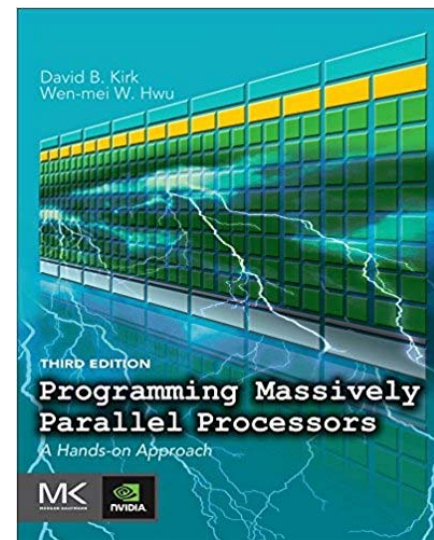


# Introduction to CUDA

## *(3) Memory And Data Locality*

# Reference

- [CUDA C Programming Guide](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html),
  - <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**



# Content

- Memory Access Efficiency
- Matrix Multiplication

# Importance of Memory Access Efficiency

```
    for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {  
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {  
  
5.          int curRow = Row + blurRow;  
6.          int curCol = Col + blurCol;  
          // Verify we have a valid image pixel  
7.          if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {  
8.              pixVal += in[curRow * w + curCol];  
9.              pixels++; // Keep track of number of pixels in the avg  
          }  
      }  
    }
```

**FIGURE 4.1:** The most executed part of the image blurring kernel in Fig. 3.8.

## Compute-to-global-memory-access Ratio

- Global memory bandwidth is 1TB/s, with 4 bytes in each single precision float value,
- No more than  $1000/4 = 250$  G single-precision ops can be loaded in one second.
- If Compute-to-global-memory-access ratio is 1, no more than 250GFLOPS can be achieved, only 2% of 12TFLOPS of recent device.
- The ratio need to be improved to 48 or higher.

# Matrix Multiplication Example

## A Simple Host Version in C

// Matrix multiplication on the (CPU) host in single precision

```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
```

```
{
```

```
    for (int i = 0; i < Width; ++i)
```

```
        for (int j = 0; j < Width; ++j) {
```

```
            float sum = 0;
```

```
            for (int k = 0; k < Width; ++k) {
```

```
                float a = M[i * Width + k];
```

```
                float b = N[k * Width + j];
```

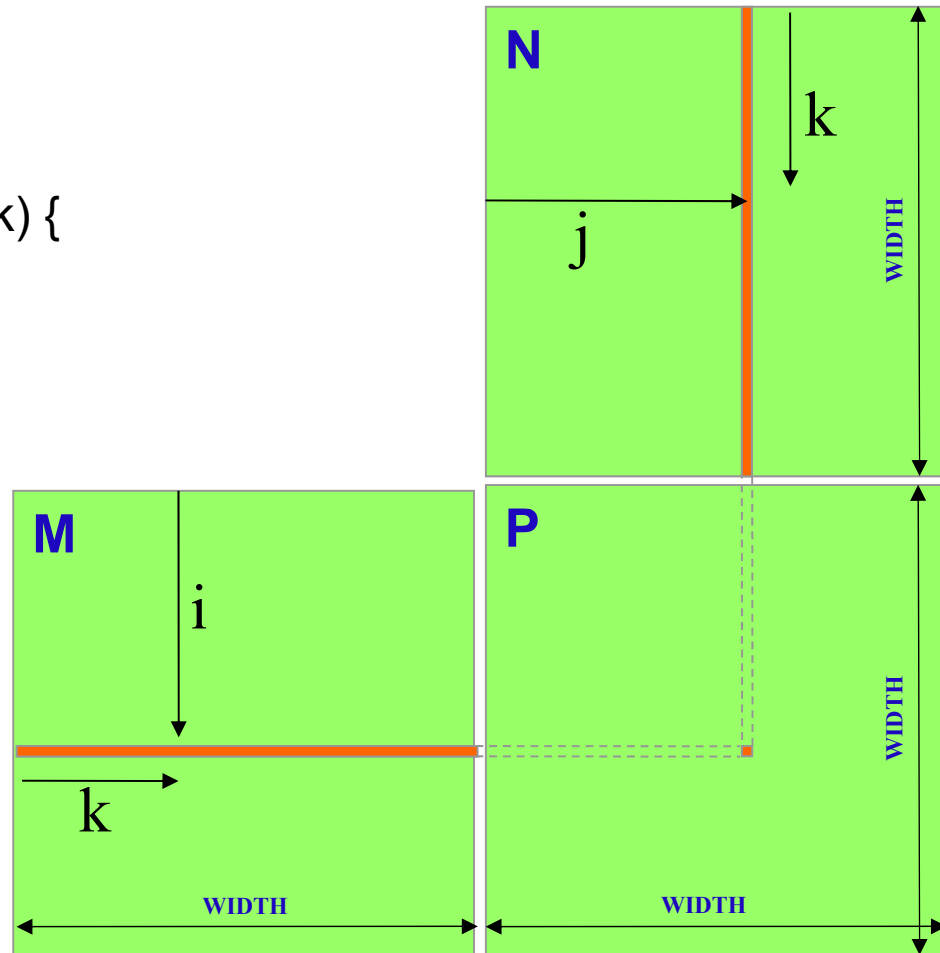
```
                sum += a * b;
```

```
            }
```

```
            P[i * Width + j] = sum;
```

```
        }
```

```
    }
```



