.