

# The Graphics Processor Unit (GPU) as a data parallel computing device

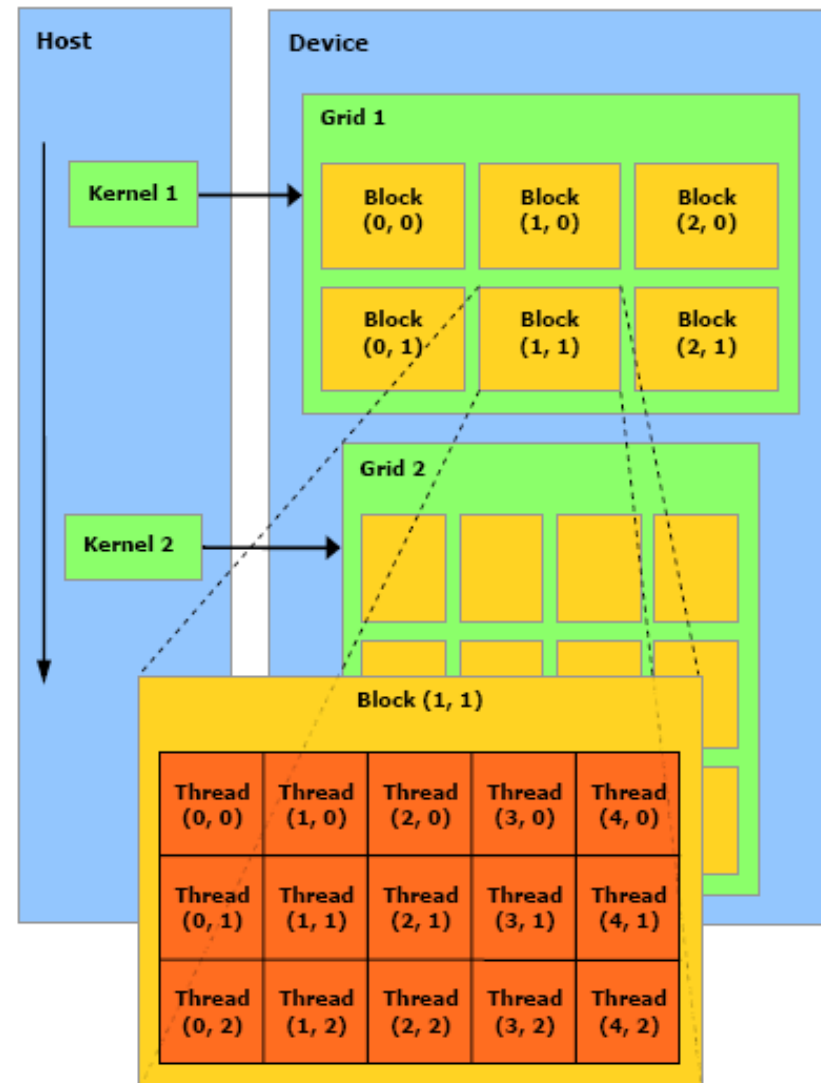
- Using GPUs for general-purpose computing is becoming increasingly popular, because of the cost of the device.
- Unfortunately, programming GPUs today is cumbersome (low productivity) as we will become evident below.
- The approach is to identify kernels of a program that can be programmed in SIMD form and have them executed by the GPU (device). the rest of the program is executed by the host processor.



- For NVIDIA devices, we can use the CUDA (Compute Unified Device Architecture) programming interface, a C extension.

## Overall organization of CUDA programs

- The GPU executes the kernels. Kernels have the form of a 2D grid of blocks with each block being a 2D or 3D grid of tasks.
- Blocks must be completely independent. Cannot communicate and synchronize.
- Conceptually, we can assume for now that blocks are executed one at a time and all threads within the block are executed together



# Identifying threads and blocks

- Can use 2 indices (x and y) in the case of blocks and 3 indices (x, y and z) in the case of threads.
- Can also use an id.
- For a two-dimensional block (grid of tasks) of shape  $(D_x, D_y)$ , the ID is  $x + y \times D_x$ , and for a three-dimensional grid of shape  $(D_x, D_y, D_z)$  is  $x + y \times D_x + z \times D_x \times D_y$ .
- Similarly for a grid of blocks.

