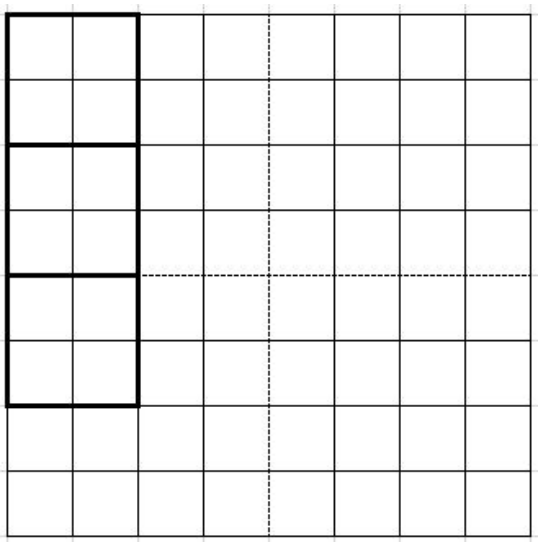
Capítulo 4 Memory and data locality

1. Consider matrix addition. Can one use shared memory to reduce the global memory bandwidth consumption? Hint: Analyze the elements accessed by each thread and see if there is any commonality between threads.

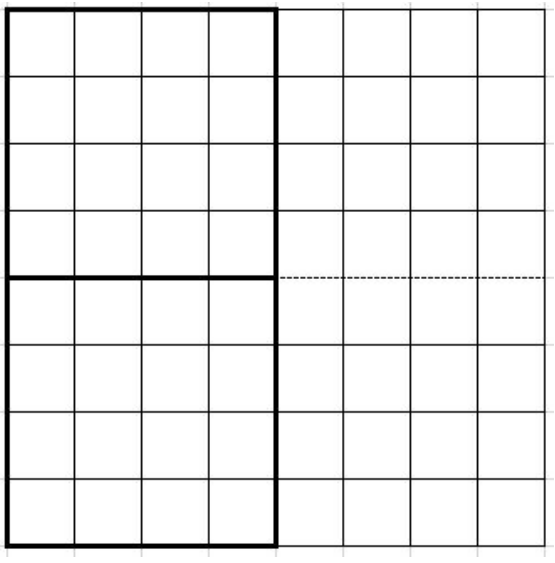
   Como cada thread va a calcular la suma de un elemento de la matriz, tenemos que, los datos de entrada para cada thread son diferentes, no se repiten por lo que no se puede utilizar el uso compartido de lectura mediante la shared memory.

2. Draw the equivalent of Fig. 4.14 for an 8× 8 matrix multiplication with 2× 2 tiling and 4× 4 tiling. Verify that the reduction in global memory bandwidth is indeed proportional to the dimensions of the tiles.

Matrix de 8 x 8 con tiles de 2 x 2



Matrix de 8 x 8 con tiles de 4 x 4



Al cargar tiles de *n* x *n* en la shared memory, se reduce proporcionalmente en factor de *n* (como en el ejemplo de arriba). Al tener mayor *n*, más elementos de un mismo bloque pueden ser utilizados sin usar la global memory.

3. What type of incorrect execution behavior can happen if one or both __syncthreads() are omitted in the kernel of Fig. 4.16?

   Hay dos __syncthreads(), uno en la línea 11 y el otro en la 14. Si se omite el primero, los threads seguirán ejecutándose, sin asegurarse primero de que todos los tiles d_M y d_N sean guardados en la memoria compartida (Mds y Nds).
   El otro caso es que se omita el segundo __syncthreads(), podría suceder que algún thread cargaría los elementos demasiado pronto y dañaría los valores de entrada de otros threads. Esta barrera asegura que todos los hilos hayan terminado de usar los elementos M y N en la memoria compartida antes de que cualquiera de ellos pase a la siguiente iteración y cargue los elementos de los siguientes tiles.

4. Assuming that capacity is not an issue for registers or shared memory, give one important reason why it would be valuable to use shared memory instead of registers to hold values fetched from global memory? Explain your answer.

   Como hemos visto en CUDA Memory Types, la shared memory tiene mayor latencia que los registros porque los registros son individuales para cada thread por lo que los datos almacenados ahí son privados para cada thread. Por otro lado, la shared memory es para el bloque entero, esto a su vez es una ventaja ya que al estar disponible para el bloque, todos los threads de ese bloque pueden acceder a la shared memory pero la ventaja es que las variables que residen en la memoria compartida son accesibles por todos los threads en un bloque, si queremos intercambiar información eficientemente y con un buen bandwidth entre threads la memoria compartida es la mejor opción.