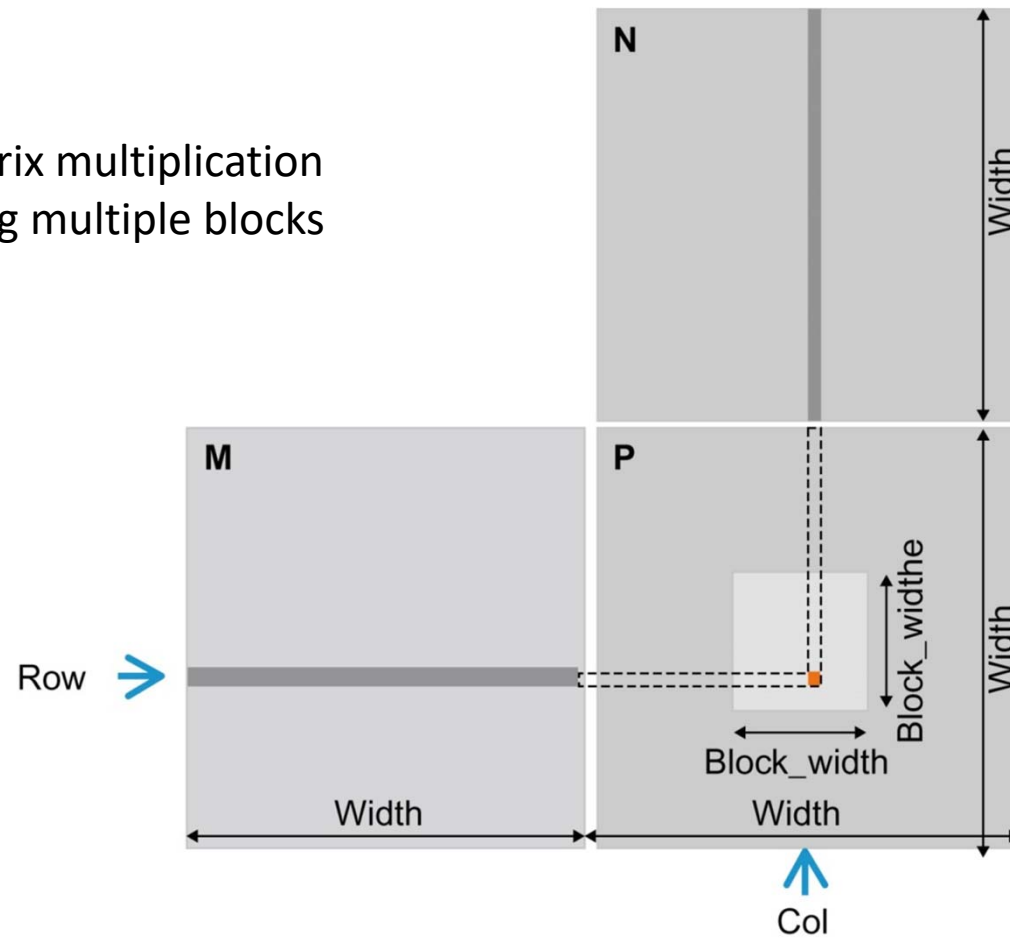
# Initial matrix multiplication version

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
int Width) {
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
}
```

A simple matrix multiplication kernel using one thread to compute one P element.

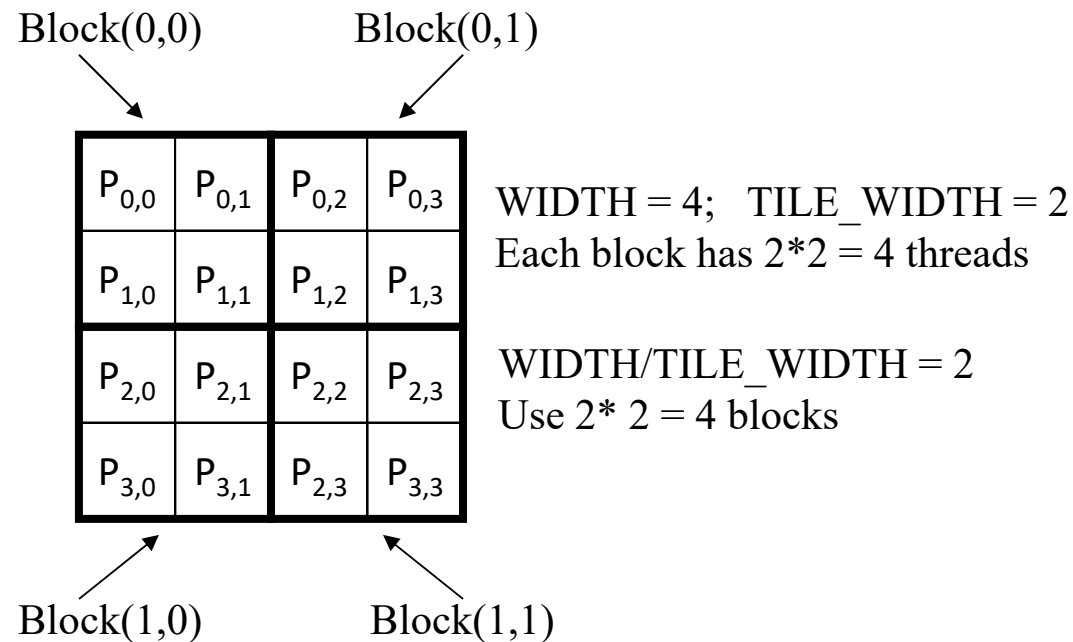
# Initial matrix multiplication version

Matrix multiplication  
using multiple blocks



# Kernel Function - A Small Example

- Have each 2D thread block to compute a  $(\text{TILE\_WIDTH})^2$  sub-matrix (tile) of the result matrix
  - Each has  $(\text{TILE\_WIDTH})^2$  threads
- Generate a 2D Grid of  $(\text{WIDTH}/\text{TILE\_WIDTH})^2$  blocks





# A Slightly Bigger Example

(TILE\_WIDTH = 2)

P <sub>0,0</sub>	P <sub>0,1</sub>	P <sub>0,2</sub>	P <sub>0,3</sub>	P <sub>0,4</sub>	P <sub>0,5</sub>	P <sub>0,6</sub>	P <sub>0,7</sub>
P <sub>1,0</sub>	P <sub>1,1</sub>	P <sub>1,2</sub>	P <sub>1,3</sub>	P <sub>1,4</sub>	P <sub>1,5</sub>	P <sub>1,6</sub>	P <sub>1,7</sub>
P <sub>2,0</sub>	P <sub>2,1</sub>	P <sub>2,2</sub>	P <sub>2,3</sub>	P <sub>2,4</sub>	P <sub>2,5</sub>	P <sub>2,6</sub>	P <sub>2,7</sub>
P <sub>3,0</sub>	P <sub>3,1</sub>	P <sub>3,2</sub>	P <sub>3,3</sub>	P <sub>3,4</sub>	P <sub>3,5</sub>	P <sub>3,6</sub>	P <sub>3,7</sub>
P <sub>4,0</sub>	P <sub>4,1</sub>	P <sub>4,2</sub>	P <sub>4,3</sub>	P <sub>4,4</sub>	P <sub>4,5</sub>	P <sub>4,6</sub>	P <sub>4,7</sub>
P <sub>5,0</sub>	P <sub>5,1</sub>	P <sub>5,2</sub>	P <sub>5,3</sub>	P <sub>5,4</sub>	P <sub>5,5</sub>	P <sub>5,6</sub>	P <sub>5,7</sub>
P <sub>6,0</sub>	P <sub>6,1</sub>	P <sub>6,2</sub>	P <sub>6,3</sub>	P <sub>6,4</sub>	P <sub>6,5</sub>	P <sub>6,6</sub>	P <sub>6,7</sub>
P <sub>7,0</sub>	P <sub>7,1</sub>	P <sub>7,2</sub>	P <sub>7,3</sub>	P <sub>7,4</sub>	P <sub>7,5</sub>	P <sub>7,6</sub>	P <sub>7,7</sub>

WIDTH = 8; TILE\_WIDTH = 2  
Each block has  $2*2 = 4$  threads

WIDTH/TILE\_WIDTH = 4  
Use  $4*4 = 16$  blocks

# A Slightly Bigger Example (cont.)

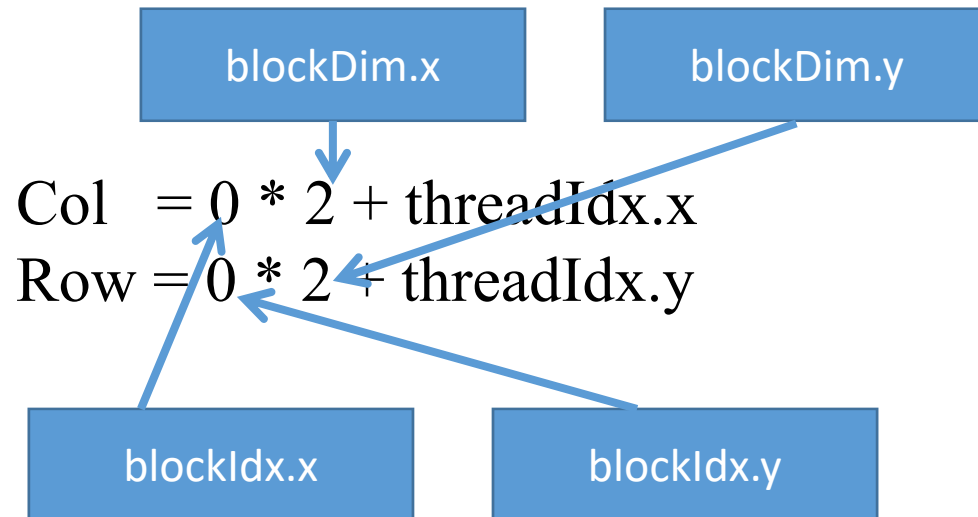
(TILE\_WIDTH = 4)

P <sub>0,0</sub>	P <sub>0,1</sub>	P <sub>0,2</sub>	P <sub>0,3</sub>	P <sub>0,4</sub>	P <sub>0,5</sub>	P <sub>0,6</sub>	P <sub>0,7</sub>
P <sub>1,0</sub>	P <sub>1,1</sub>	P <sub>1,2</sub>	P <sub>1,3</sub>	P <sub>1,4</sub>	P <sub>1,5</sub>	P <sub>1,6</sub>	P <sub>1,7</sub>
P <sub>2,0</sub>	P <sub>2,1</sub>	P <sub>2,2</sub>	P <sub>2,3</sub>	P <sub>2,4</sub>	P <sub>2,5</sub>	P <sub>2,6</sub>	P <sub>2,7</sub>
P <sub>3,0</sub>	P <sub>3,1</sub>	P <sub>3,2</sub>	P <sub>3,3</sub>	P <sub>3,4</sub>	P <sub>3,5</sub>	P <sub>3,6</sub>	P <sub>3,7</sub>
P <sub>4,0</sub>	P <sub>4,1</sub>	P <sub>4,2</sub>	P <sub>4,3</sub>	P <sub>4,4</sub>	P <sub>4,5</sub>	P <sub>4,6</sub>	P <sub>4,7</sub>
P <sub>5,0</sub>	P <sub>5,1</sub>	P <sub>5,2</sub>	P <sub>5,3</sub>	P <sub>5,4</sub>	P <sub>5,5</sub>	P <sub>5,6</sub>	P <sub>5,7</sub>
P <sub>6,0</sub>	P <sub>6,1</sub>	P <sub>6,2</sub>	P <sub>6,3</sub>	P <sub>6,4</sub>	P <sub>6,5</sub>	P <sub>6,6</sub>	P <sub>6,7</sub>
P <sub>7,0</sub>	P <sub>7,1</sub>	P <sub>7,2</sub>	P <sub>7,3</sub>	P <sub>7,4</sub>	P <sub>7,5</sub>	P <sub>7,6</sub>	P <sub>7,7</sub>

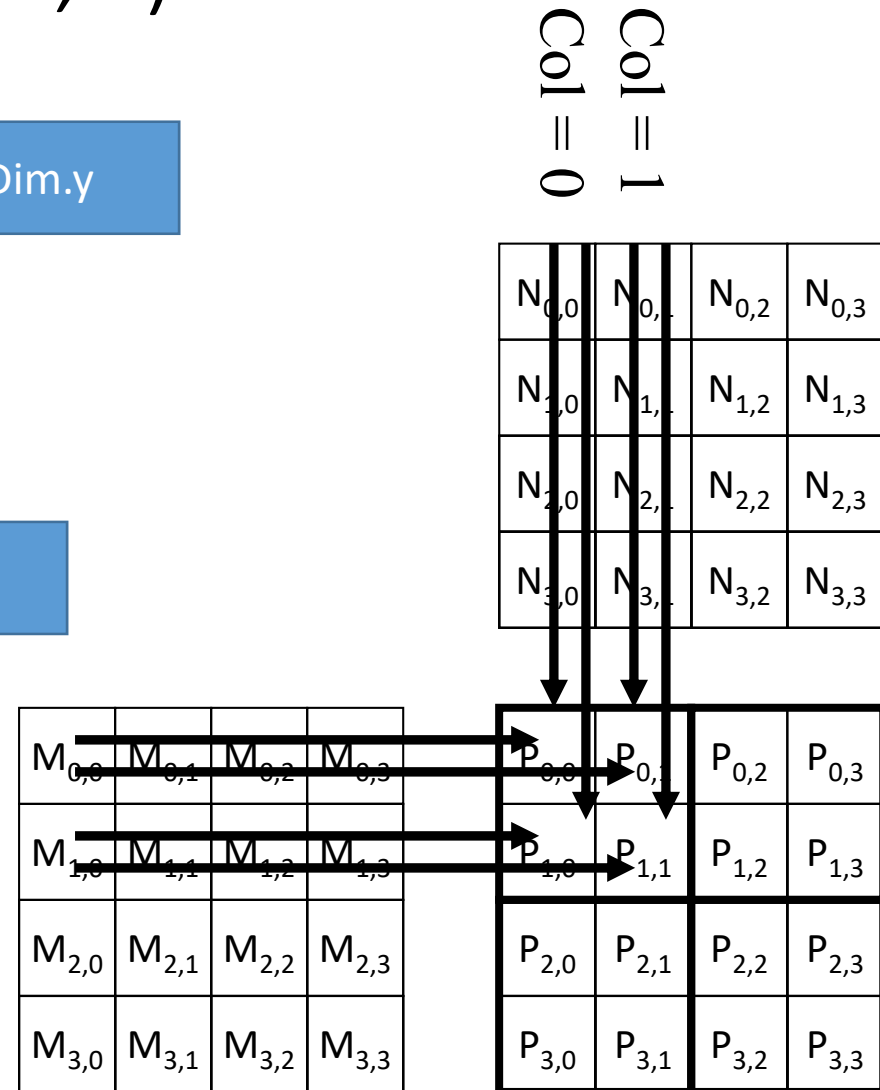
WIDTH = 8; TILE\_WIDTH = 4  
Each block has 4\*4 = 16 threads

WIDTH/TILE\_WIDTH = 2  
Use 2\* 2 = 4 blocks

# Work for Block (0,0)



Row = 0  
Row = 1



# Work for Block (0,1)

$$\text{Col} = 1 * 2 + \text{threadIdx.x}$$

$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$

blockIdx.x

blockIdx.y

Row = 0

Row = 1

Col = 2

Col = 3

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$	$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

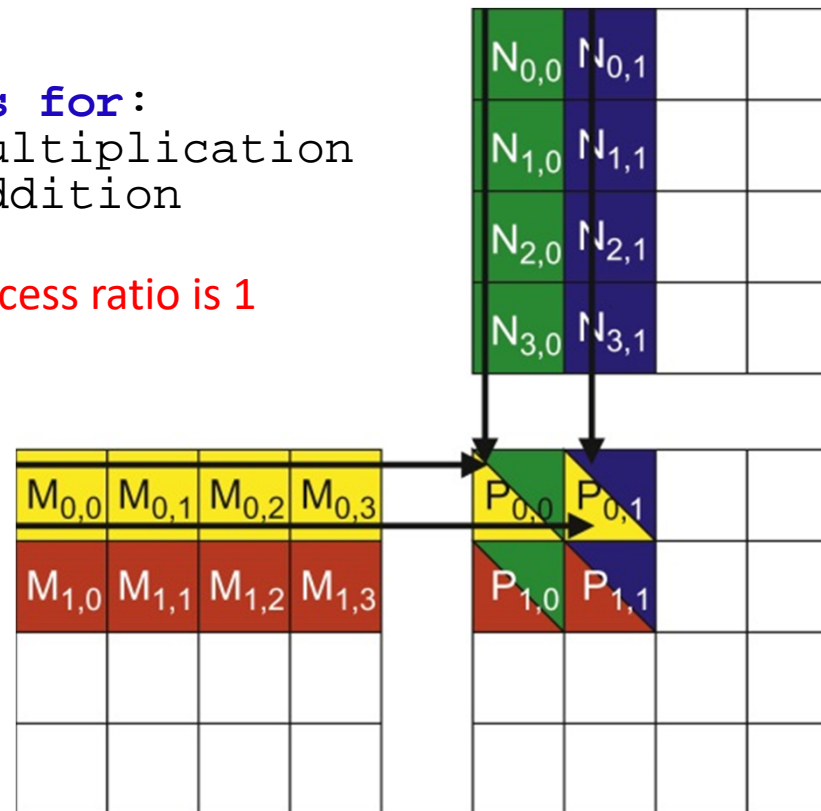
# Walk through the for loop

