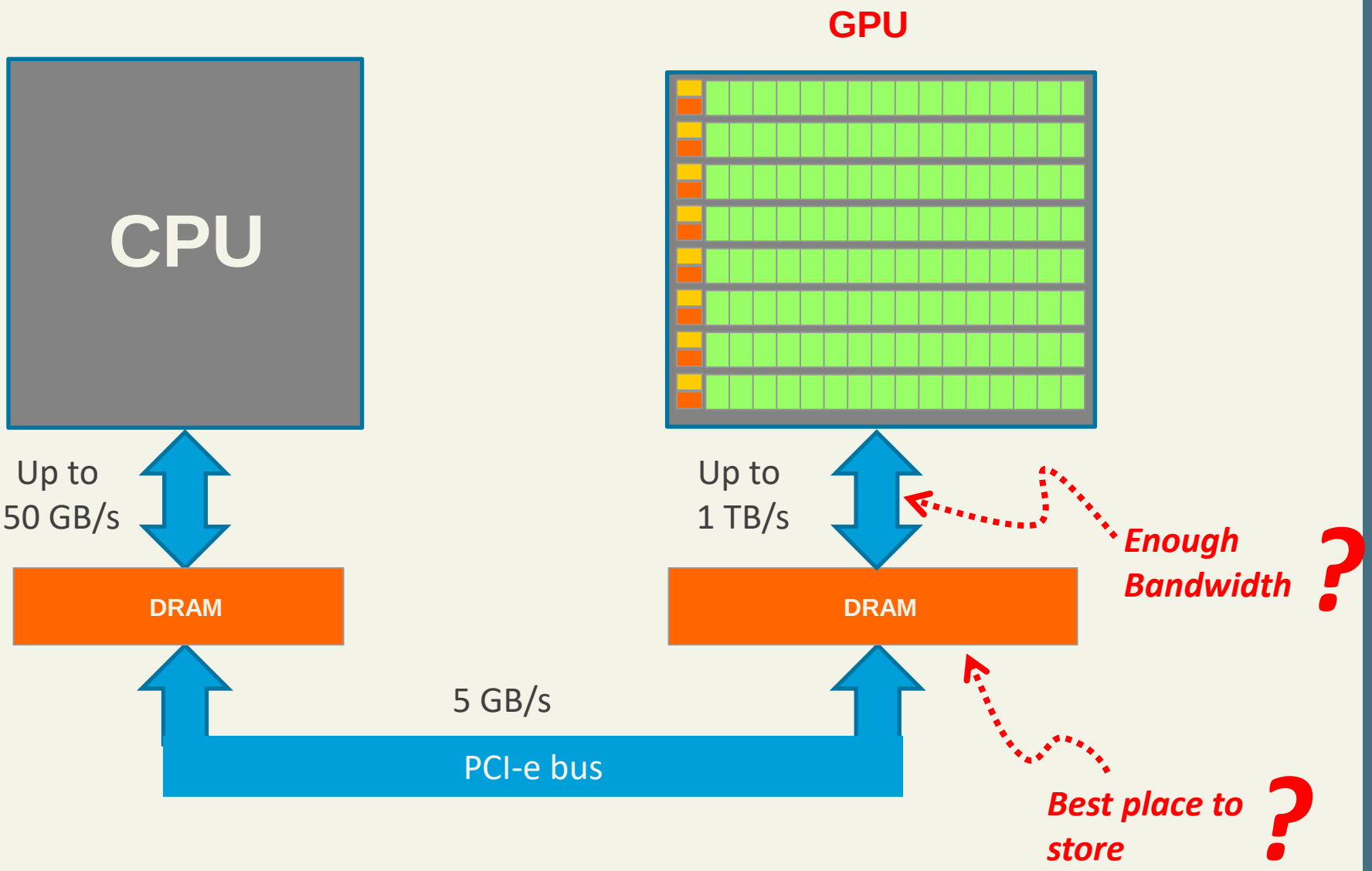
COSC 407
Intro to Parallel Computing

CUDA Memories and Performance Revisited

Enough Bandwidth?



Measuring Performance: Timing

Timing can be done using CPU timers or GPU timers

(1) Using CPU timers:

```
cudaMemcpy(...);

double t = clock();
kernel<<<..,..>>>(..);
cudaDeviceSynchronize();     //block host till kernel finishes
t = 1000 * (clock()-t) / CLOCKS_PER_SEC;    //milliseconds

cudaMemcpy(...);
```

Measuring Performance: Timing

(2) Using GPU timers

```
cudaEvent_t start, stop;      //create two events
cudaEventCreate(&start);
cudaEventCreate(&stop);

cudaMemcpy(...);

cudaEventRecord(start);       //record start event
kernel<<<..,..>>>(..);
cudaEventRecord(stop);        //record stop event
cudaEventSynchronize(stop);   //block host till 'stop' is recorded
cudaMemcpy(...);

float time = 0;
cudaEventElapsedTime(&time, start, stop); //time in milliseconds
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Measuring Performance: Bandwidth (BW)

- ***Bandwidth (BW)*** - the rate at which data can be transferred from/to GPU global memory.
- ***Theoretical Bandwidth***:
  - Can be calculated based on GPU specifications
  - **Example**: *Tesla*: clock rate=1.85 GHz, memory interface width=384 bit, thus

    BW = (1.85 G) x (384/8 B) x (2 for double data rate) = **177.6** GB/s
- ***Effective Bandwidth*:**
  - Calculated for **specific program**

$$BW_{Effective} = (R_B + W_B) / time$$

R_B (or W_B) is # of bytes read (or written) per kernel during `time`

  - **How?** Time your kernel. Compute the effective bandwidth based on the amount of data the kernel reads and writes per unit of time.
  - **Example**: for 1024 x 1024 float matrix copy, R_B=W_B = 1024^2 x 4 Bytes

$$BW_{effective} = 2 \times 1024^2 \times 4 / time$$

CGMA

***C**ompute to **G**lobal **M**emory **A**ccess**: *number of floating point calculations performed for each access to the global memory*

max computation = CGMA * (Bandwidth / #bytes_per_memory_access)

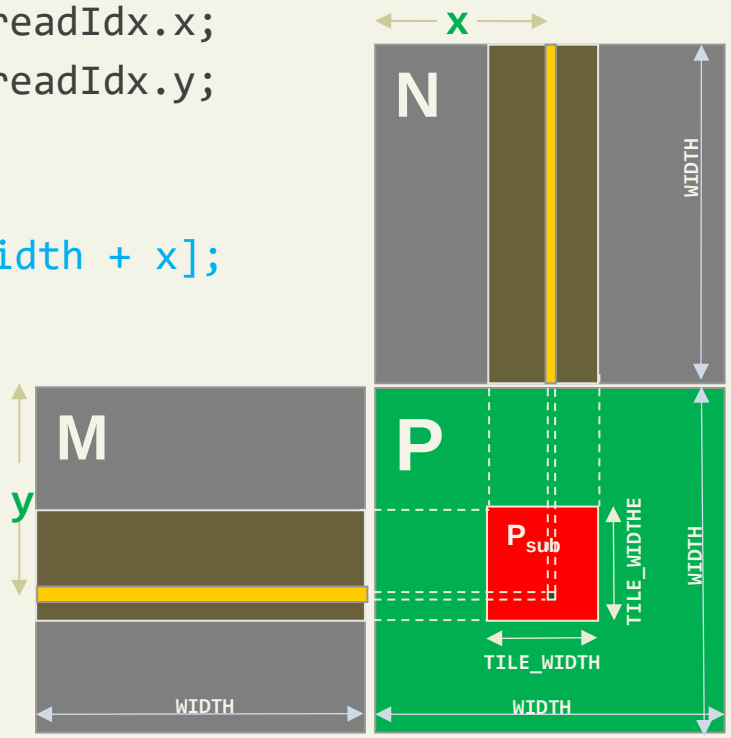
$$\text{CGMA} = \frac{\max computation}{\max number\ of\ memory\ transactions}$$

- Units: (bandwidth/#bytes−per−mem−access)
- The highest achievable floating-point calculation throughput is limited by the rate at which the input data can be loaded from the global memory.

Example

Consider Matrix Multiplication code

```
__global__ void MatrixMul(float* M, float* N, float* P, int width) {
    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;
    float value = 0;
    for (int k = 0; k < width; k++)
         value += M[y*width + k] * N[k*width + x];
    P[y*width+x] = value;
}
```



In the for loop, we have 2 Floating Point operations (FLOP) and 2 memory accesses

- Let's compute their ratio:

CGMA ratio = 1.0

***Compute to Global Memory Access**: number of floating point calculations performed for each access to the global memory*

Example

Problem!

- **Let's say** the global memory bandwidth = ***n*** GB/s
- Each access of a float is 4 Bytes.
- CGMA = 1.0 means each FLOP requires reading one float from global memory. This means GPU can only perform **n**/4 FLOPs per sec.
  - Example:
    - G80: Memory Bandwidth: 86.4 GB/s → only **21.6** GFLOPS
    - G80 processing rate is > **370** GFLOPS
    - This means, G80 will run at only 6% of its power *due to* the LOW CGMA

- *The lower CGMA the lower the performance*

- **AIM**: we should always try to ***increase CGMA***
  - i.e. reduce the number of general memory accesses with respect to the number of FLOPs.

Question

Rank the memory operations from **Fastest** to **Slowest**

A. memory, shared memory, local memory, registers
B. Registers, shared memory, local memory, global memory
C. Local memory, registers, shared memory, global memory
D. Registers, local memory, shared memory, global memory
E. They all have the same speed

Question

Which of the following statements is True?

A. Threads in the same block can access a variable in that block's shared memory
B. Threads in the same SM can access a variable in the global memory
C. Threads in the same block have their private variables in their registers
D. Threads in the same block can access each other's results stored in the block shared memory and in the global memory
E. All of the above are true

Question

What is the order of statements from fastest to slowest (assuming the compiler does not do any kind of code optimization)?

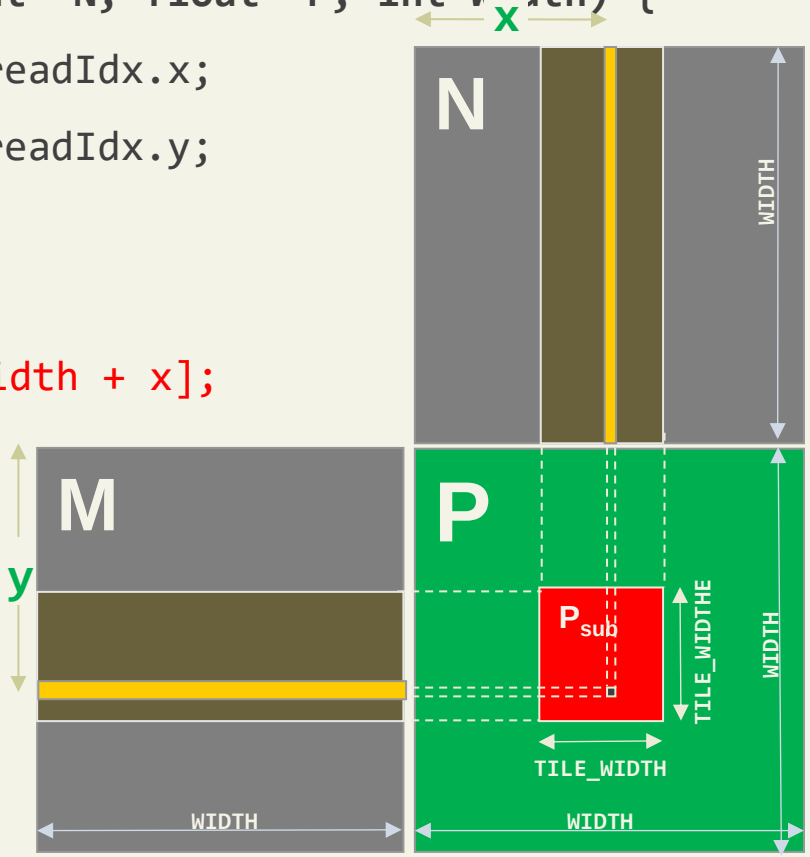
```
__global__ void amethod(int* arr, int x){
        __shared__ int sh1 = 8 , sh2;
        int y, z;
        y = x;                  //1
        arr[0] = arr[1];        //2
        sh2 = sh1;              //3
        z = arr[10];            //4
}
```

Register	very fast
Local	slow
Shared	fast
Global	slow
Constant	fast

a) 1, 4, 3, 2
b) 3, 1, 4, 2
c) 1, 3, 4, 2
d) 3, 4, 1, 2
e) 4, 1, 2, 3

Poor Performance!

