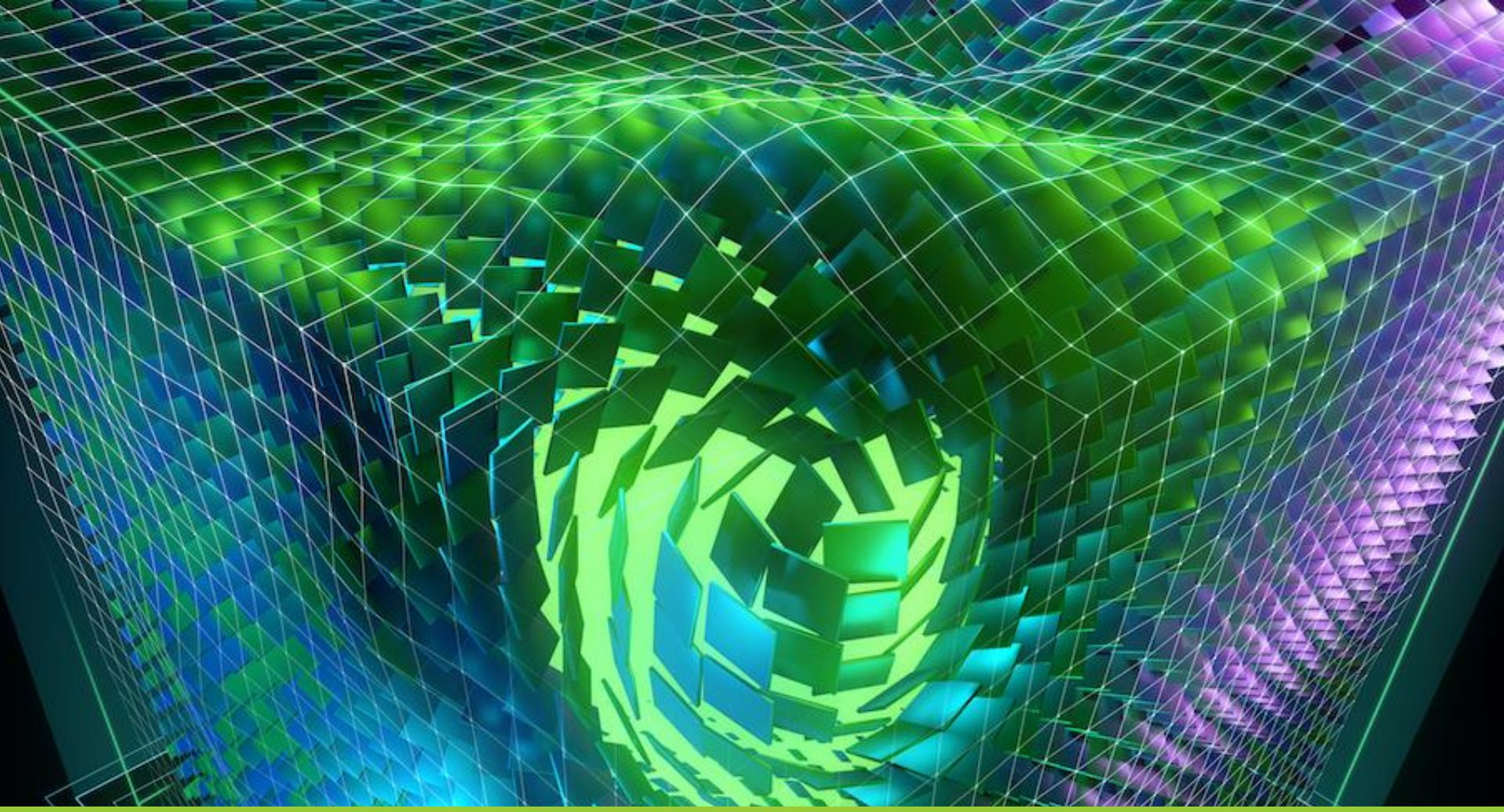


INF351 – Computación de Alto Desempeño

# Memoria Compartida

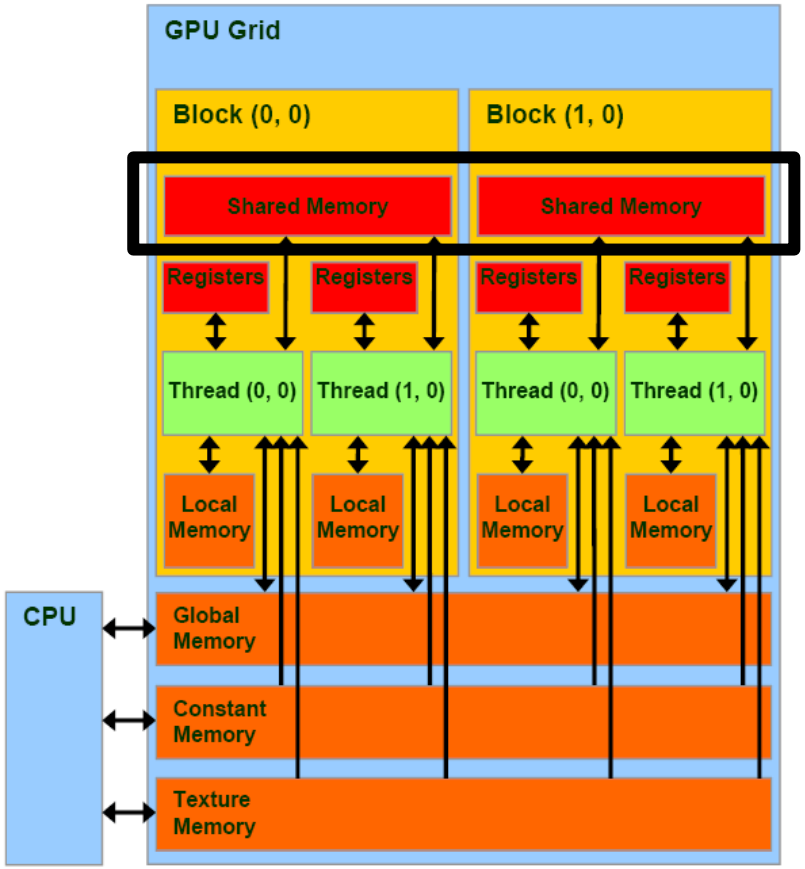
PROF. ÁLVARO SALINAS

# Memoria Compartida

Uno de los tipos de memoria del dispositivo. Es la más rápida después de la memoria de registros.

Físicamente, se encuentra dentro de la GPU. A esto se debe su baja latencia.

Solo puede ser asignada dentro de un kernel y es únicamente visible para las hebras pertenecientes al bloque que trabaja con ella.



# Memoria Compartida

La memoria compartida, o shared memory, es un tipo de memoria de la GPU ubicada dentro de cada SM.

Comparte su espacio físico con el caché L1.

Su ubicación le permite presentar latencias muy bajas, usualmente 100 veces más bajas respecto a la memoria global.

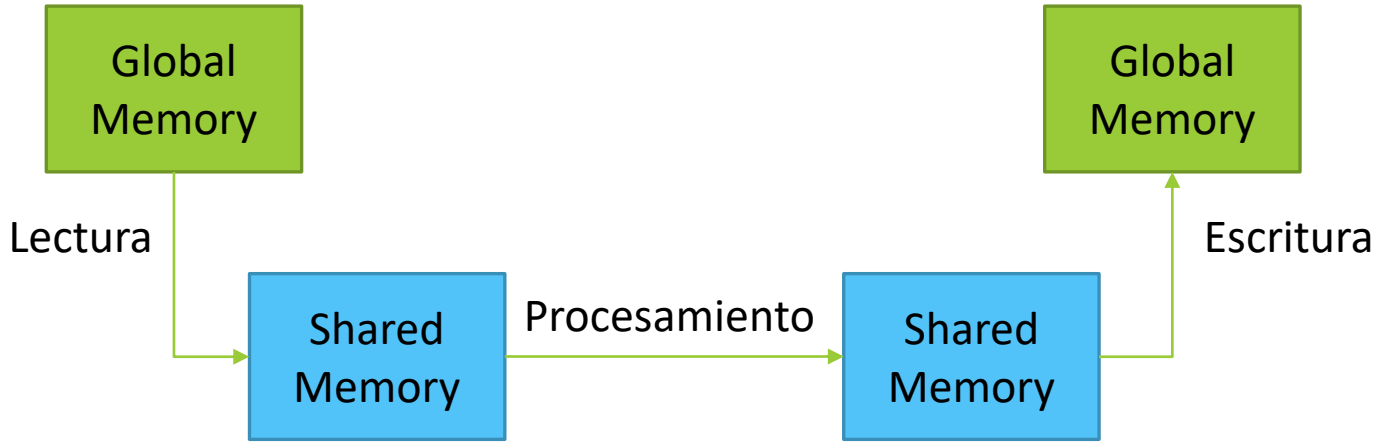
Al igual que la cantidad de registros utilizados, la cantidad de memoria compartida es también un limitante para la ocupancia, pues cada SM posee una cantidad máxima.

Su utilización puede provocar una gran mejora al desempeño de la aplicación, pero solo será útil en algunos casos.

# ¿Cómo se usa?

Ya que la memoria compartida debe ser asignada dentro de un kernel, a menos que sean generados en el kernel, los datos que deseemos almacenar en ella deben ser leídos desde memoria global.

