

---

**CS133**  
**Parallel & Distributed Computing**  
**Performance Optimization with CUDA**

**Instructor: Jason Cong**  
**[cong@cs.ucla.edu](mailto:cong@cs.ucla.edu)**

# ***Review of Previous Lecture***

---

## **◆ CUDA programming model**

- **Two level of parallelism**
- **Threads and thread blocks**

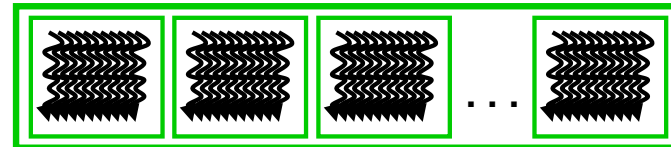
## **◆ CPU-GPU communication**

# CUDA/OpenCL – Execution Model

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD (Single Program Multiple Data) kernel C code

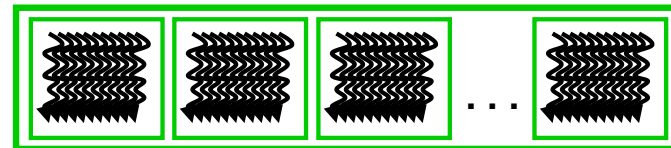
Serial Code (host)

Parallel Kernel (device)  
`KernelA<<< nBlk, nTid >>>(args);`



Serial Code (host)

Parallel Kernel (device)  
`KernelB<<< nBlk, nTid >>>(args);`



# Heterogeneous Computing

## vecAdd Host Code

```
#include <cuda.h>
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n* sizeof(float);
    float* A_d, B_d, C_d;
    ...
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code - to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```

```

void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float* A_d, B_d, C_d;

1. // Transfer A and B to device memory
   cudaMalloc((void **) &A_d, size);
   cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
   cudaMalloc((void **) &B_d, size);
   cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

   // Allocate device memory for
   cudaMalloc((void **) &C_d, size);

2. // Kernel invocation code - to be shown later
   ...
3. // Transfer C from device to host
   cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
   // Free device memory for A, B, C
   cudaFree(A_d); cudaFree(B_d); cudaFree(C_d);
}

```

# Example: Vector Addition Kernel

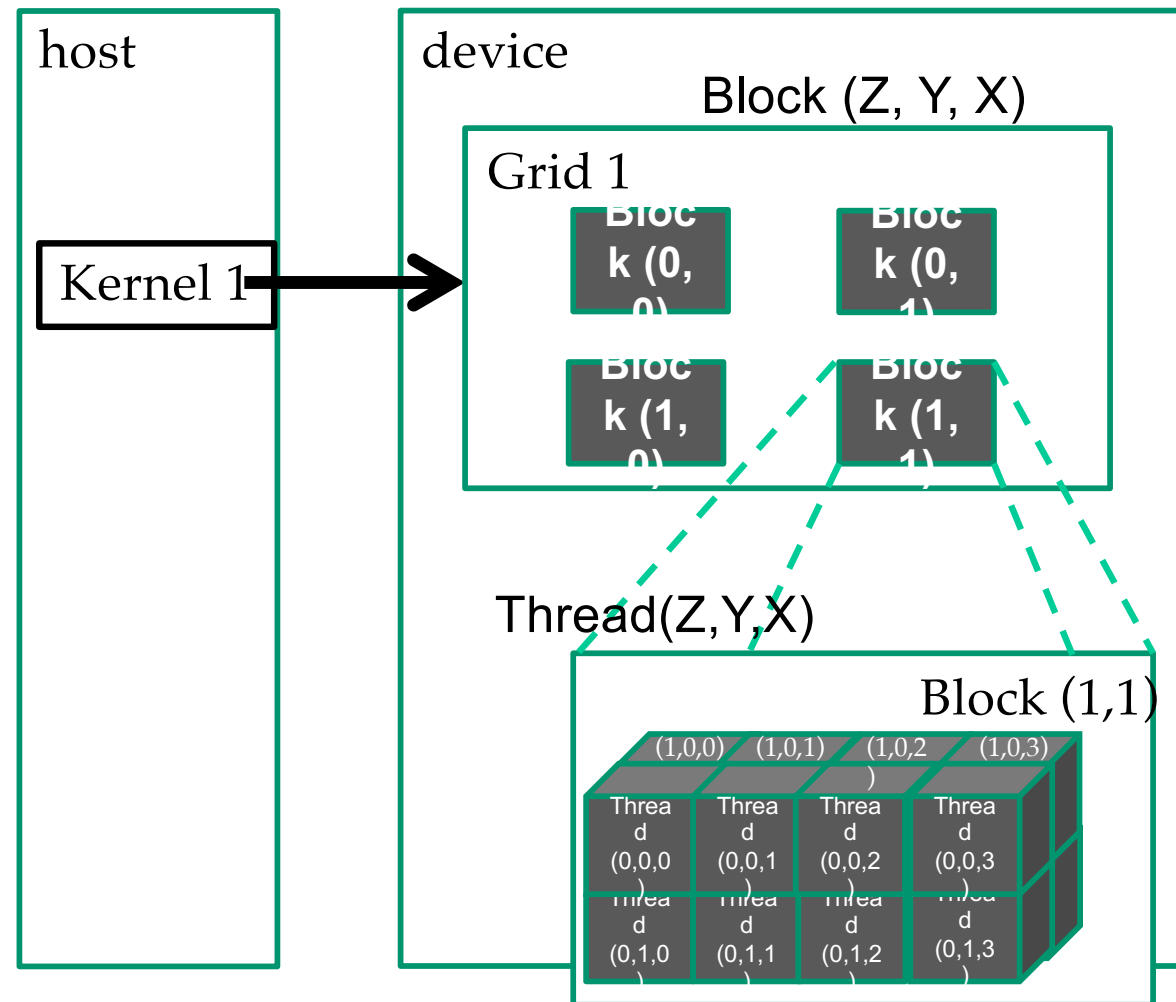
## Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if(i < n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>>(A_d, B_d, C_d, n);
}
```

# CUDA Thread Grids are Multi-Dimensional



# Lecture Today

- Tiled matrix multiplication example
- Understand the performance implications of global memory accesses
- Use of shared memory
- Memory coalescing



# MM Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant, assuming square matrix
dim3 dimGrid(ceil(Width/(TILE_WIDTH*1.0)),      X dimension
              ceil(Width/(TILE_WIDTH*1.0)), 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

You need to extend the code to handle  
rectangular matrix by yourself

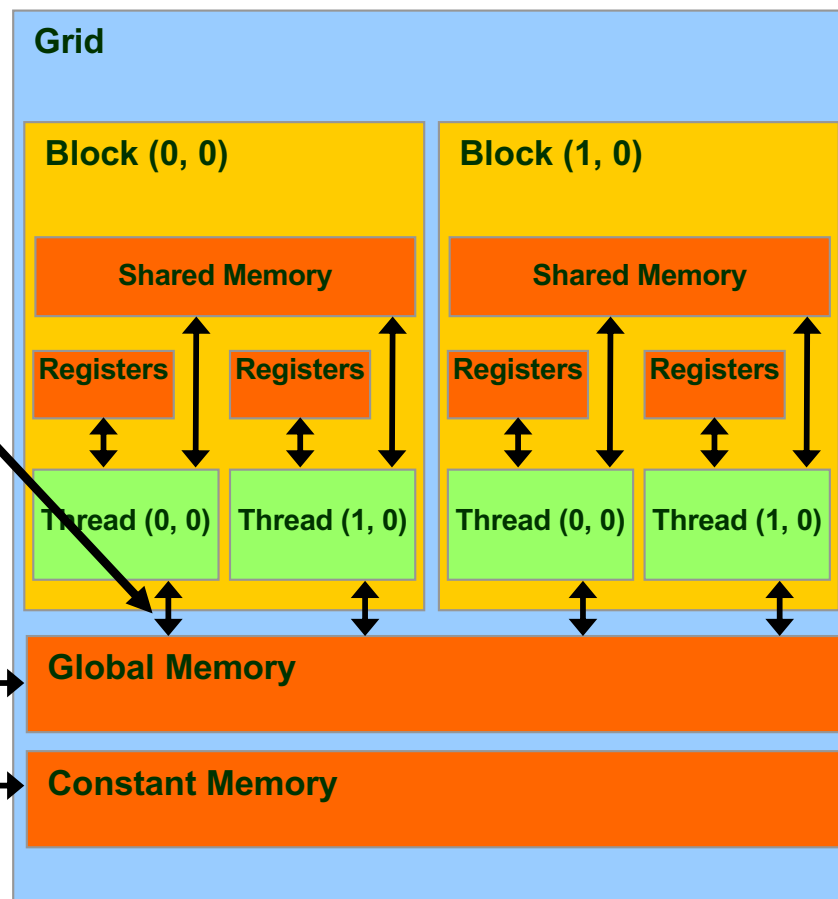
# A Simple Matrix Multiplication Kernel (Simplified Dimension and Syntax!)

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column index of d_P and d_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 += d_M[Row][k] * d_N[k][Col];
        }
        d_P[Row][Col] = Pvalue;
    }
}
```

# 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 single-precision floating point multiply-add
  - Two operands need to be fetched for each two floating-point operations (\* and +)
  - Each floating-point operation needs 4 bytes of operand
  - 150 GB/s limits the code at 37.5 (150/4) 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



## ***You Would Say: I know the Problem!***

---

- ◆ **No data reuse! Too much data movement!!**
- ◆ **Need to tile the computation so that all the tiles fit in cache to enable data reuse.**
- ◆ **But, where is the cache on GPU?**



# Tiled Matrix-Matrix Multiplication using Shared Memory

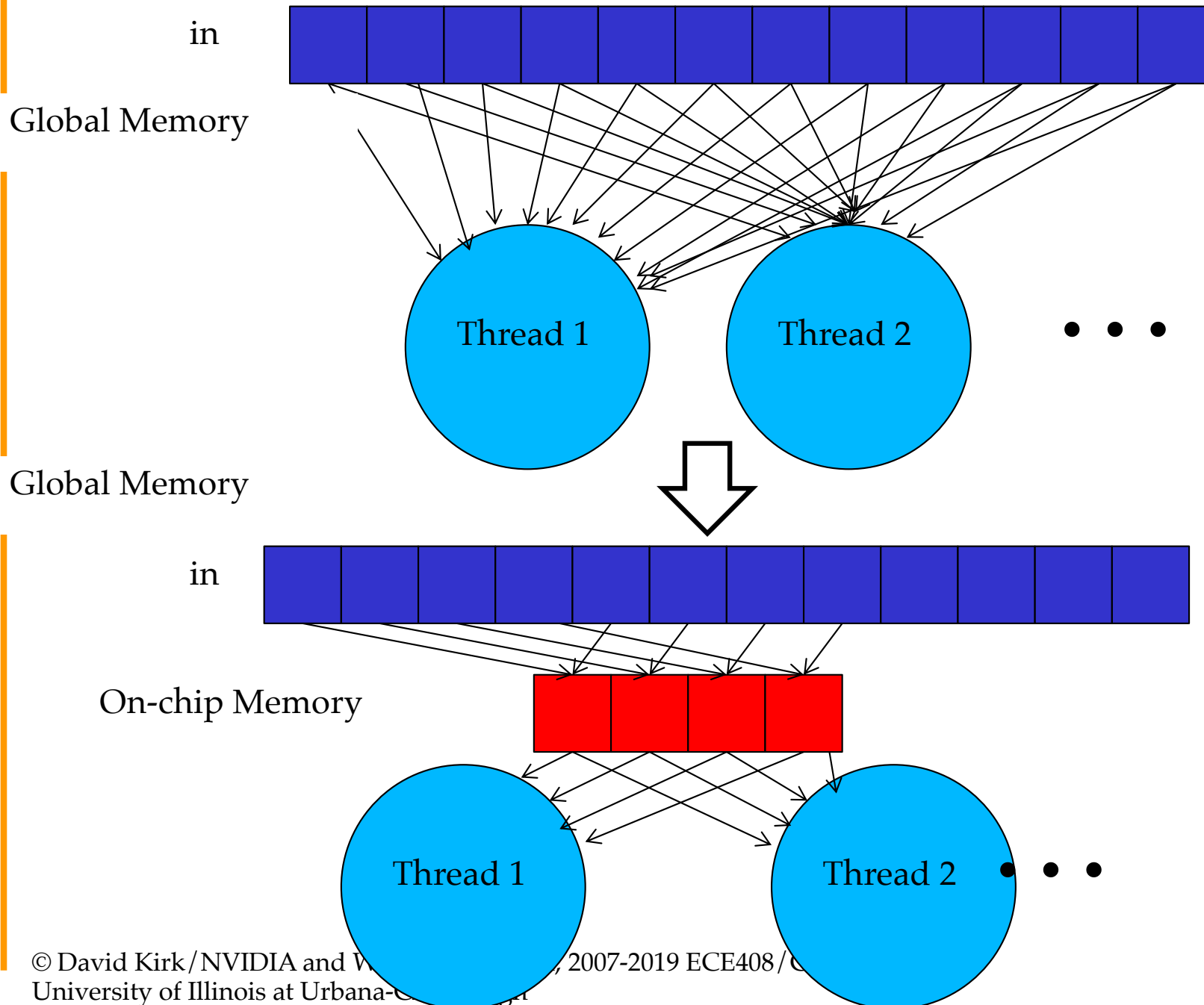
# A Common Programming Strategy

- Global memory is implemented with DRAM - slow
- 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 the (smaller but faster) *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 access any data element
    - Copying results from shared memory to global memory
  - Tiles are also called blocks in the literature

# Declaring Shared Memory Arrays