```
for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width+k] * d_N[k*Width+Col];
```

**2 global memory access for:**

One floating point multiplication  
One floating point addition

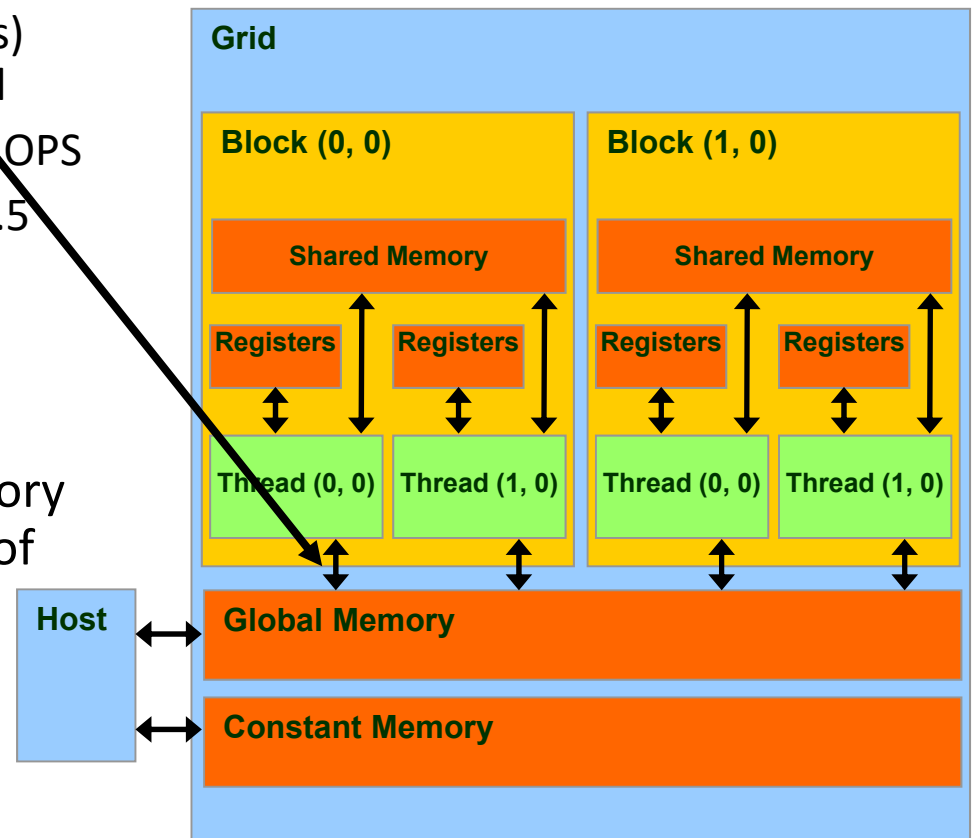
Compute-to-global-memory-access ratio is 1



**FIGURE 4.5:** Matrix multiplication actions of one thread block.

# How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements
  - Two memory accesses (8 bytes) per floating point multiply-add
  - 4B/s of memory bandwidth/FLOPS
  - 150 GB/s limits the code at 37.5 GFLOPS
- The actual code runs at about 25 GFLOPS
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS

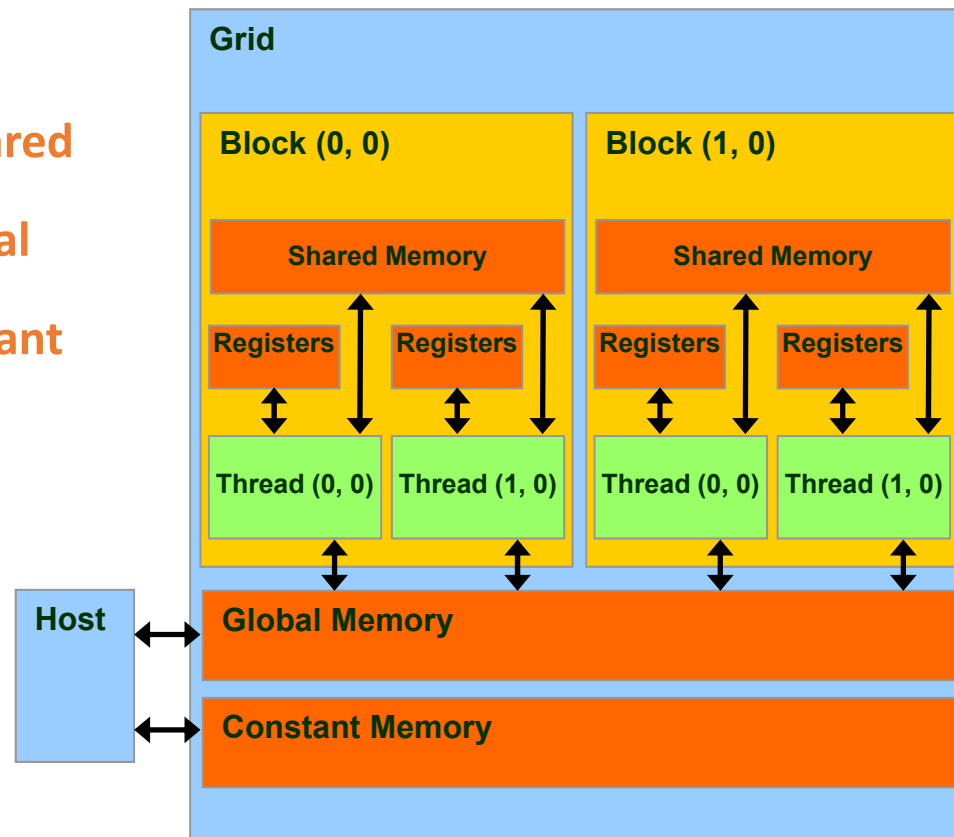


# Content

- Memory Types

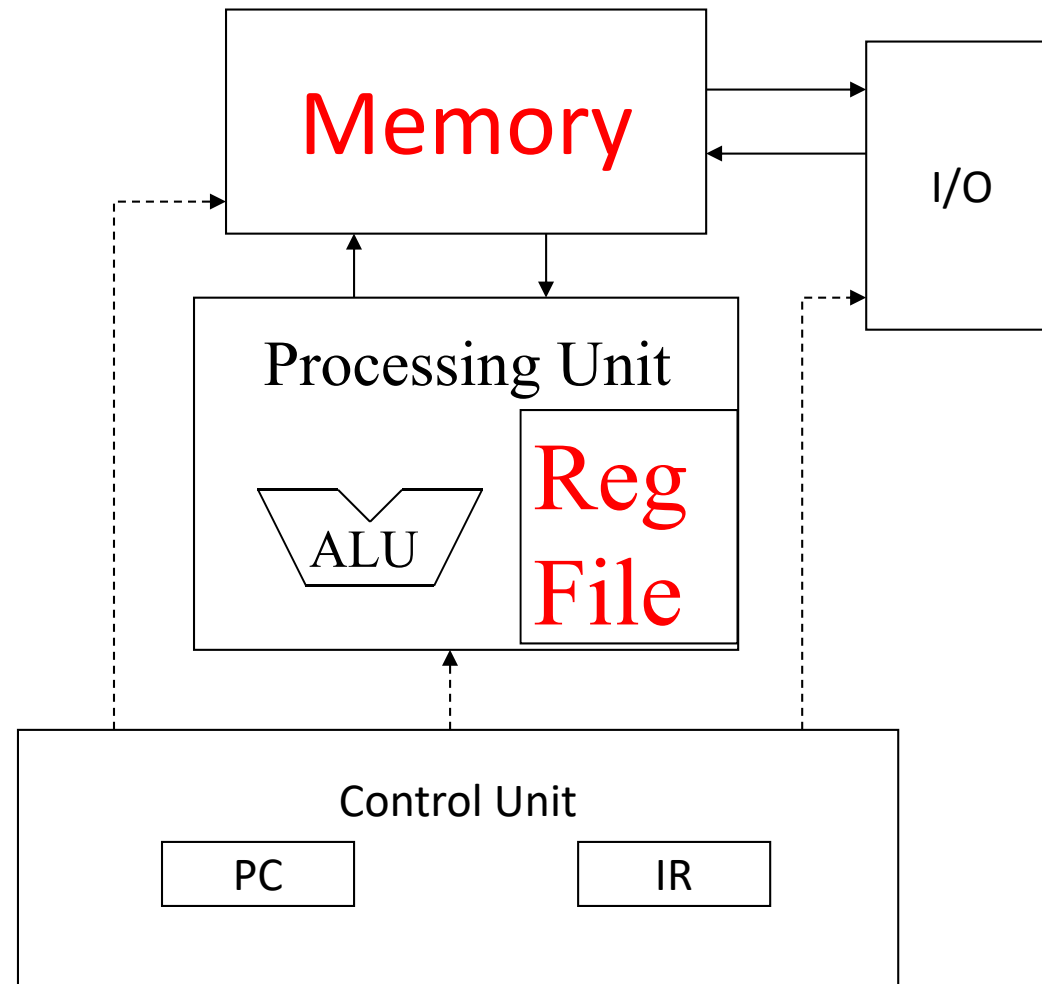
# Programmer View of CUDA Memories

- Each thread can:
  - Read/write per-thread **registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**
  - Read/only per-grid **constant memory (~5 cycles with caching)**





# The Von-Neumann Model



# Going back to the program

- Every instruction needs to be fetched from memory, decoded, then executed.
  - The decode stage typically accesses register file
- Instructions come in three flavors: Operate, Data transfer, and Program Control Flow.
- An example instruction cycle is the following:

Fetch | Decode | Execute | Memory

# Operate Instructions

- Example of an operate instruction:

ADD R1, R2, R3

- Instruction cycle for an operate instruction:

Fetch | Decode | Execute | Memory

# Memory Access Instructions

- Examples of memory access instruction:

LDR R1, R2, #2

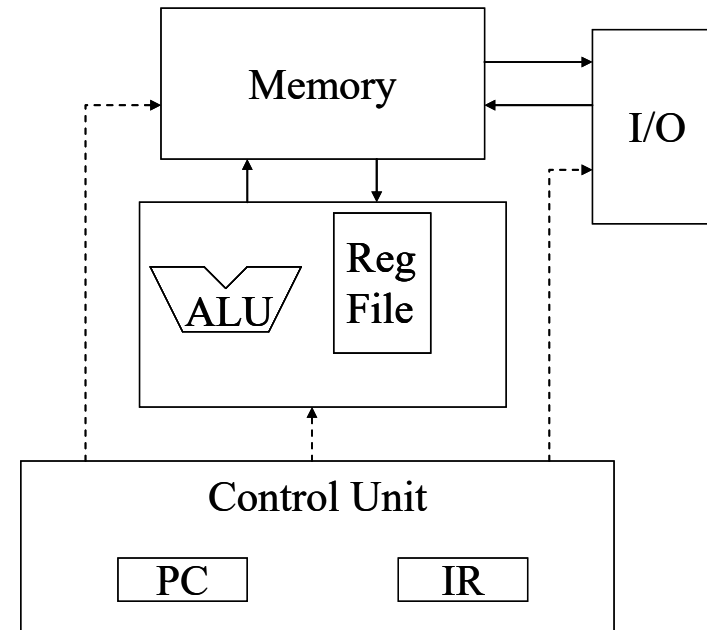
STR R1, R2, #2

- Instruction cycle for an operate instruction:

Fetch | Decode | Execute | Memory

# Registers vs Memory

- Registers are “free”
  - No additional memory access instruction
  - Very fast to use, however, there are very few of them
- Memory is expensive (slow), but very large



# CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	<code>int LocalVar;</code>	Register*	thread	kernel
<code>__device__ __shared__</code>	<code>int SharedVar;</code>	shared	block	Kernel
<code>__device__</code>	<code>int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__</code>	<code>int ConstantVar;</code>	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables
  - without any qualifier reside in a **register**
  - Private copy for each thread (do not exceed limit)
    - **\*Except per-thread arrays** that reside in global memory

# CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	<code>int LocalVar;</code>	Register*	thread	kernel
<code>__device__ __shared__</code>	<code>int SharedVar;</code>	shared	block	Kernel
<code>__device__</code>	<code>int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__</code>	<code>int ConstantVar;</code>	constant	grid	application

- **Shared variables**
  - All threads in a block see the same version
  - Private copy for each block
  - Efficient means for threads in block to collaborate
  - Hold portion of global memory data that heavily used in kernel execution phase.

# CUDA Variable Type Qualifiers

Variable declaration		Memory	Scope	Lifetime
Automatic variables	<code>int LocalVar;</code>	Register*	thread	kernel
<code>__device__ __shared__</code>	<code>int SharedVar;</code>	shared	block	Kernel
<code>__device__</code>	<code>int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__</code>	<code>int ConstantVar;</code>	constant	grid	application

- **Global variables**
  - Slow, but recent devices use cache to improve latency and throughput
  - Can be used for threads to collaborate across blocks,
  - Or pass information from one kernel invocation to another kernel invocation.



# CUDA Variable Type Qualifiers

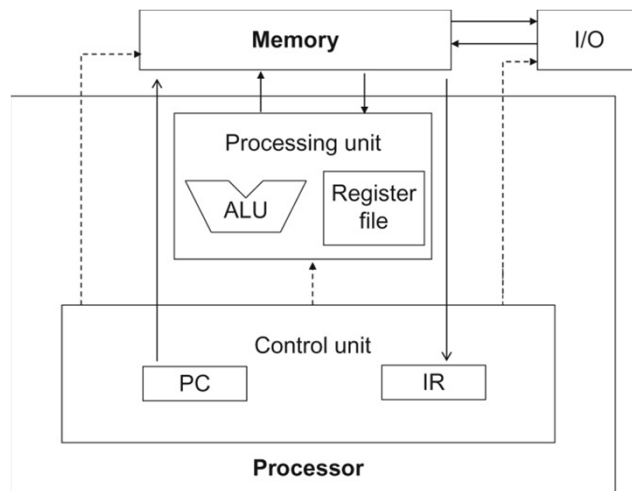
Variable declaration		Memory	Scope	Lifetime
Automatic variables	<code>int LocalVar;</code>	Register*	thread	kernel
<code>__device__ __shared__</code>	<code>int SharedVar;</code>	shared	block	Kernel
<code>__device__</code>	<code>int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__</code>	<code>int ConstantVar;</code>	constant	grid	application

- **Constant variables**
  - Must outside any function body
  - All threads in all grids see the same version
  - Often used for variables that provide input values to kernel functions.

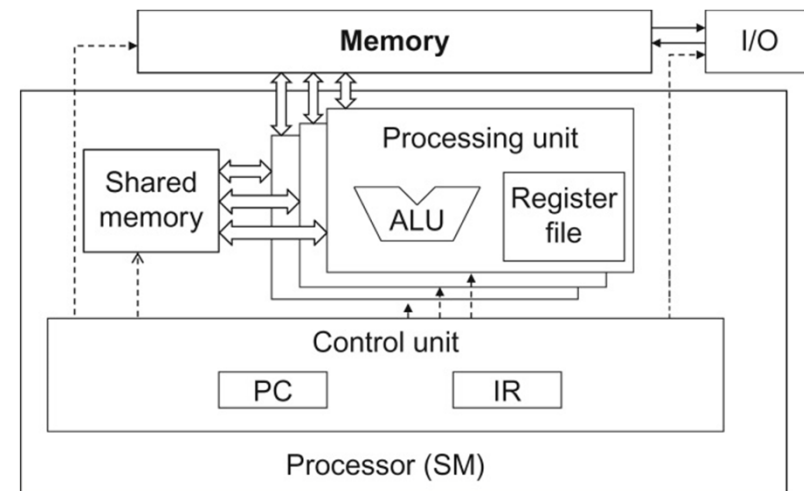