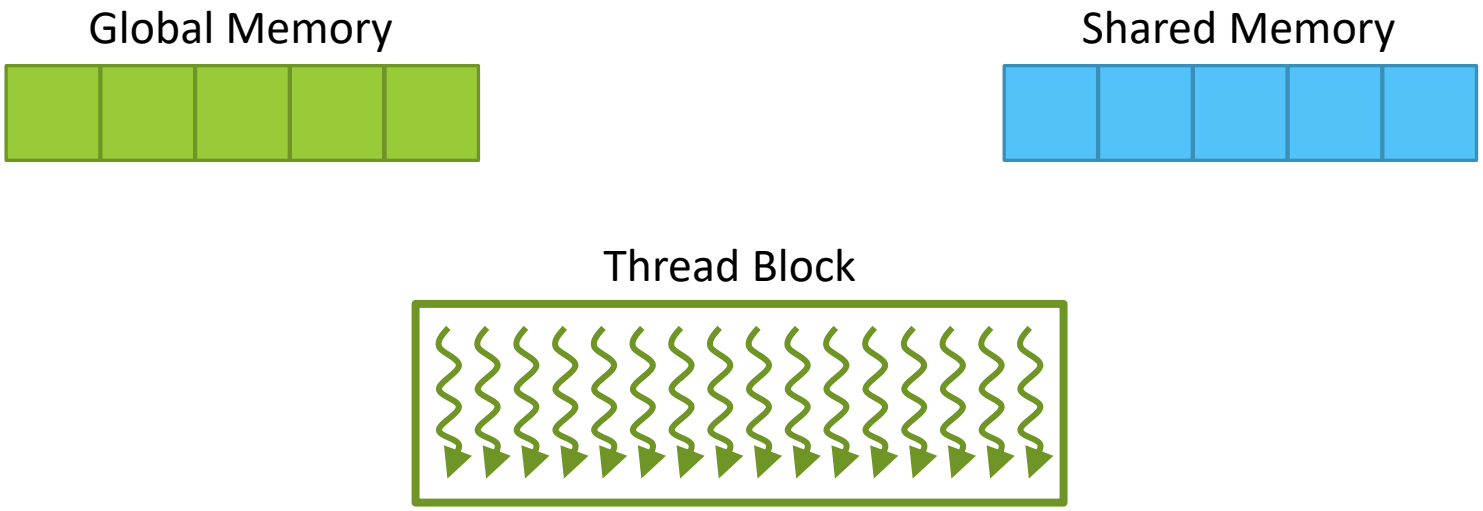
Una vez que los datos hayan sido procesados, actualizados o se hayan generado resultados almacenados en shared memory, debemos escribir estos en global memory. De no hacerlo, la información desaparecerá junto con la memoria compartida al finalizar el kernel.



# ¿Cómo funciona?

Cuando una hebra comienza su ejecución del kernel, lee un dato desde memoria global (tal y como han sido nuestras implementaciones hasta ahora). Dicho dato es almacenado por la hebra en memoria compartida.

Una vez que el dato se encuentre en shared memory, todas las hebras pertenecientes al mismo bloque de la hebra que escribió dicho dato pueden leerlo.



# ¿Cómo funciona?

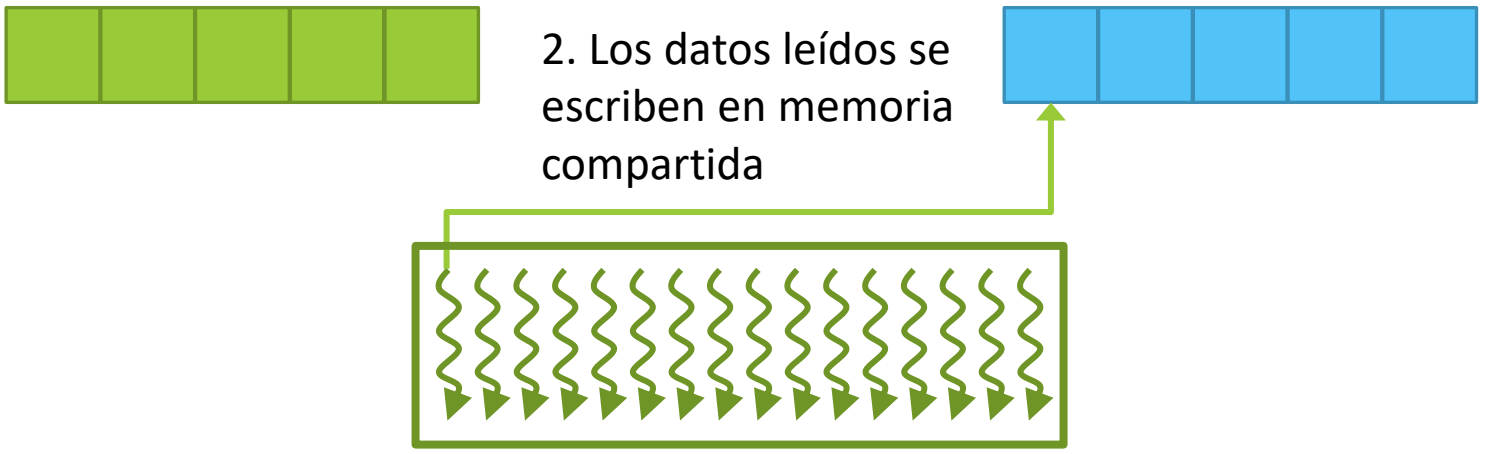
Cuando una hebra comienza su ejecución del kernel, lee un dato desde memoria global (tal y como han sido nuestras implementaciones hasta ahora). Dicho dato es almacenado por la hebra en memoria compartida.

Una vez que el dato se encuentre en shared memory, todas las hebras pertenecientes al mismo bloque de la hebra que escribió dicho dato pueden leerlo.

# ¿Cómo funciona?

Cuando una hebra comienza su ejecución del kernel, lee un dato desde memoria global (tal y como han sido nuestras implementaciones hasta ahora). Dicho dato es almacenado por la hebra en memoria compartida.

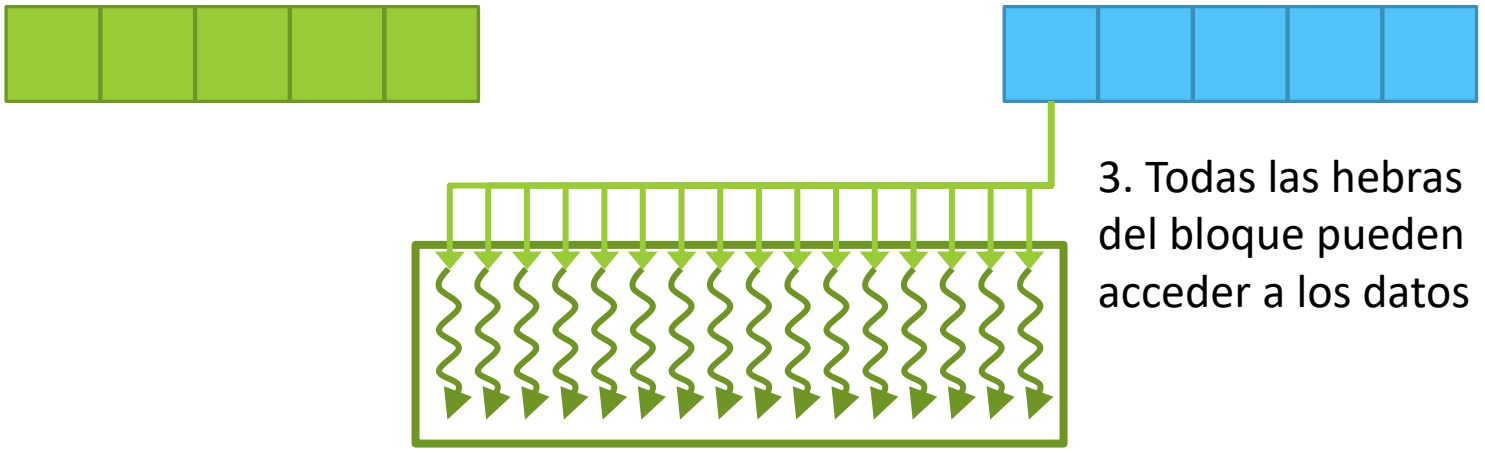
Una vez que el dato se encuentre en shared memory, todas las hebras pertenecientes al mismo bloque de la hebra que escribió dicho dato pueden leerlo.



# ¿Cómo funciona?

Cuando una hebra comienza su ejecución del kernel, lee un dato desde memoria global (tal y como han sido nuestras implementaciones hasta ahora). Dicho dato es almacenado por la hebra en memoria compartida.

Una vez que el dato se encuentre en shared memory, todas las hebras pertenecientes al mismo bloque de la hebra que escribió dicho dato pueden leerlo.



# ¿Cuándo se usa?

A partir de lo anterior, podemos deducir que trabajar con memoria compartida no significa evitar el uso de memoria global, sino solo copiar la información a una memoria de acceso más rápido para su procesamiento.

Esto suena bastante similar al uso de registros, con la excepción de que estos últimos solo son visibles por la hebra que los crea. Entonces, ¿en qué casos conviene utilizar memoria compartida?

La memoria compartida es de utilidad solamente en casos en donde un mismo dato será utilizado por varias hebras pertenecientes al mismo bloque o donde se realizan múltiples lecturas/escrituras secuenciales en memoria global.

Anteriormente se utilizaba para lograr accesos de memoria coalescentes en casos especiales.

# Manejo de Shared Memory

Para crear variables residentes en memoria compartida debemos utilizar la etiqueta *__shared__*.

La memoria compartida puede ser asignada de forma estática o dinámica.

En el primer caso, debemos conocer el largo del array en tiempo de compilación. Un ejemplo sería:

*__shared__ int a[64];*

con lo que creamos un arreglo de 64 enteros que serán visibles por todas las hebras del bloque.

# Manejo de Shared Memory

Si no conocemos el largo del array en tiempo de compilación, la memoria debe ser asignada dinámicamente. Para ello debemos utilizar el descriptor *extern* como en el siguiente ejemplo:

*extern __shared__ int a[];*

De todas formas debemos darle una dimensión a la memoria que se está asignando. Esto se logra al especificar un tercer parámetro de configuración de ejecución en la llamada al kernel.

La siguiente invocación:

*kernel<<<grid_size, block_size, n*sizeof(int)>>>(n, …);*

permite indicar que la memoria reservada será la de *n* enteros.