```
__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 Tiling Basic Idea



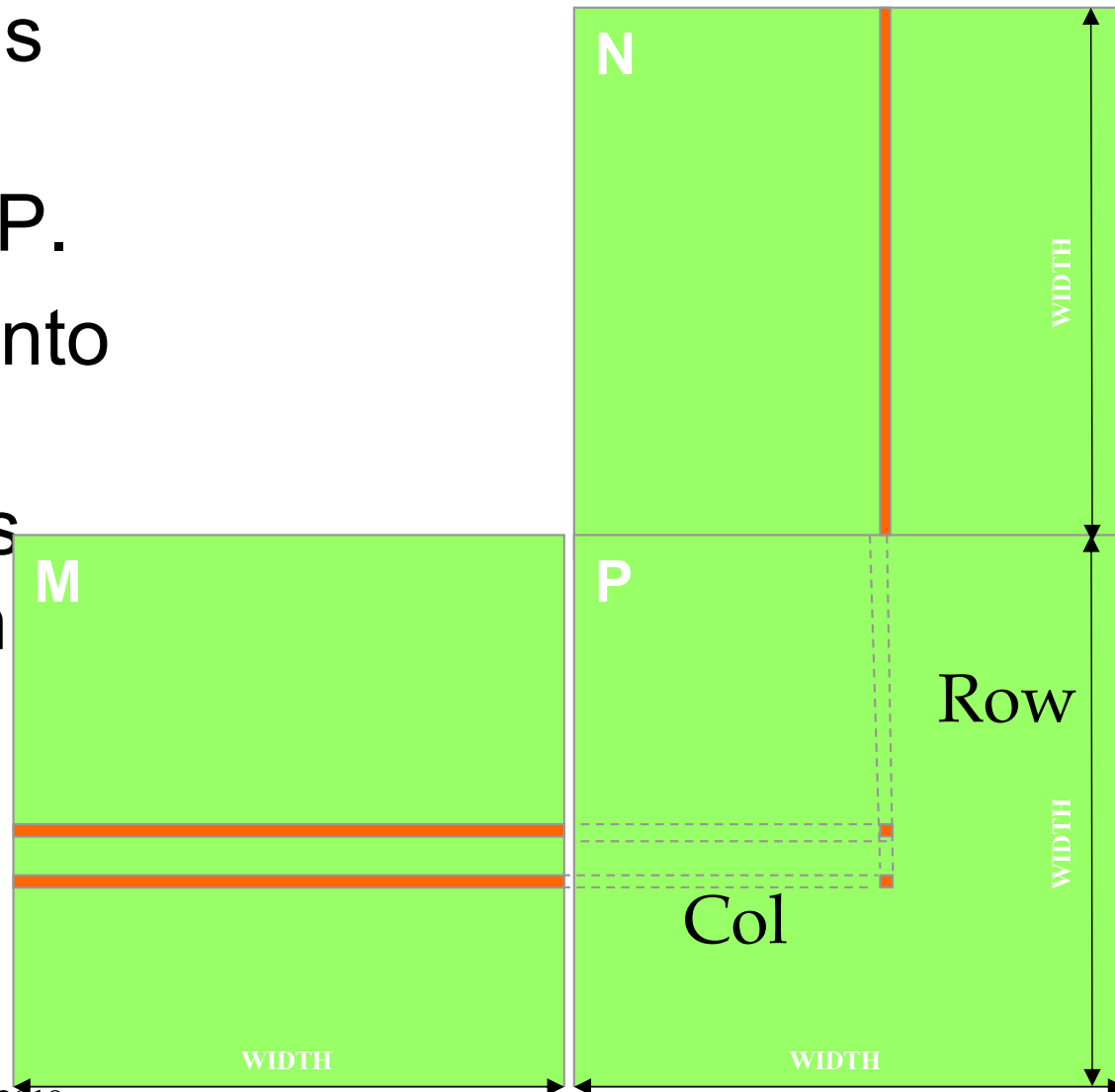


# Outline of Technique

- 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
- > You (the programmer) are the “cache replacement controller”

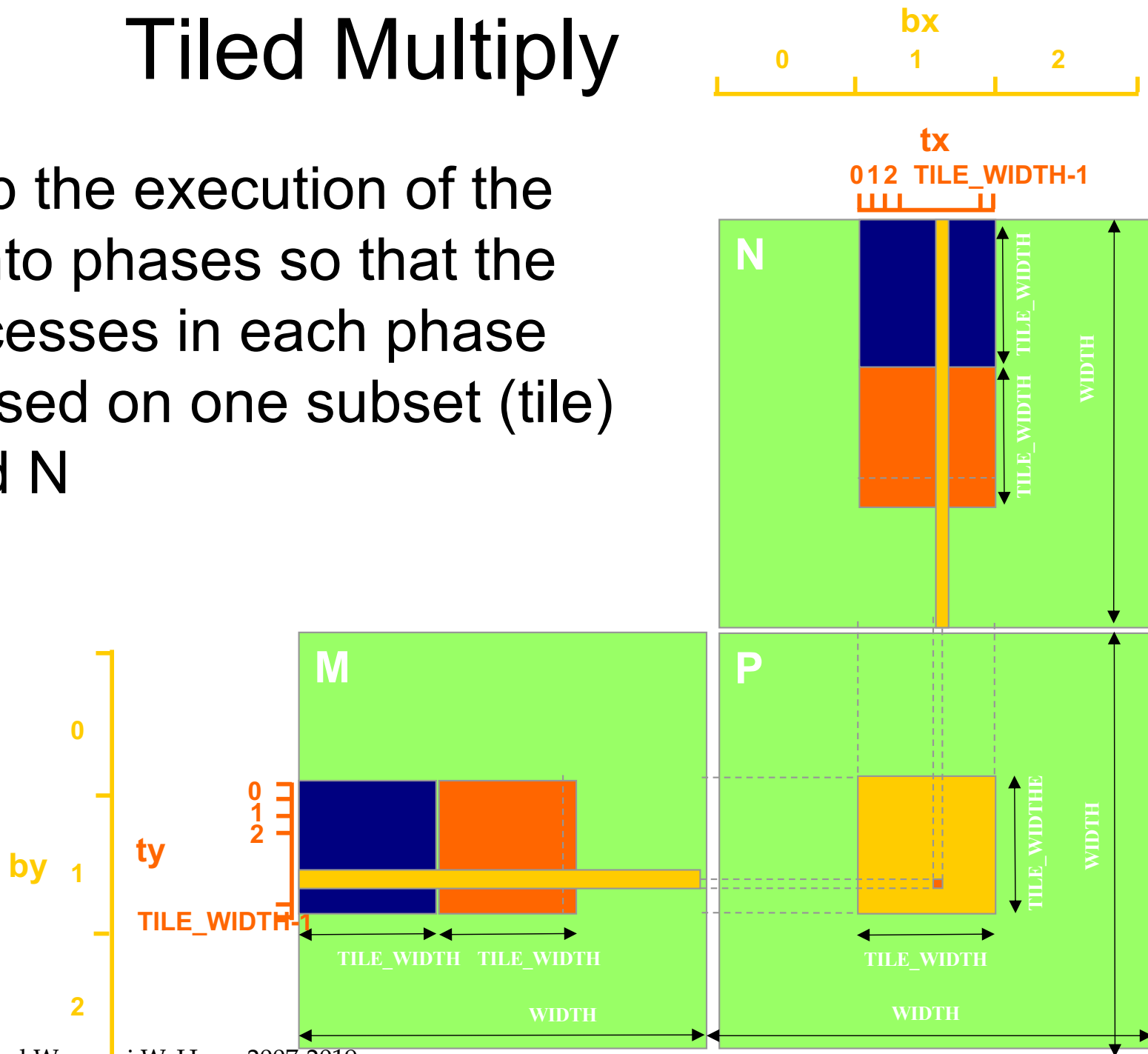
# Idea: Place global memory data into Shared Memory for **reuse**

- Each input element is used in calculating WIDTH elements in P.
- Load each element into Shared Memory and have several threads use the local version reduce the memory bandwidth



# Tiled Multiply

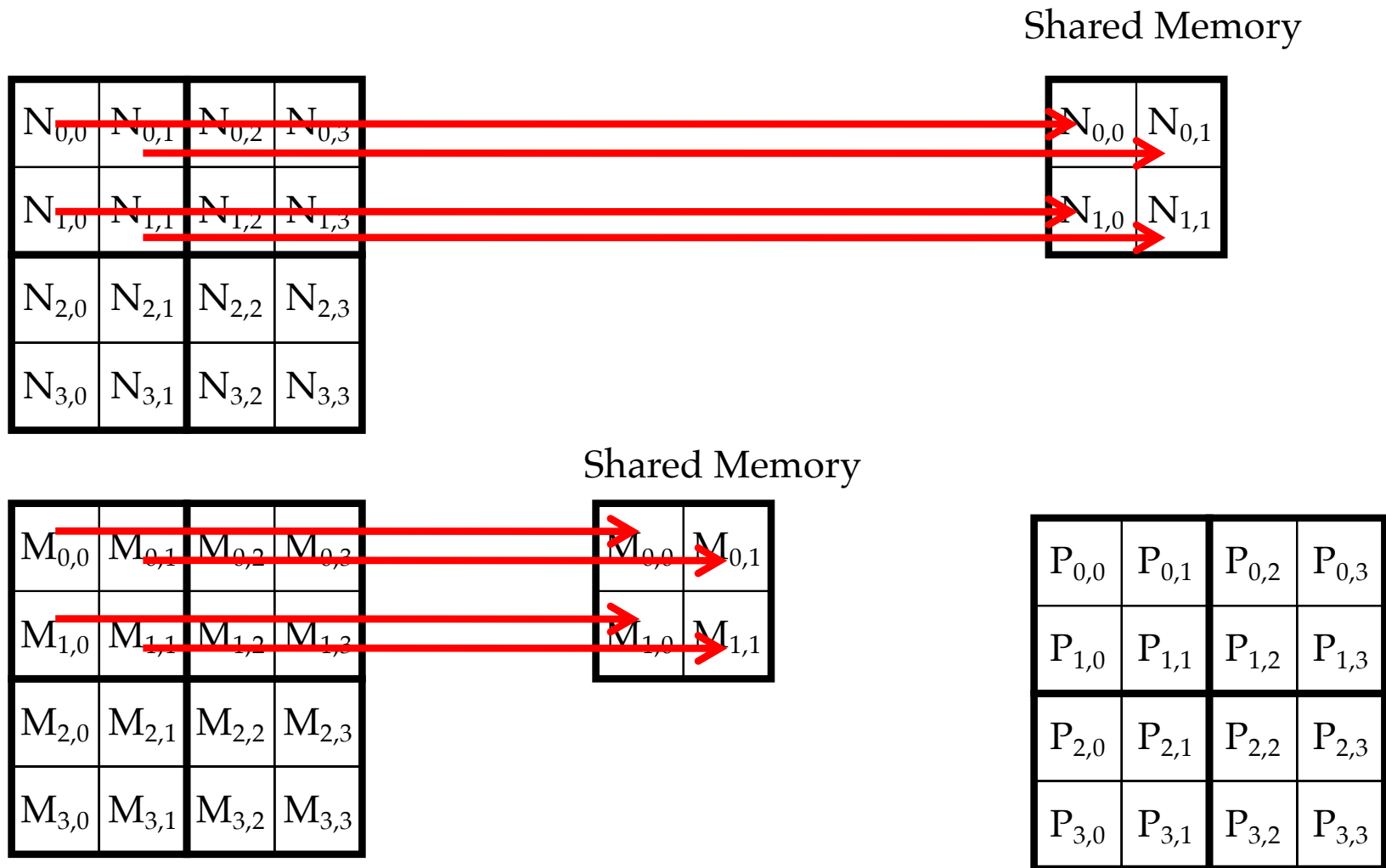
- Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one subset (tile) of M and N



# Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in basic tiling code
- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).