5. For our tiled matrix–matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices M and N?

   C. 1/32 of the original usage.

   Como en una respuesta anterior, hemos visto que la reducción se hace en relación al tamaño del tile dado, en este caso, cada elemento en el tile es usado 32 veces.

6. Assume that a CUDA kernel is launched with 1,000 thread blocks, with each having 512 threads. If a variable is declared as a local variable in the kernel, how many versions of the variable will be created through the lifetime of the execution of the kernel?

   D. 512000

   Si es una local variable, se crea una copia de esta variable en cada thread de forma privada. Como tenemos 512 threads por bloque, se crean 512 copias de variable por

bloque, pero hay 1000 bloques entonces la cantidad total de copias sería: 512 x 1000 = 512000.

7. In the previous question, if a variable is declared as a shared memory variable, how many versions of the variable will be created throughout the lifetime of the execution of the kernel?

   B. 1000

   La memoria compartida a diferencia de la local se crea por cada bloque que estemos usando, independientemente de la cantidad de threads por bloque, como tenemos 1000 bloques, entonces tenemos 1000 copias de esa variable.

8. Consider performing a matrix multiplication of two input matrices with dimensions N × N. How many times is each element in the input matrices requested from global memory in the following situations?

   A. There is no tiling.

      Como no se usa la shared memory en este caso, la multiplicación de matrices es de filas por columnas, entonces cada elemento de la primera matriz será leído N veces porque tiene N filas, del mismo modo, de la segunda matriz cada elementos será leído N veces porque tiene N columnas, en este caso al ser la matriz cuadrada, la cantidad de veces que cada elemento va a ser leído va a depender del tamaño de la matriz (en este caso N).

   B. Tiles of size T × T are used.

      En este caso sí usaremos los tiles, como ya se ha visto en ejercicios anteriores (2 y 5), la cantidad de veces que cada elemento va a ser usado va a ser el tamaño de la matriz cuadrada (que sí cumple N x N) y el size del tile (T), por lo que tendríamos N/T veces que un elemento será accesado.

9. A kernel performs 36 floating-point operations and 7 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.

   A. Peak FLOPS= 200 GFLOPS, Peak Memory Bandwidth= 100 GB/s

   Compute:

$$\frac{36}{200*10^9} = 0.18 * 10^{-9}s$$

   Memory:

$$\frac{(7)(4)}{100*10^9} = 0.28 * 10^{-9}s$$

Dado que el tiempo en memory es mayor, entonces es memory-bound.

B. Peak FLOPS= 300 GFLOPS, Peak Memory Bandwidth= 250 GB/s

Compute:

$$\frac{36}{300*10^9} = 0.12 * 10^{-9}s$$

Memory:

$$\frac{(7)(4)}{250*10^9} = 0.112 * 10^{-9}s$$

Dado que el tiempo en CPU es mayor, entonces es compute-bound.

10. To manipulate tiles, a new CUDA programmer has written the following device kernel, which will transpose each tile in a matrix. The tiles are of size BLOCK_WIDTH by BLOCK_WIDTH, and each of the dimensions of matrix A is known to be a multiple of BLOCK_WIDTH. The kernel invocation and code are shown below. BLOCK_WIDTH is known at compile time, but could be set anywhere from 1 to 20.

```
dim3 blockDim(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 gridDim(A_width/blockDim.x,A_height/blockDim.y);

BlockTranspose<<<gridDim, blockDim>>>(A, A_width, A_height);

__global__ void BlockTranspose(float* A_elements, int A_width,
int A_height)
{
  __shared__ float blockA[BLOCK_WIDTH][BLOCK_WIDTH];
  int baseIdx=blockIdx.x * BLOCK_SIZE + threadIdx.x;
  baseIdx += (blockIdx.y * BLOCK_SIZE + threadIdx.y) * A_width;
  blockA[threadIdx.y][threadIdx.x]=A_elements[baseIdx];
  A_elements[baseIdx]=blockA[threadIdx.x][threadIdx.y];
}
```

A. Out of the possible range of values for BLOCK_SIZE, for what values of BLOCK_SIZE will this kernel function execute correctly on the device?

Va a funcionar bien con un BLOCK_SIZE desde 1 a 5. Dado que al encontrar la cantidad total de threads no debe exceder la cantidad de threads en un warp.

B. If the code does not execute correctly for all BLOCK_SIZE values, suggest a fix to the code to make it work for all BLOCK_SIZE values.

Usar una barrera __syncthreads() entre las líneas que leen y escriben la matriz de memoria compartida.