# Manejo de Shared Memory

Este tercer parámetro que indica el tamaño de la memoria a asignar es único, por lo que si deseamos crear más de un array en memoria compartida, debemos crear uno y dividirlo mediante punteros. Por ejemplo, la siguiente asignación:

```
extern __shared__ int a[];
int *a_int = a;
float *a_float = (float*)&a_int[nint];
char *a_char = (char*)&a_float[nfloat];
```

Permite crear un array de *nint* enteros, uno de *nfloat* flotantes y uno de *nchar* chars con la siguiente invoación:

```
int size = nint*sizeof(int)+nfloat*sizeof(float)+nchar*sizeof(char);
kernel<<< grid_size, block_size, size>>>(nint, nfloat, nchar, …);
```

# Sincronización

Dado que cada hebra lee un dato desde memoria global y lo almacena en memoria compartida para que las demás hebras pertenecientes a su bloque puedan leerlo, es necesario manejar las posibles condiciones de carrera que pudiesen surgir.

# Sincronización

Dado que cada hebra lee un dato desde memoria global y lo almacena en memoria compartida para que las demás hebras pertenecientes a su bloque puedan leerlo, es necesario manejar las posibles condiciones de carrera que pudiesen surgir.

- *cudaDeviceSyncrhonize();*

# Sincronización

Dado que cada hebra lee un dato desde memoria global y lo almacena en memoria compartida para que las demás hebras pertenecientes a su bloque puedan leerlo, es necesario manejar las posibles condiciones de carrera que pudiesen surgir.

- *cudaDeviceSyncrhonize();*
- Sincronización implícita entre kernels.

# Sincronización

Dado que cada hebra lee un dato desde memoria global y lo almacena en memoria compartida para que las demás hebras pertenecientes a su bloque puedan leerlo, es necesario manejar las posibles condiciones de carrera que pudiesen surgir.

- *cudaDeviceSyncrhonize();*
- Sincronización implícita entre kernels.
- *__syncthreads();*

# Sincronización

Dado que cada hebra lee un dato desde memoria global y lo almacena en memoria compartida para que las demás hebras pertenecientes a su bloque puedan leerlo, es necesario manejar las posibles condiciones de carrera que pudiesen surgir.

- *cudaDeviceSyncrhonize();*
- Sincronización implícita entre kernels.
- *__syncthreads();*

El comando *__syncthreads()* nos permite sincronizar todas las hebras pertenecientes al mismo bloque. Debe ser llamada desde un kernel.

# Bank Conflict

Al igual que ocurre con accesos de memoria no coalescentes con la memoria global, los accesos a memoria compartida efectuados por un warp también pueden ser serializados. Esto ocurre en presencia de un bank conflict.

La memoria compartida se encuentra repartida en banks de 32 bits. El número de banks es un valor fijo que depende de la compute capability de la GPU utilizada.

# Bank Conflict

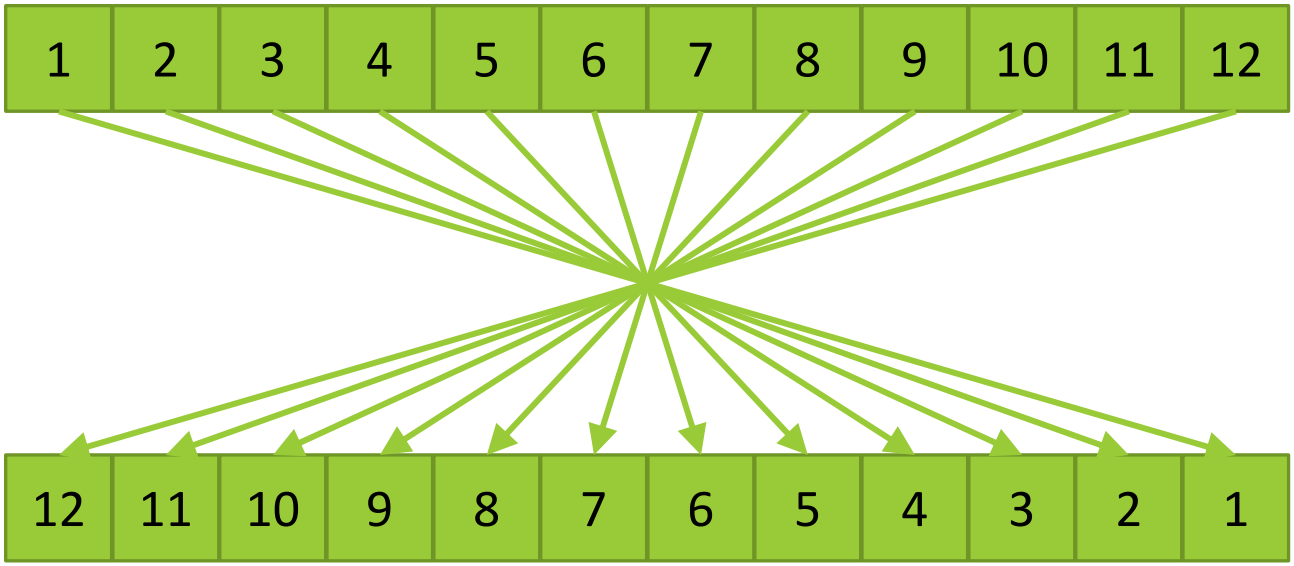
Un mismo bank posee varias direcciones de memoria, las cuales están separadas por $n \times 4$ bytes si consideramos $n$ banks.

| Bank Number | 1 | 2 | 3 | … | 31 | 32 |
|---|---|---|---|---|---|---|
| Memory Adresses | 0 | 4 | 8 | … | 120 | 124 |
| | 128 | 132 | 136 | | 248 | 252 |

Un conflict ocurre cuando hebras del mismo warp intentan acceder a direcciones de memoria pertenecientes al mismo bank. En dicho caso, esos accesos se realizan de forma secuencial.

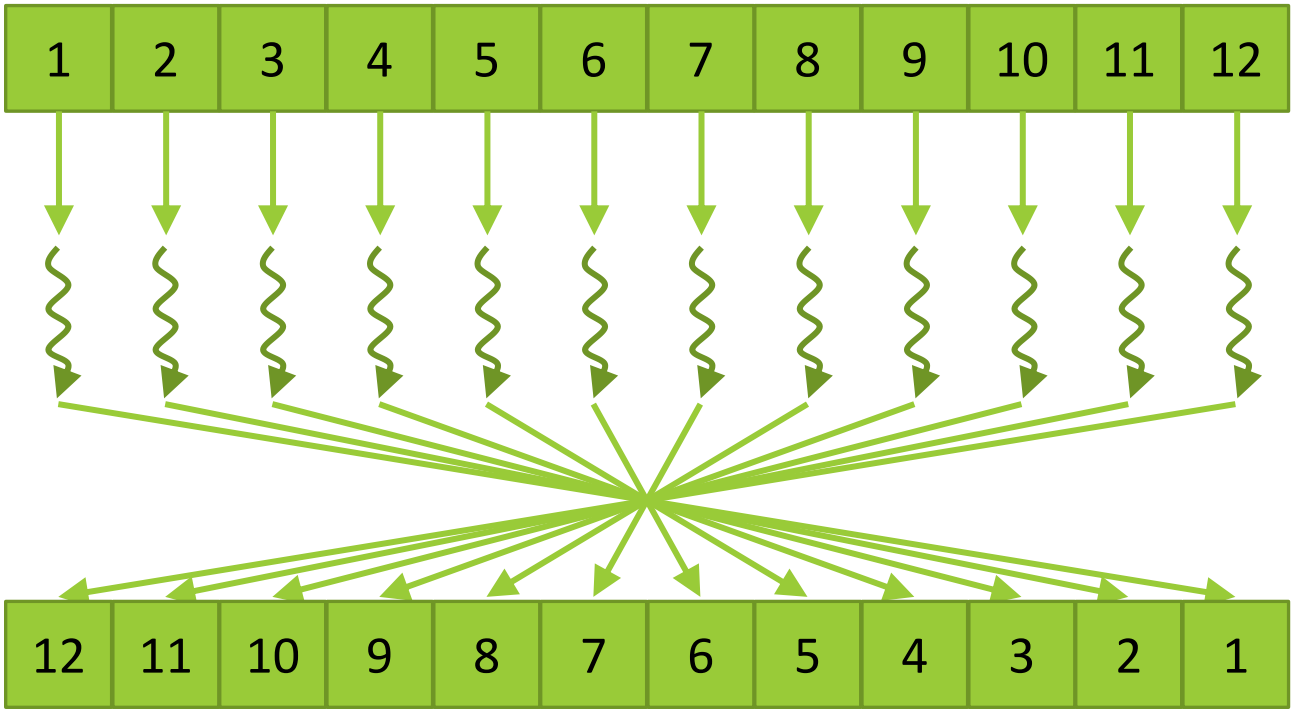
La única excepción a esto es cuando todas las hebras del warp acceden al mismo dato, produciéndose un broadcast realizado en una única transacción de memoria.