# Work for Block (0,0)

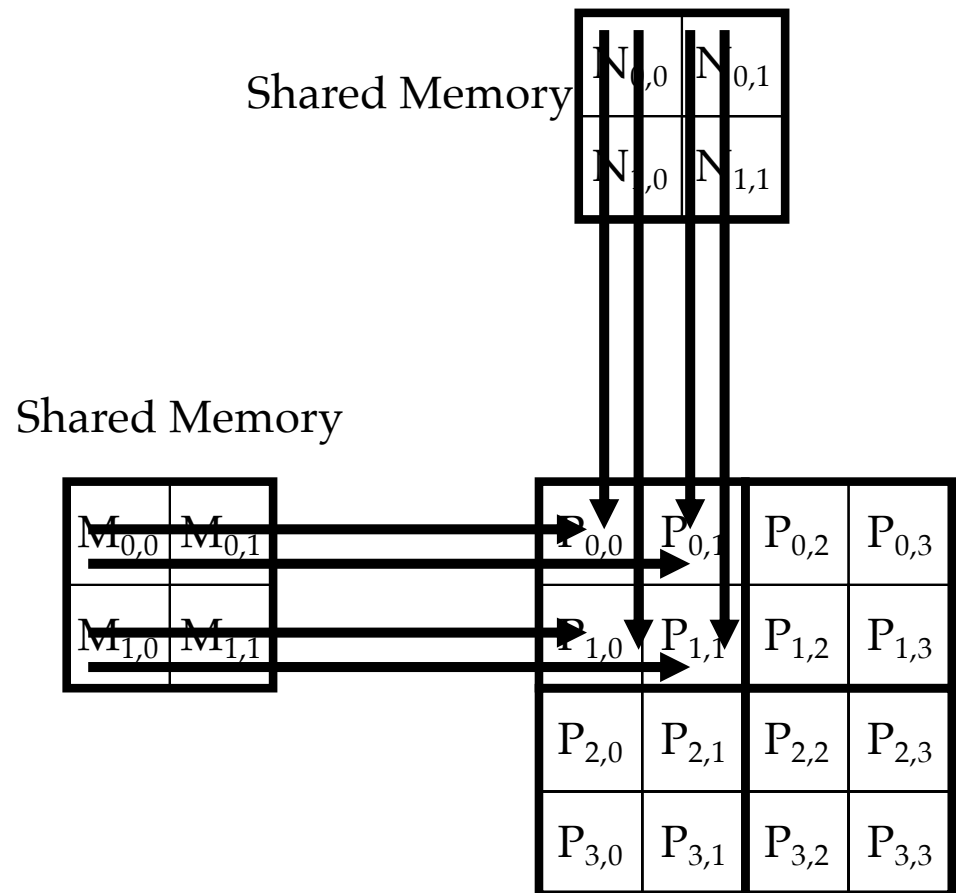


# Work for Block (0,0)

Threads use shared memory data in step 0.

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

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

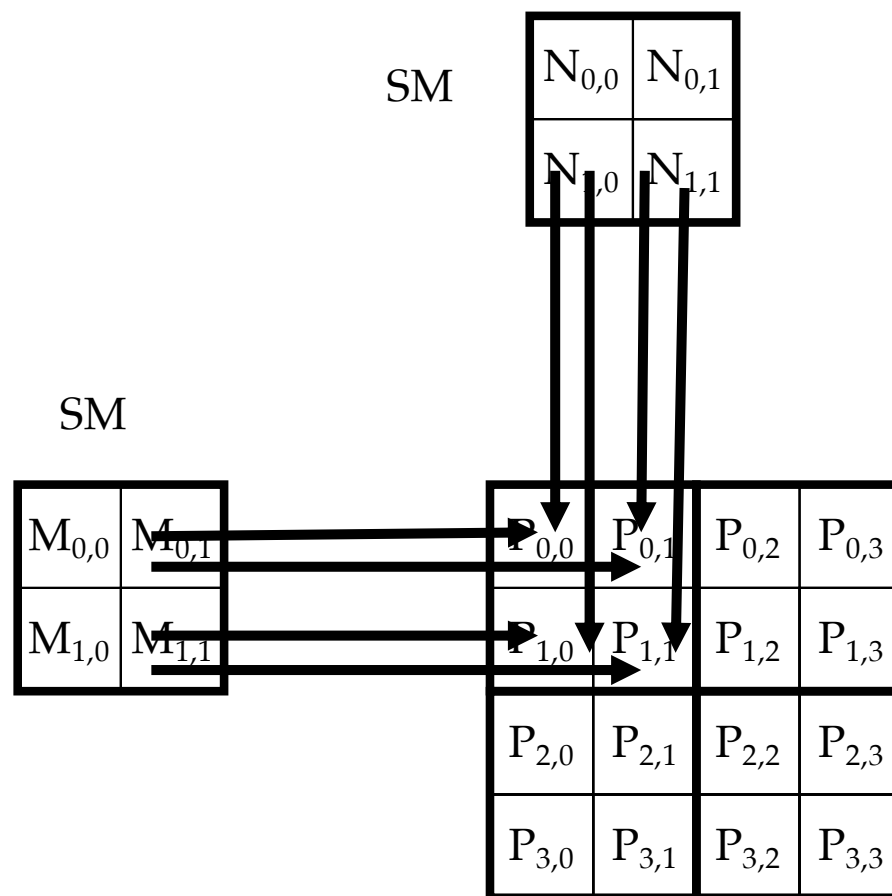


# Work for Block (0,0)

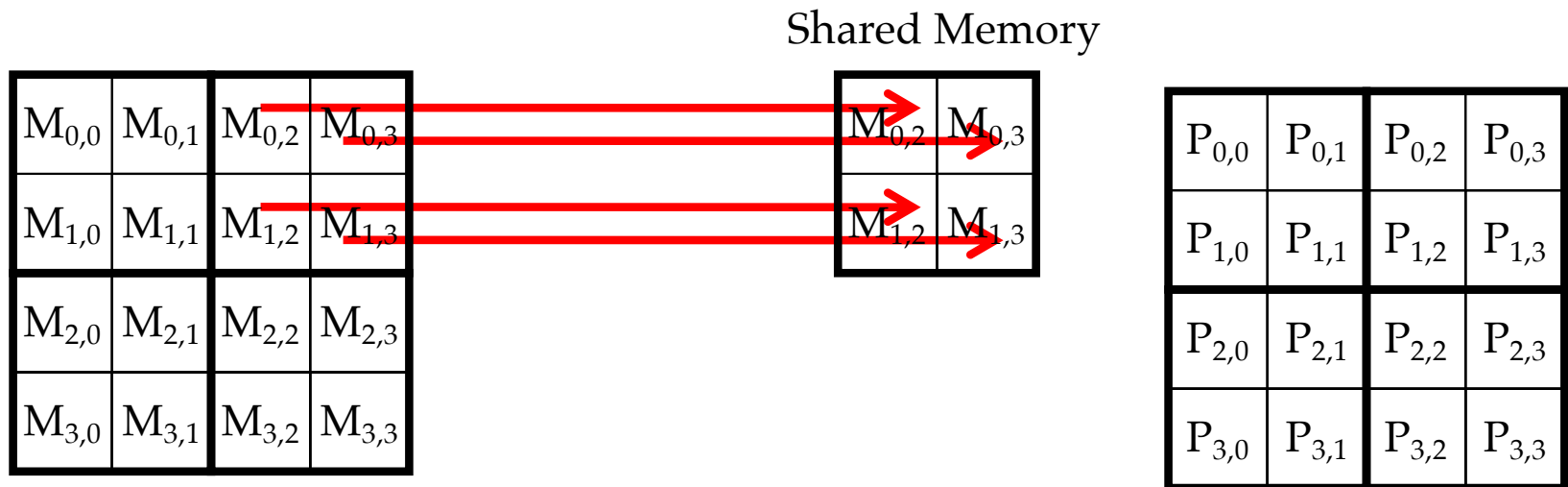
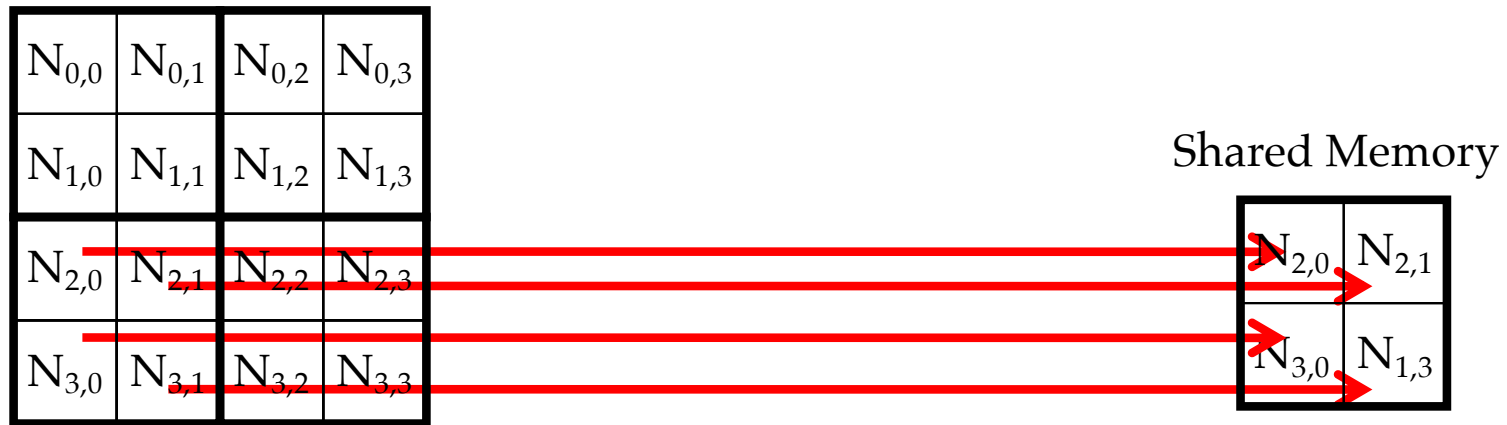
Threads use shared memory data in step 1.