# The code

- The code for each kernel is the same. **All threads** in a kernel execute the same code.
- This is called SPMD programming (single program multiple data). SPMD is typically used for the programming of distributed-memory multiprocessors (e.g. with MPI), but it can also be used to represent SIMD computation.
- In the case of CUDA, the code is mainly C. There are no array extensions.
- To achieve SIMD execution, subsets of the threads within a block are executed in lockstep. For now, let us assume that all threads execute in lockstep to simplify the discussion.



## An example

Consider the array operations:

$$C(1:n) = X(1:n) + 2$$

$$A(1:n) = B(1:n) + C(2:n+1)$$

Assume a grid with a single block containing a array of  $1 \times n$  threads. The code would look as follows if all threads were executed in lockstep:

```
int i = threadIdx.x // there is a copy of i for each thread
c[i] = x[i] + 2
a[i] = b[i] + c[i+1]
```

Here `threadIdx` is a built-in variable of type `dim3` which contains the thread index within the block. We only need the component `x` because we assume that the block contains a  $1 \times n$  array of threads.

**Please remember that the assumption that all threads execute in lockstep is not always true as we will see below.**



# The control problem

The interpretation of SPMD programs as a sequence of SIMD array operations is easy as long as each thread writes to a different memory location and all statements are assignment statements.

By proceeding in lockstep, each thread executes the same operation at each step. the operation could be a load from memory, an add of two registers, a store, or any other machine instruction.

What happens when there are control constructs such as `if` statements and `while` loops ?

To achieve the uniformity needed by SIMD computation, a program with control structures has to be translated to something that looks like a sequence of `where` statements.



# Examples of translation

Assuming again  $n$  threads in a block, consider

```
int i = threadIdx.x
if (a[i] > 0) {
    b[i] = 0;
    c[i] = 1;
} else {
    b[i] = 1;
    c[i] = 2;
}
```

This could be implemented as:

```
m(1:n) = a(1:n) > 0
where (m(1:n)) b(1:n) = 0
where (m(1:n)) c(1:n) = 1
where (.not.m(1:n)) b(1:n) = 1
where (.not.m(1:n)) c(1:n) = 2
```

or,

```
m(1:n) = a(1:n) > 0
if (any(m(1:n))) then
    where (m(1:n)) b(1:n) = 0
    where (m(1:n)) c(1:n) = 1
end if
if (.not.all(m(1:n))) then
    where (.not.m(1:n)) b(1:n) = 1
    where (.not.m(1:n)) c(1:n) = 2
end if
```

More efficient, sometimes.