```
__global__ void MatrixMul(float* M, float* N, float* P, int width) {
    int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
    int y = blockIdx.y * TILE_WIDTH + threadIdx.y;
    float value = 0;
    for (int k = 0; k < width; k++)
        value += M[y*width + k] * N[k*width + x];
    P[y*width+x] = value;
}
```



Why poor performance again?

CGMA = 1.0 which limits GFLOPS.

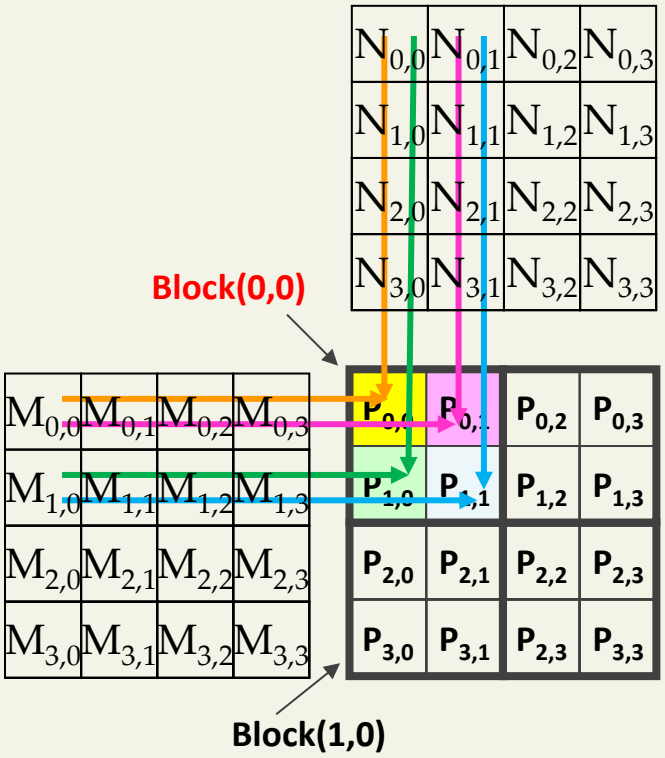
- Actual ~22 GFLOPS
- Card can do > 370 GFLOPS

Need to drastically cut down global memory access to improve.

Example:
Global Memory Access by Threads in Block (0,0)

Let's assume TILE_WIDTH = 2. We notice That **Each element** of M and N **is used exactly twice within each block.**

```
//each thread(y,x) runs this loop to compute P(y,x)
for (int k=0; k<width; k++)
        value += M[y*width + k] * N[k*width + x];
```



$P_{0,0} = M_{0,0}*N_{0,0} + M_{0,1}*N_{1,0} + M_{0,2}*N_{2,0} + M_{0,3}*N_{3,0}$

time

$thread_{0,0}$ $P_{0,0}$	$thread_{1,0}$ $P_{1,0}$	$thread_{0,1}$ $P_{0,1}$	$thread_{1,1}$ $P_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{1,0} * N_{0,0}$	$M_{0,0} * N_{0,1}$	$M_{1,0} * N_{0,1}$
$M_{0,1} * N_{1,0}$	$M_{1,1} * N_{1,0}$	$M_{0,1} * N_{1,1}$	$M_{1,1} * N_{1,1}$
$M_{0,2} * N_{2,0}$	$M_{1,2} * N_{2,0}$	$M_{0,2} * N_{2,1}$	$M_{1,2} * N_{2,1}$
$M_{0,3} * N_{3,0}$	$M_{1,3} * N_{3,0}$	$M_{0,3} * N_{3,1}$	$M_{1,3} * N_{3,1}$

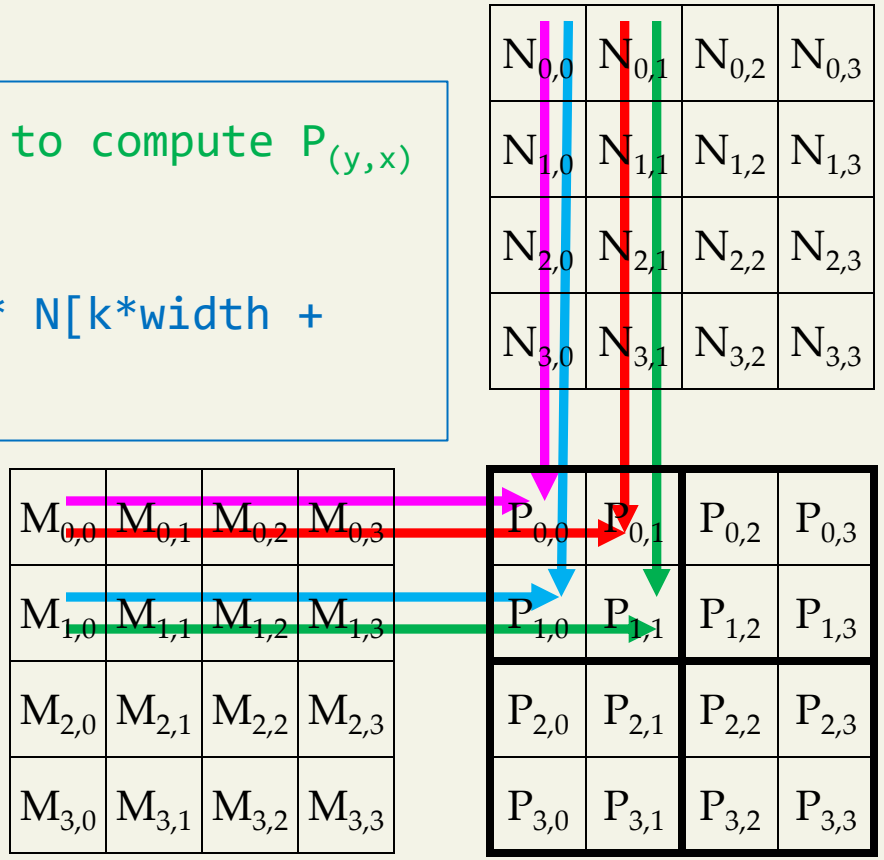
Table: global memory access by threads in Block(0,0

Example:
Global Memory Access by Threads in Block (0,0)

That means in general, each item in M or N is accessed TILE_WIDTH times by the threads within the same block

- In our current example, each element is accessed twice by the threads in Block (0,0)