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

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



# Work for Block (0,0)



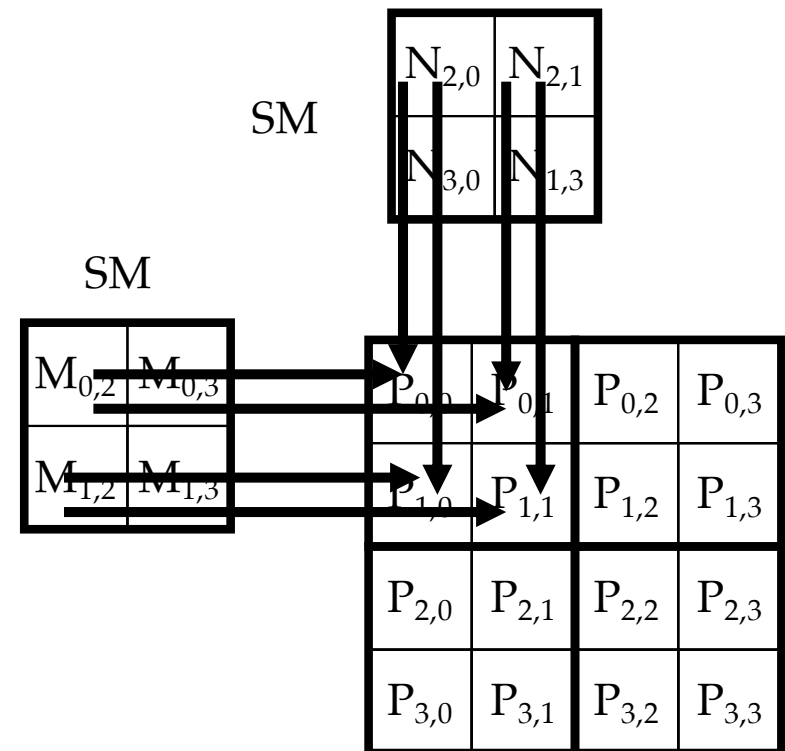


# Work for Block (0,0)

Threads use shared memory data in step 2.

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

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

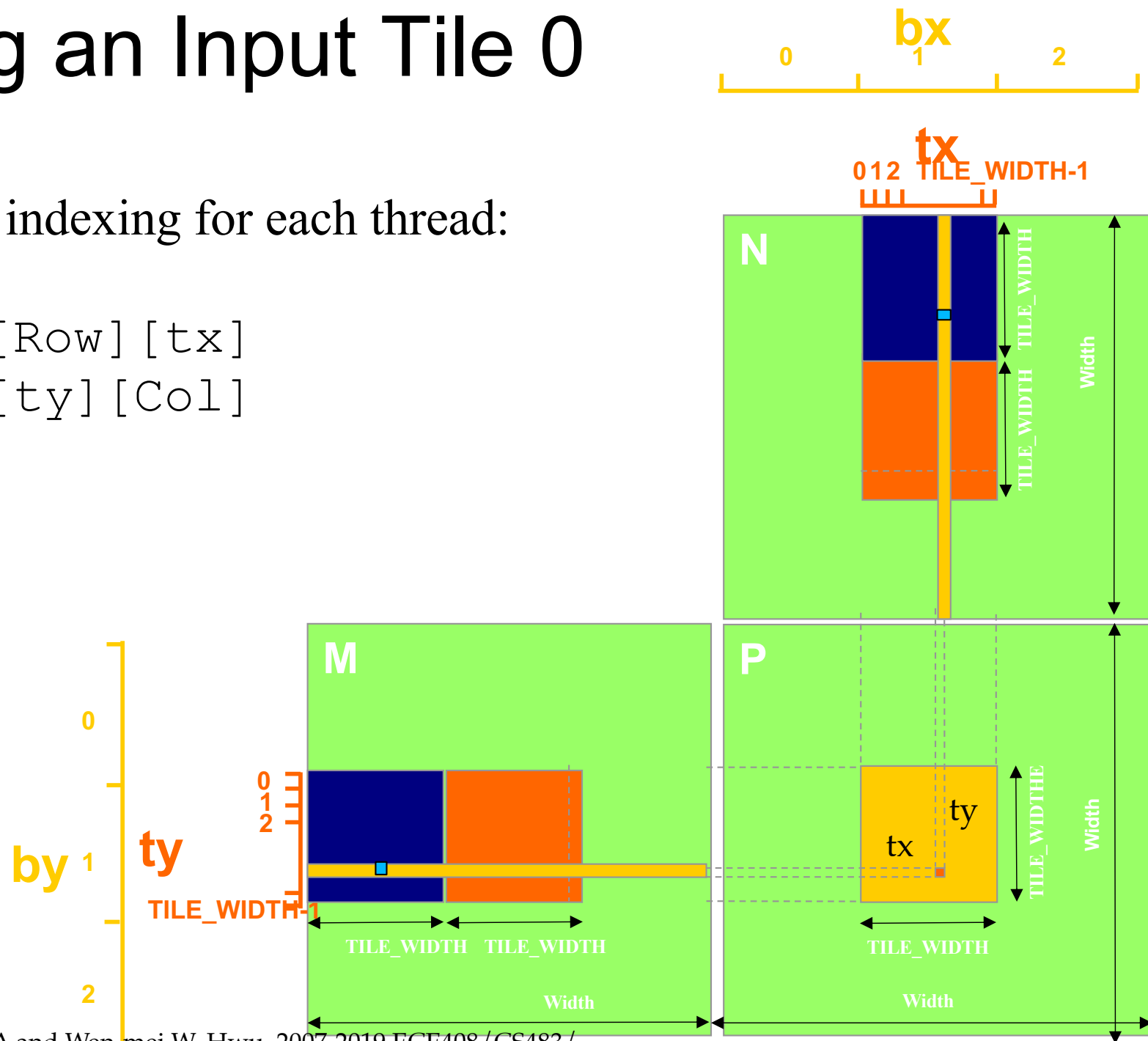


# Loading an Input Tile 0

Tile 0 2D indexing for each thread:

$M[\text{Row}][\text{tx}]$

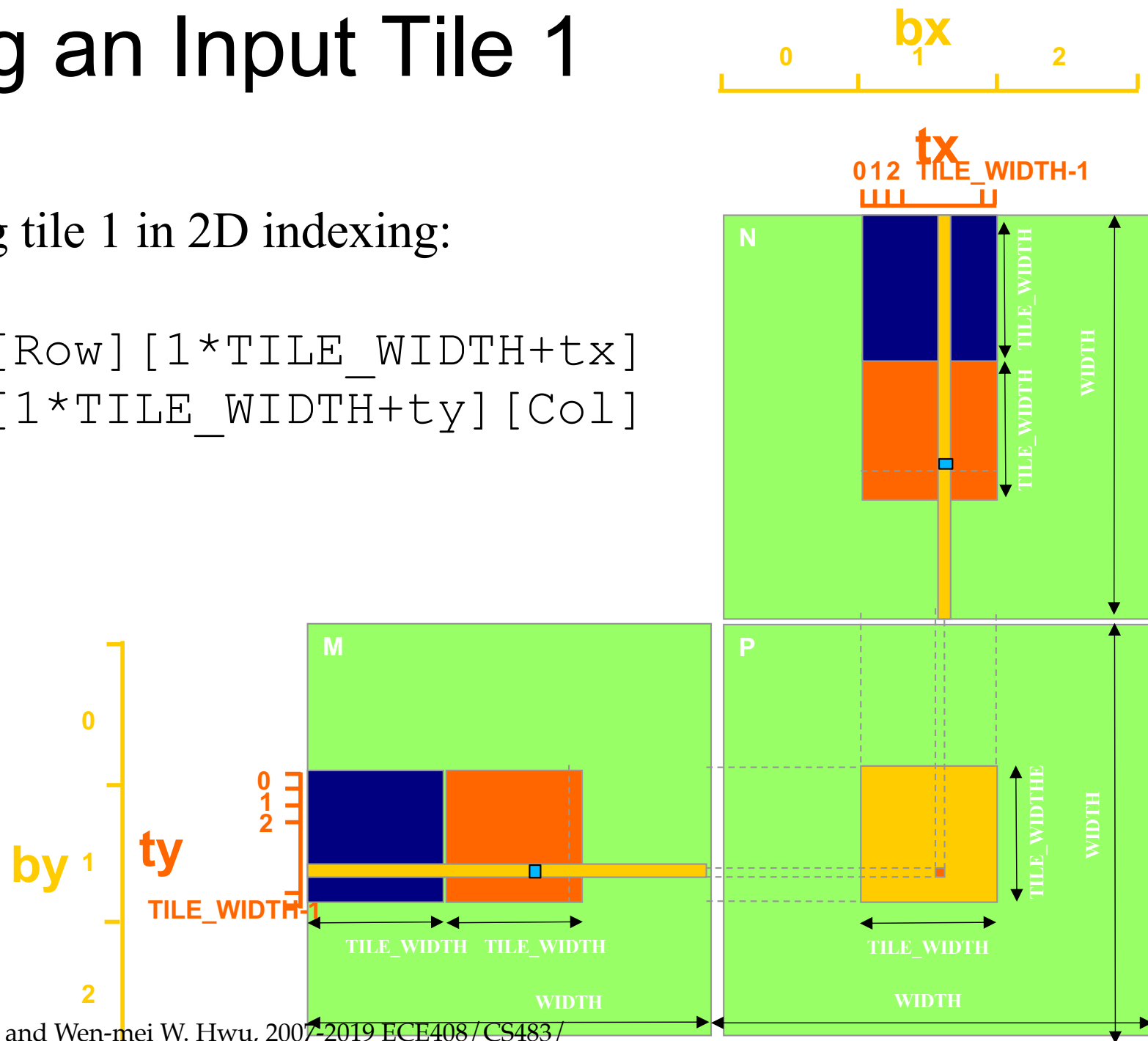
$N[\text{ty}][\text{Col}]$



# Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

```
M[Row][1*TILE_WIDTH+tx]
N[1*TILE_WIDTH+ty][Col]
```



# Loading an Input Tile m

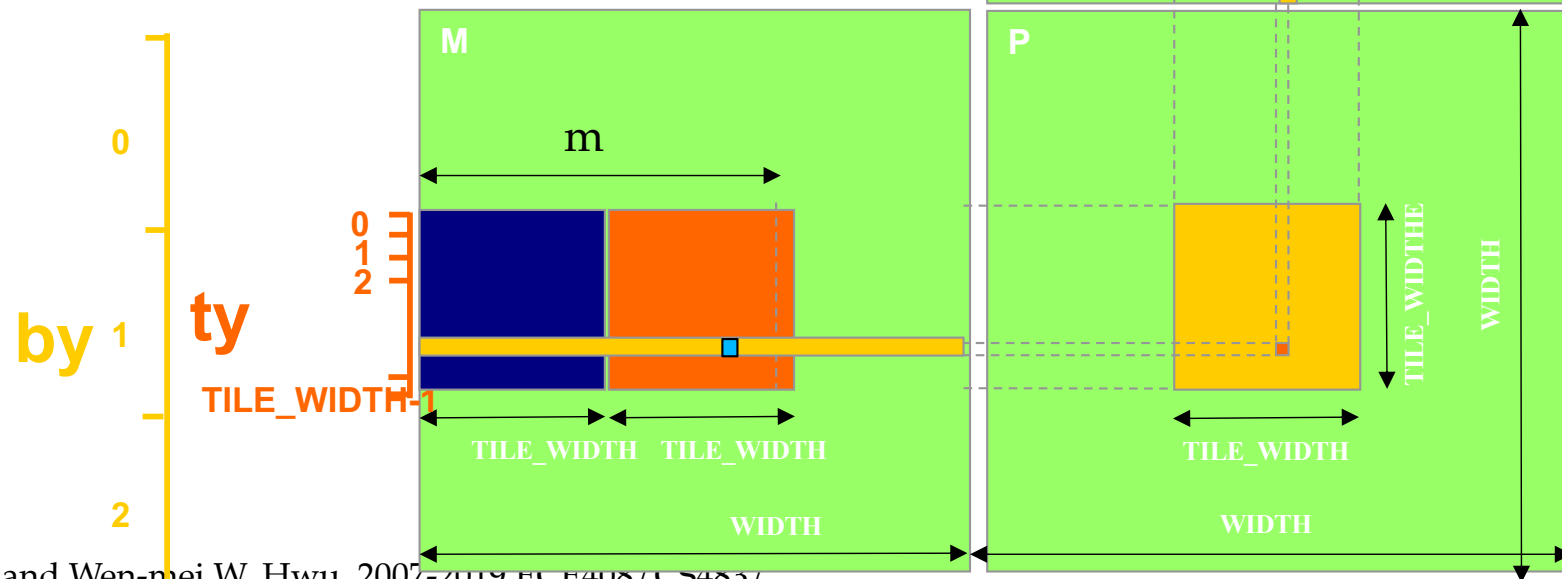
However, recall that M and N are dynamically allocated and can only use 1D indexing:

```
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]
```



# Accessing a Tile

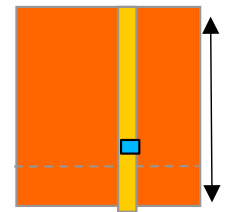
To perform the  $k^{\text{th}}$  step of the product within the tile:

`subTileM[ty][k]`

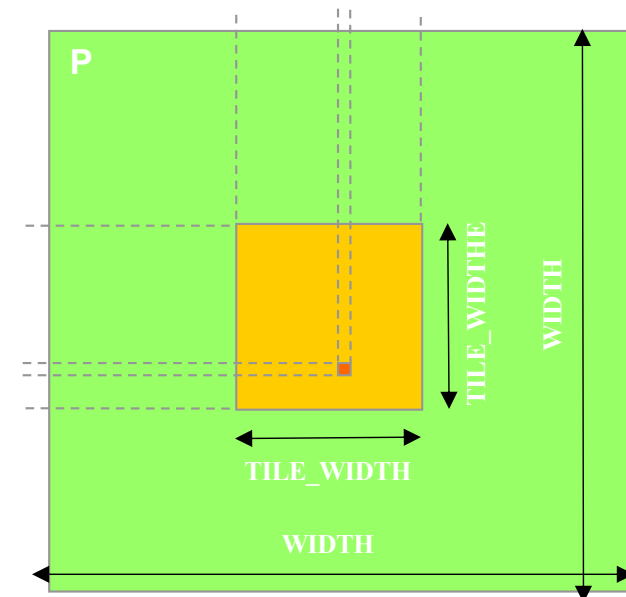
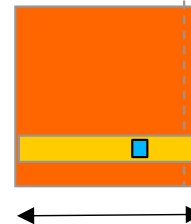
`subTileN[k][tx]`



`subTileN`

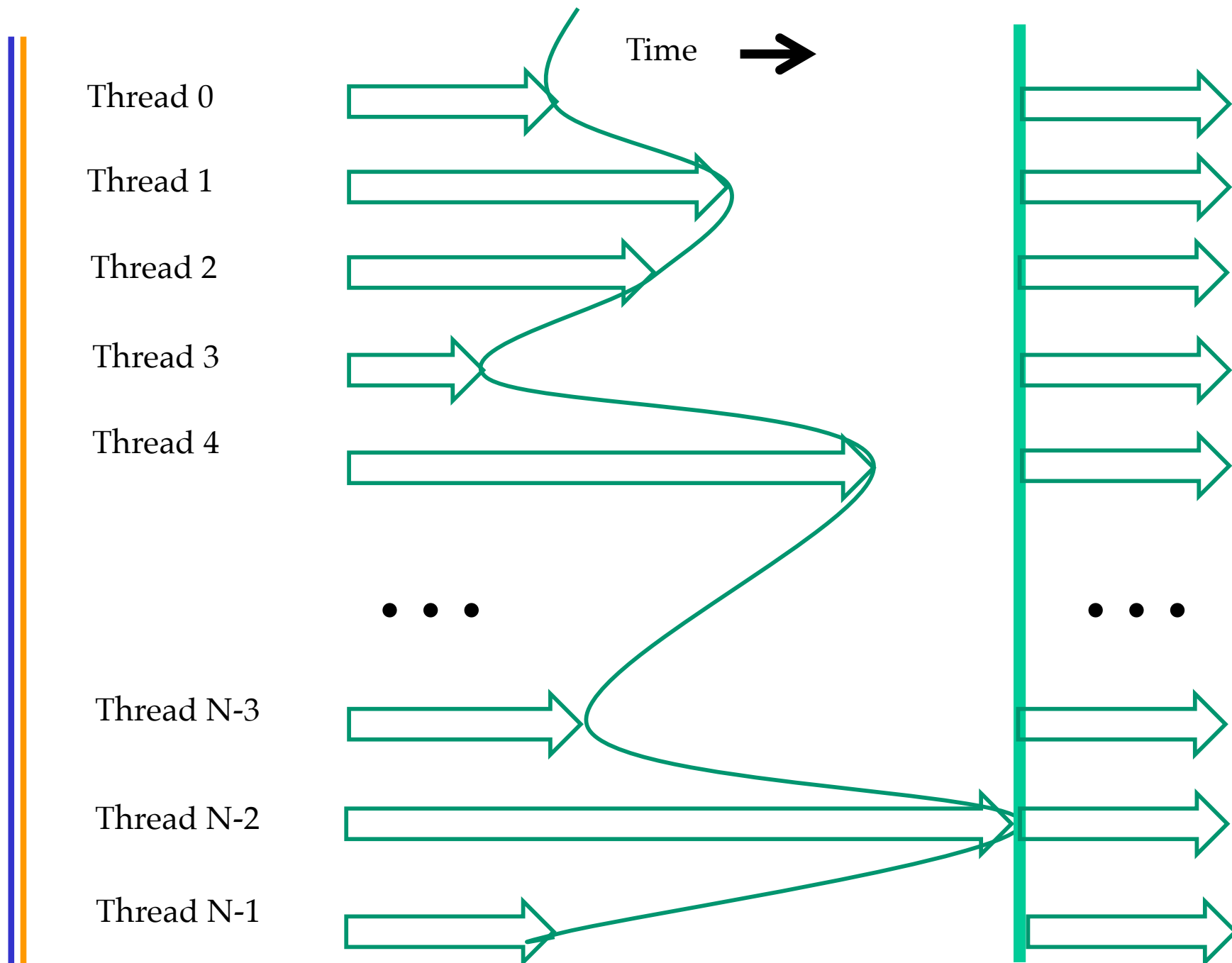


`subTileM`



# 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



# 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
    // The code assumes that the Width is a multiple of TILE_WIDTH!
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;
}
```



# Compare with Basic MM Kernel

```
__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;
    }
}
```

# Shared Memory and Threading

- Each SM in Maxwell GM107 has 64KB shared memory (48KB max per block)
  - Shared memory size is implementation dependent!
  - For `TILE_WIDTH = 16`, each thread block uses  $2 \times 256 \times 4B = 2KB$  of shared memory.
    - Shared memory can potentially support up to 32 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 8
    - This allows up to  $8 \times 512 = 4,096$  pending loads. (2 per thread, 256 threads per block)
  - `TILE_WIDTH 32` would lead to  $2 \times 32 \times 32 \times 4B = 8KB$  shared memory usage per thread block,
    - Shared memory can potentially support up to 8 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 2
    - This allows up to  $2 \times 2,048 = 4,096$  pending loads (2 per thread, 1024 threads per block)

## **CS 133 Worksheet: #13**

**Name 1:**

**Name 2:**



**What is the maximum tile size that you may use for the tiled matrix multiplication problem on the Maxwell GM204 GPU (used in Lab3)? How much global memory access that such an implementation can reduce compared to the basic MM kernel (no tiling)? Please explain why.**

## ***Announcement – CS Town Hall this Wed.***

---

The CS Town Hall is a way for the department to hear and respond to student concerns and feedback. Students will have an opportunity to ask their professors questions at the event and share their opinion about CS classes at UCLA by filling out a survey beforehand. The event will be held on Wednesday, May 11th from 6 pm to 8 pm in the Mong Learning Center and will also be catered by Bella Pita. The links for the RSVP and survey can be found at [uclaacm.com/town-hall](https://uclaacm.com/town-hall) and are given below as well.

Link to RSVP: <https://tinyurl.com/22town-hall-rsvp>

Link to Survey: <https://tinyurl.com/cs-town-hall-s22>

# Memory Bandwidth Consumption

- Using 16x16 tiling, we reduce the global memory by a factor of 16
  - Each operand is now used by 16 floating-point operations
  - The 150GB/s bandwidth can now support  $(150/4)*16 = 600$  GFLOPS!
- Using 32x32 tiling, we reduce the global memory accesses by a factor of 32
  - Each operand is now used by 32 floating-point operations
  - The 150 GB/s bandwidth can now support  $(150/4)*32 = 1,200$  GFLOPS!
  - The memory bandwidth is no longer a limiting factor for performance!

# Device Query

- Number of devices in the system

```
int dev_count;  
cudaGetDeviceCount( &dev_count);
```

- Capability of devices

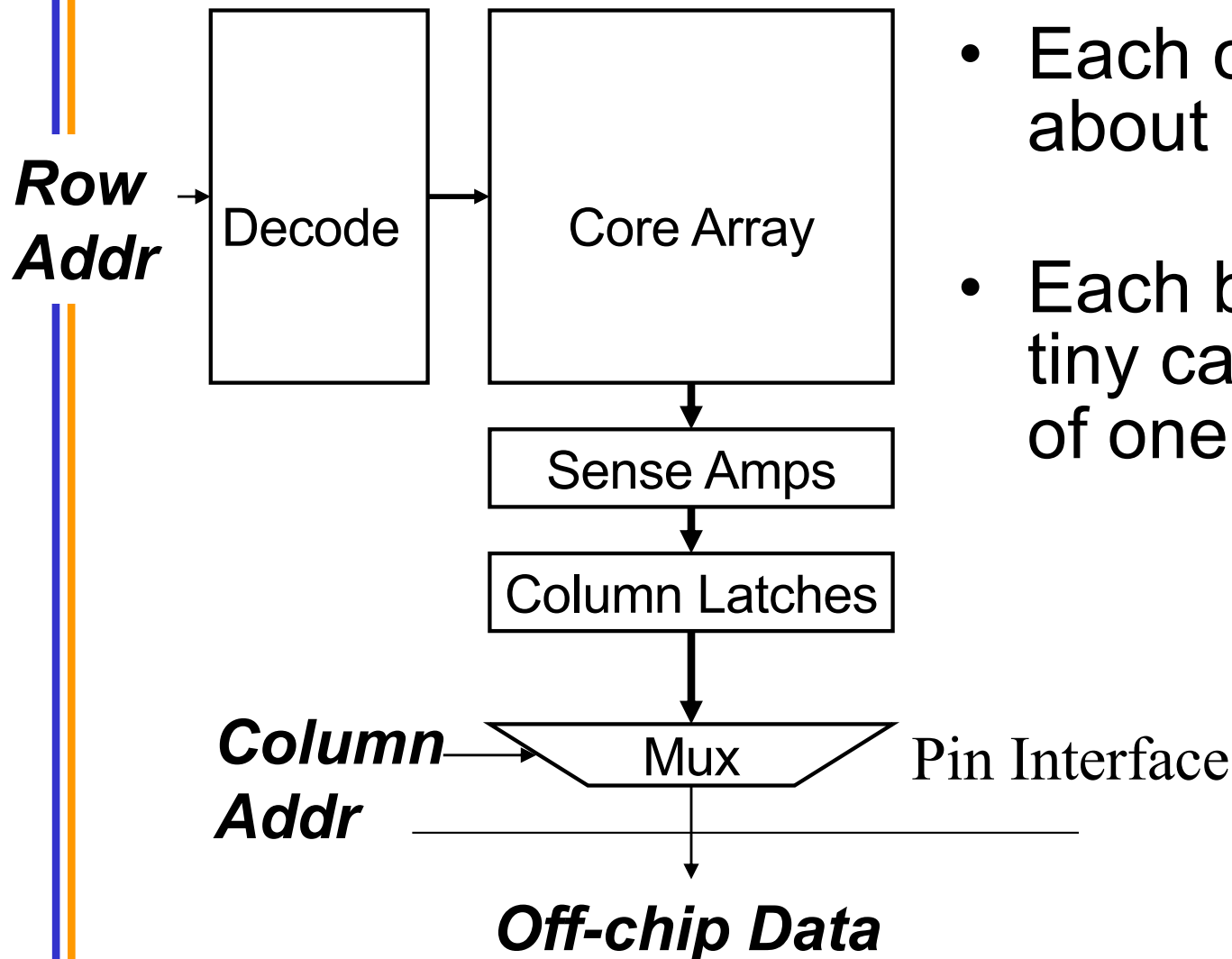
```
cudaDeviceProp      dev_prop;  
for (i = 0; i < dev_count; i++) {  
    cudaGetDeviceProperties( &dev_prop, i);  
  