## The code

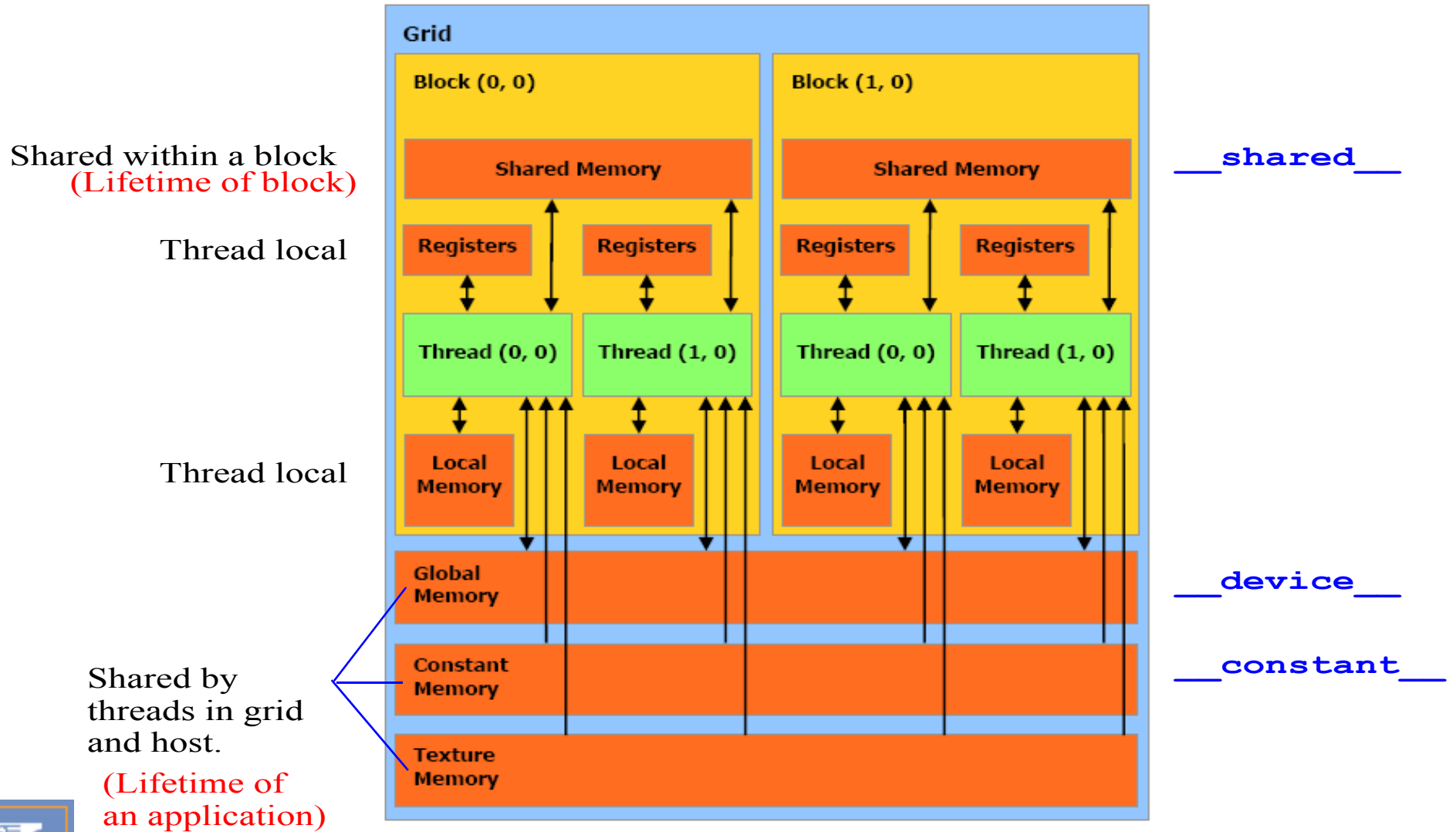
```
int i = threadIdx.x;
while (c[i] > 1) {
    c[i] /= 2;
    d[i] += 1;
}
```

## Can be translated to

```
m(1:n) = c(1:n) > 1;
do while (any(m(1:n)))
    where (m(1:n)) c(1:n) = c(1:n) / 2
    where (m(1:n)) d(1:n) = d(1:n) + 1
    m(1:n) = c(1:n) > 1;
end do
```



# Memory



## A. Language extensions

1. Function type qualifiers are placed before the declaration of a C function.
  - `__global__` function implements a kernel. Callable  
from host only. Example:

```
__global__ void KernelFunc(...);  
dim3 DimGrid(100, 50); // 5000 blocks of threads  
dim3 DimBlock(4, 8, 8); // 256 threads per block  
size_t SharedMemBytes = 64; // 64 bytes of shared memory  
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```

Notice that a `__global__` function must be called with `<<<...>>>` parameters to specify shape of kernel and amount of shared memory **dynamically allocated** per block.

- `__device__` function is executed in device and callable  
from device only (i.e. a `__global__` or another `__device__`  
function).



## 2. Built-in variables

- `dim3 gridDim` Dimensions of the grid.
- `uint3 blockIdx` Block index.
- `dim3 blockDim` Dimensions of the block.
- `uint3 threadIdx` Thread index within block.



## B. Runtime library

It has three parts

- A host component (runs on host)
- A device component (runs on device)
- A common component. built-in types and subset of C standard library that runs on both host and device.



## B.1 Common components

- Built-in vector types such as `char1`, `char2`, `char3`, `char4`, ...  
`int1`, `int2`, `int3`, `int4`,... `float1`, `float2`, `float3`,  
`float4` components are `x`, `y`, `z`, and `w`.
- `dim3` is another built-in data type derived from `uint3`.  
Unspecified components are 1.



## B.3 Host runtime component

Two APIs, one low level (*driver API*) and the other high-level (*runtime API*). Only the latter discussed.

- Memory allocation in the device

`cudaMalloc()` allocates an object in the global memory of the device and `cudaFree()` frees it.