# Content

- Tiling for reduced memory traffic
- Matrix-Matrix Multiplication using Shared Memory

# Registers vs Memory



von Neumann model



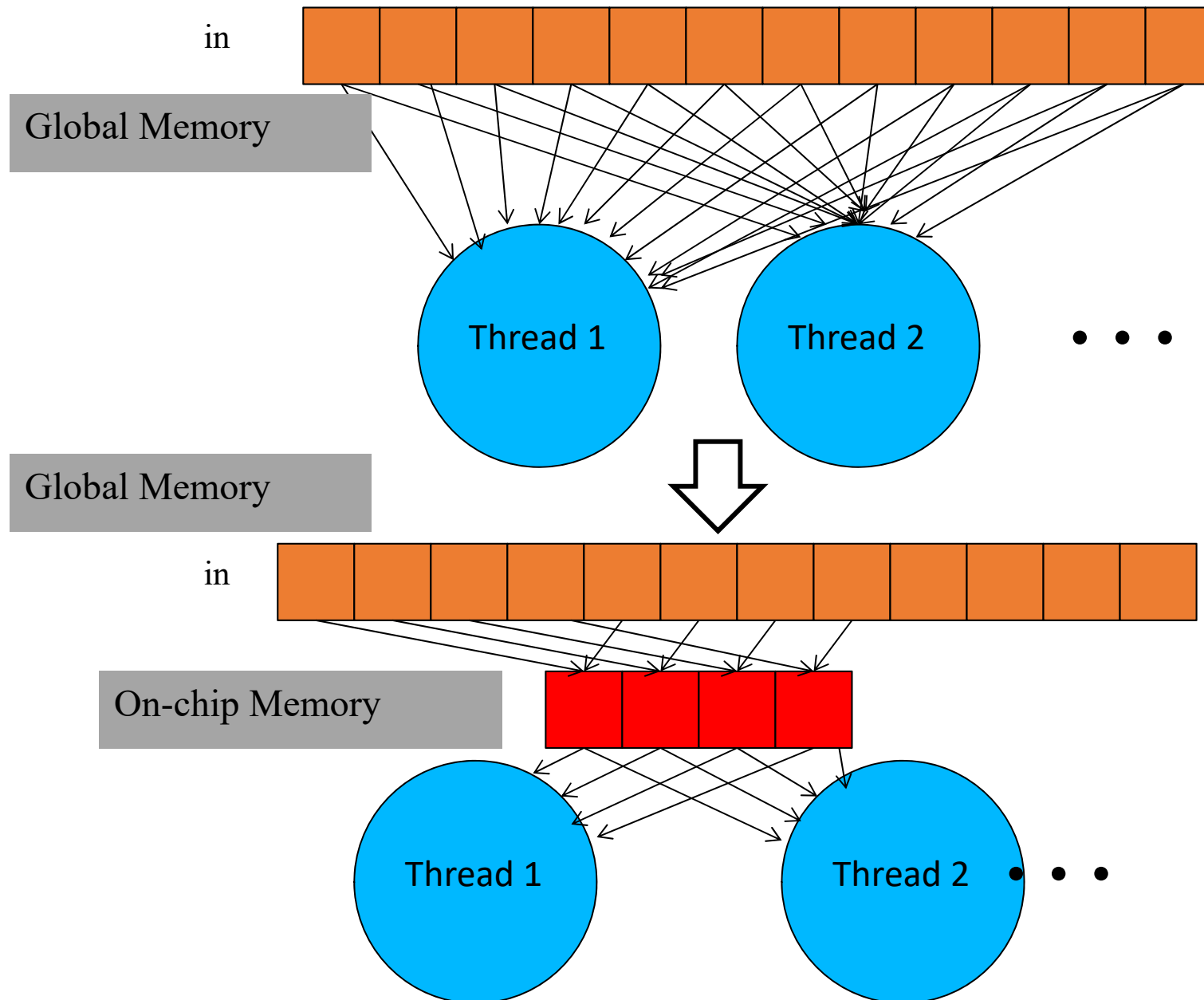
CUDA device SM

# A Common Programming Strategy

- Global memory resides in device memory (DRAM)
- A profitable way of performing computation on the device is to **tile the input data** to take advantage of fast shared memory:
  - **Partition** data into **subsets** (tiles) that fit into shared memory
  - Handle **each data subset with one thread block** by:
    - Loading the subset from global memory to shared memory, **using multiple threads to exploit memory-level parallelism**
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

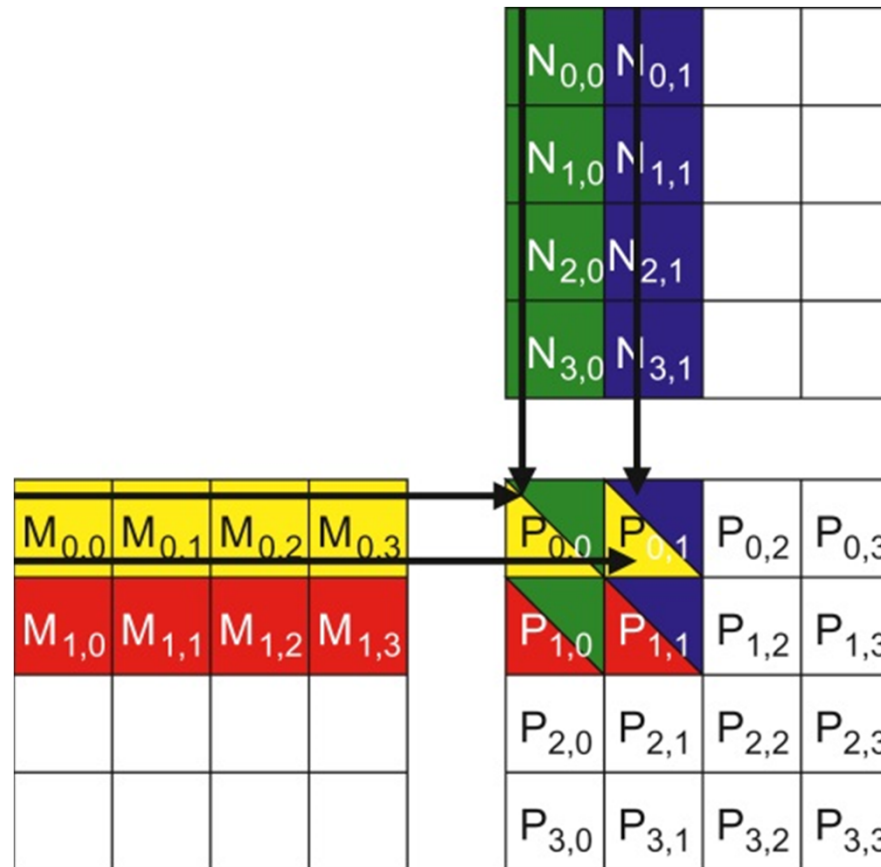
# Shared Memory Blocking Basic Idea



# Outline of Technique

- Identify a tile of global data that are accessed by multiple threads
- Load the 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

# The small example




**FIGURE 4.9:** For brevity, we show  $M[y*Width + x]$ ,  $N[y*Width + x]$ ,  $P[y*Width + x]$  as  $M_{y,x}$ ,  $N_{y,x}$ ,  $P_{y,x}$ .



# The small example

Memory access of Block(0,0)

Access order 

thread <sub>0,0</sub>	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
thread <sub>0,1</sub>	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
thread <sub>1,0</sub>	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
thread <sub>1,1</sub>	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

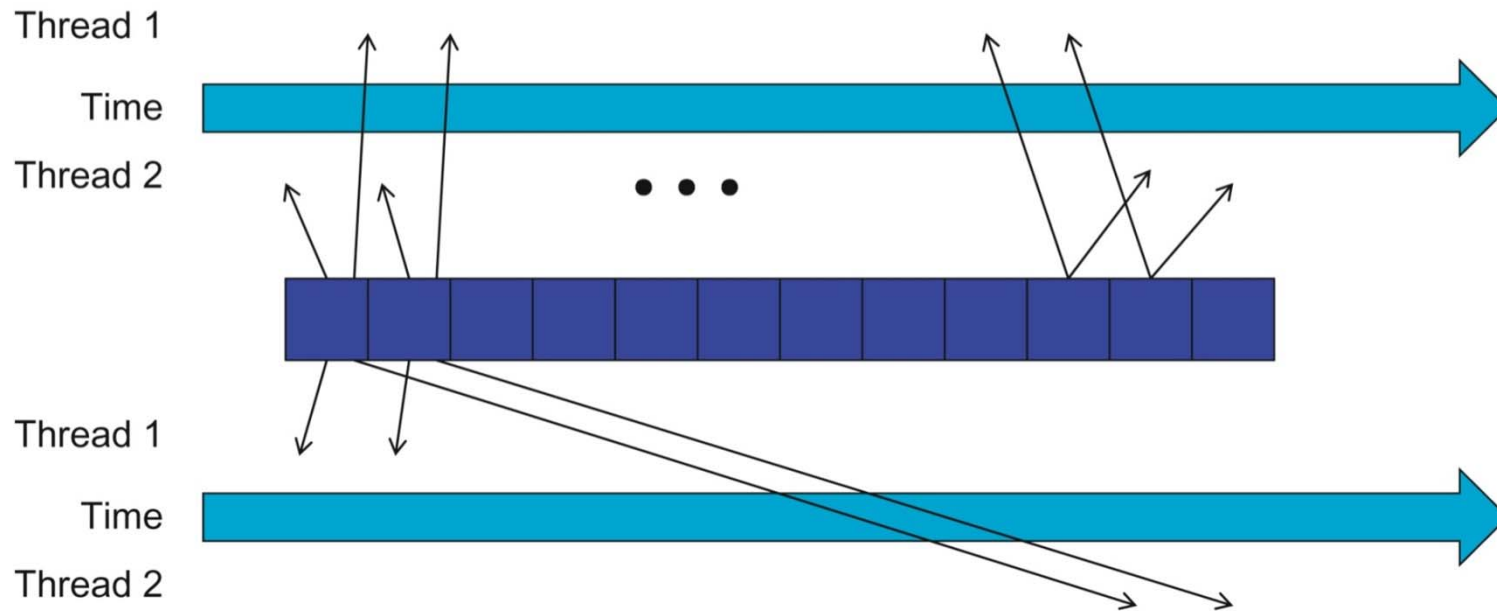
**Every M and N element is accessed twice.**

**The potential reduction in global memory traffic is proportional to the dimension of blocks used.**



**Tiled algorithm are highly similar to carpooling arrangement.**  
With extra effort.

**Good — threads have similar access timing**

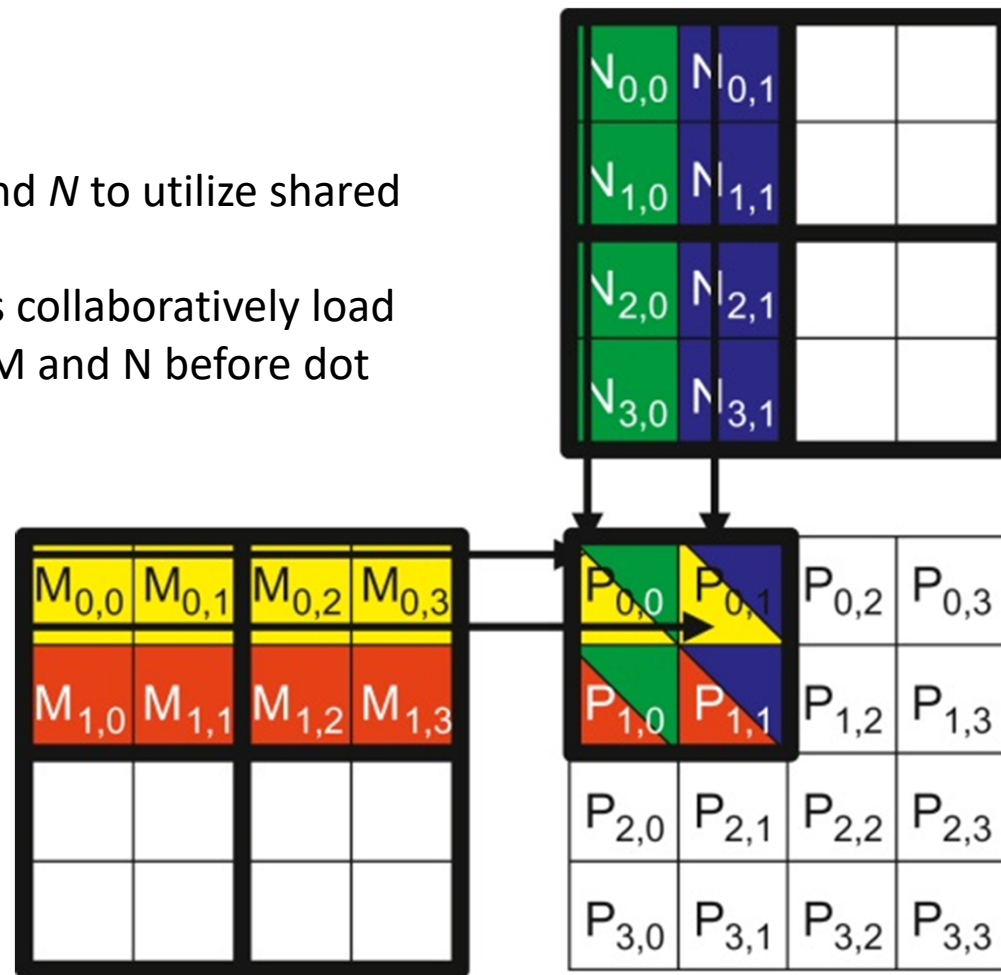


**Bad — threads have very different timing**

**FIGURE 4.13:** Tiled Algorithms require synchronization among threads.

# The small example

Tiling  $M$  and  $N$  to utilize shared memory  
All threads collaboratively load subset of  $M$  and  $N$  before dot product.



# The small example

	Phase 1			Phase 2		
thread <sub>0,0</sub>	<b>M<sub>0,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>N<sub>0,0</sub></b> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>0,1</sub> *Nds <sub>1,0</sub>	<b>M<sub>0,2</sub></b> ↓ Mds <sub>0,0</sub>	<b>N<sub>2,0</sub></b> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>0,1</sub> *Nds <sub>1,0</sub>
thread <sub>0,1</sub>	<b>M<sub>0,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>N<sub>0,1</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>0,1</sub> += Mds <sub>0,0</sub> *Nds <sub>0,1</sub> + Mds <sub>0,1</sub> *Nds <sub>1,1</sub>	<b>M<sub>0,3</sub></b> ↓ Mds <sub>0,1</sub>	<b>N<sub>2,1</sub></b> ↓ Nds <sub>0,1</sub>	PValue <sub>0,1</sub> += Mds <sub>0,0</sub> *Nds <sub>0,1</sub> + Mds <sub>0,1</sub> *Nds <sub>1,1</sub>
thread <sub>1,0</sub>	<b>M<sub>1,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>N<sub>1,0</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>1,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,0</sub>	<b>M<sub>1,2</sub></b> ↓ Mds <sub>1,0</sub>	<b>N<sub>3,0</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>1,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,0</sub>
thread <sub>1,1</sub>	<b>M<sub>1,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>N<sub>1,1</sub></b> ↓ Nds <sub>1,1</sub>	PValue <sub>1,1</sub> += Mds <sub>1,0</sub> *Nds <sub>0,1</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>	<b>M<sub>1,3</sub></b> ↓ Mds <sub>1,1</sub>	<b>N<sub>3,1</sub></b> ↓ Nds <sub>1,1</sub>	PValue <sub>1,1</sub> += Mds <sub>1,0</sub> *Nds <sub>0,1</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>

time →

Execution phases of a tiled matrix multiplication.

# Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

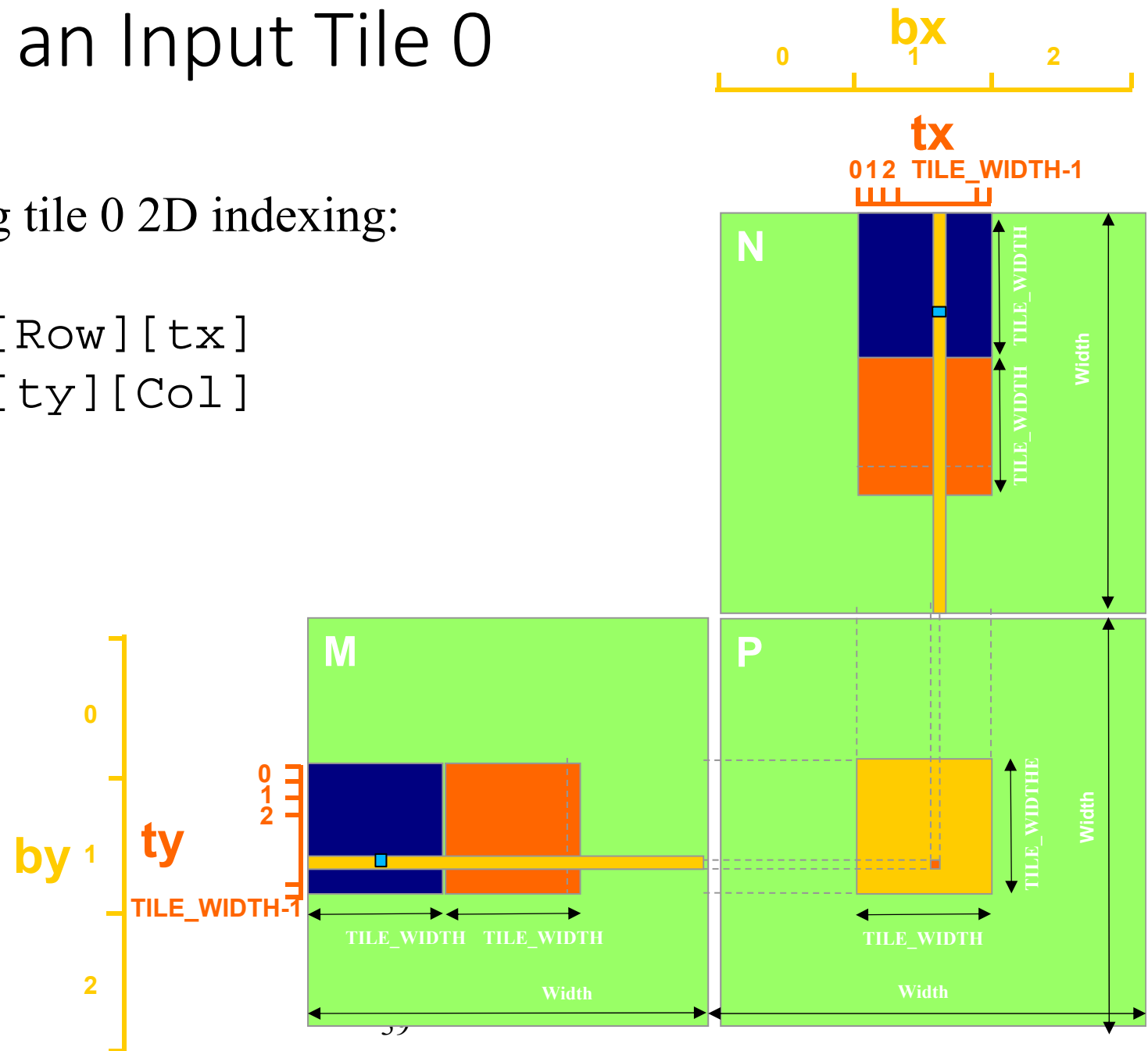
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
    // Loop over the M and N tiles required to compute the P element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of M and N tiles into shared memory
9.      subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.     subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.     __syncthreads();
15. }
16. P[Row*Width+Col] = Pvalue;
}
```