```
//each thread(y,x) runs this loop to compute P(y,x)
for (int k=0; k<width; k++)
        value += M[y*width + k] * N[k*width +
x];
```



Question

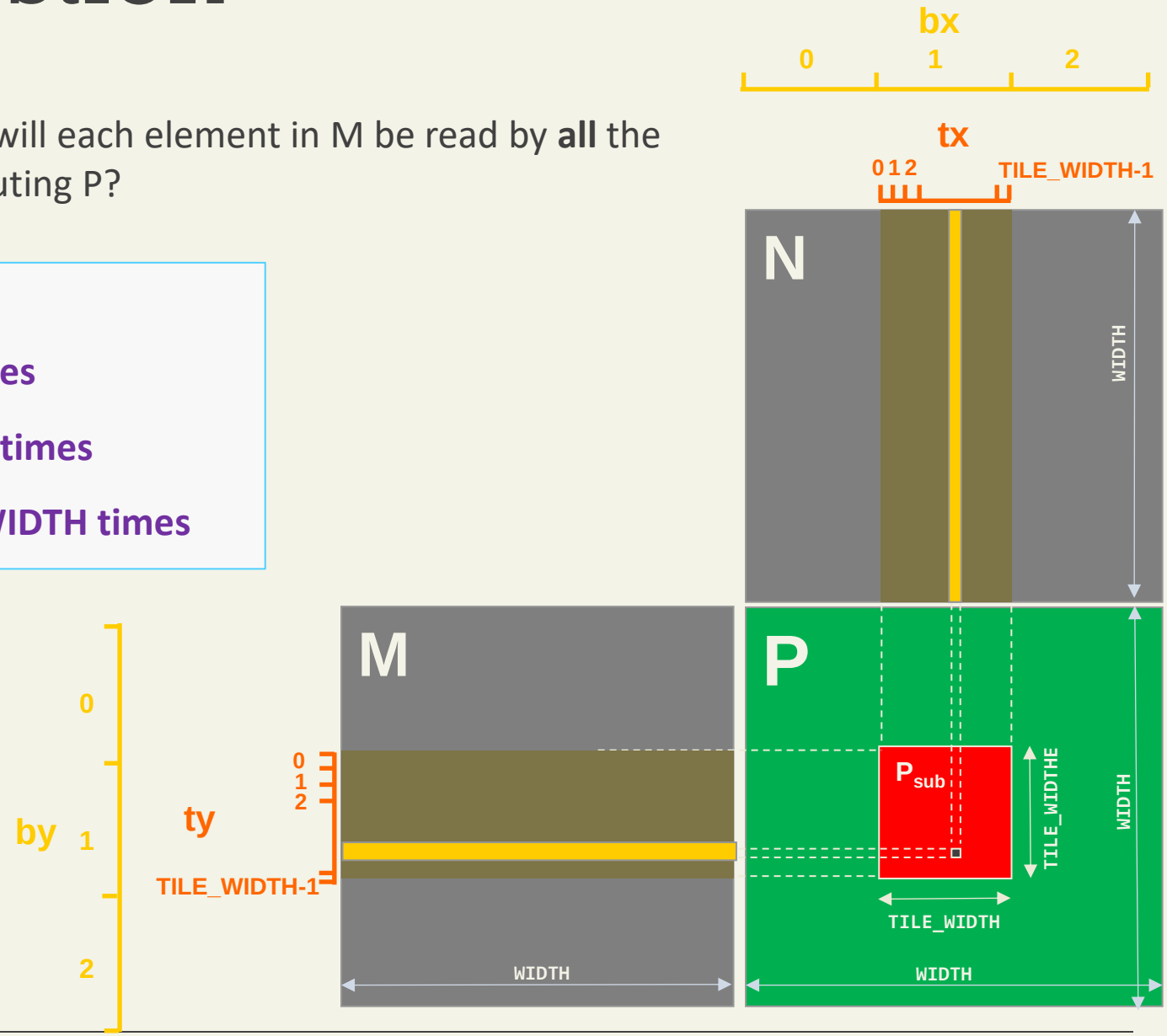
How many times will each element in M be read by **all** the threads for computing P?

a) 2 times

b) WIDTH times

c) 2 x WIDTH times

d) WIDTH x WIDTH times

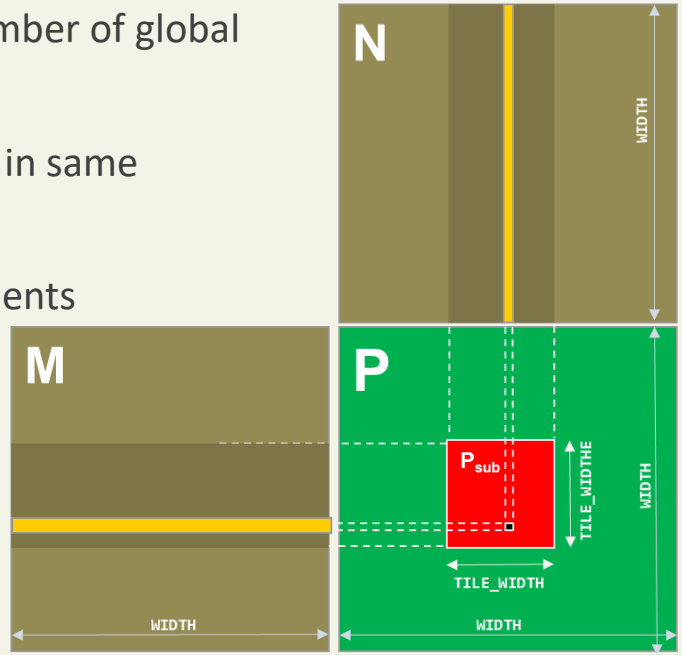


Using Fast Memories for Matrix Multiplication

Idea: to improve performance, put M and N in a fast memory.

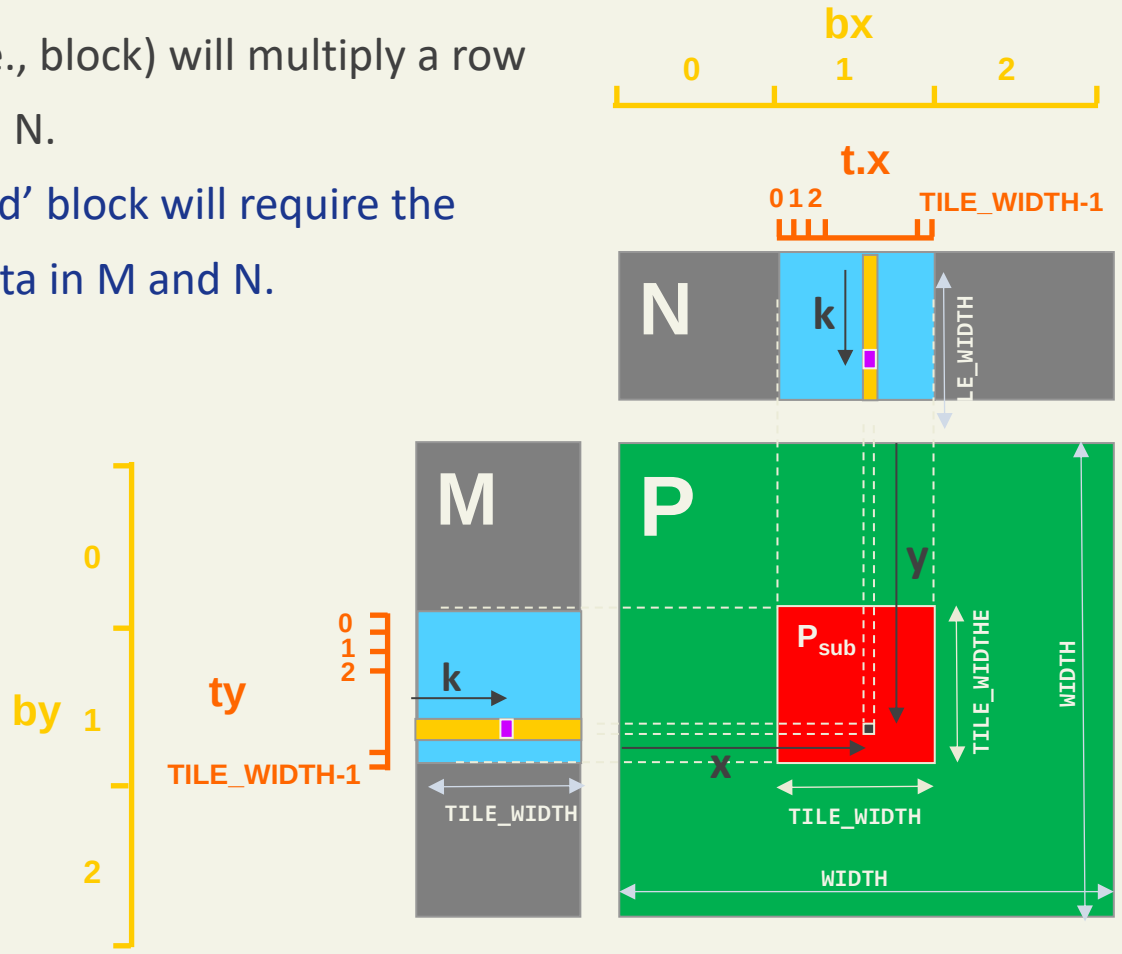
Fast memories we have:

- Registers
  - Problem 1: not enough register
  - Problem 2: Registers are private per thread. This means, we will need to copy the same M value for every thread – this doesn't solve the problem since we will have many global memory reads!
- Shared
  - Shared by all threads in same block – reduces number of global memory reads.
    - Copy M element to shared, then all threads in same block share this value
  - Problem: not enough room for ALL M and N elements
  - Solution: use Tiling
- ***Tiling***: put the data required by the current block in shared memory so that all threads use them



Using Shared Memory for Matrix Multiplication

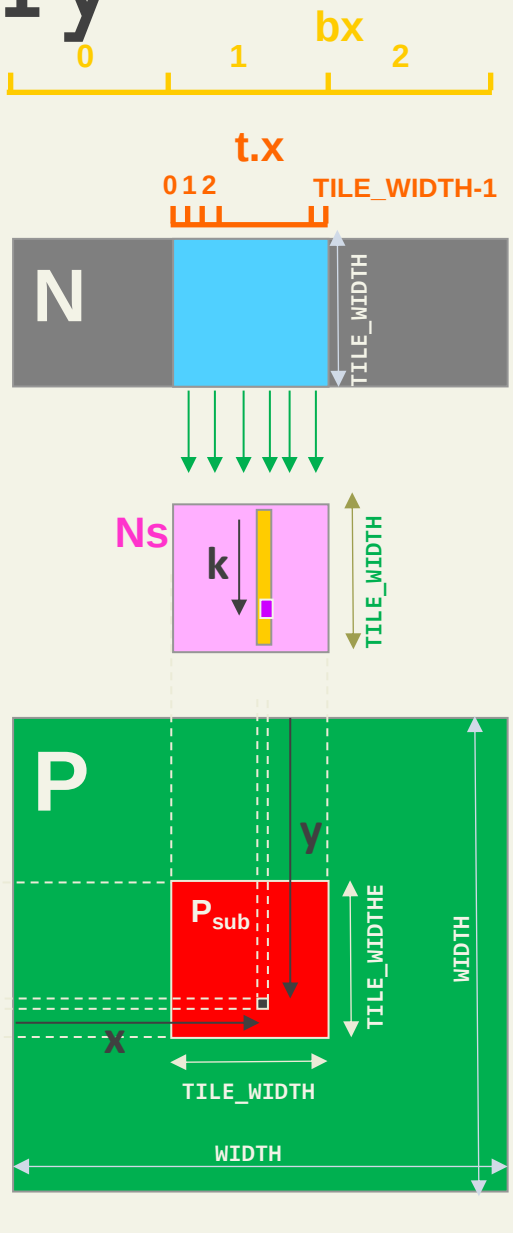
- **For simplicity**, let's start by assuming the width of M and the height of N ***are equal to*** TILE_WIDTH.
  - TILE_WIDTH = blockDim.x = blockDim.y
    - Each thread in the tile (i.e., block) will multiply a row from M by a column from N.
      - All threads in the 'red' block will require the same 'blue' input data in M and N.



Using Shared Memory

KEY IDEA:

1. Block loads **data** from **global memory** to **shared memory**
2. Synchronize threads
3. Threads work on data from shared memory in parallel.
4. Block writes **data** back from **shared** to **global memory**.

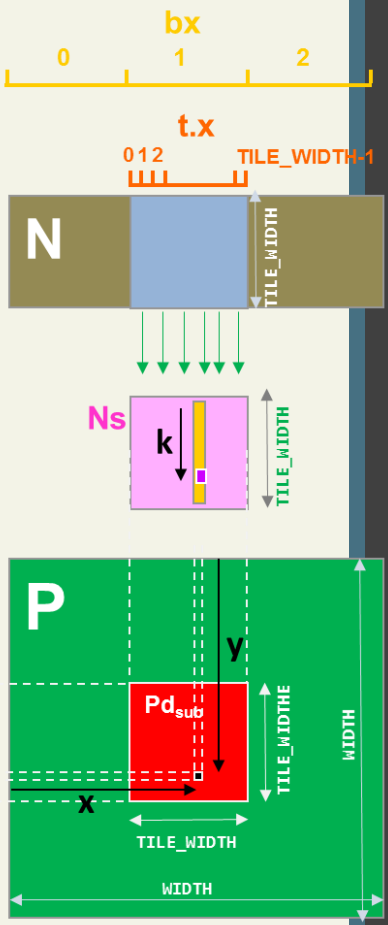


Tiled Multiplication: Kernel Code