# Ejemplo: Invirtamos un array



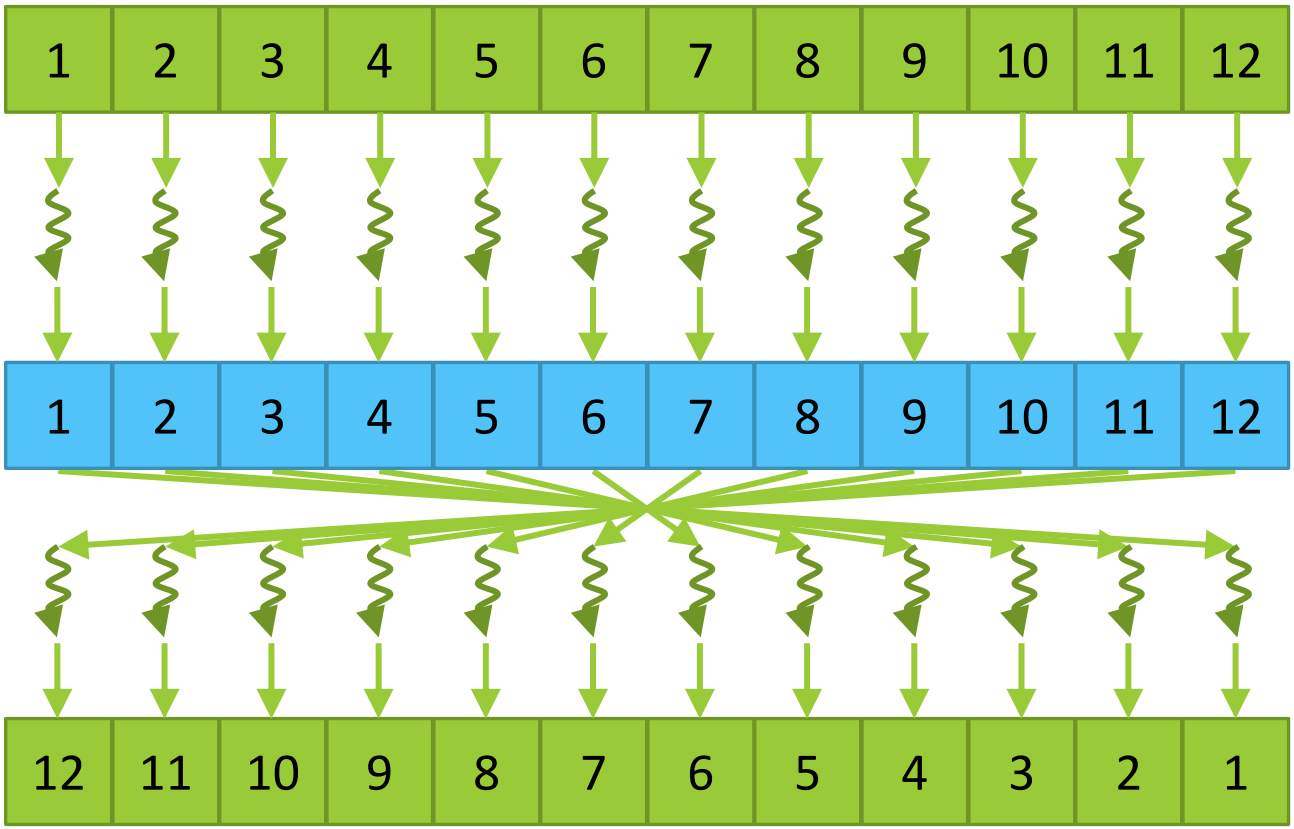
# Ejemplo: Invirtamos un array

Directamente desde memoria global:

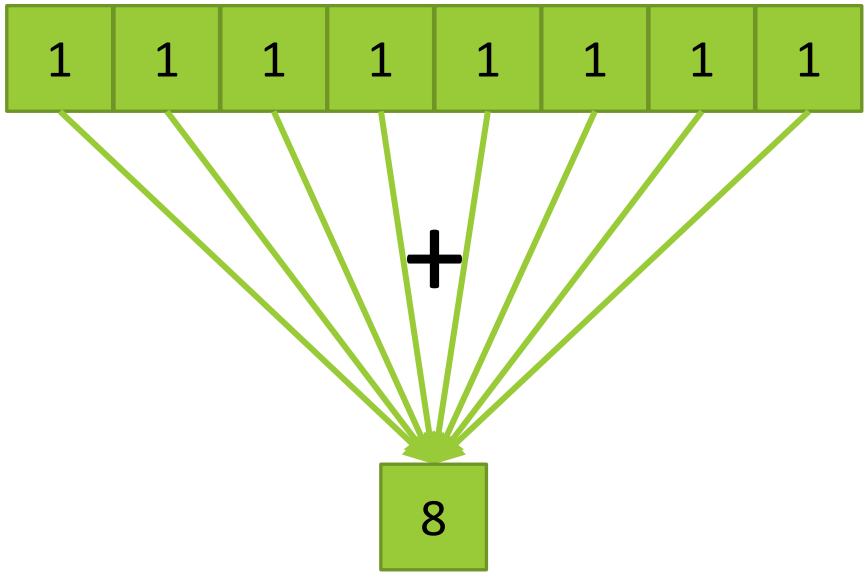


# Ejemplo: Invirtamos un array

Con memoria compartida:

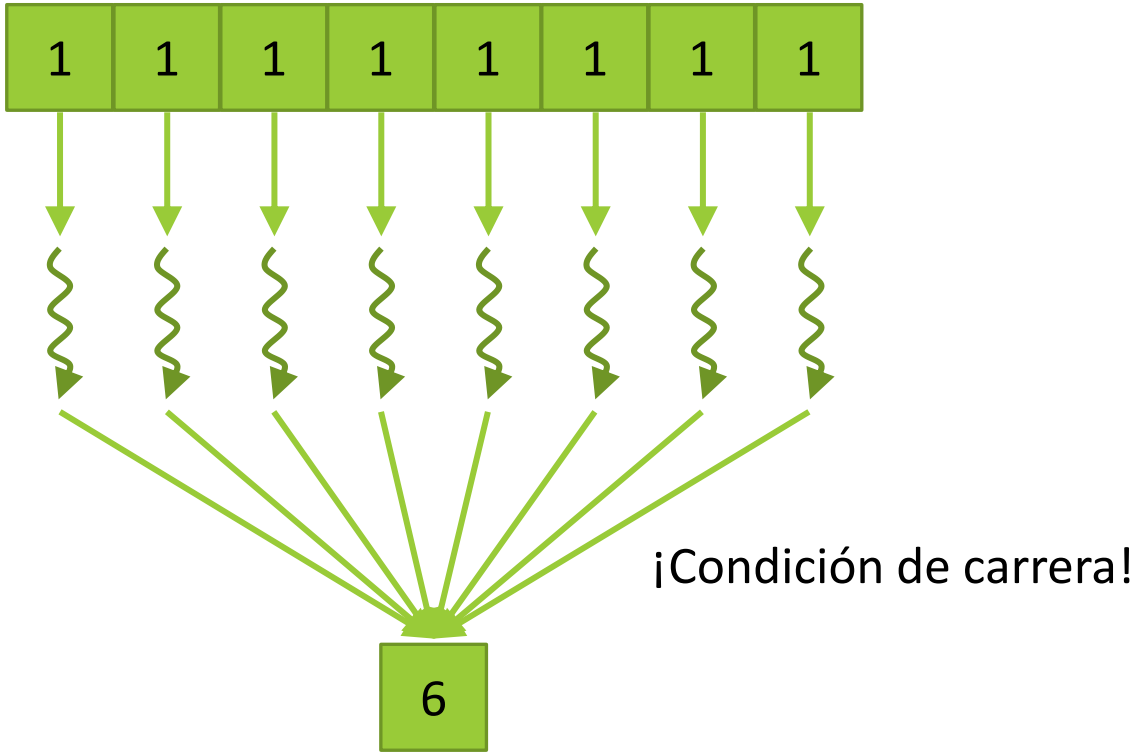


# Ejemplo: Reducción



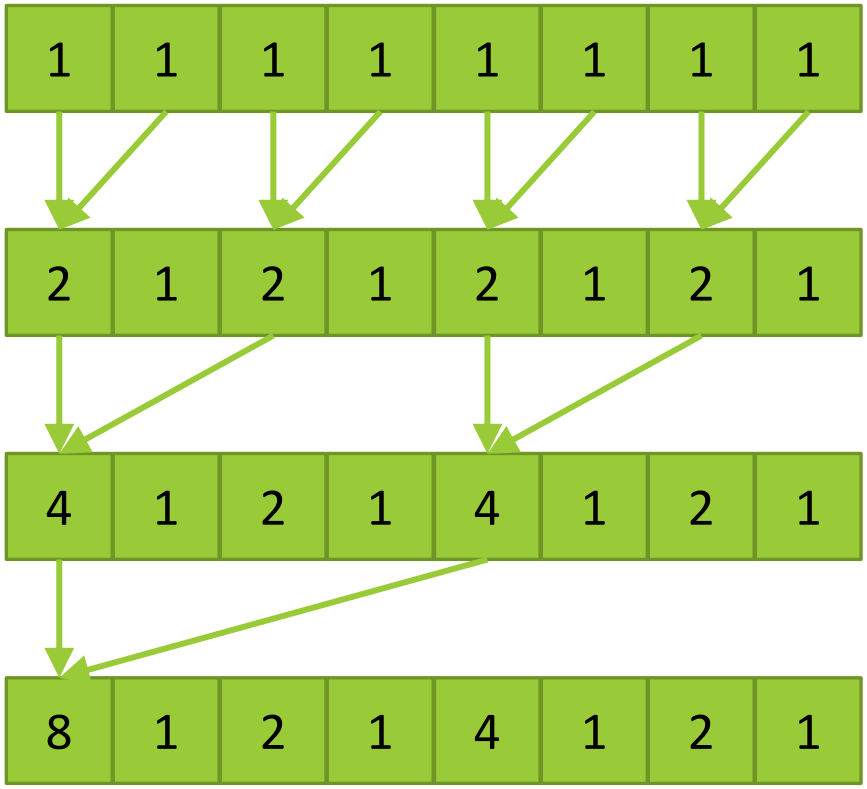
# Ejemplo: Reducción

Implementación naive:



# Ejemplo: Reducción

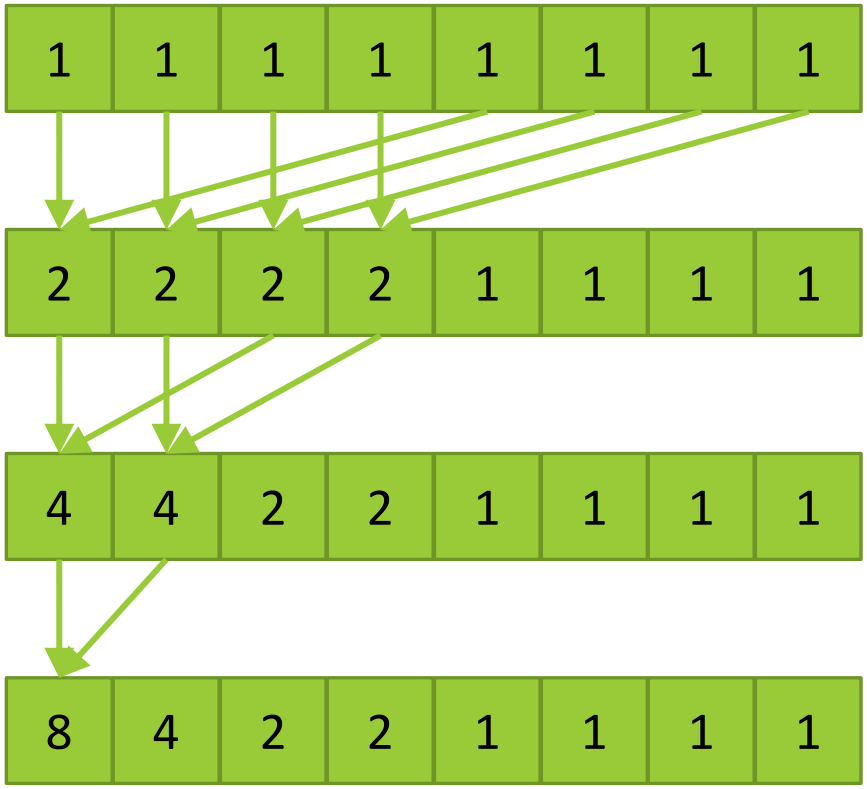
Utilizando memoria global:




¡Accesos de memoria no coalescentes!

# Ejemplo: Reducción

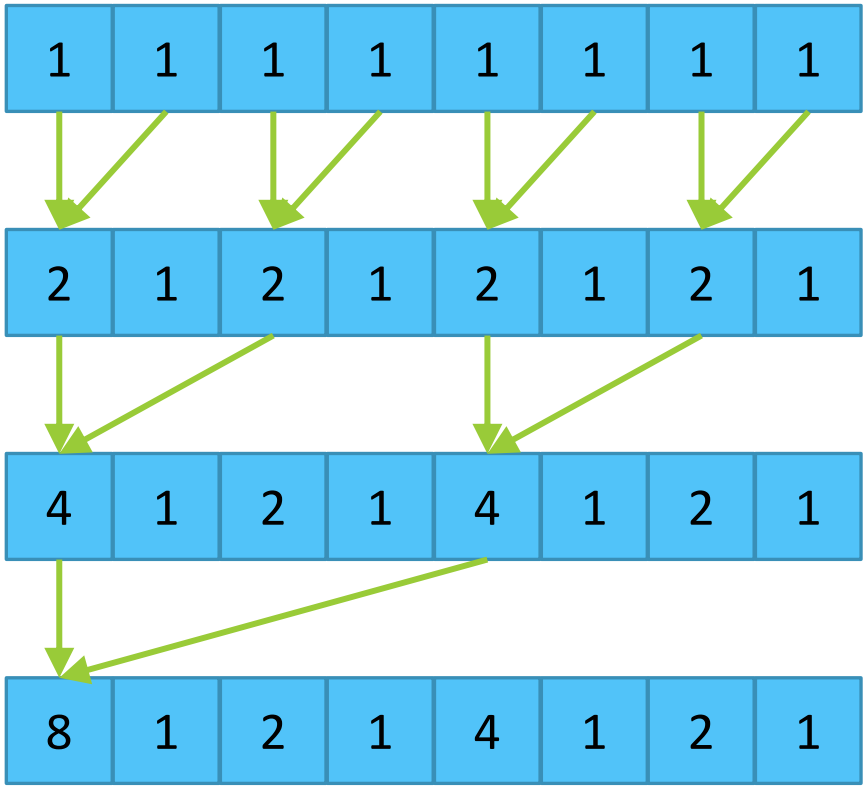
Con accesos de memoria coalescentes:




¡Muchos accesos al mismo elemento por hebras distintas!

# Ejemplo: Reducción

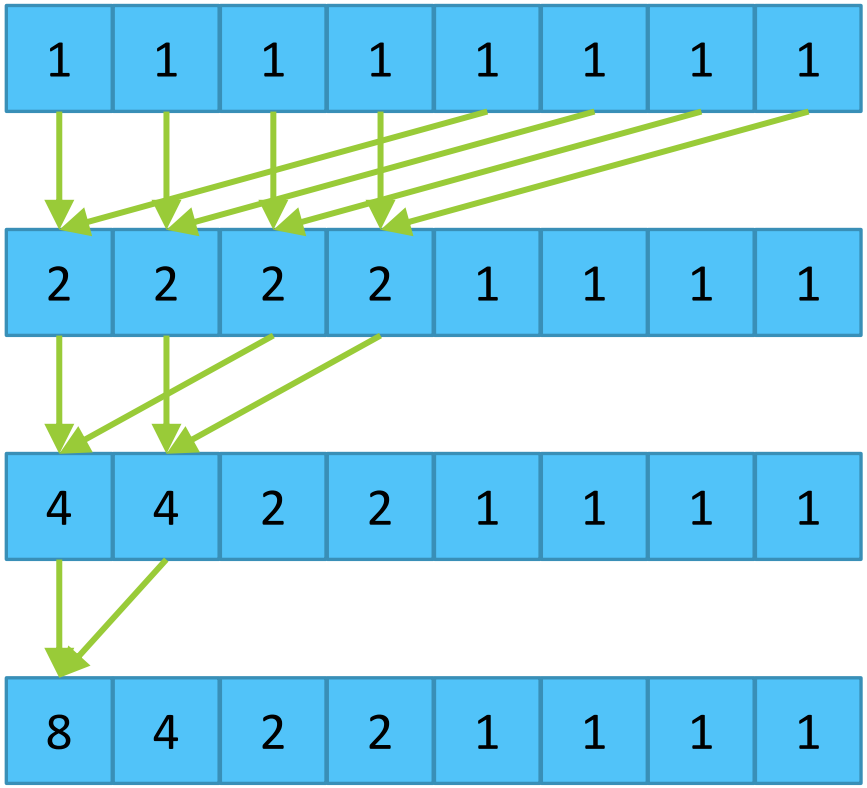
Utilizando memoria compartida:



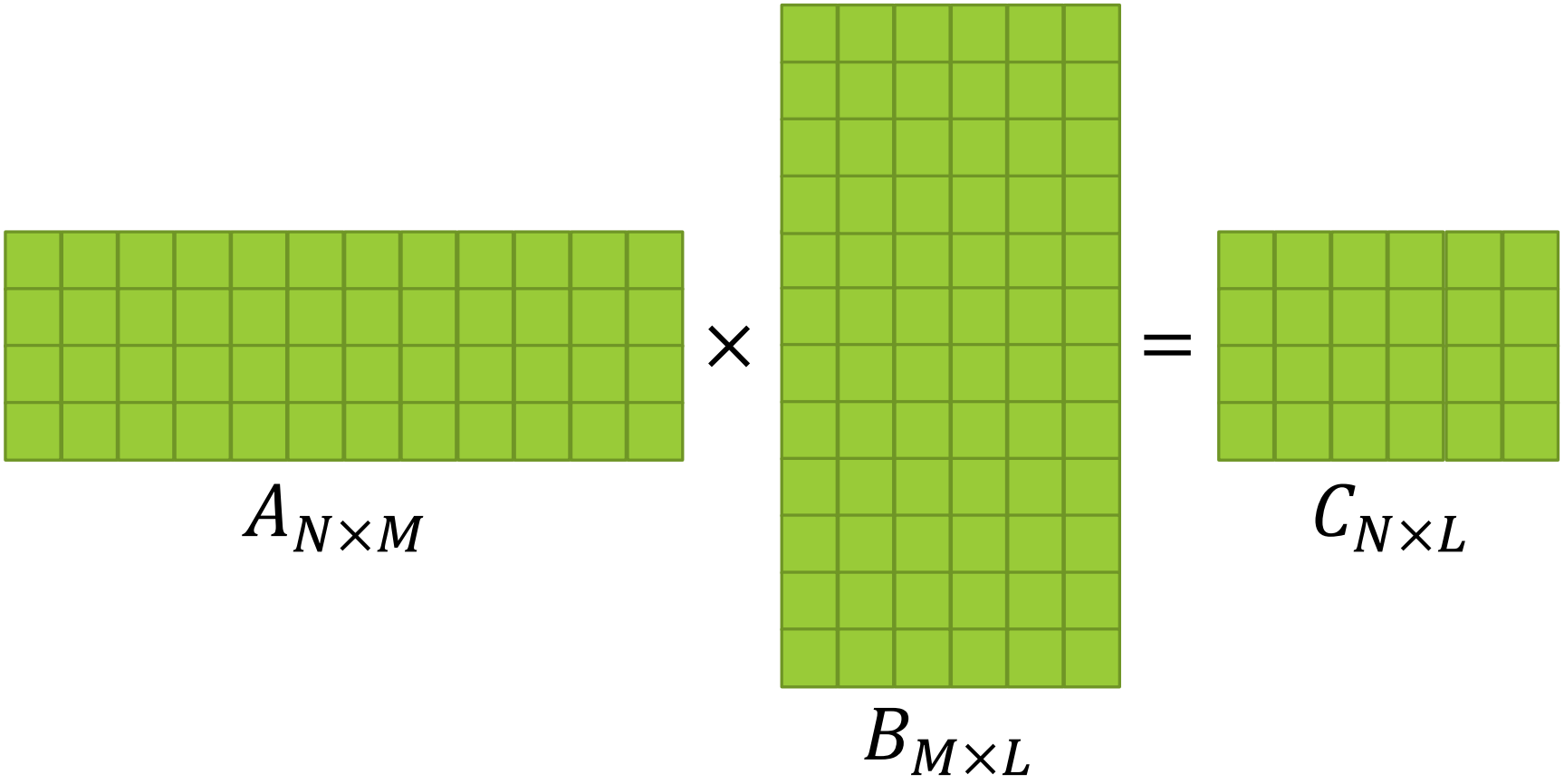

¡Bank conflicts!