```
float* x;  
cudaMalloc((void**)&x, 256*sizeof(float));  
...  
cudaFree(x);
```

- Host-device data transfer

`cudaMemcpy()` is used for memory data transfer. It accepts four parameters: destination (pointer), source (pointer), number of bytes, type of transfer.

```
float* y=new float[256];  
cudaMemcpy(x,y,256*sizeof(float),cudaMemcpyHostToDevice);  
...  
cudaMemcpy(y,x,256*sizeof(float),cudaMemcpyDeviceToHost);
```



# Execution model

As announced above, the GPU does not execute the threads in fully parallel form.

It divides the blocks into groups and executes one group at a time. the size of each group of block is determined by the memory requirements (which is why one must specify size of dynamically allocated memory when a `__global__` function is invoked).

Within a block, threads are divided into *warps* of size 32 (today). A thread scheduler periodically switches from one *warp* to another in an undefined order. Each warp contains threads with consecutive IDs.

Unfortunately, with *warps* come *races*.

**Definition:** We say there is a race if two threads may access the same memory location, at least one of the accesses is a write to memory, and there is no guaranteed order of execution for the two accesses.





## Examples of races

There are two types of races depending on the type of accesses:

1. Read-Write,
2. Write-Write.

An example of read-write was given above:

```
int i = threadIdx.x // there is a copy of i for each thread
c[i] = x[i] + 2
a[i] = b[i] + c[i+1]
```

To see why there is a race here, consider threads 32 and 33:

Thread 32:

```
c[31] = x[31] + 2
a[31] = b[31] + c[32]
```

Thread 33:

```
c[32] = x[32] + 2
a[32] = b[32] + c[33]
```

Since these two threads are part of different warps, the order of execution of the accesses `c[32]` to is not guaranteed and therefore there is a race since one of the accesses is a write.



## Avoiding races

To avoid races, we can use the device runtime function `__syncthreads()` which serves as a barrier. That is all threads must reach a particular point of invocation of this function before execution may continue.

To avoid deadlocks, it is necessary that all threads invoke this function at the same time. In other words, if the invocation appears in a conditional all threads in a block must follow the same control path.

Also, the function should only be inserted where necessary. For example, in the code

```
int i = threadIdx.x // there is a copy of i for each thread
c[i] = x[i] + 2
a[i] = b[i] + c[i]
```

There is not need to insert an invocation to `__syncthreads()` while it is needed in the code below