```
_global_ void MatrixMul(float* M, float* N, float* P, int Width){
  __shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Ns[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;     int by = blockIdx.y;
  int tx = threadIdx.x;    int ty = threadIdx.y;

  // Identify row and column of P element to work on
  int y = by * TILE_WIDTH + ty;
  int x = bx * TILE_WIDTH + tx;

  // every thread loads one piece of data into shared memory
  Ms[ty][tx] = M[y * Width + tx)];
  Ns[ty][tx] = N[ty * Width + x];
  __syncthreads();  //sync threads within same block
  float value = 0;
  for (int k = 0; k < TILE WIDTH; k++)
      value += Ms[ty][k] * Ns[k][tx];
  P[y*Width + x] = value;
}
```



Q: should we also put P in shared memory first?

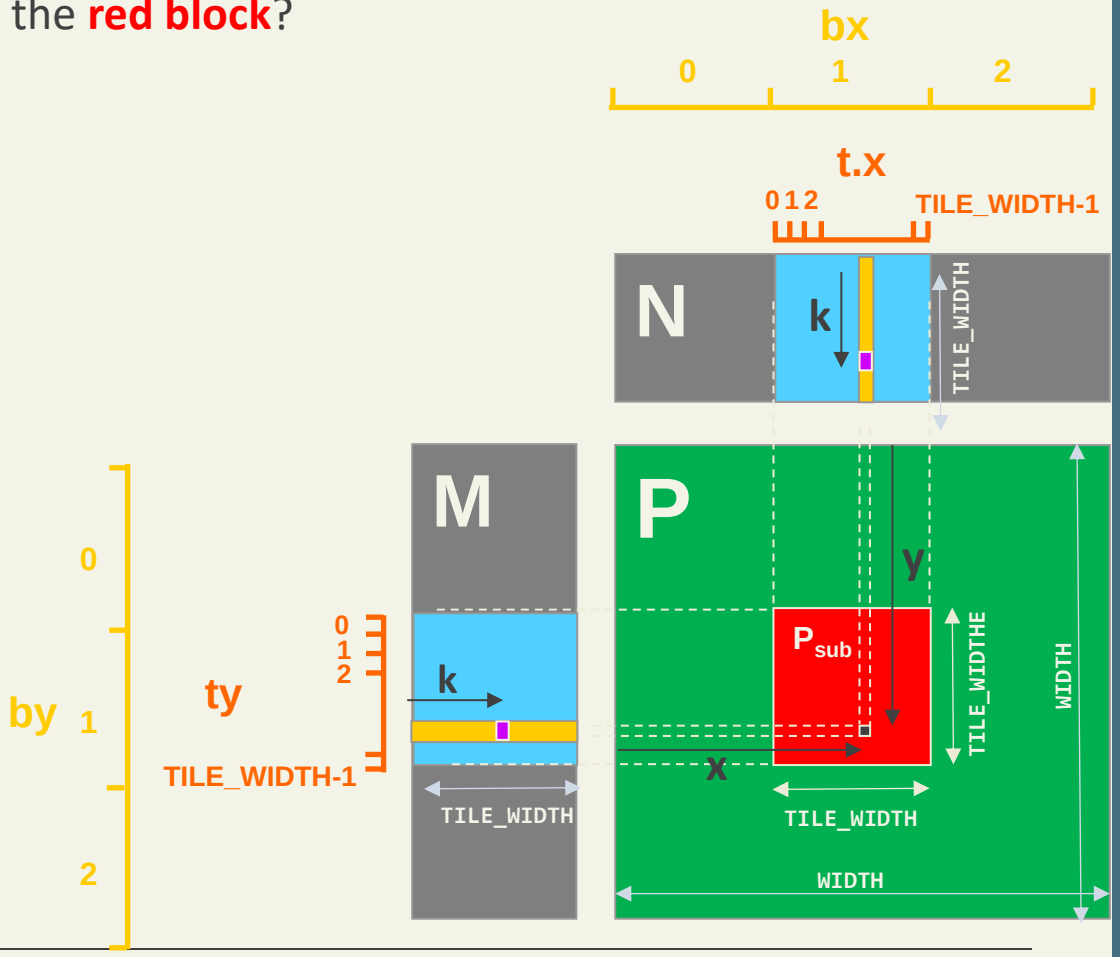
Q2: should we merge these two lines? i.e.use P[]+=Ms[][]*Ns[][];

The code assumes that x and y are < WIDTH. If not, use if(x<WIDTH && y<WIDTH)

Question

- If we use our MatrixMul program without tiling, how many times is each elements in **M** or **N** accessed by the **red block**?

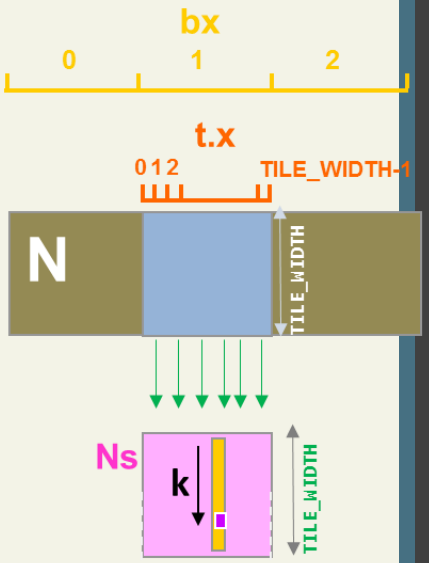
A. Once
B. Twice
C. TILE_WIDTH times
D. WIDTH times
E. Something else

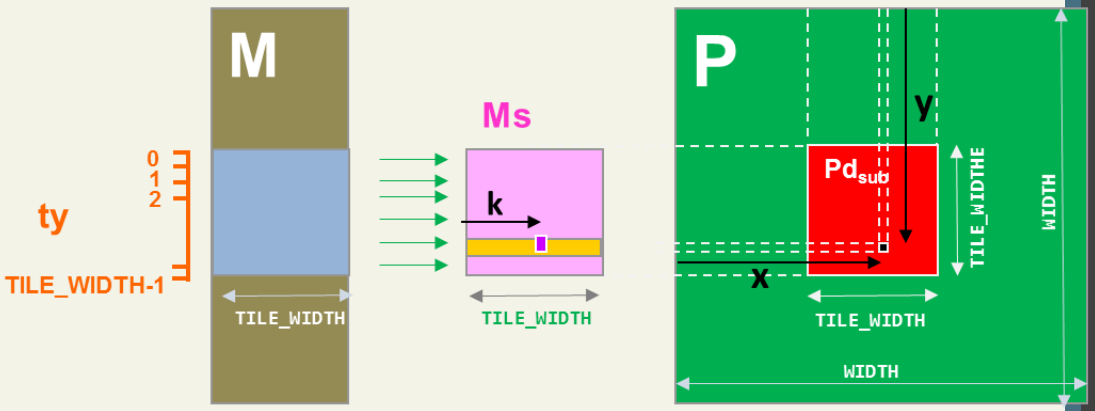


Question

- If we use our MatrixMul program for the given diagram, how many times is each elements in **M** or **N** accessed by the **red block**?

A. Once
B. Twice
C. TILE_WIDTH times
D. WIDTH times
E. Something else

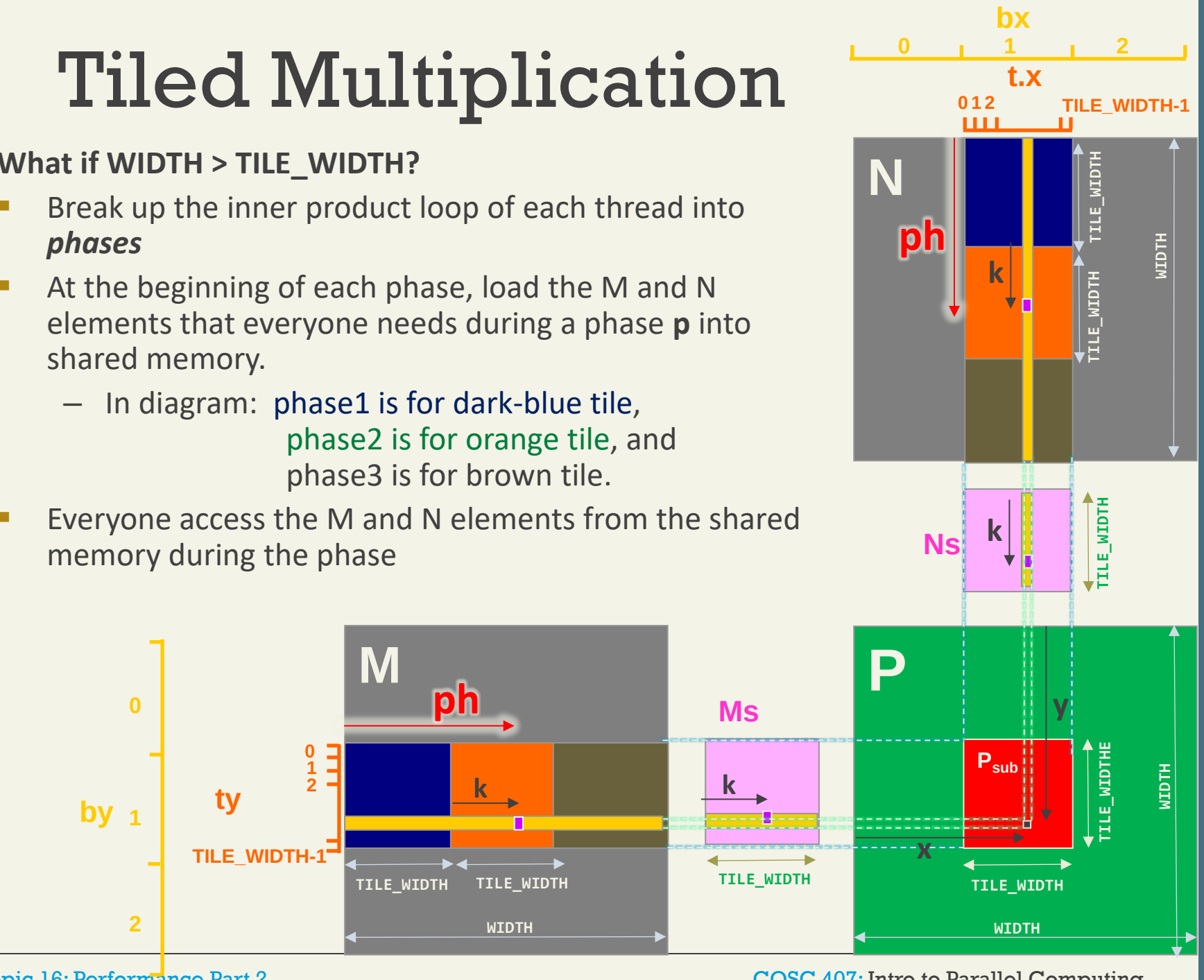




Tiled Multiplication

What if WIDTH > TILE_WIDTH?

- Break up the inner product loop of each thread into ***phases***
- At the beginning of each phase, load the M and N elements that everyone needs during a phase **p** into shared memory.
  - In diagram: phase1 is for dark-blue tile, phase2 is for orange tile, and phase3 is for brown tile.
- Everyone access the M and N elements from the shared memory during the phase



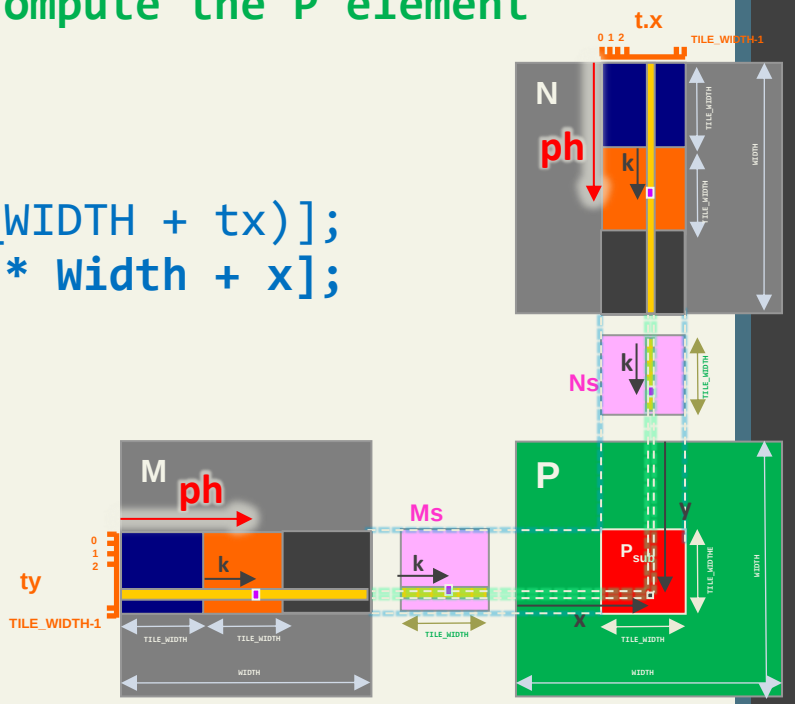
Tiled Multiplication: Kernel Code