# Ejemplo: Reducción

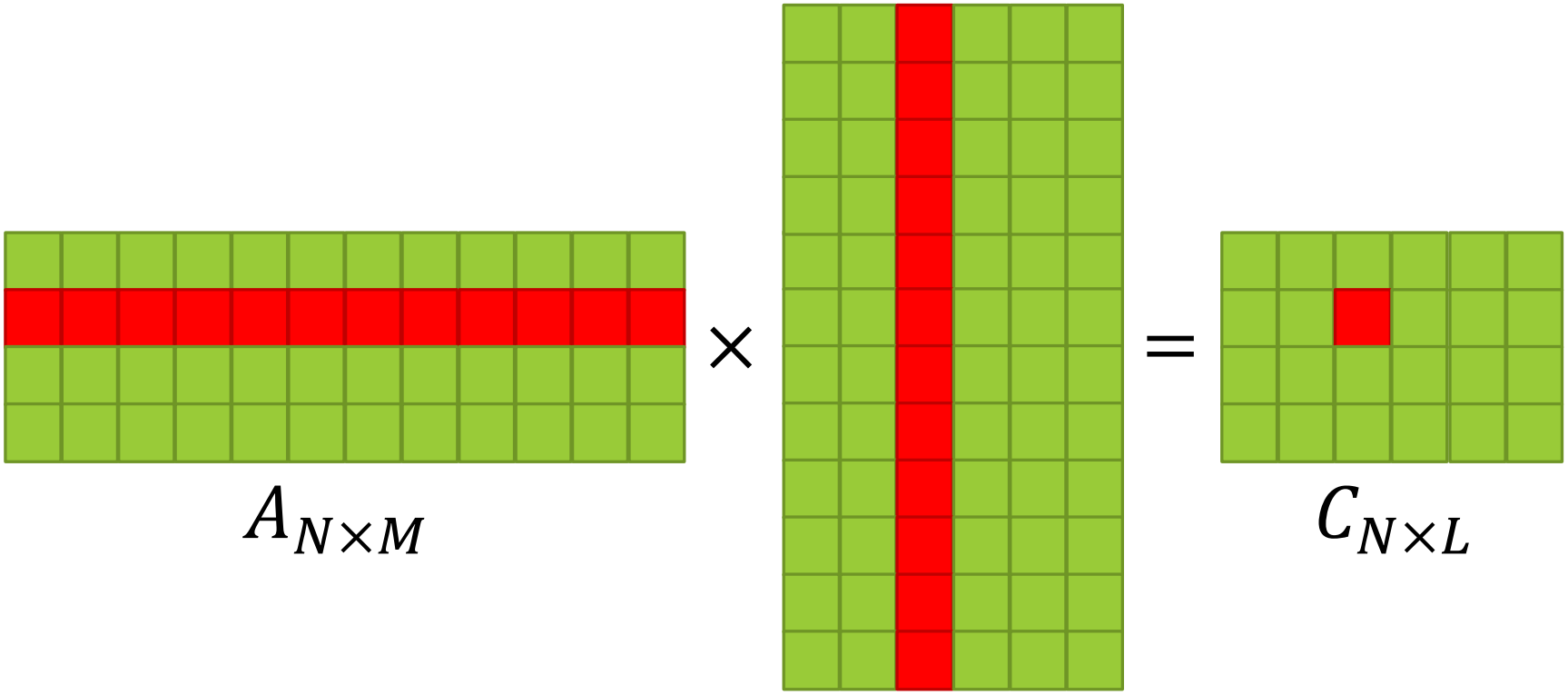
Con memoria compartida sin conflictos:



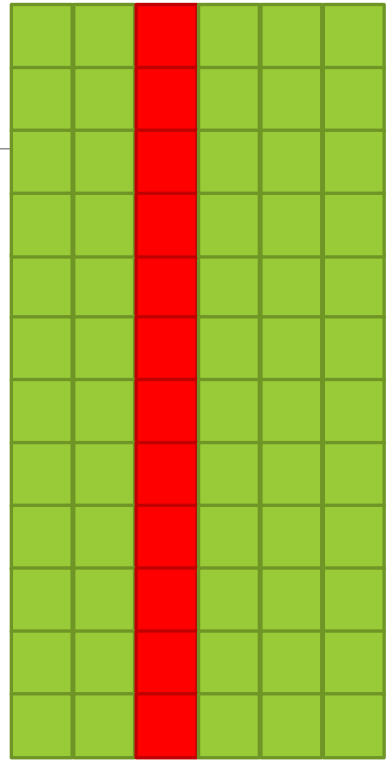
# Ejemplo: Multiplicación de Matrices

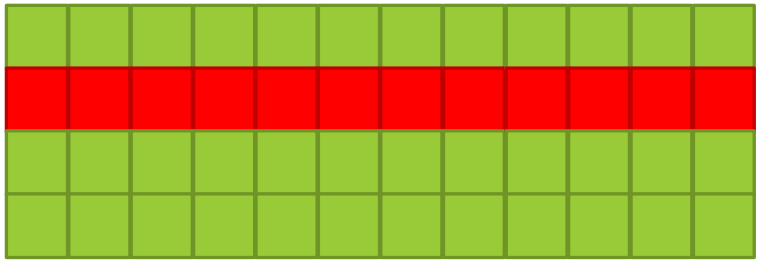


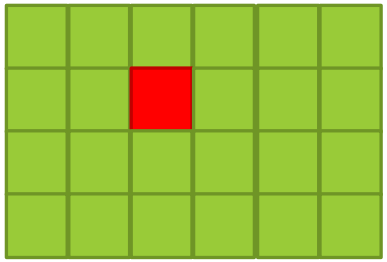
# Ejemplo: Multiplicación de Matrices



# Ejemplo: Multiplicación de Matrices

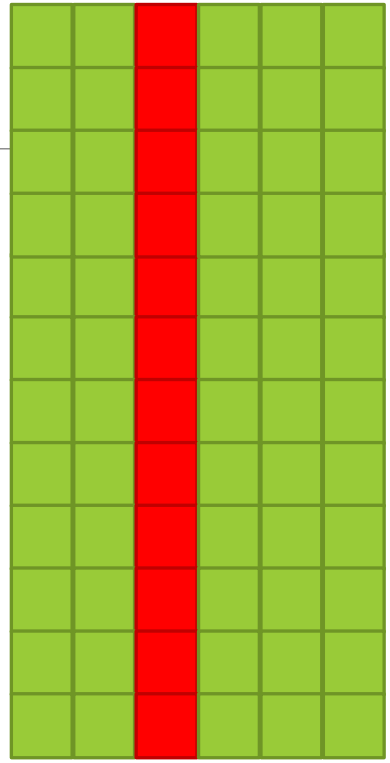


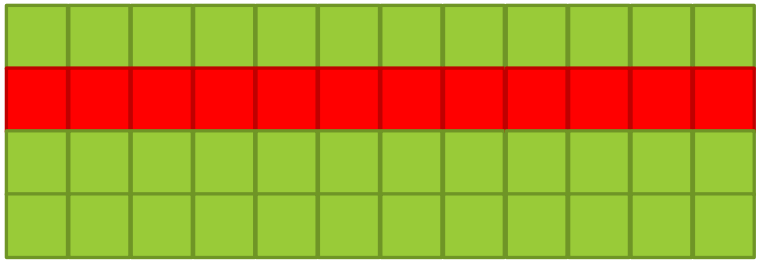


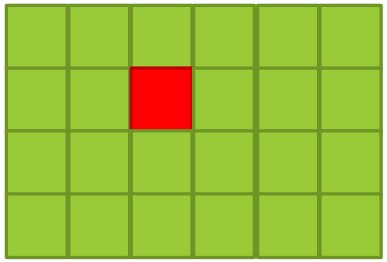


# Ejemplo: Multiplicación de Matrices

```
__global__ void kernel(int *A, int *B, int *C,
                       int N, int M, int L){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N*L){
        int col = tId % L;
        int row = tId / L;
        int value = 0;
        for (int i = 0; i < M; i++)
            value += A[i + row*M] * B[col + i*L];
        C[col + row*L] = value;
    }
}
```

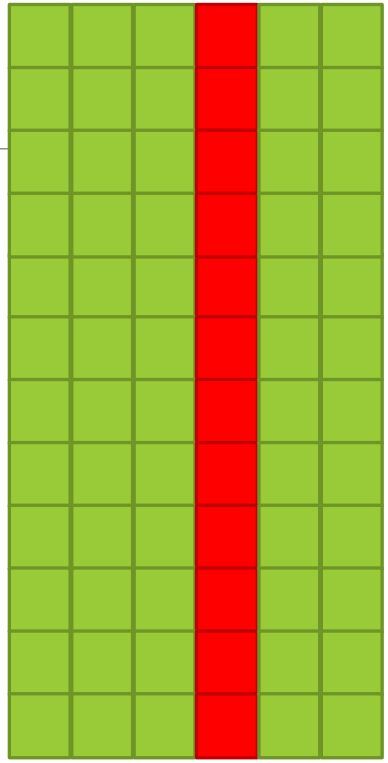


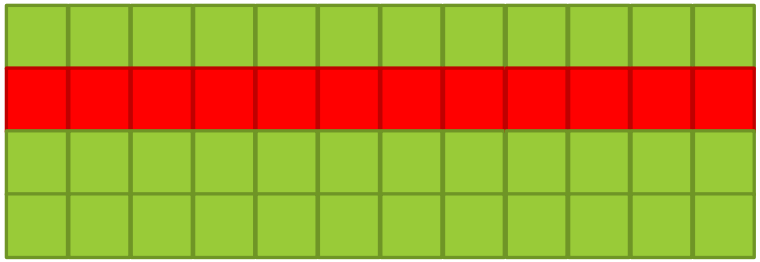


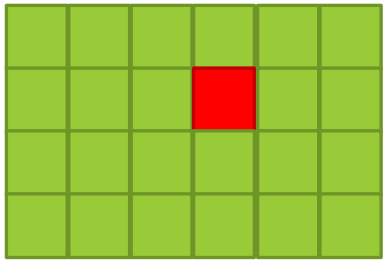


# Ejemplo: Multiplicación de Matrices

```
__global__ void kernel(int *A, int *B, int *C,
                       int N, int M, int L){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N*L){
        int col = tId % L;
        int row = tId / L;
        int value = 0;
        for (int i = 0; i < M; i++)
            value += A[i + row*M] * B[col + i*L];
        C[col + row*L] = value;
    }
}
```