```
int i = threadIdx.x // there is a copy of i for each thread
c[i] = x[i] + 2
__syncthreads()
a[i] = b[i] + c[i+1]
```



# Memory Bandwidth

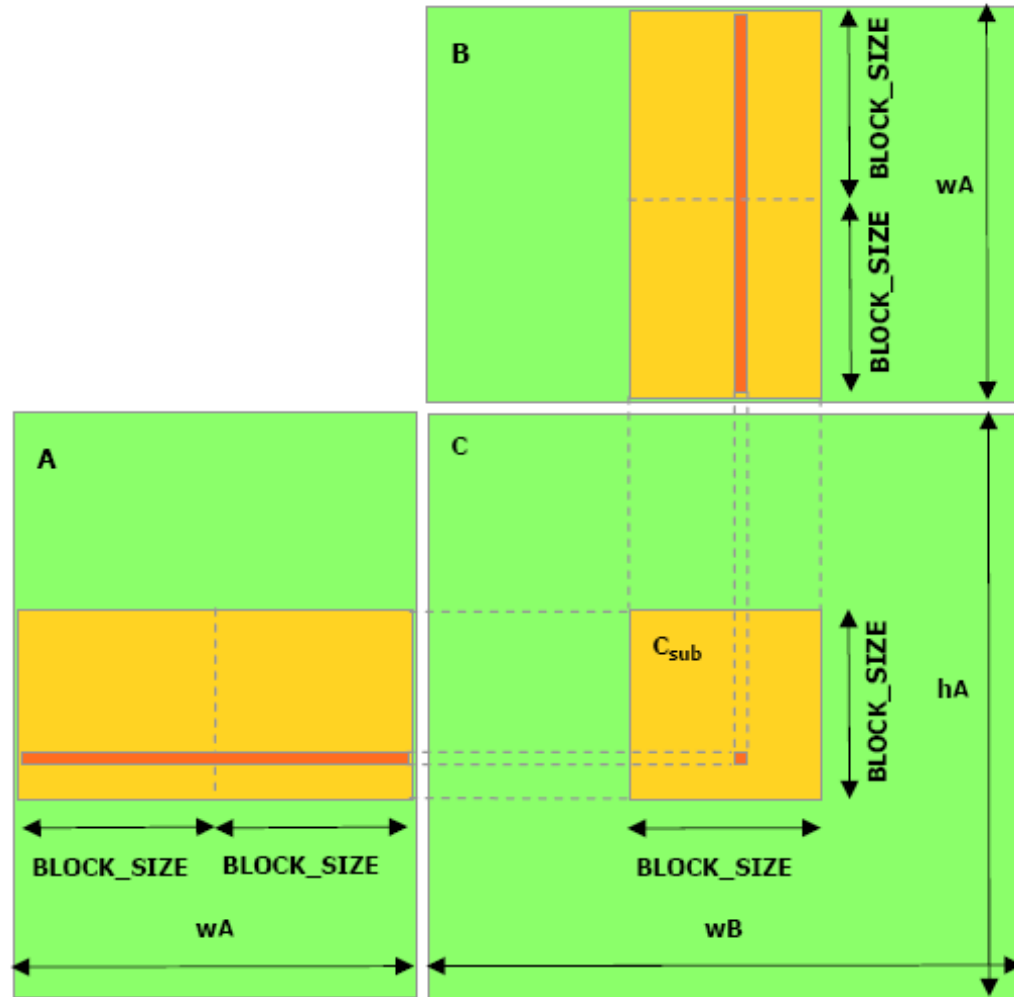
The global memory space is not cached.

With the right access pattern we can improve the memory bandwidth by arranging the global memory addresses simultaneously accessed by each thread of a half warp so that they together form a contiguous block. (see the NVIDIA CUDA Programming Guide manual for details).

The shared memory consist (today) of 16 banks. Data must be organized so that each half warp can access the 16 banks without conflicts as discussed earlier.



# Example code: Matrix-matrix multiplication



Each thread block computes one sub-matrix  $C_{\text{sub}}$  of C. Each thread within the block computes one element of  $C_{\text{sub}}$ .

```

#define BLOCK_SIZE 16
// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);
// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;
    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);
    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);
    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);
    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);
    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);
    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}

```



```

// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB,
float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;
    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;
    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;
    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;
    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;
    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;
    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;
    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
        a <= aEnd;
        a += aStep, b += bStep) {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

```

```

// Load the matrices from global memory to shared memory;
// each thread loads one element of each matrix
As[ty][tx] = A[a + wA * ty + tx];
Bs[ty][tx] = B[b + wB * ty + tx];
// Synchronize to make sure the matrices are loaded
__syncthreads();
// Multiply the two matrices together;
// each thread computes one element
// of the block sub-matrix
for (int k = 0; k < BLOCK_SIZE; ++k)
    Csub += As[ty][k] * Bs[k][tx];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}
// Write the block sub-matrix to global memory;
// each thread writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;

```