```
_global_ void MatrixMul(float* M, float* N, float* P, int Width){
  __shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Ns[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;        int by = blockIdx.y;
  int tx = threadIdx.x;       int ty = threadIdx.y;

  // Identify the row and column of the P element to work on
  int y = by * TILE_WIDTH + ty;
  int x = bx * TILE_WIDTH + tx;
  float value = 0;

  // Loop over M and N tiles required to compute the P element
  int num_phases = Width/TILE_WIDTH;
  for (int ph = 0; ph < num_phases; p++) {
     // load tile m into shared memory
     Ms[ty][tx] = M[y * Width + (ph *TILE_WIDTH + tx)];
     Ns[ty][tx] = N[(ty + ph *TILE_WIDTH) * Width + x];
     __syncthreads();
     for (int k = 0; k < TILE_WIDTH; k++)
        value += Ms[ty][k] * Ns[k][tx];
     __synchthreads();
  }
  P[y*Width + x] = value;
}
```

we still need
if(x<width && y<width){…}



Loading an Input Tile

Let's see how the loop (in red in previous slide) works

- ph = 0: accessing tile 0 in 2D indexing:

```
N[ty][x]
M[y][tx]
```

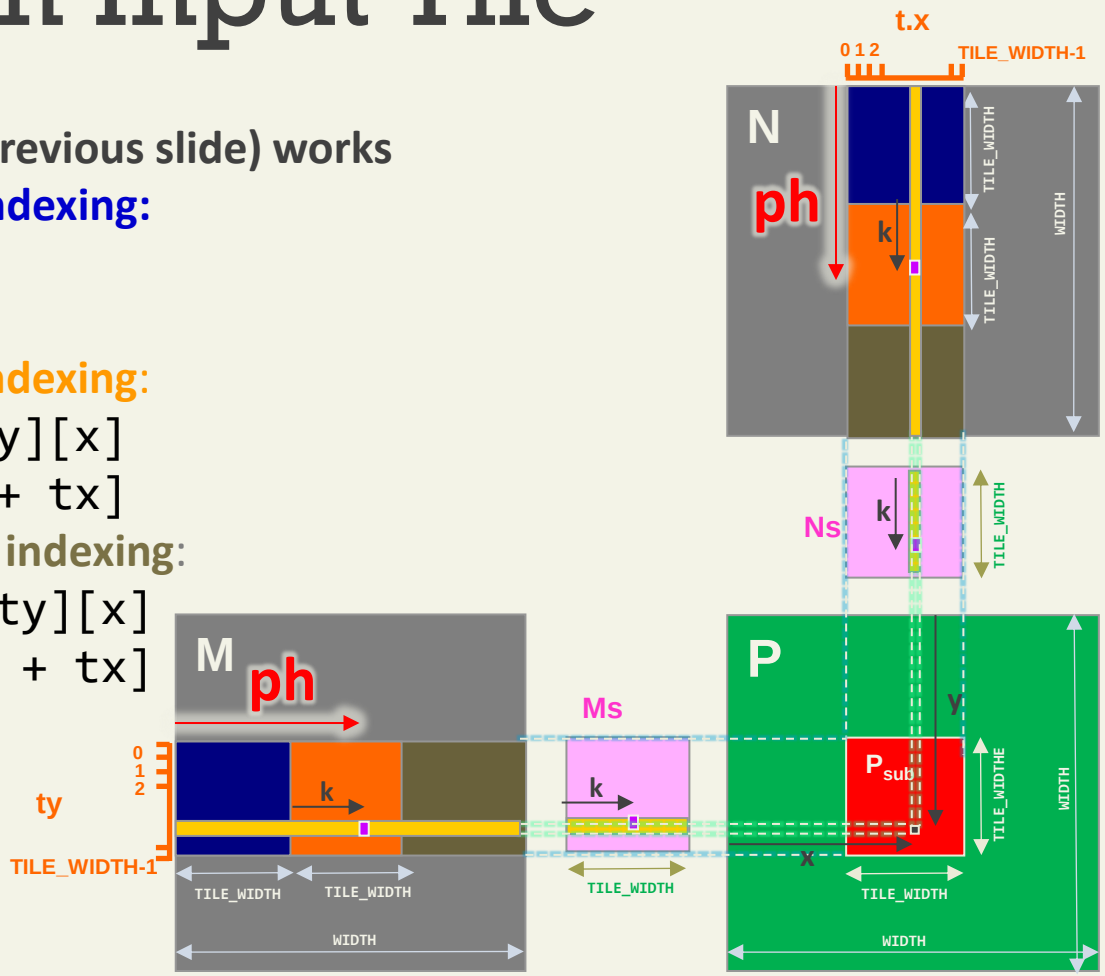
- ph = 1: accessing tile 1 in 2D indexing:

```
N[1*TILE_WIDTH + ty][x]
M[y][1*TILE_WIDTH + tx]
```

- ph = …: accessing tile ph in 2D indexing:

```
N[ph*TILE_WIDTH + ty][x]
M[y][ph*TILE_WIDTH + tx]
```

. . .

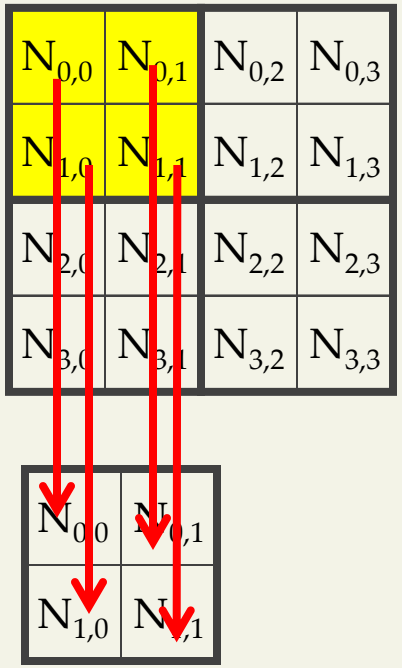


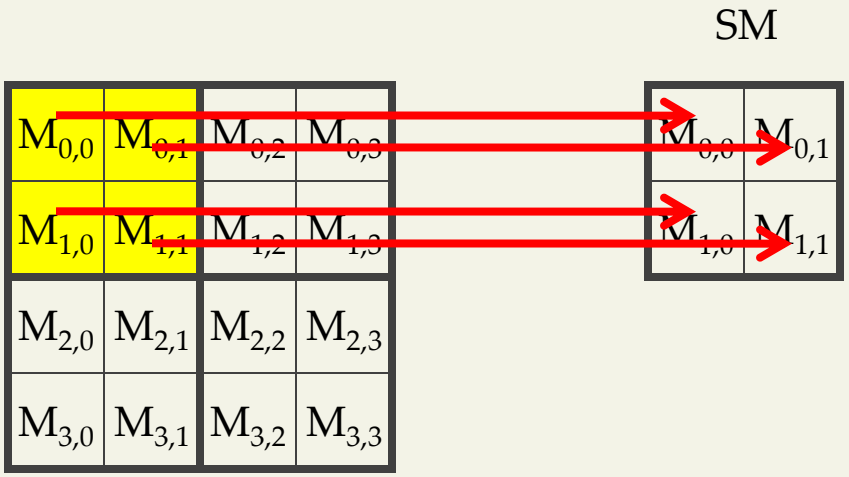
Remember, M and N are dynamically allocated and can only use 1D indexing:

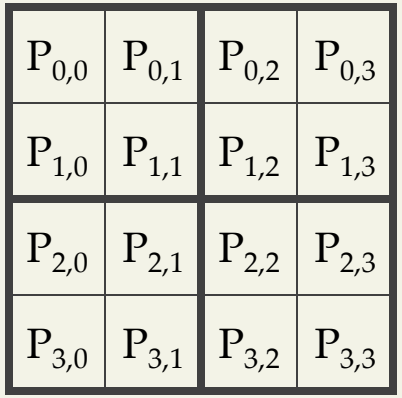
```
N[(ph*TILE_WIDTH+ty)*Width+x]
M[y*Width+ph*TILE_WIDTH + tx]
```