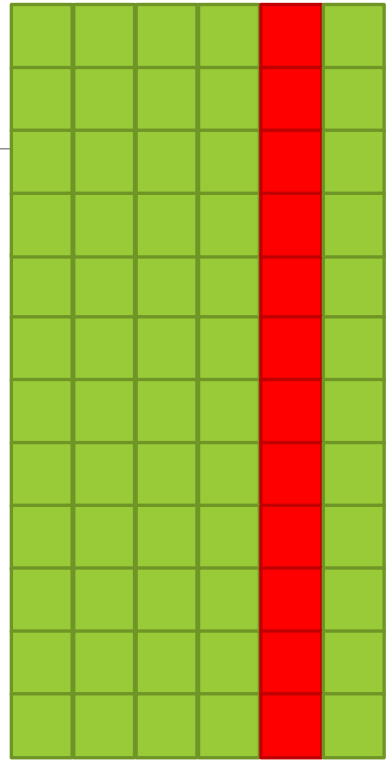


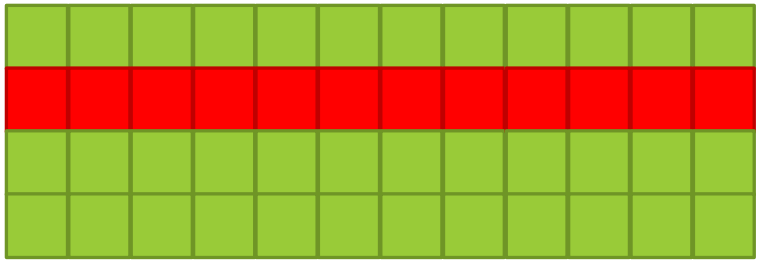


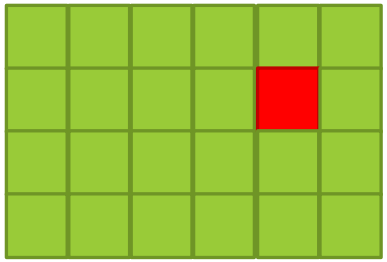


# Ejemplo: Multiplicación de Matrices

```
__global__ void kernel(int *A, int *B, int *C,
                       int N, int M, int L){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    if (tId < N*L){
        int col = tId % L;
        int row = tId / L;
        int value = 0;
        for (int i = 0; i < M; i++)
            value += A[i + row*M] * B[col + i*L];
        C[col + row*L] = value;
    }
}
```

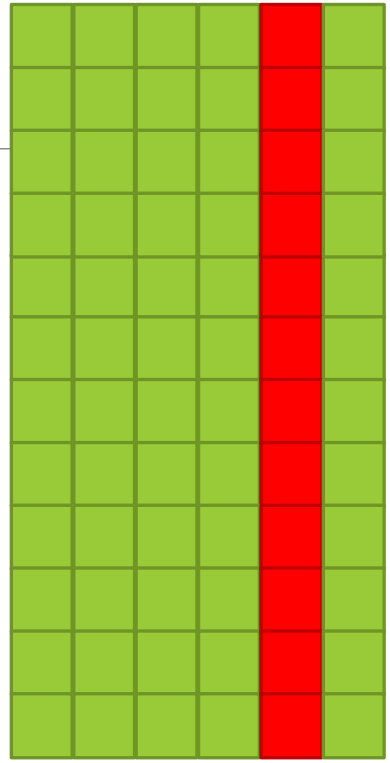


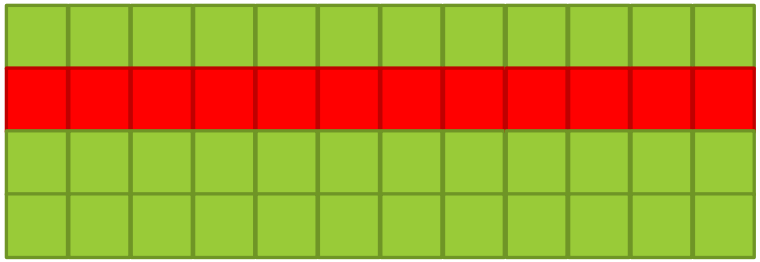


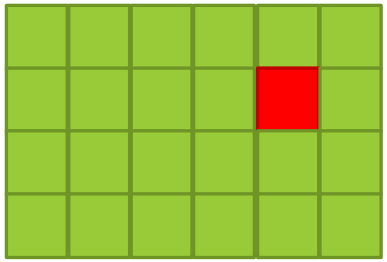


# Ejemplo: Multiplicación de Matrices

Cada elemento de $A$ será leído $L$ veces (número de columnas de $B$).

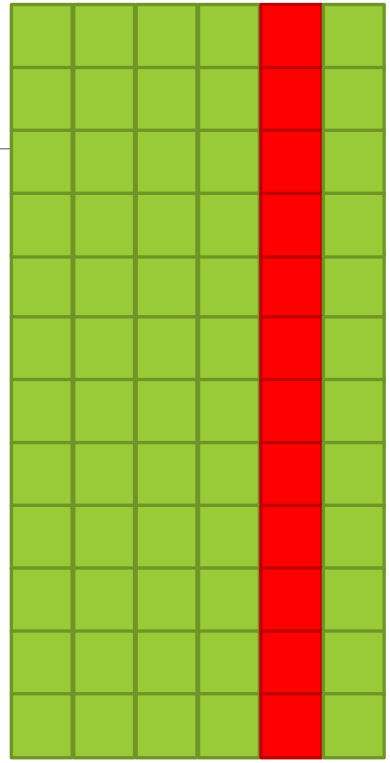


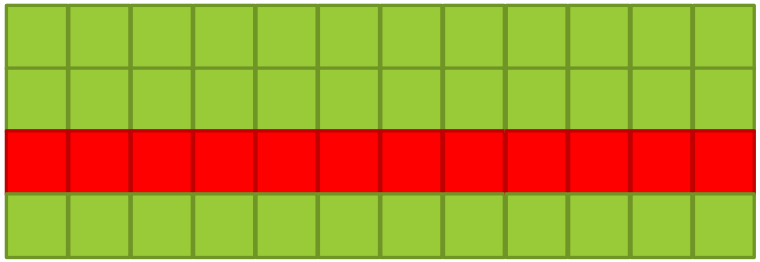


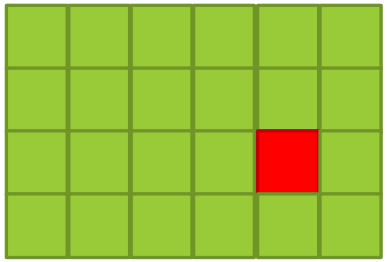


# Ejemplo: Multiplicación de Matrices

Cada elemento de $A$ será leído $L$ veces (número de columnas de $B$).



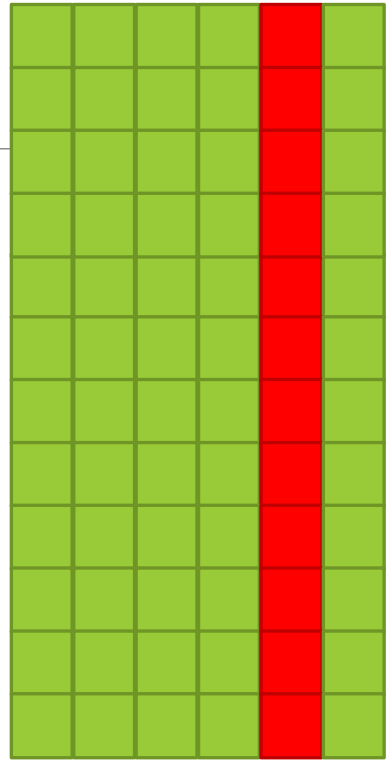


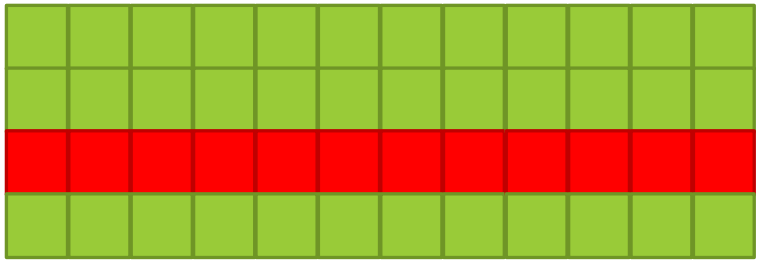


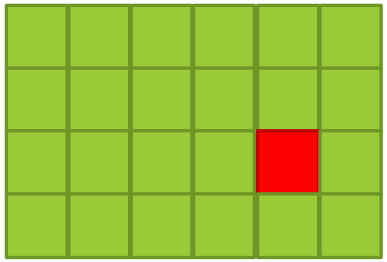
# Ejemplo: Multiplicación de Matrices

Cada elemento de $A$ será leído $L$ veces (número de columnas de $B$).

Cada elemento de $B$ será leído $N$ veces (número de filas de $A$).