# Loading an Input Tile 0

Accessing tile 0 2D indexing:

`M[Row][tx]`

`N[ty][Col]`

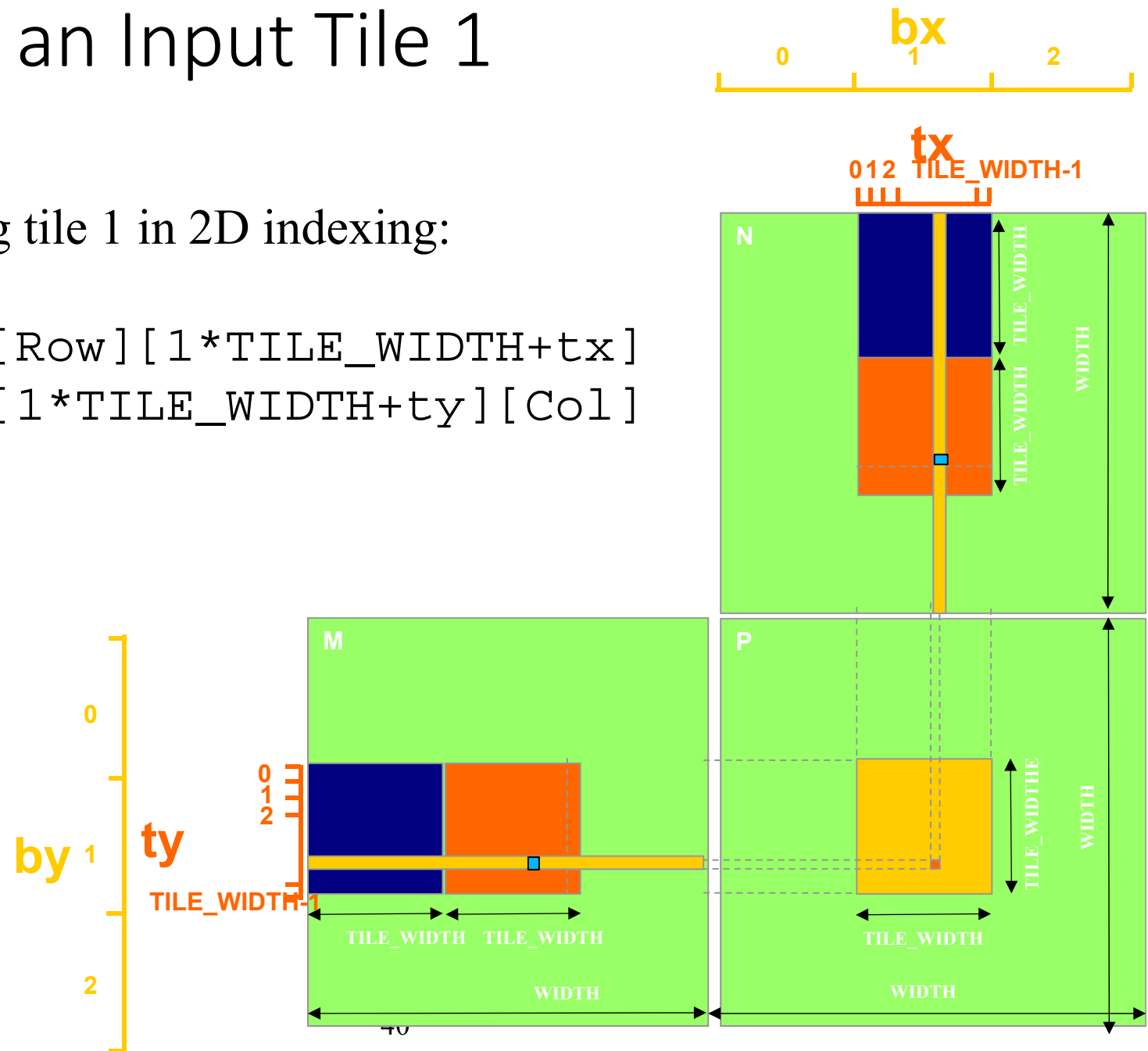


# Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

$M[\text{Row}][1 * \text{TILE\_WIDTH} + tx]$

$N[1 * \text{TILE\_WIDTH} + ty][\text{Col}]$





# Loading an Input Tile m

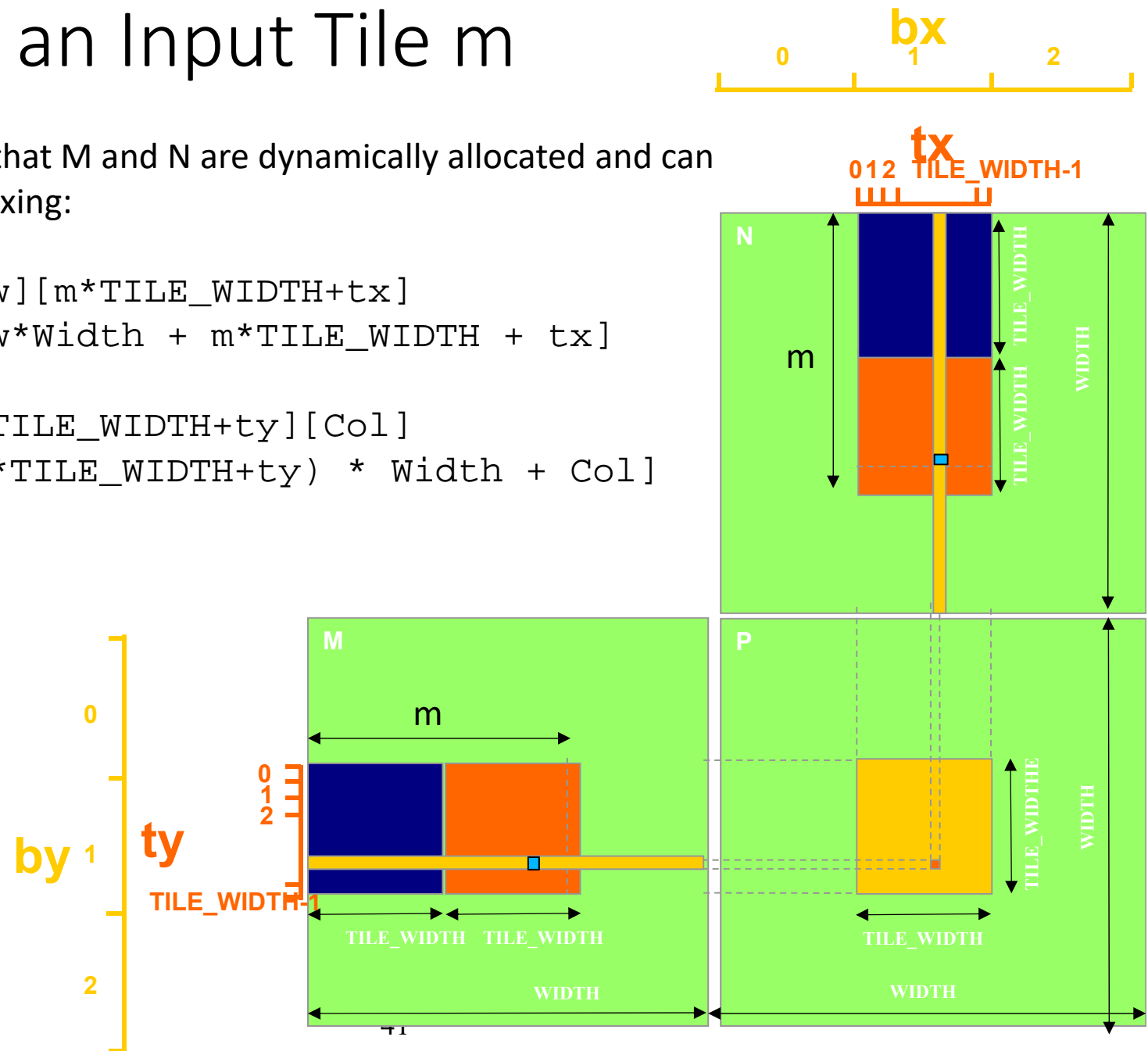
However, recall that M and N are dynamically allocated and can only use 1D indexing:

```
M[Row][m*TILE_WIDTH+tx]
```

```
M[Row*Width + m*TILE_WIDTH + tx]
```

```
N[m*TILE_WIDTH+ty][Col]
```

```
N[(m*TILE_WIDTH+ty) * Width + Col]
```



# Barrier Synchronization

- An API function call in CUDA
  - `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed

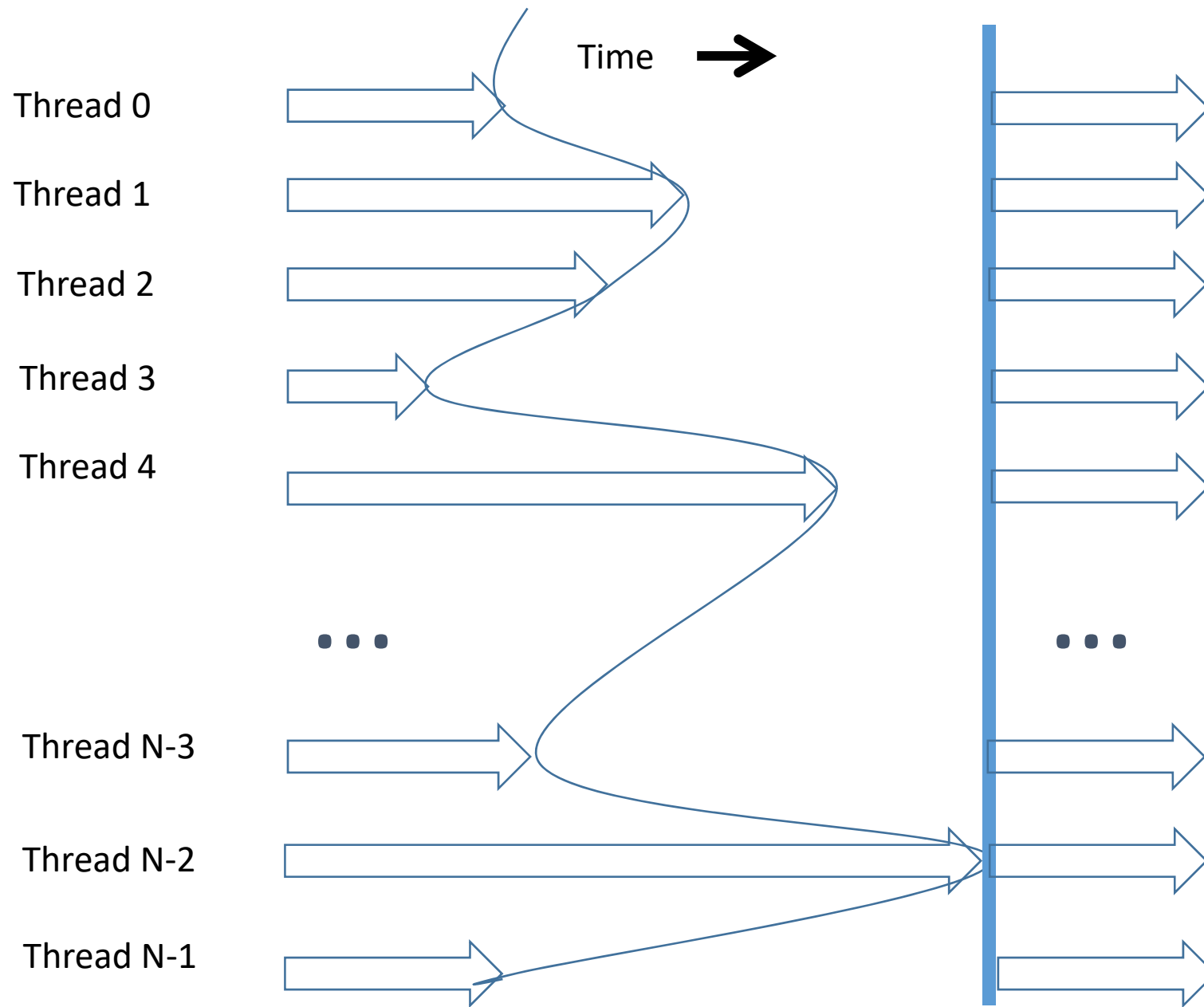
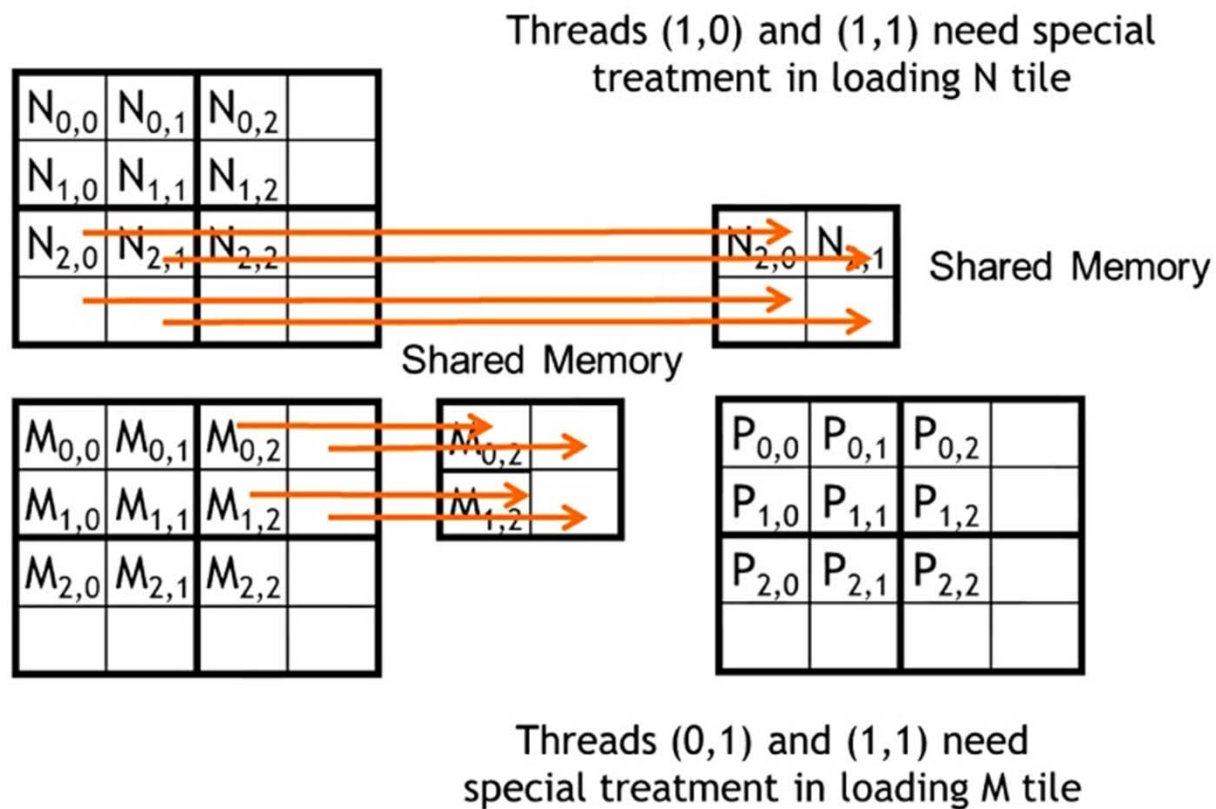


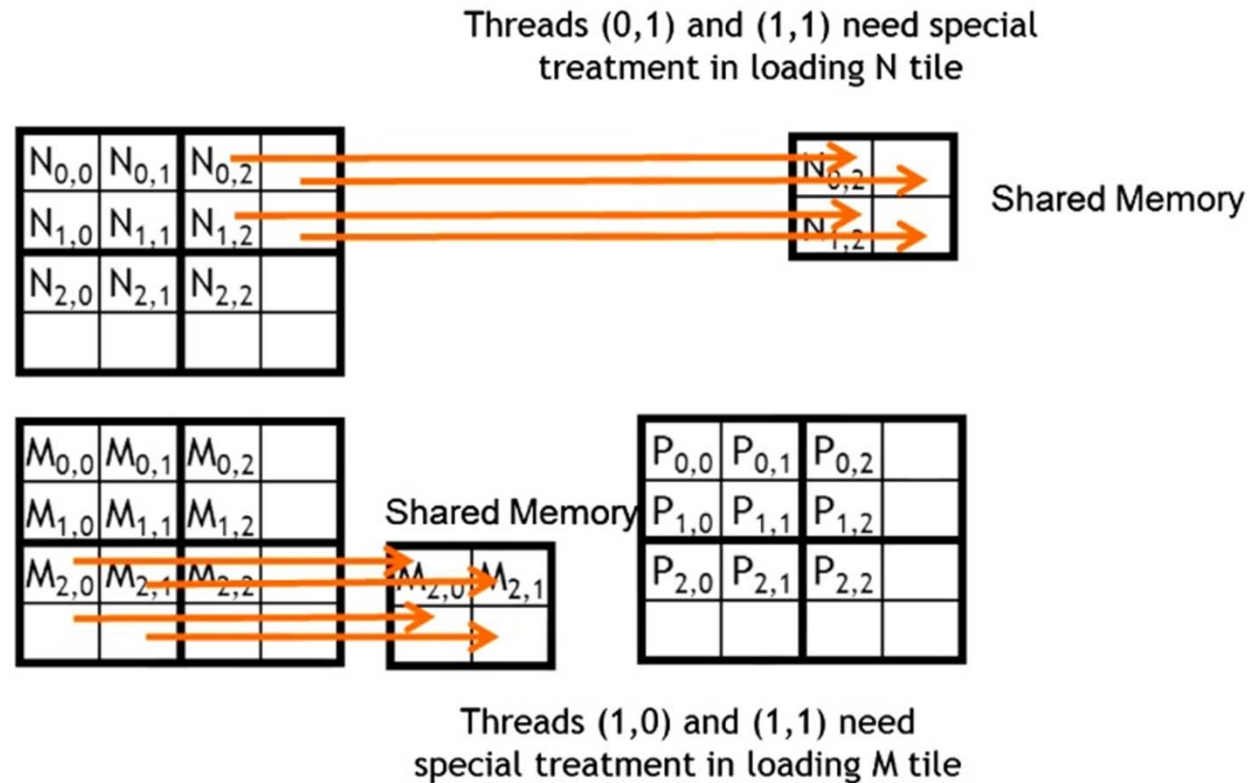
Figure 4.11 An example execution timing of barrier synchronization.