```
y = by*TILE_WIDTH+ty
x = bx*TILE_WIDTH+tx
```

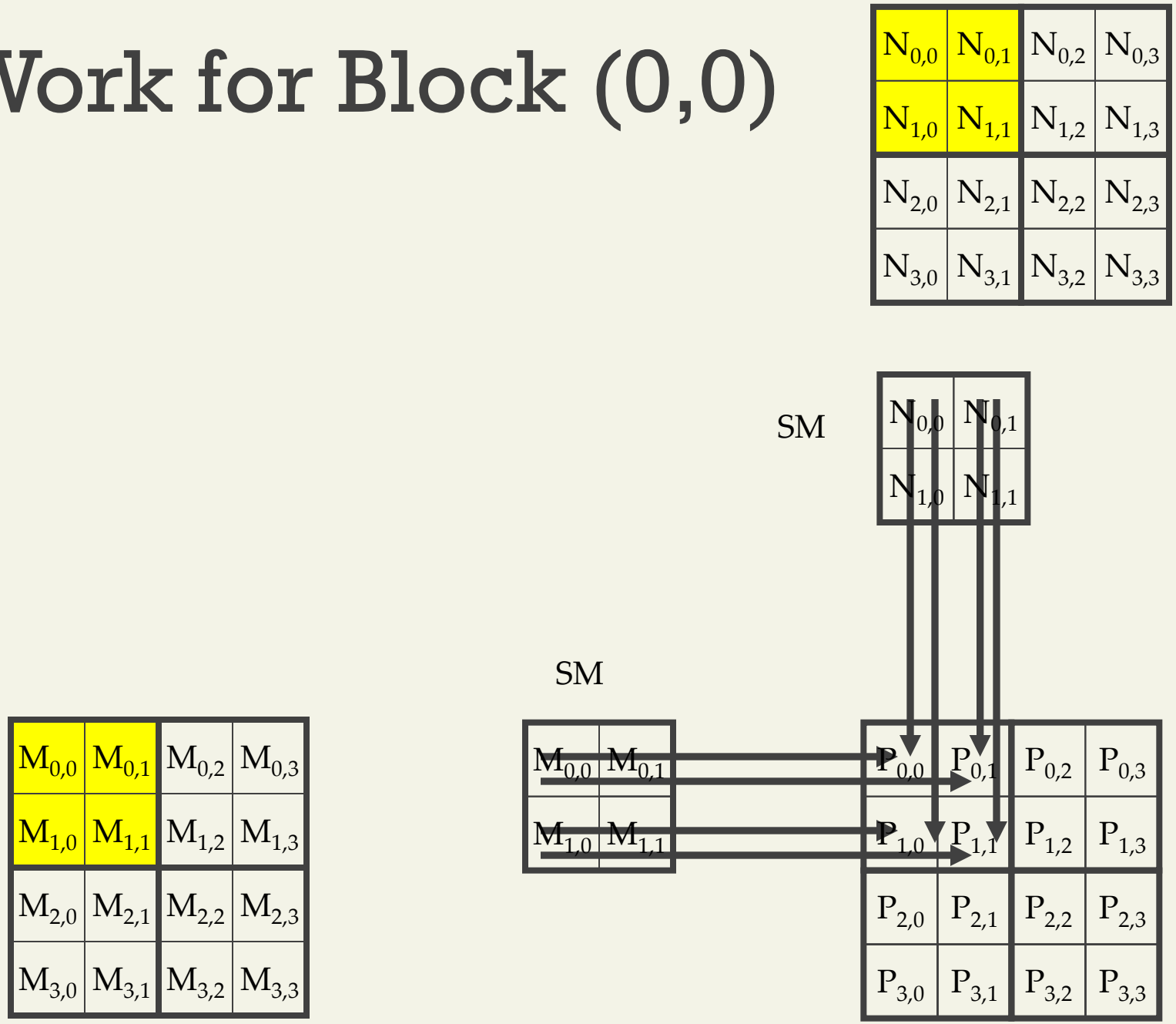
Work for Block (0,0)



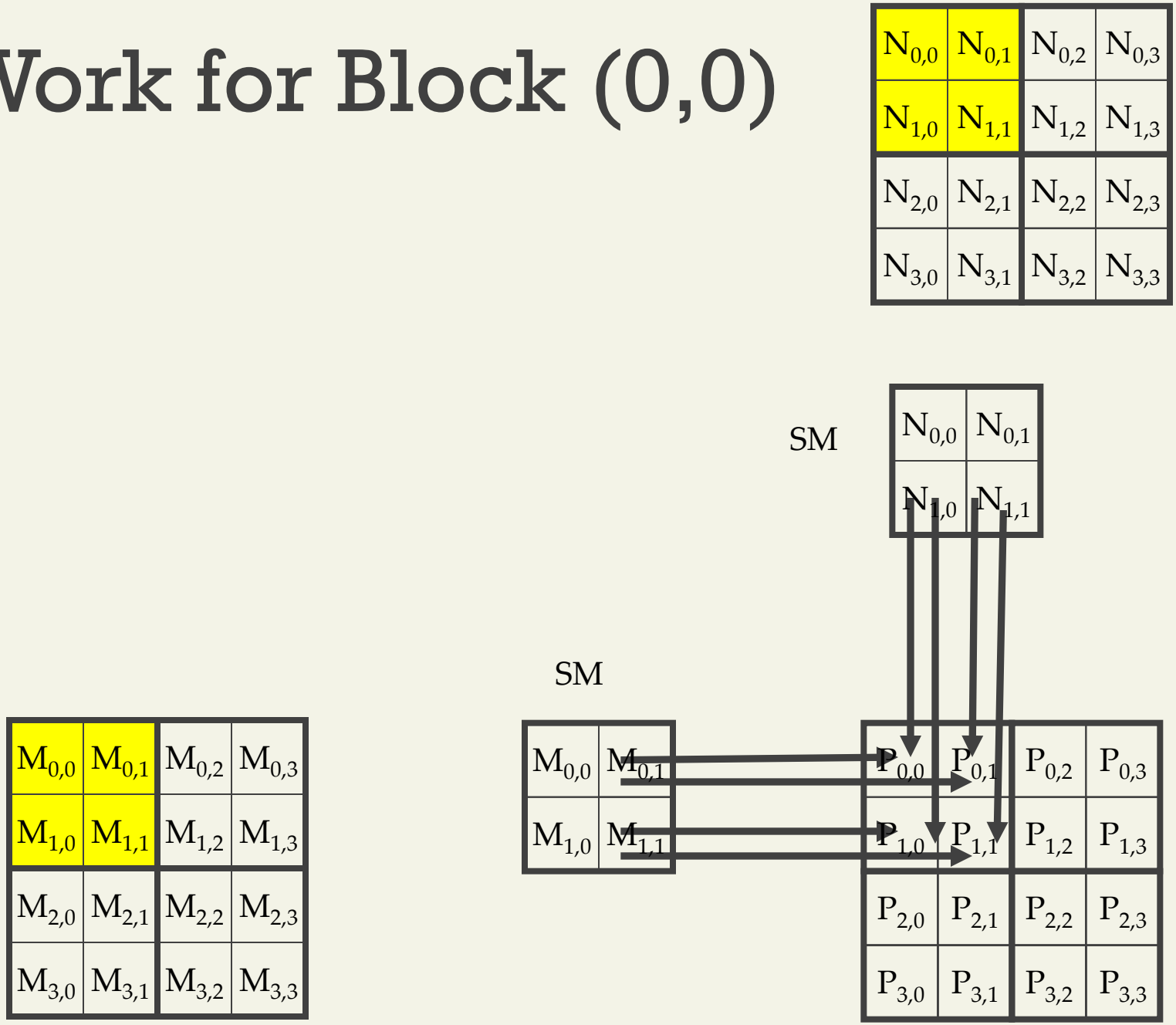




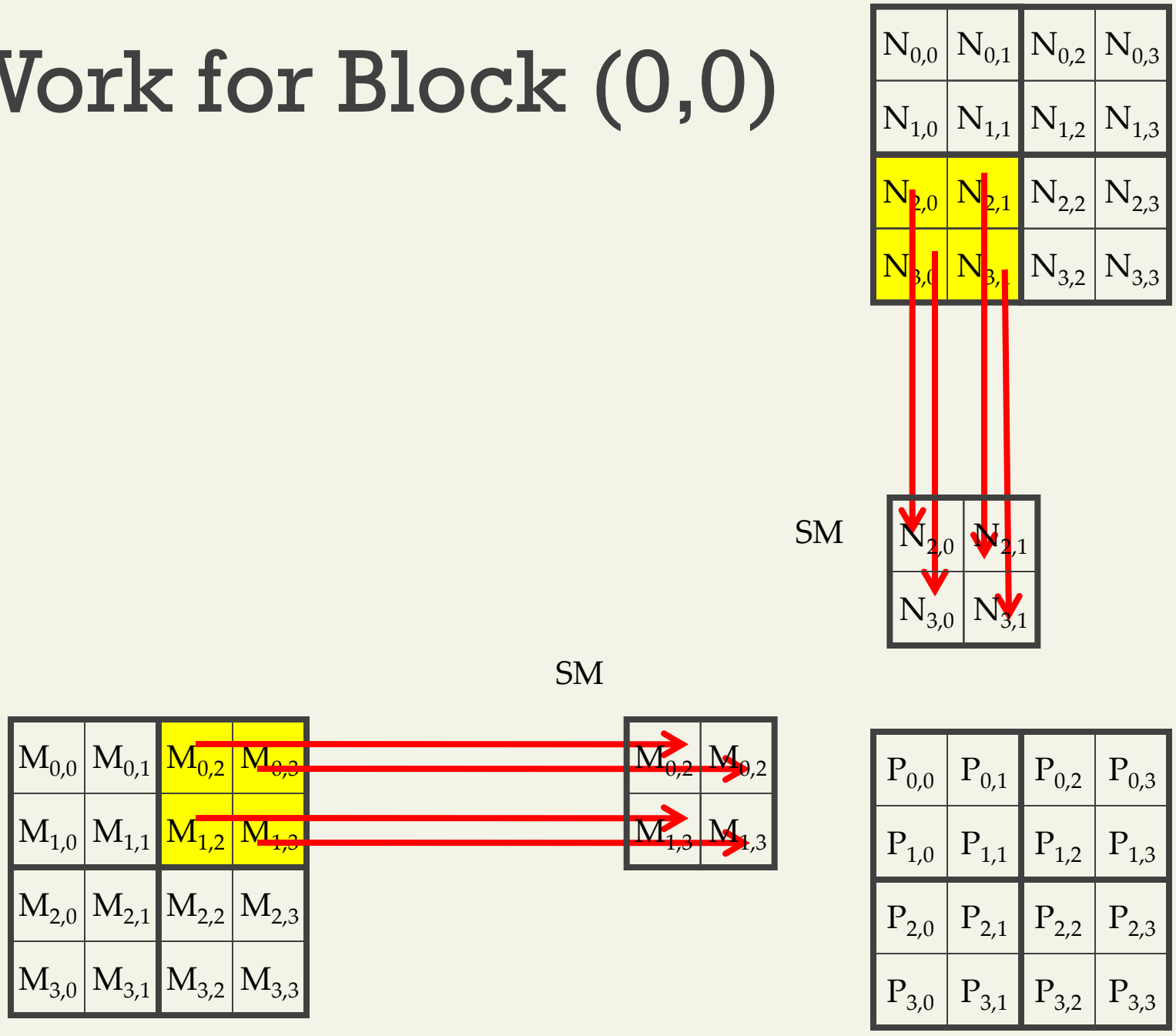
Work for Block (0,0)



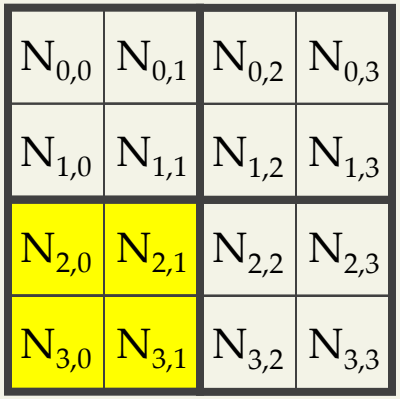
Work for Block (0,0)

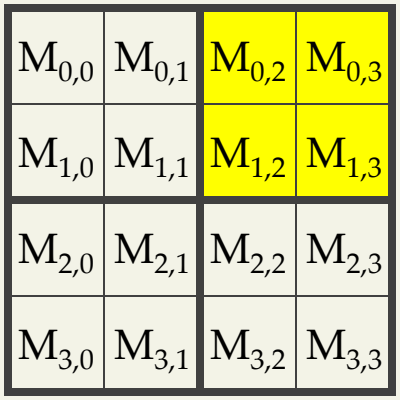


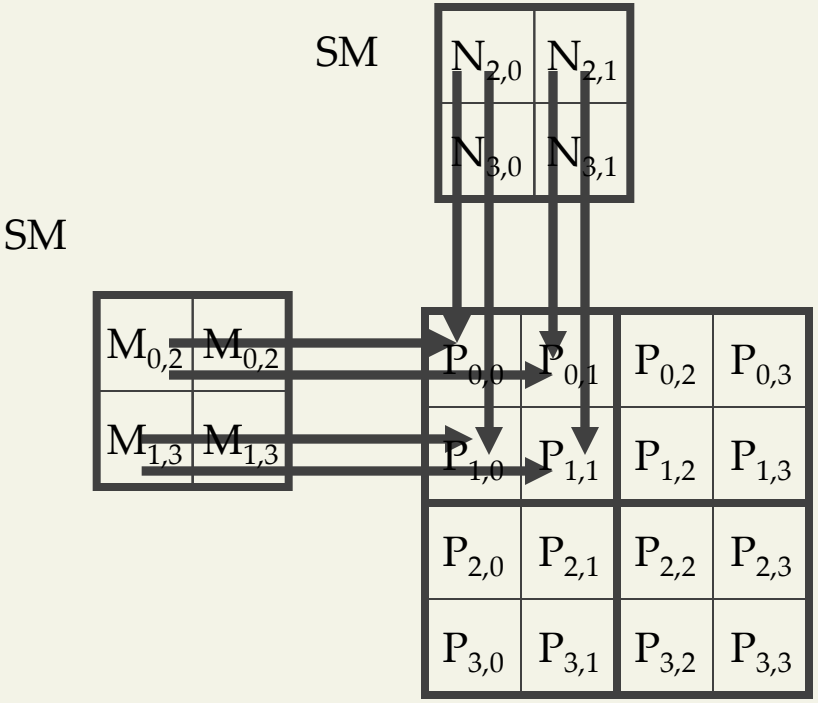
Work for Block (0,0)



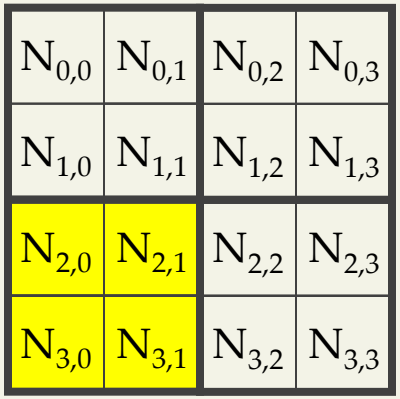
Work for Block (0,0)

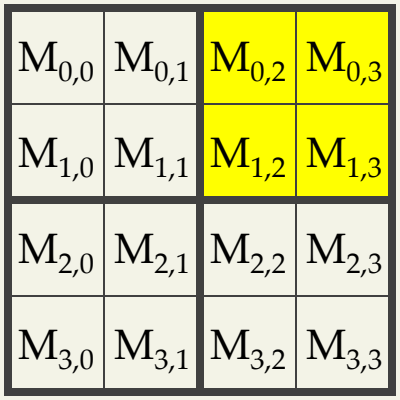


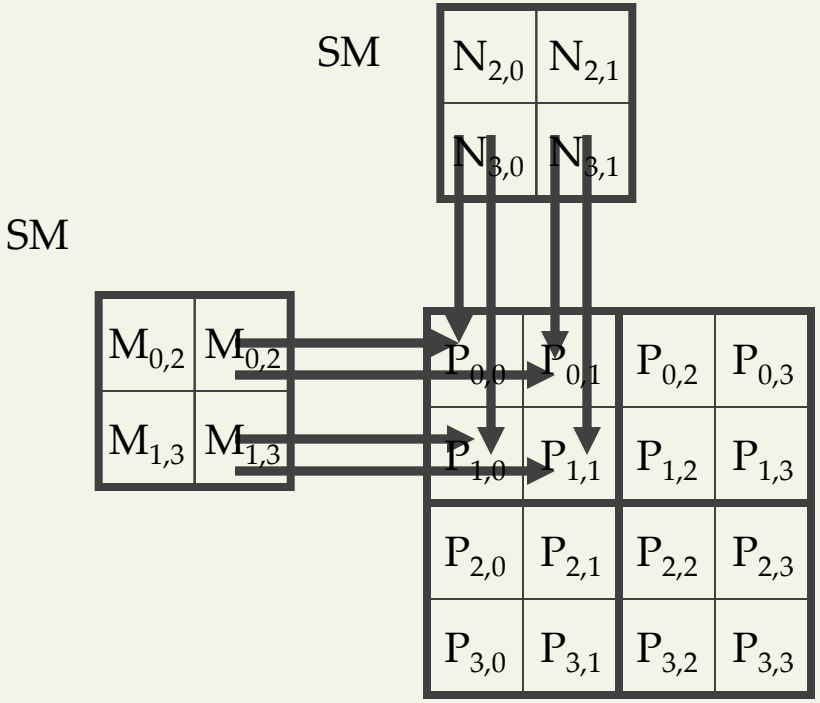




Work for Block (0,0)







Tiled Multiplication: Execution Configuration

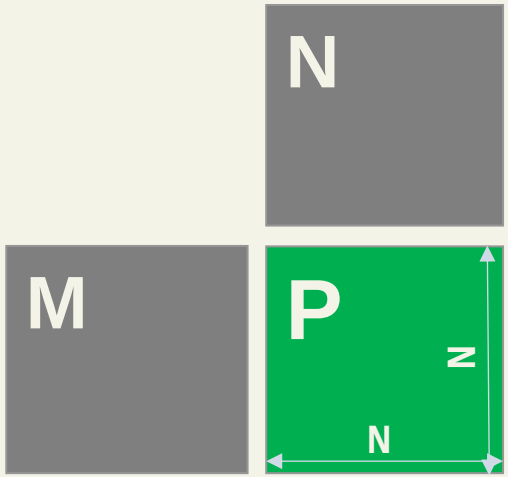
- Setup the execution configuration:
  - `dim3 blockSize(TILE_WIDTH, TILE_WIDTH);`
  - `dim3 gridSize(Width/TILE_WIDTH, Width/TILE_WIDTH);`

- Above assumes width is divisible by TILE_WIDTH. If not, you should:
  - Use the general formula we learned before, i.e.
    - `int nblk_x = (WIDTH - 1) / TILE_WIDTH + 1;`
    - `int nblk_y = (HEIGHT - 1)/ TILE_HEIGHT + 1;`
    - `dim3 gridSize(nblk_x, nblk_y);`
  - Use an if statement to disable threads operating in ranges outside the array
    - E.g. `if(x<WIDTH && y<HEIGHT)`

Question