    // decide if device has sufficient resources and capabilities  
}
```

- `cudaDeviceProp` is a built-in C structure type
  - `dev_prop.dev_prop.maxThreadsPerBlock`
  - `Dev_prop.sharedMemoryPerBlock`
  - ...

# Memory Coalescing

- After row-major linearization, adjacent threads in each warp access adjacent memory locations
  - Accesses for threads in the warp consolidated into one (or two) DRAM access (burst)
- Better efficiency using DRAM bursts
  - Current DRAM burst size is 64-128 bytes
  - Any distance (gap) between locations accessed by adjacent threads in a warp reduces DRAM efficiency
- When adjacent threads in a warp access identical locations, accesses for threads in the warp consolidated into one DRAM access

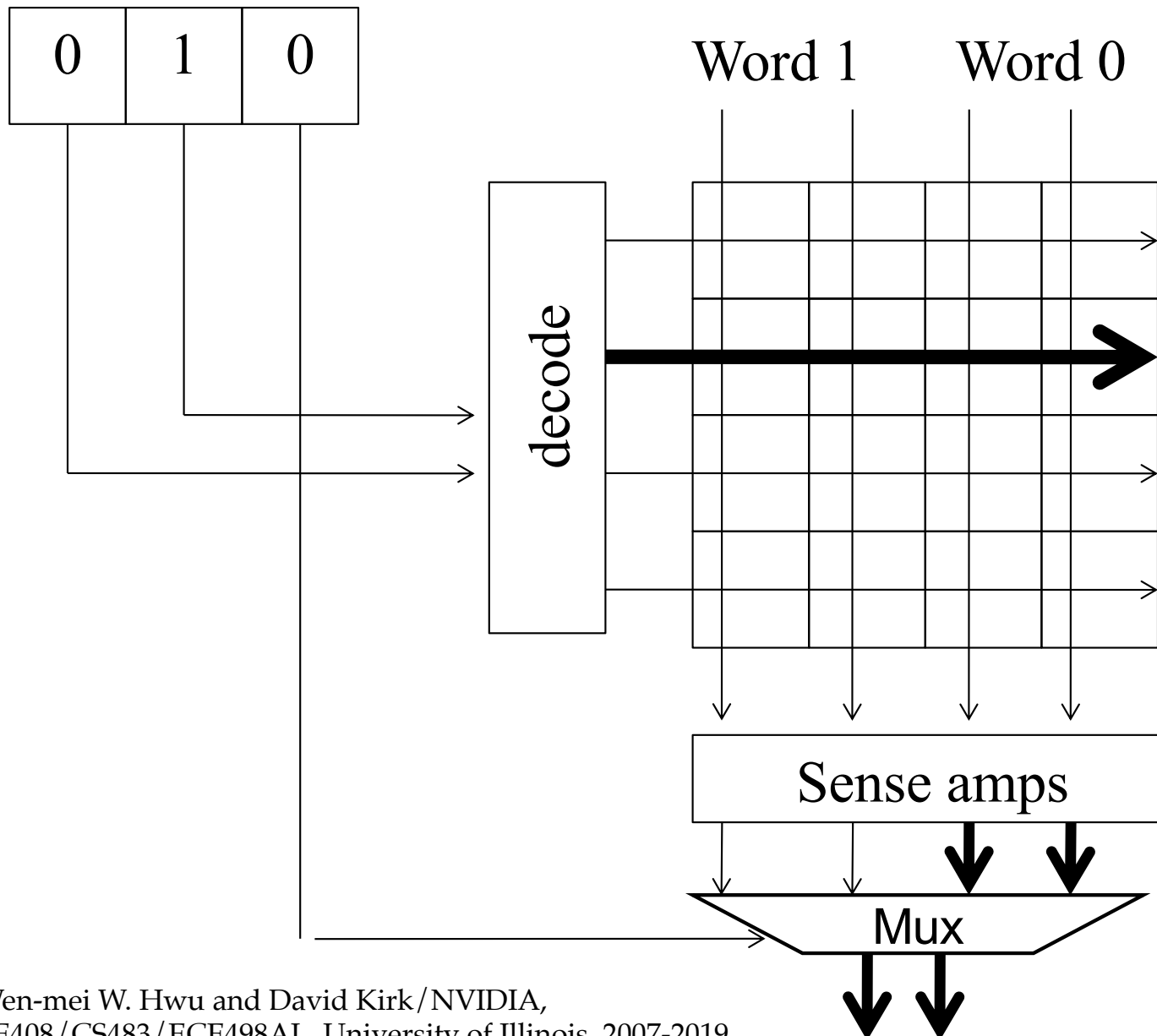
# DRAM Bank Organization



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

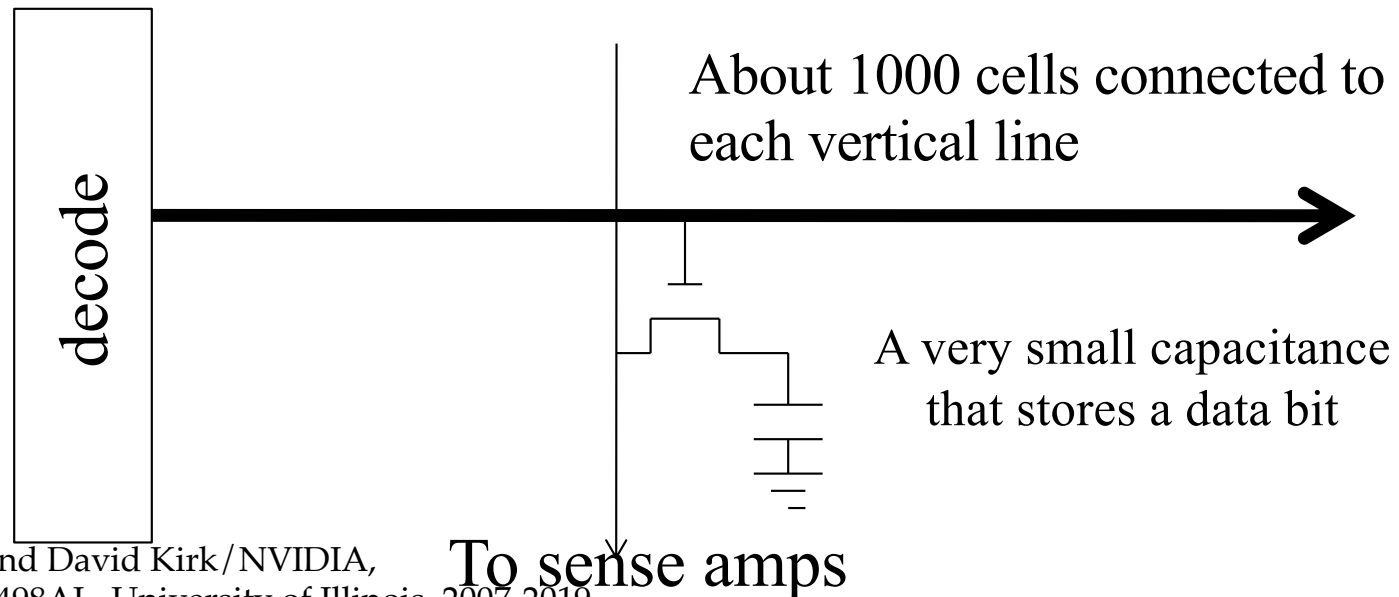


# A very small (8x2 bit) DRAM Bank

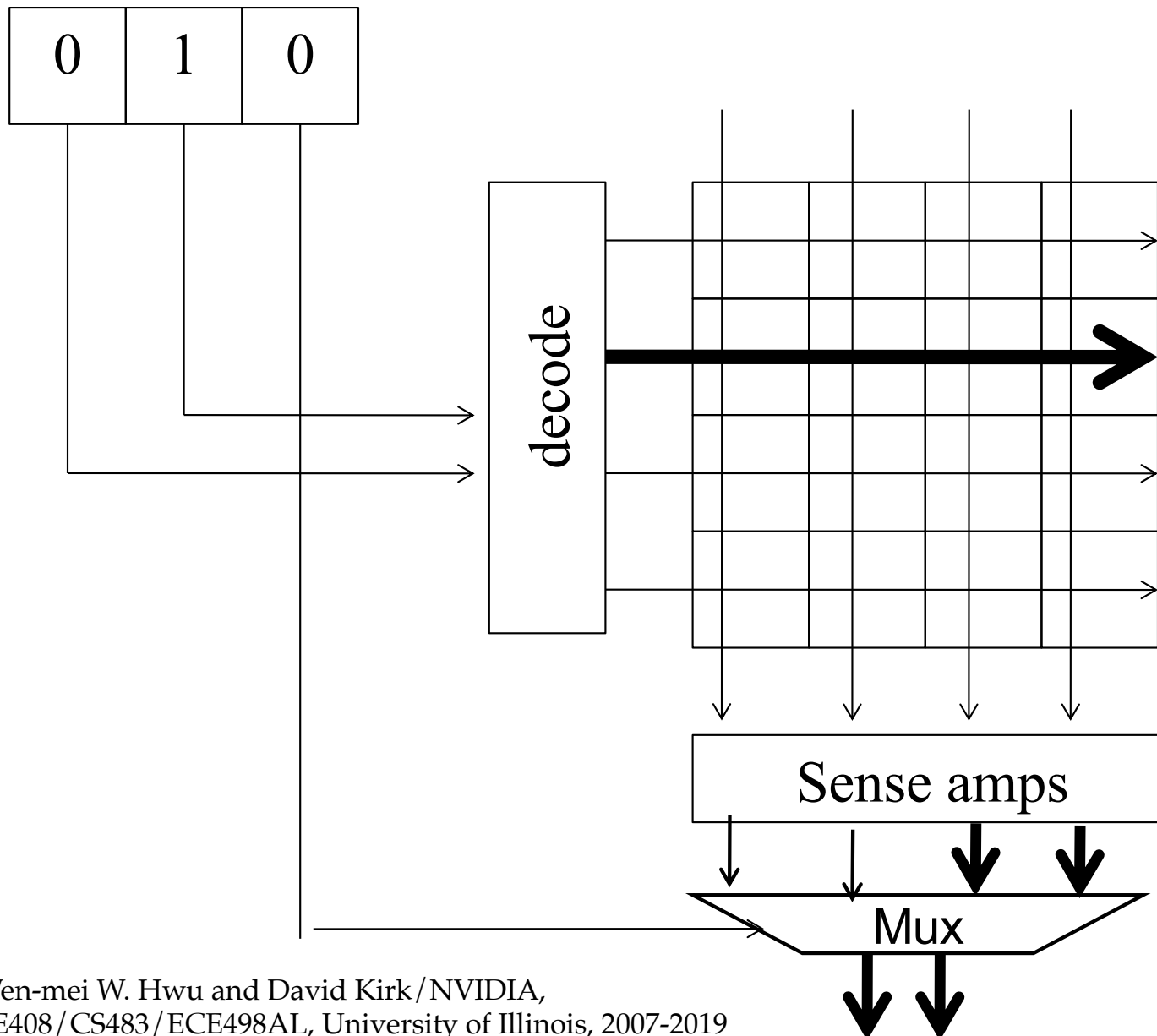