# Boundary Checks



**FIGURE 4.18:** Loading input matrix elements that are close to the edge—phase 1 of Block0,0.

# Boundary Checks



**FIGURE 4.19:** Loading input elements during phase 0 of block1,0.

# Boundary Checks

```
// Loop over the M and N tiles required to compute P element
8.   for (int ph = 0; ph < ceil(Width/(float)TILE_WIDTHH); ++ph) {

        // Collaborative loading of M and N tiles into shared memory
9.   if ((Row< Width) && (ph*TILE_WIDTHH+tx)< Width)
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTHH + tx];
10.  if ((ph*TILE_WIDTHH+ty)<Width && Col<Width)
        Nds[ty][tx] = N[(ph*TILE_WIDTHH + ty)*Width + Col];

11.  __syncthreads();

12.  for (int k = 0; k < TILE_WIDTHH; ++k) {
13.      Pvalue += Mds[ty][k] * Nds[k][tx];
        }
14.  __syncthreads();
    }
15.  if ((Row<Width) && (Col<Width)) P[Row*Width + Col] = Pvalue;
```

**FIGURE 4.20:** Tiled matrix multiplication kernel with boundary condition checks.

# Content

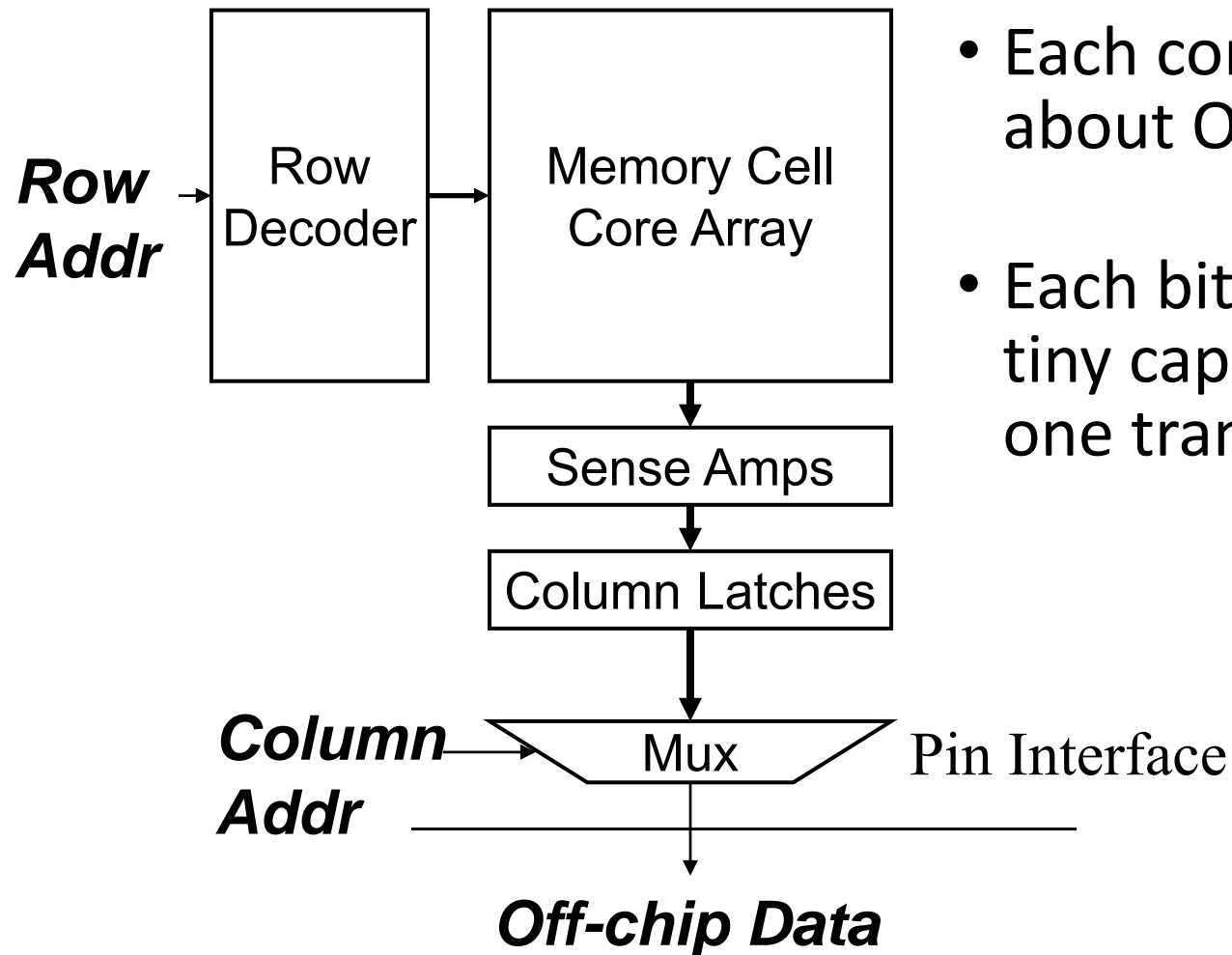
- Global Memory (DRAM) Bandwidth
- Ideal



- Reality



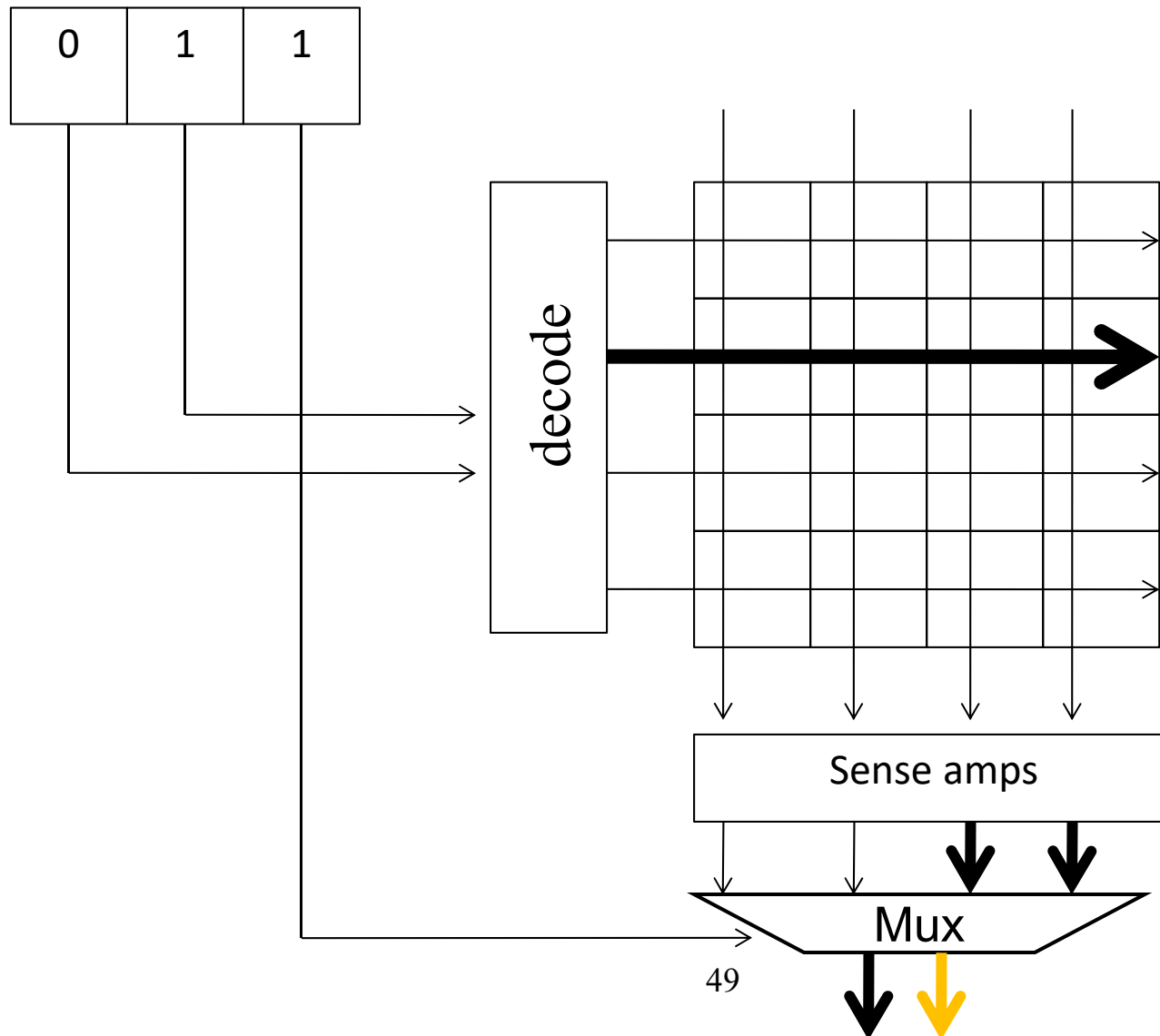
# DRAM Bank Organization



- Each core array has about  $O(1M)$  bits
- Each bit is stored in a tiny capacitor, made of one transistor