- Consider performing a matrix multiplication of two input matrices with dimensions NxN. How many times is each element in the input matrices requested from global memory when there is **NO tiling**?

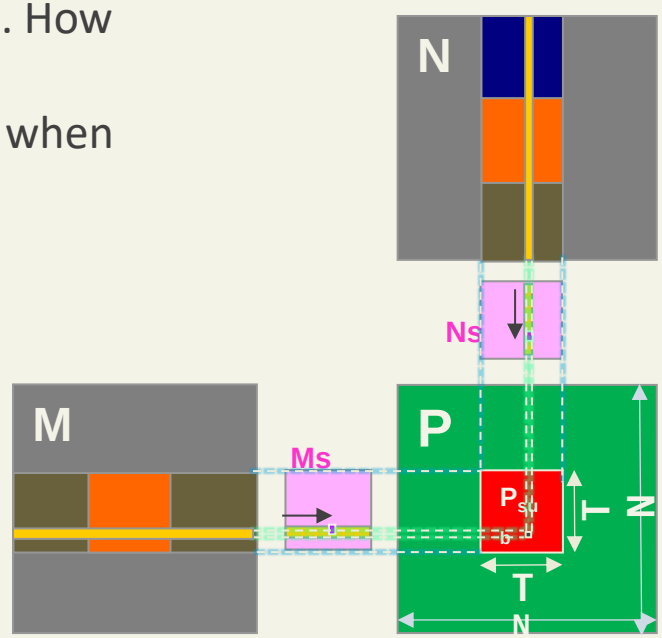
A. 1
B. N
C. N^2
D. blockWidth/N
E. blockWidth



Question

- Consider performing a matrix multiplication of two input matrices with dimensions NxN. How many times is each element in the input matrices requested from global memory when tiles of TxT are used?

A. 1
B. N
C. N^2
D. N/T
E. N^2/T^2

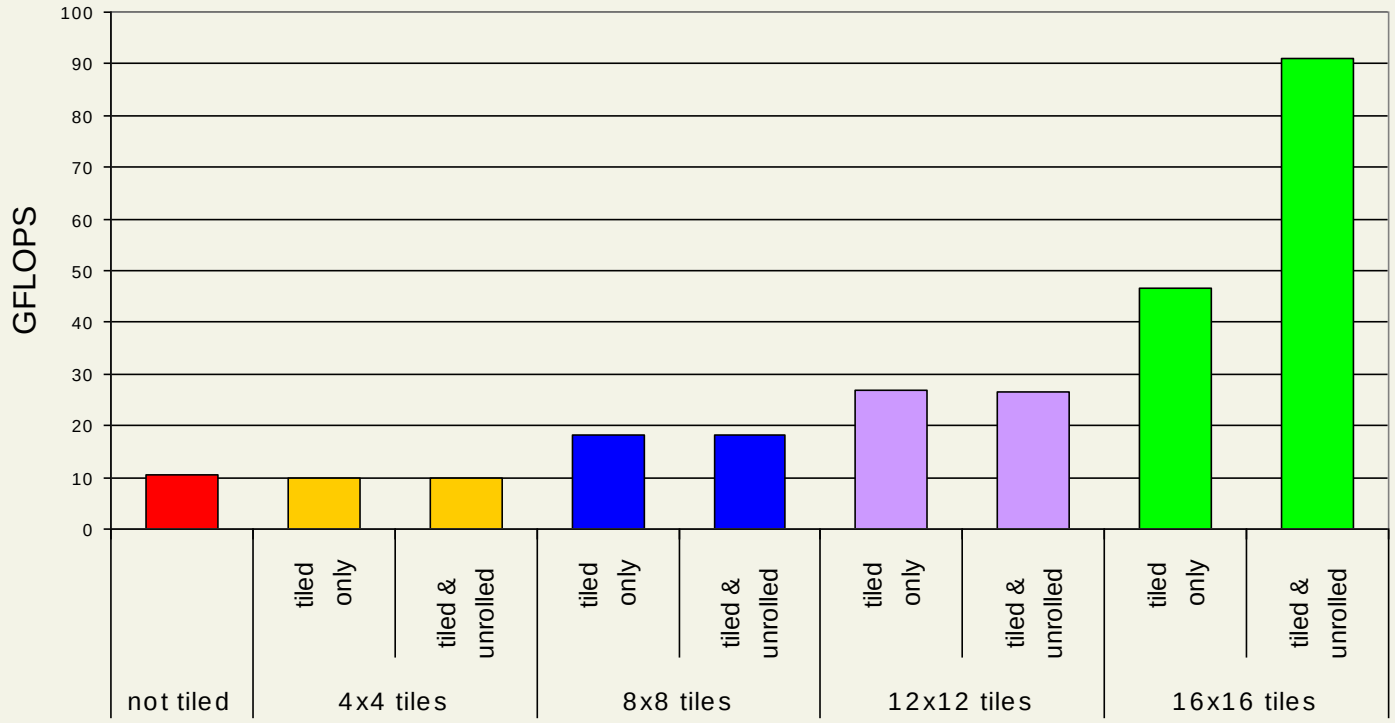


Computing CGMA when *Tiling* is used

- Each thread block should have many threads. For example:
  - TILE_WIDTH of 16 gives 16*16 = 256 threads
  - TILE_WIDTH of 32 gives 32*32 = 1024 threads
- With TILE_WIDTH = 16, each block performs
  - 2*256 = **512** float loads from global memory
    - 256 threads, each loading 2 floats (one from M and one from N)
  - 256 * (**2***16) = **8,192** mul/add operations
    - 256 threads, each performs one multiplication and one addition in a loop that runs 16 times
  - Hence, **CGMA** = 8,192/512 = **16**, which means,
    - **Using 16x16 tiling**, we reduce the accesses to the global memory by a **factor of 16**
    - The 86.4GB/s bandwidth can now support (86.4/4)*16 = 347.6 GFLOPS!