# DRAM core arrays are slow.

- Reading from a cell in the core array is a very slow process
  - DDR: Core speed =  $\frac{1}{2}$  interface speed
  - DDR2/GDDR3: Core speed =  $\frac{1}{4}$  interface speed
  - DDR3/GDDR4: Core speed =  $\frac{1}{8}$  interface speed
  - ... likely to be worse in the future

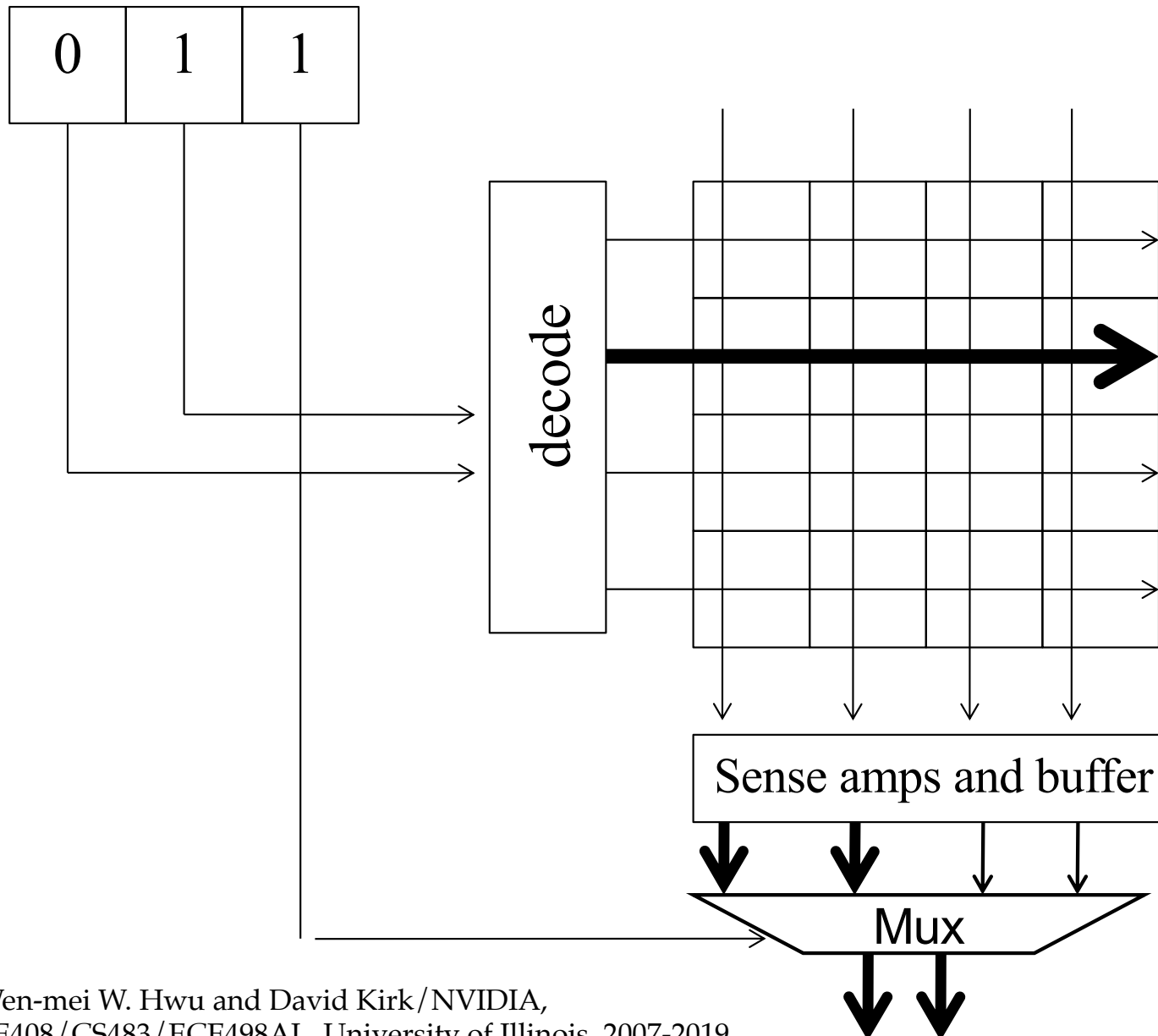


# DRAM Bursting (burst size = 4 bits)

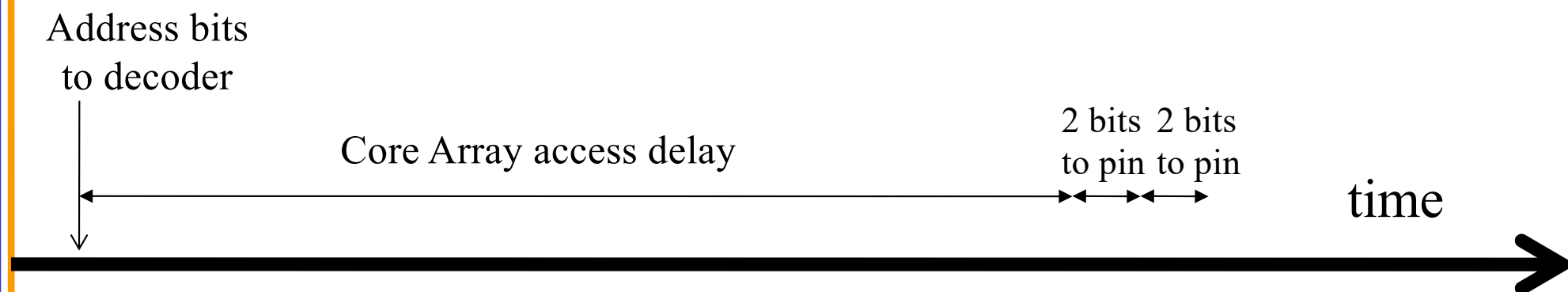


# DRAM Bursting (cont.)

## second part of the burst



# DRAM Bursting for the 8x2 Bank



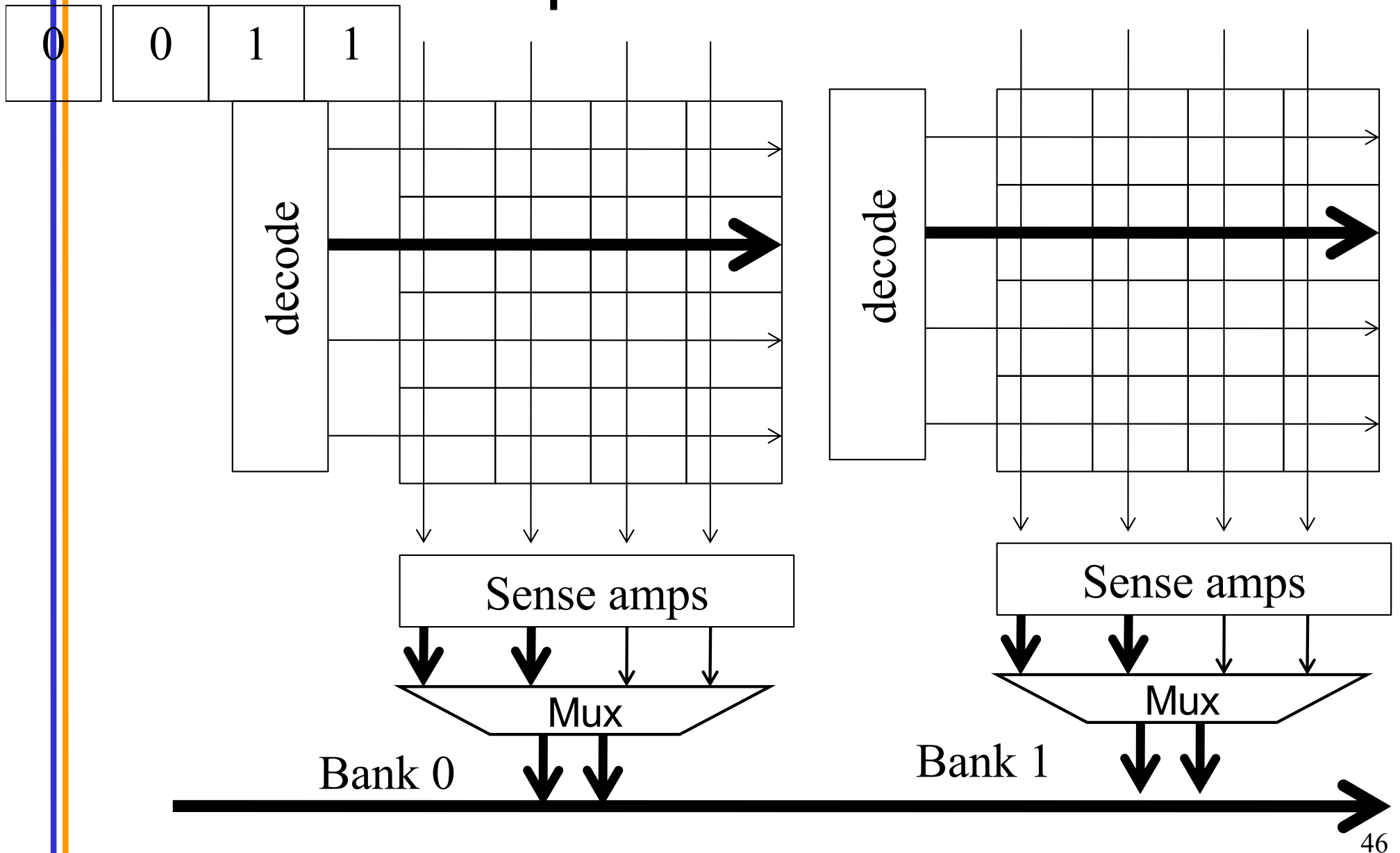
Non-burst timing



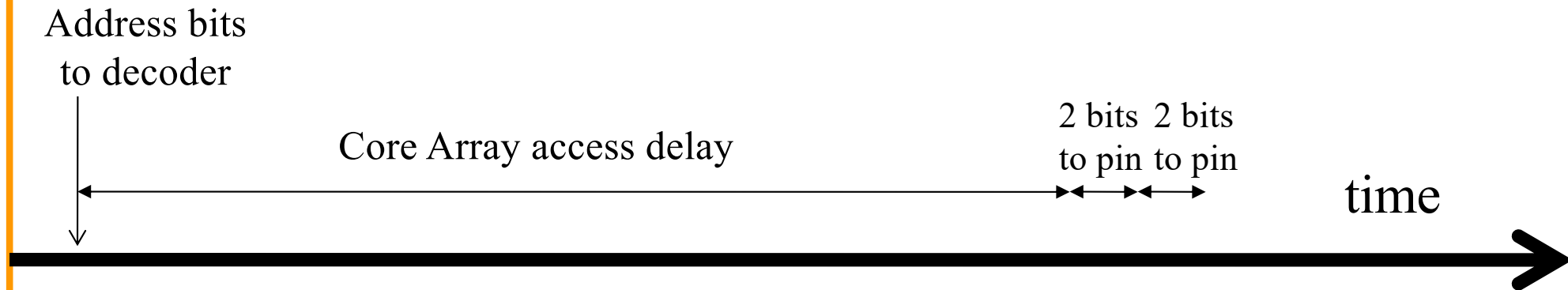
Burst timing

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 and Banking



Single-Bank burst timing, dead time on interface



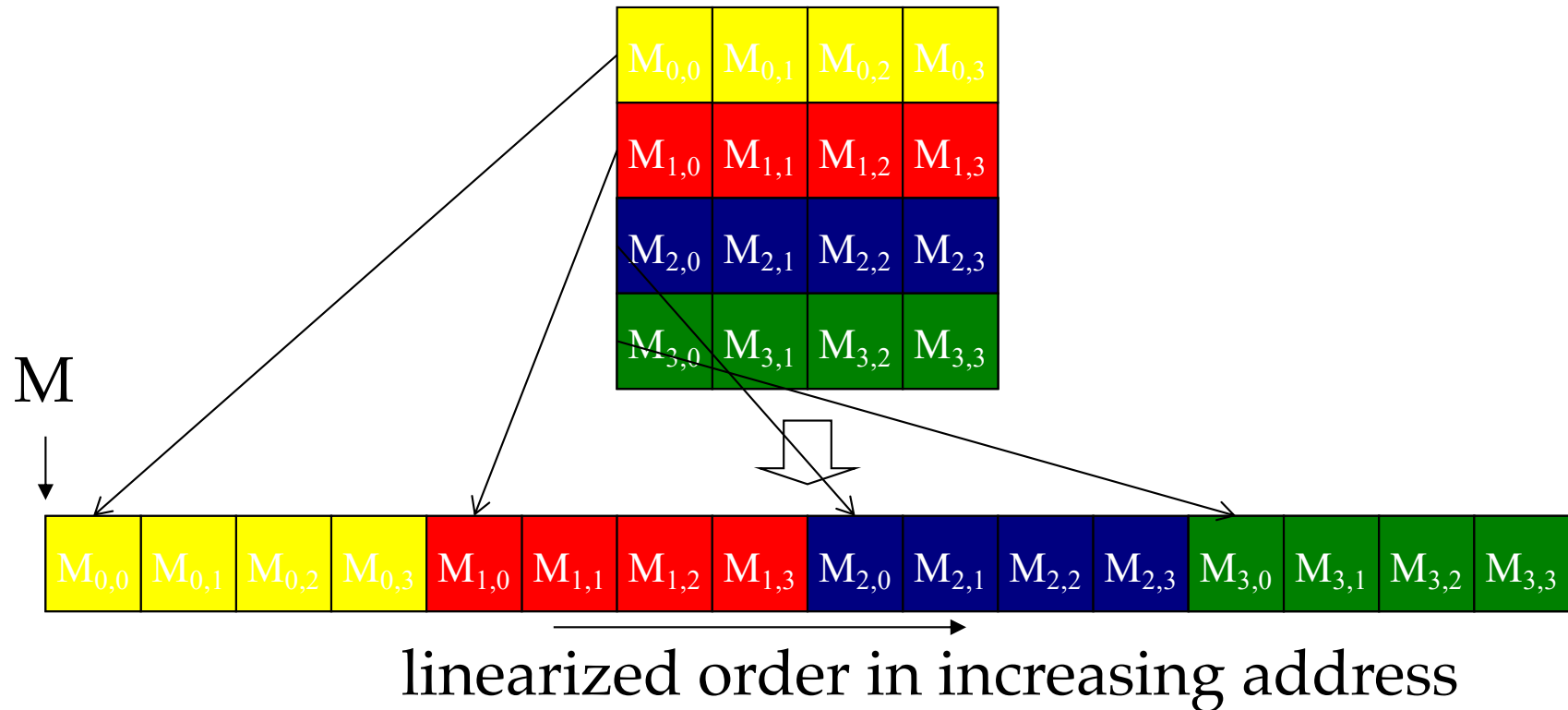
Multi-Bank burst timing, reduced dead time

# Memory Accesses by a Warp of Threads

- Recall that thread blocks are broken into warps
- All threads in a warp execute in SIMD style
  - So, each memory access in the kernel will be executed as 32 memory accesses by a warp