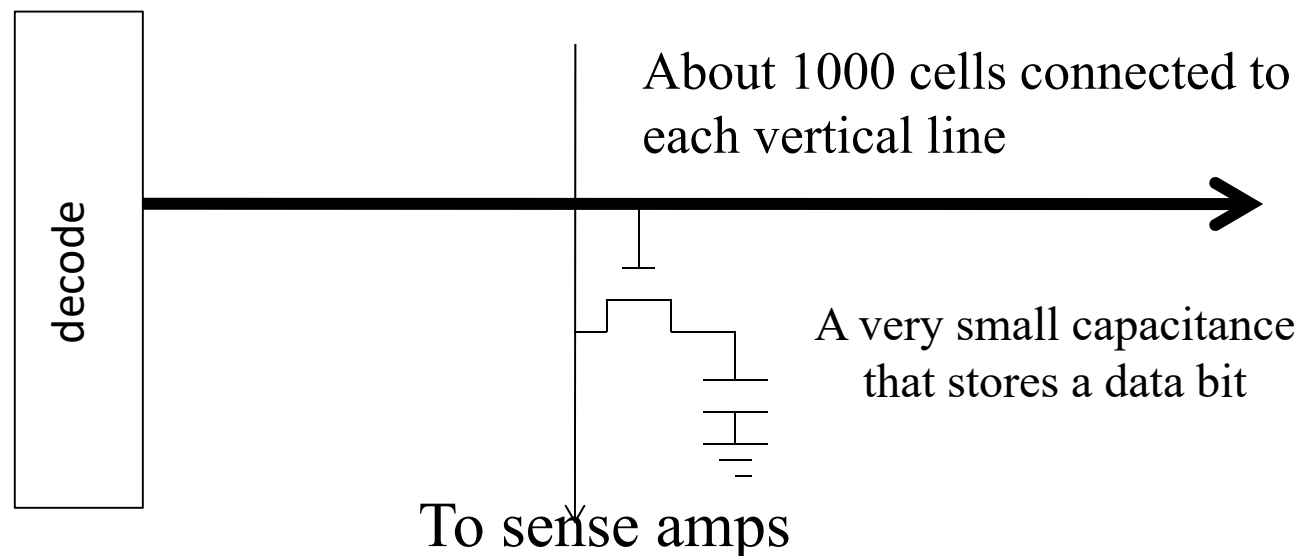


# A very small DRAM Bank

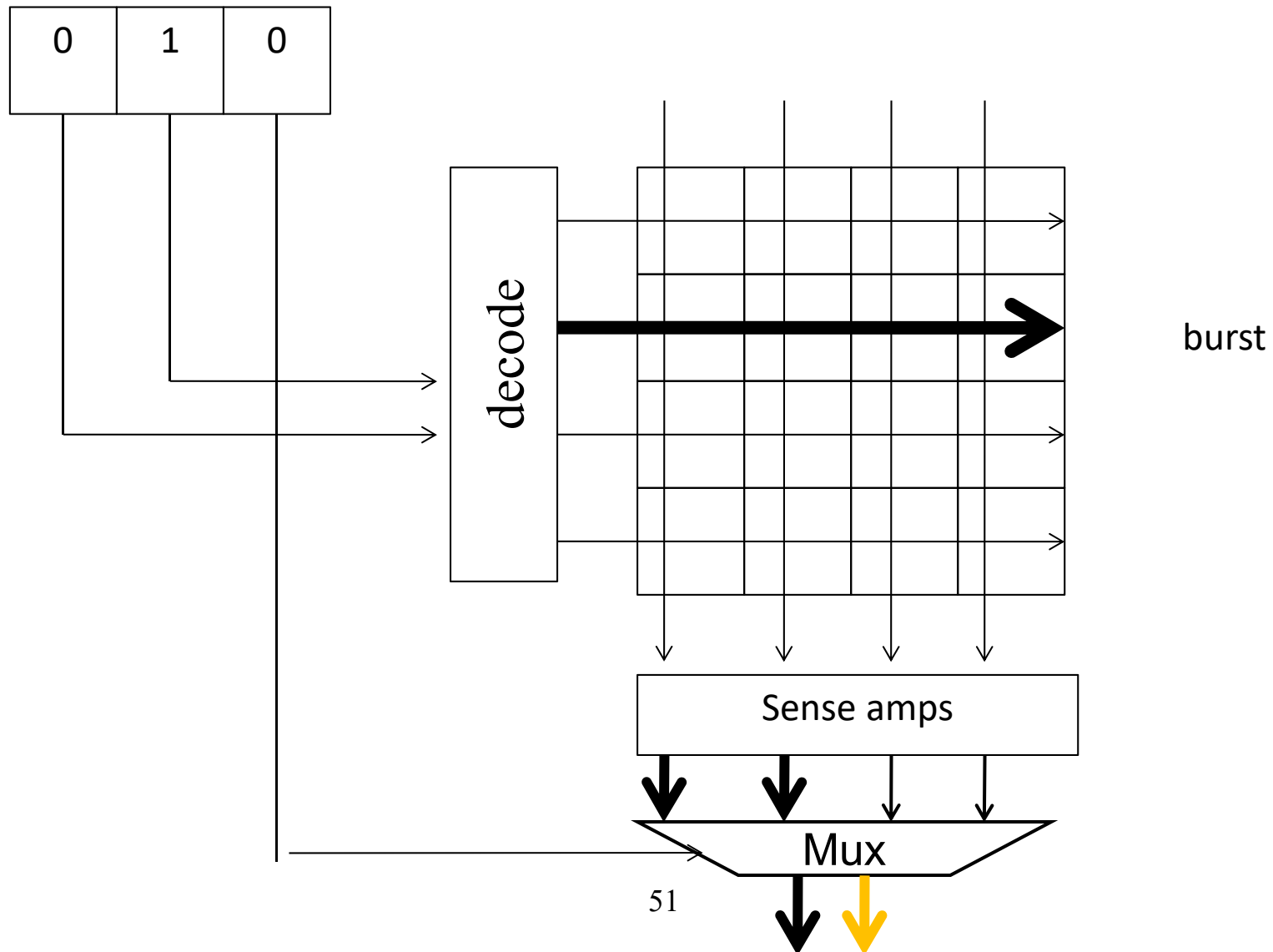


# DRAM core arrays are slow.

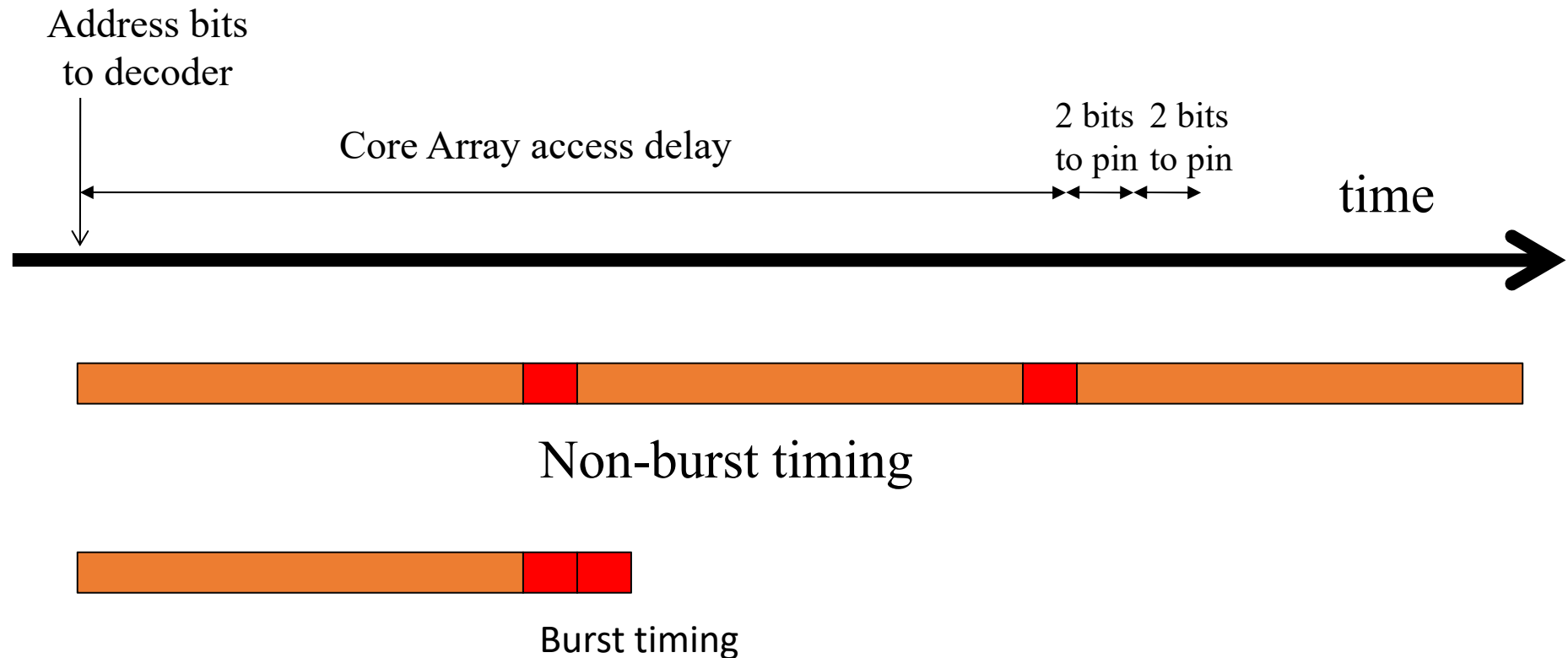
- Reading from a cell in the core array is a very slow process
  - 10s of nanoseconds v.s. sub-nanosecond clock cycle
  - Like determine the flavor of cup of coffee by smelling far away



# DRAM Bursting (burst size = 4 bits)

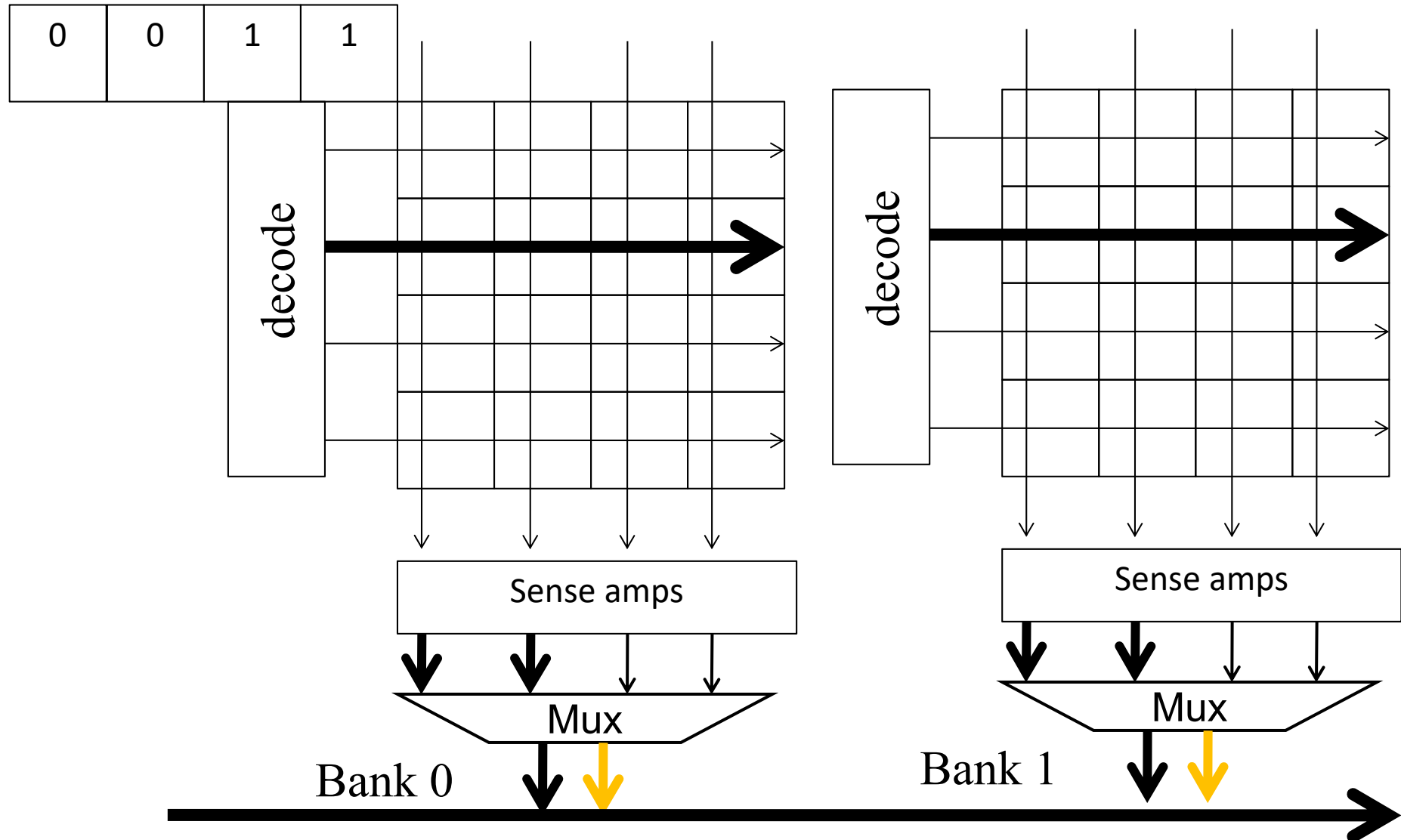


# DRAM Bursting for the 8x2 Bank

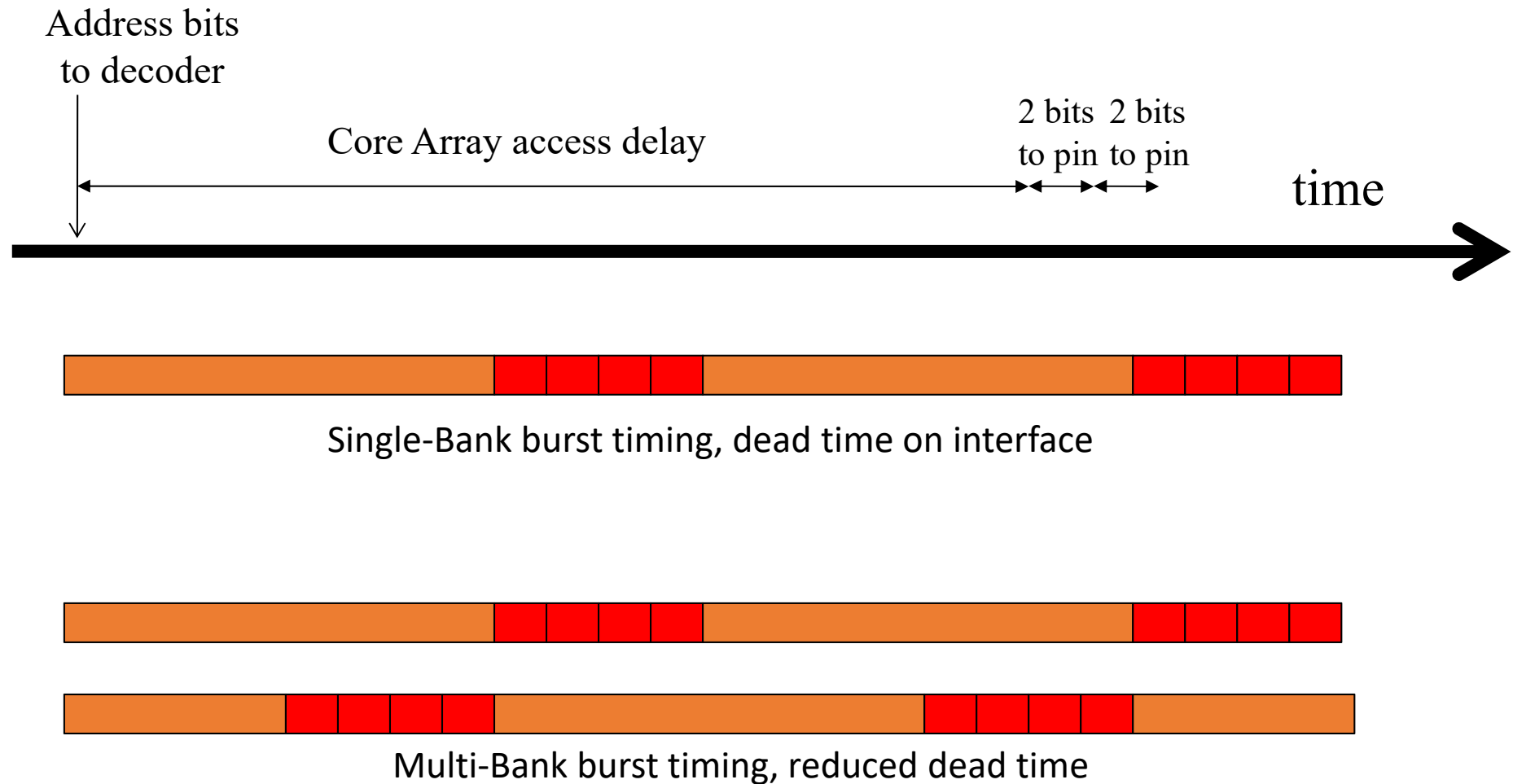


Modern DRAM systems are designed to **be always accessed in burst mode**.  
Burst bytes are transferred but discarded when accesses are not to sequential locations.

# Multiple DRAM Banks



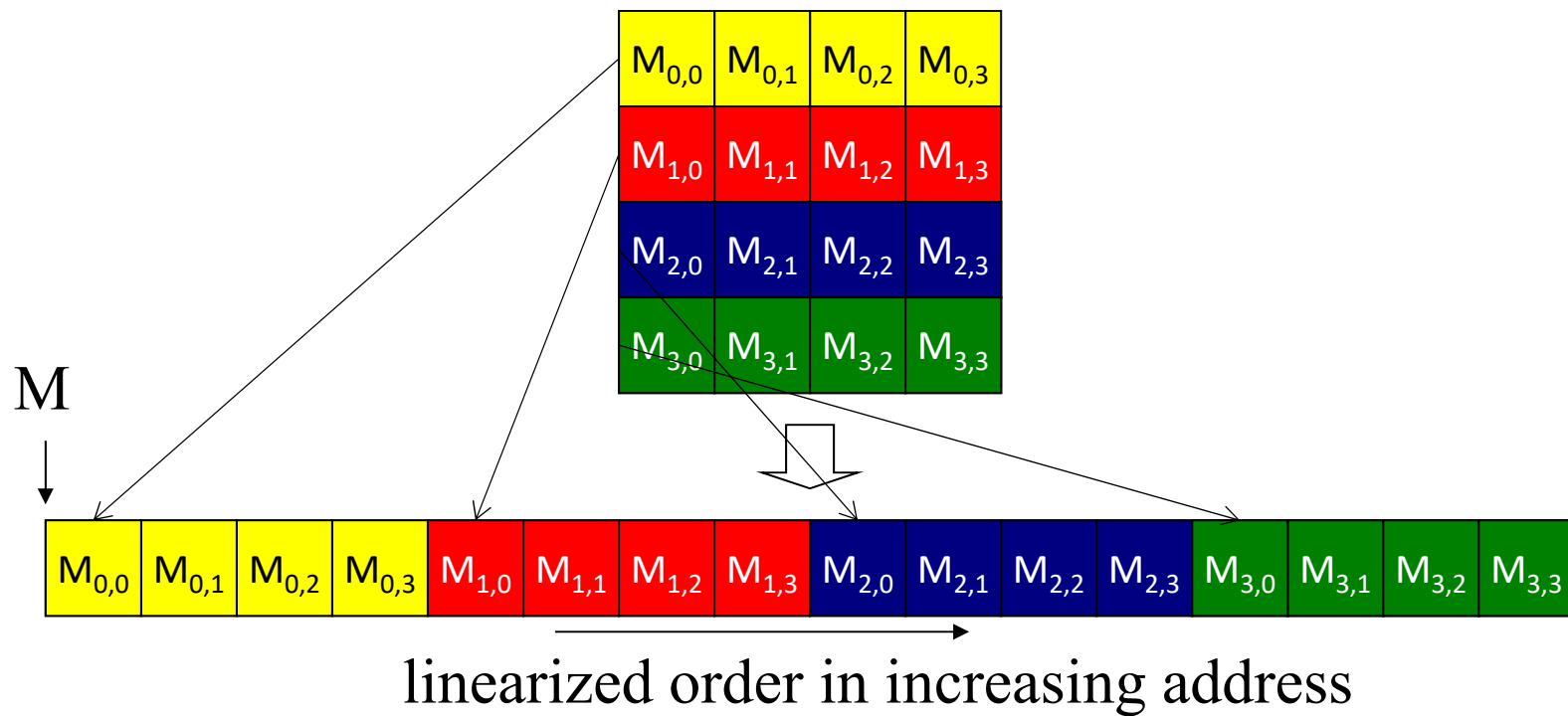
# DRAM Bursting for the 8x2 Bank



# Coalesce

- Take advantages of the fact that threads in a **warp** execute the same instruction at any given time.
- Modern DRAM systems are designed to be always accessed in **burst** mode.
- The favorable access pattern is achieved When all threads in a warp access consecutive global memory locations.
- In this case, the hardware combines, or **coalesces**, all these accesses into a consolidated access to consecutive DRAM locations (burst).