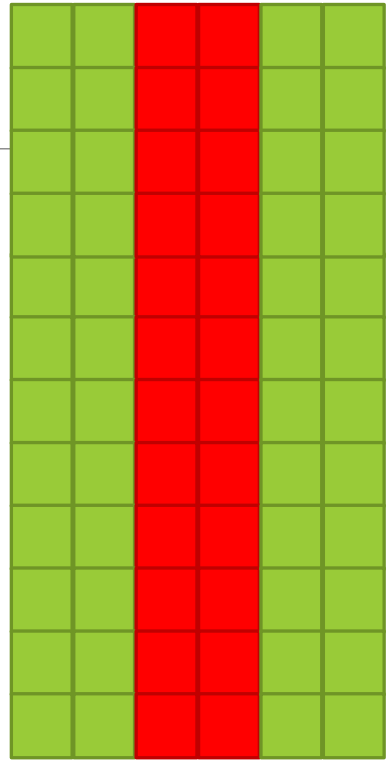


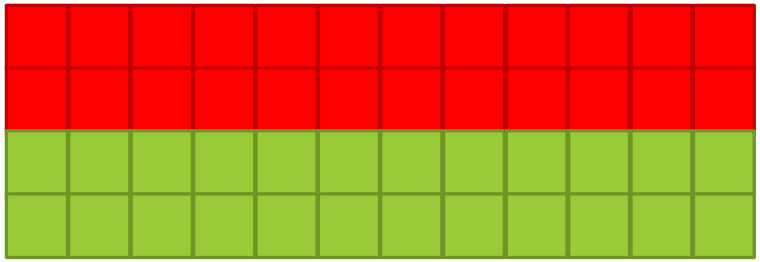


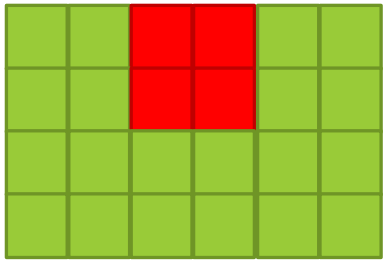
# Ejemplo: Multiplicación de Matrices

Ahora pensemos en submatrices (tiles) de tamaño $T \times T$ que dividen la matriz $C$.

Haremos que cada bloque se encargue de un tile, por lo que cada bloque tendrá $T \times T$ hebras.



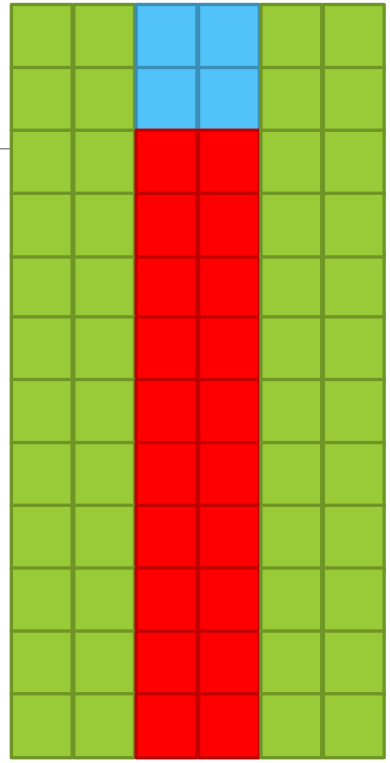


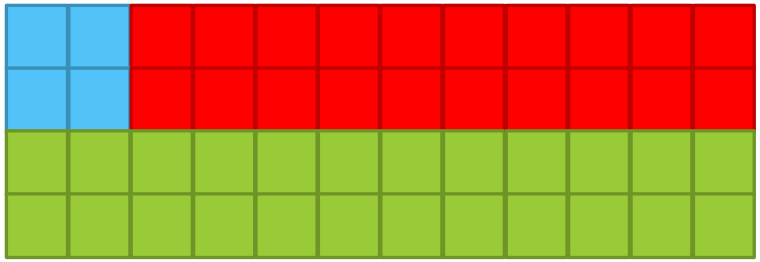


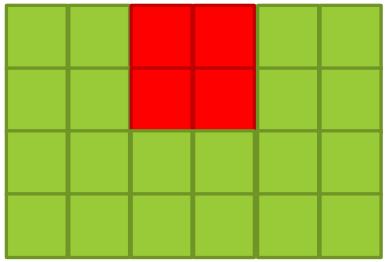
# Ejemplo: Multiplicación de Matrices

Cada bloque almacenará iterativamente en memoria compartida submatrices de $A$ y $B$.

En cada iteración, cada hebra del bloque obtendrá el “aporte” que esas submatrices realizan a la suma total.



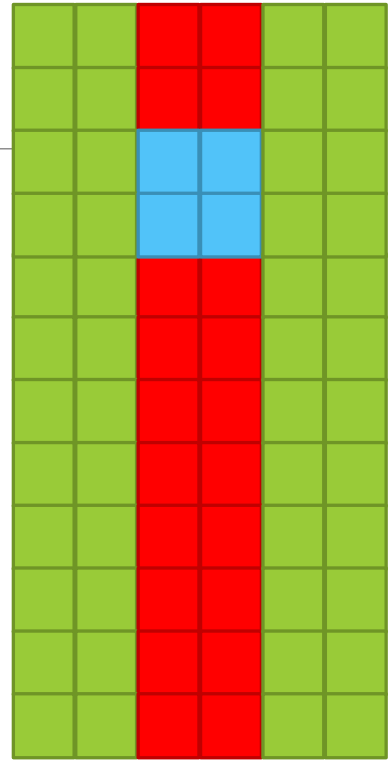


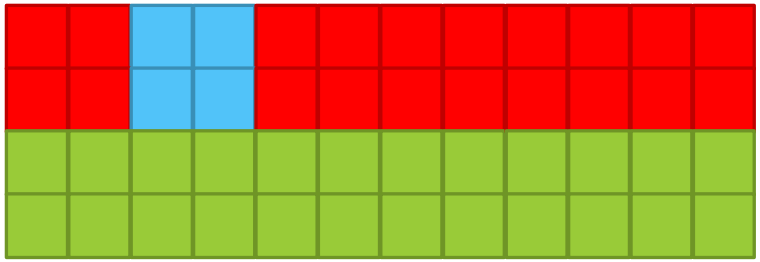


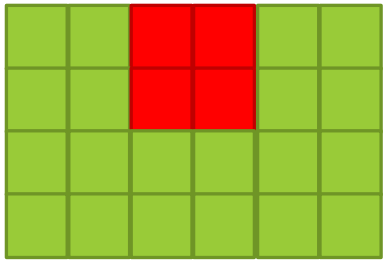
# Ejemplo: Multiplicación de Matrices

Cada bloque almacenará iterativamente en memoria compartida submatrices de $A$ y $B$.

En cada iteración, cada hebra del bloque obtendrá el “aporte” que esas submatrices realizan a la suma total.



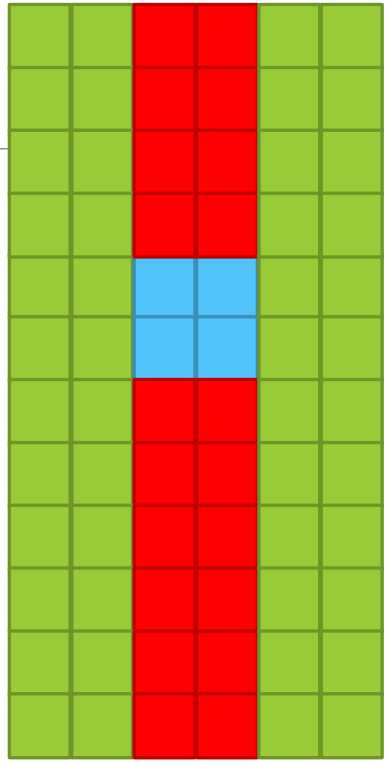


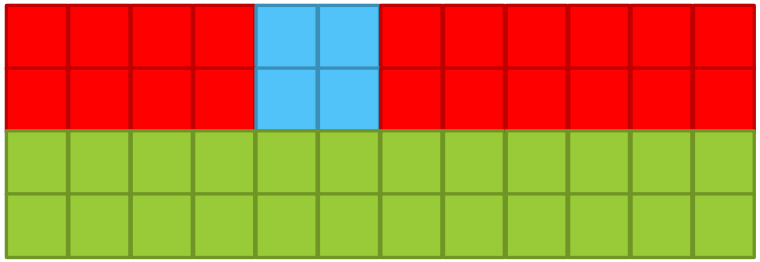


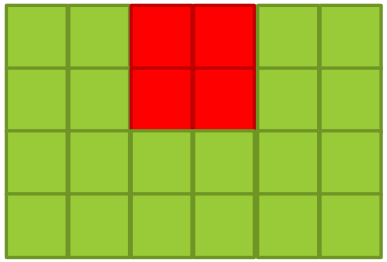
# Ejemplo: Multiplicación de Matrices

Cada elemento de $A$ será leído desde memoria global $L/T$ veces.

Cada elemento de $B$ será leído desde memoria global $N/T$ veces.







# Ejemplo: Multiplicación de Matrices

```
__global__ void kernel(int *A, int *B, int *C,
                       int N, int M, int L){
    int tId = threadIdx.x + blockIdx.x * blockDim.x;
    int col = tId % L, row = tId / L, value = 0;
    int Tcol = threadIdx.x % T;
    int Trow = threadIdx.x / T;
    __shared__ int As[T][T];
    __shared__ int Bs[T][T];
    for (int i = 0; i < M/T; i++){
        As[Trow][Tcol] = A[i*T + Tcol + (row)*M];
        Bs[Trow][Tcol] = B[col + (i*T + Trow)*L];
        __syncthreads();
        for (int j = 0; j < T; j++)
            value += As[Trow][j]
                     * Bs[j][Tcol];
        __syncthreads();
    }
    C[col + row*L] = value;
}
```