```
_global_ void MatrixMul(float*M, float*N, float*P,int Width){
  __shared__ float Ms[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Ns[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;                int by = blockIdx.y;
  int tx = threadIdx.x;               int ty = threadIdx.y;
  // Identify the row and column of the P element to work on
  int y = by * TILE_WIDTH + ty;
  int x = bx * TILE_WIDTH + tx;
  float value = 0;
  // Loop over M and N tiles required to compute the P element
  int num_phases = Width/TILE_WIDTH;
  for (int ph = 0; ph < num_phases; p++) {
    // load tile m into shared memory
    Ms[ty][tx] = M[y * Width + (ph *TILE_WIDTH + tx)];
    Ns[ty][tx] = N[(ty + ph *TILE_WIDTH) * Width + x];
    __syncthreads();
    for (int k = 0; k < TILE_WIDTH; k++)
                value += Ms[ty][k] * Ns[k][tx];
    __synchthreads();
  }
  P[y*Width + x] = value;
}
```

Tiling Size Effects



Question

The data of the arrays **M**, **N**, and **P** in this code reside in___

```
__global__ void MatrixMul(float* M, float* N, float* P, int width){
   int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
   int y = blockIdx.y * TILE_WIDTH + threadIdx.y;
   float value = 0;
   for (int k = 0; k < width; k++)
         value += M[y*width + k] * N[k*width + x];
   P[y*width+x] = value;
}
```

A. Global memory
B. Shared memory
C. Local memory
D. Register

Question

The variable value in this code resides in

```
__global__ void MatrixMul(float* M, float* N, float* P, int width){
   int x = blockIdx.x * TILE_WIDTH + threadIdx.x;
   int y = blockIdx.y * TILE_WIDTH + threadIdx.y;
   float value = 0;
   for (int k = 0; k < width; k++)
        value += M[y*width + k] * N[k*width + x];
   P[y*width+x] = value;
}
```

A. Constant memory
B. Global memory
C. Shared memory
D. Local memory
E. Register

Image Processing on CUDA

- Data parallel processing is a natural choice for image processing
  - Each pixel can be mapped to one threads
  - Lots of data is shared between threads
    - E.g. when you apply a blur filter, each pixel could become the average of the pixels around it.
- Examples:
  - Pixel based operations: desaturate, negative, hue shift, etc.
    - As in OpenMP assignment A3 – can also be easily done in CUDA.
  - Convolution (e.g. blur, edge detection, etc)
    - As in CUDA assignment A7
  - Histogram operations
  - Image compression
  - Noise reduction
  - Image correlation

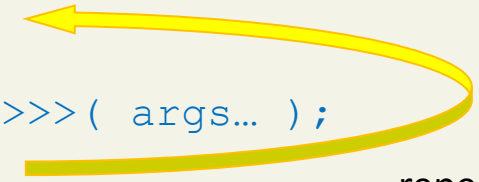
One Last Thing…

- Global memory access includes using cache memory (L1 and L2). The way cache works is, when reading a byte from global memory, it is stored in the cache (for better performance) along with its neighbours (i.e. neighbouring data in the global)
  - Reads:
    - L1 and L2 cache memories are used.
      - If L1 is enabled, a load request is serviced by a 128-byte memory transaction
      - If L1 is disabled, L2 is used and a load request is serviced by memory transactions of one, two, or four 32-byte segments.
  - Writes
    - Only L2 is used
      - Write requests are also serviced by memory transactions of one, two, or four 32-byte segments.

Summary - Code

Typical Structure of a CUDA Program

- Global variables declaration
  - Use **`__device__`**, **`__constant__`**, **`__host__`**
- Kernel function: `void kernel (args)`
  - variables declaration – auto (default), **`__shared__`**
    - auto variables transparently assigned: primitives→ registers. Arrays → local mem
  - Tiling (if needed)…
  - **`__syncthreads()`**…
- Other functions
  - `float foo(args){…}`
  - Use **`__device__`**, **`__host__`**, etc in their declaration.
- `main()`
  - allocate memory space on the device – `cudaMalloc`
  - transfer data from host to device – `cudaMemCpy`
  - execution configuration setup
  - kernel call: `kernel<<<execution configuration>>>( args… );`
  - transfer results from device to host – `cudaMemCpy`
  - optional: compare against golden (host computed) solution



repeat as needed

Summary: Tiling Technique

- Identify a block/tile of global memory content that are accessed by multiple threads
- Load the block/tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next block/tile

Summary – other concepts

- Constant memory also resides in device memory (DRAM) - much slower access than shared memory
  - But… cached!
  - Highly efficient access for read-only data

- Carefully divide data according to access patterns
  - R/Only → constant memory (very fast if in cache)
  - R/W shared within Block → shared memory (very fast)
  - R/W within each thread → registers (very very fast)
  - R/W inputs/results → global memory (very slow)

Performance Improvement Guidelines

- Use many threads on the GPU
- Use # blocks > # SMs
  - to make sure all SMs are busy (balanced load)
- Choose block dimensions that, not only make sense to your problem, but also fully utilize the SM capabilities.
  - e.g., G80 can handle up to 8 blocks or 768 threads simultaneously. Using 256 threads/block (e.g., 16x16) will fully utilize the SM resources. 64 threads/block (e.g., 8x8) will be underutilization of the SM.
- Reduce global memory traffic by copying frequently accessed data to faster memory
  - E.g., using tiling technique

Summary, cont'd

- To make best use of GPU computing, you need to:
  - Identify compute intensive parts of an application
  - Minimal data transfer GPU←→CPU (data transfer is expensive)
  - Adopt scalable algorithms
  - Optimize data arrangements to maximize locality
  - Performance Tuning
  - Pay attention to code portability and maintainability

Conclusion

Today:

- Example: Improving Performance of Matrix Multiplication
- More optimizations
- Bandwidth
- iCLicker questions!

Next Lecture:

- Introduction to MPI