# Placing a 2D C array into linear memory space (review)





# A Simple Matrix Multiplication Kernel (review)

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

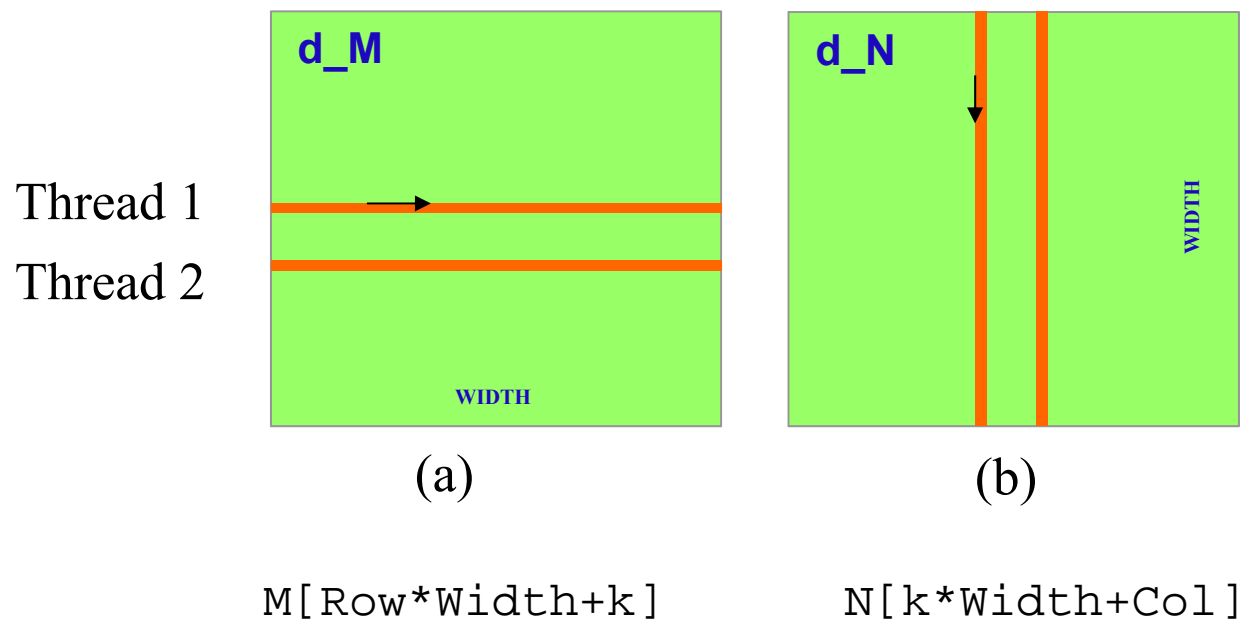
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k)
            Pvalue += M[Row*Width+k] * N[k*Width+Col];

        P[Row*Width+Col] = Pvalue;
    }
}
```

# Two Access Patterns

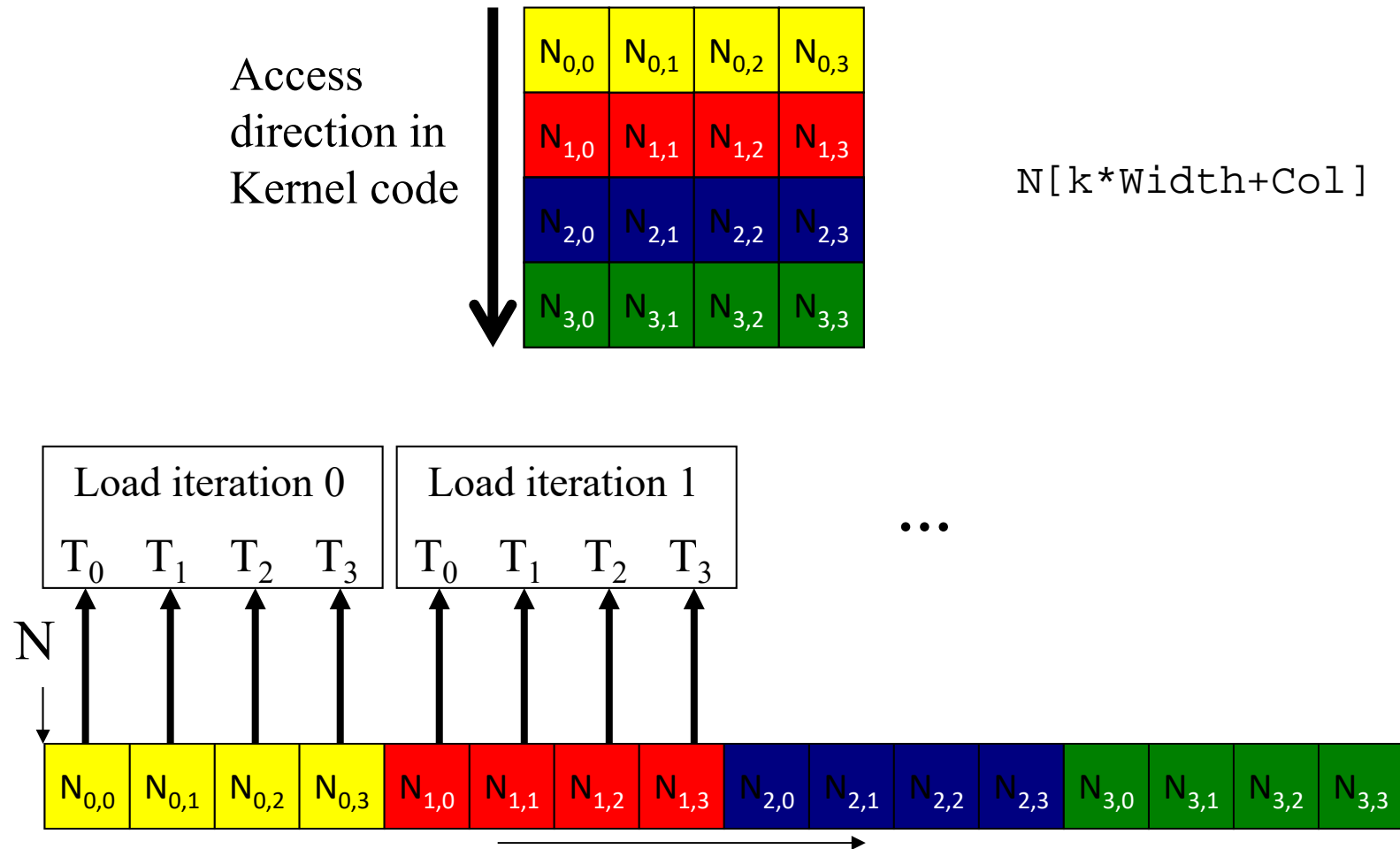
For M, threads in a warp read adjacent rows;

For N, threads in a warp read adjacent columns;

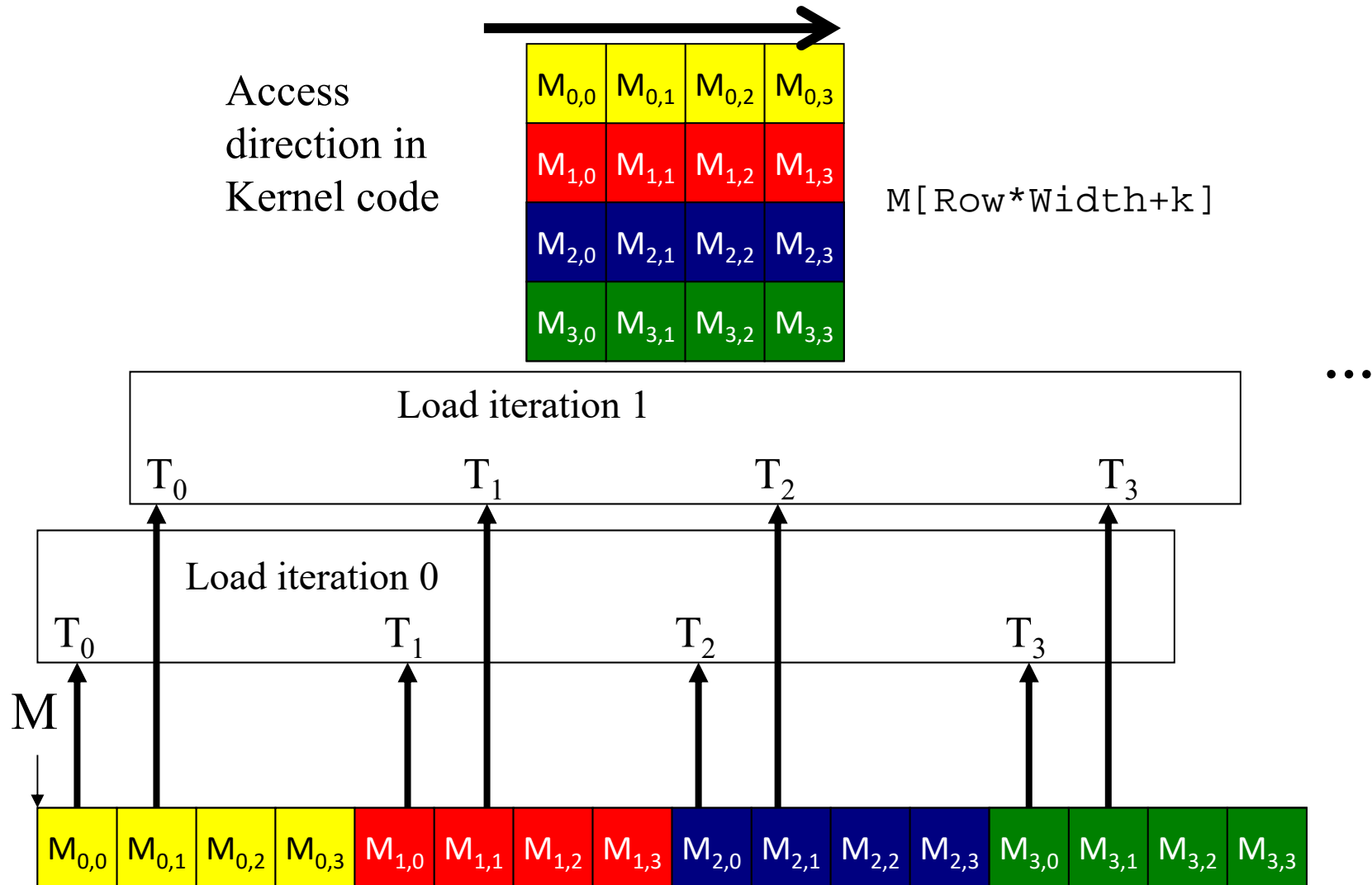


k is loop counter in the inner product loop of the kernel code

N accesses are coalesced.

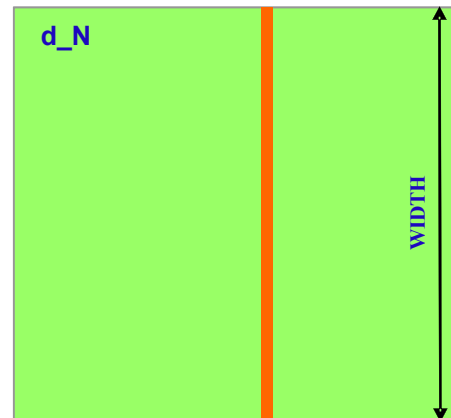
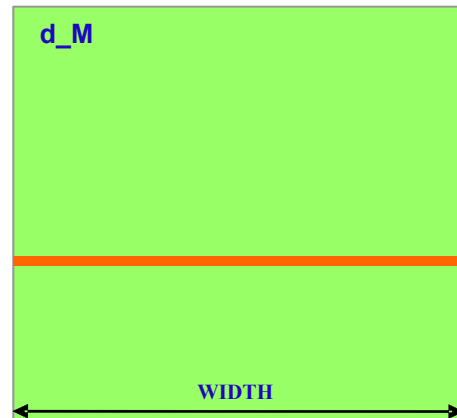


# M accesses are not coalesced.



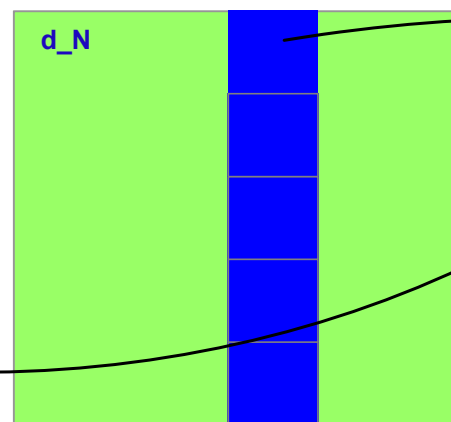
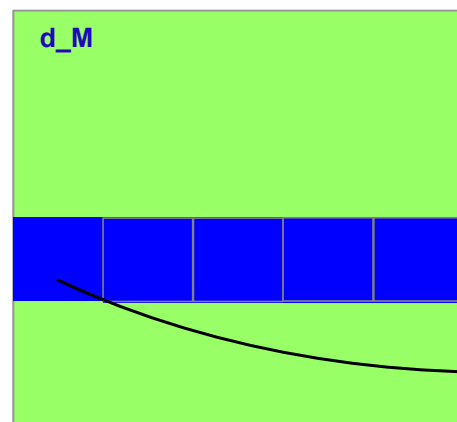
# Use shared memory to enable coalescing in tiled matrix multiplication

Original  
Access  
Pattern

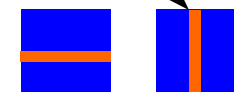


Corner turning

Tiled  
Access  
Pattern



Copy into  
scratchpad  
memory



Perform  
multiplication  
with scratchpad  
values

# Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
7.  float Pvalue = 0;
    // Loop over the M and N tiles required to compute the P element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of M and N tiles into shared memory
9.      subTileM[m][tx] = M[
10.         ?
11.     ];
12.     subTileN[m][ty] = N[
13.         ?
14.     ];
15.     __syncthreads();
16.     for (int k = 0; k < TILE_WIDTH; ++k)
17.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
18.     __syncthreads();
19. }
20. P[Row*Width+Col] = Pvalue;
}
```