- If all threads in a warp access consecutive locations within a burst, their accesses can be served just one burst
  - Only one DRAM transaction/transfer is needed
  - This is called **memory coalescing**
  - Accesses in a warp are effectively combined into just one access to the DRAM system



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



# A Simple Matrix Multiplication Kernel (review)

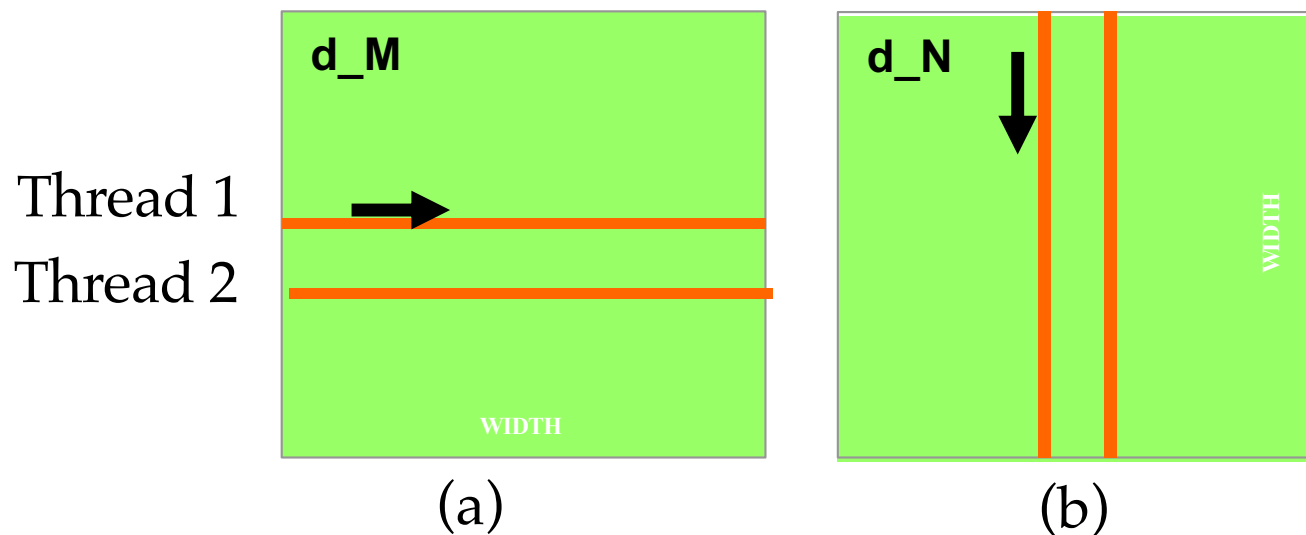
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    // Calculate the row index of the P element and M
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    // Calculate the column index of P and N
    int Col = blockIdx.x * blockDim.x + threadIdx.x;

    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

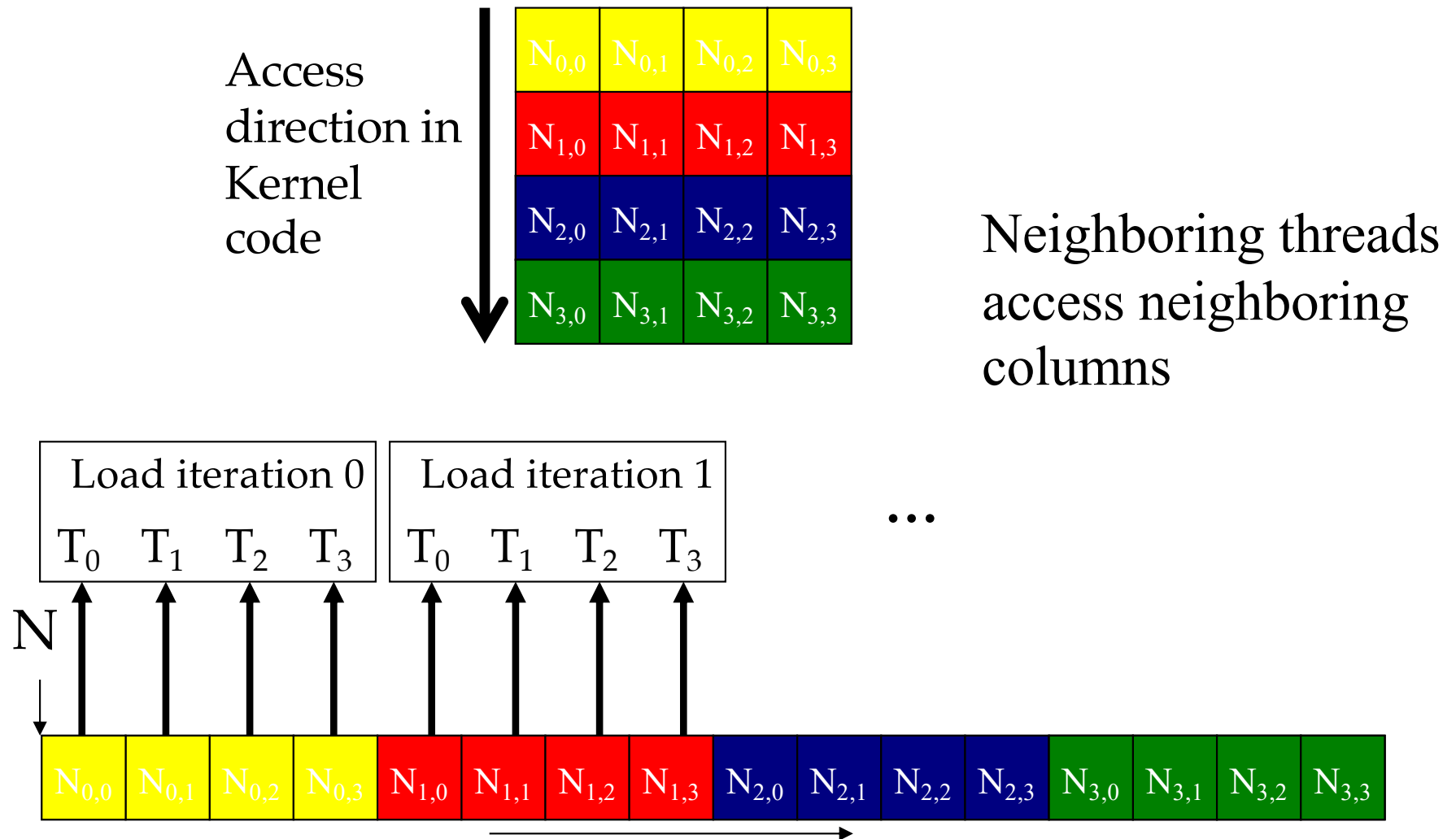


$M[\text{Row} * \text{Width} + k]$

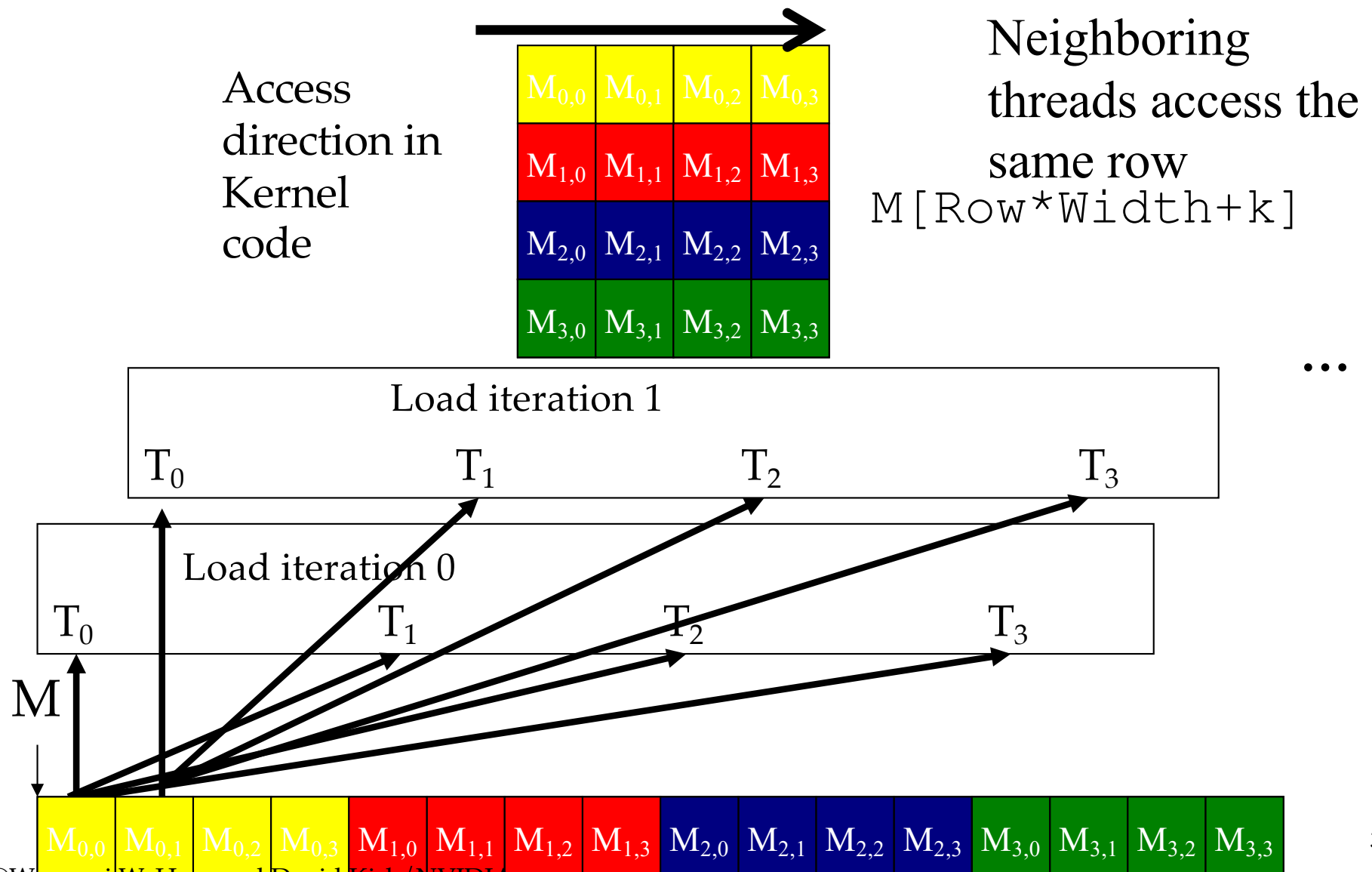
$N[k * \text{Width} + \text{Col}]$

$k$  is loop counter in the inner product loop of the kernel code<sub>51</sub>

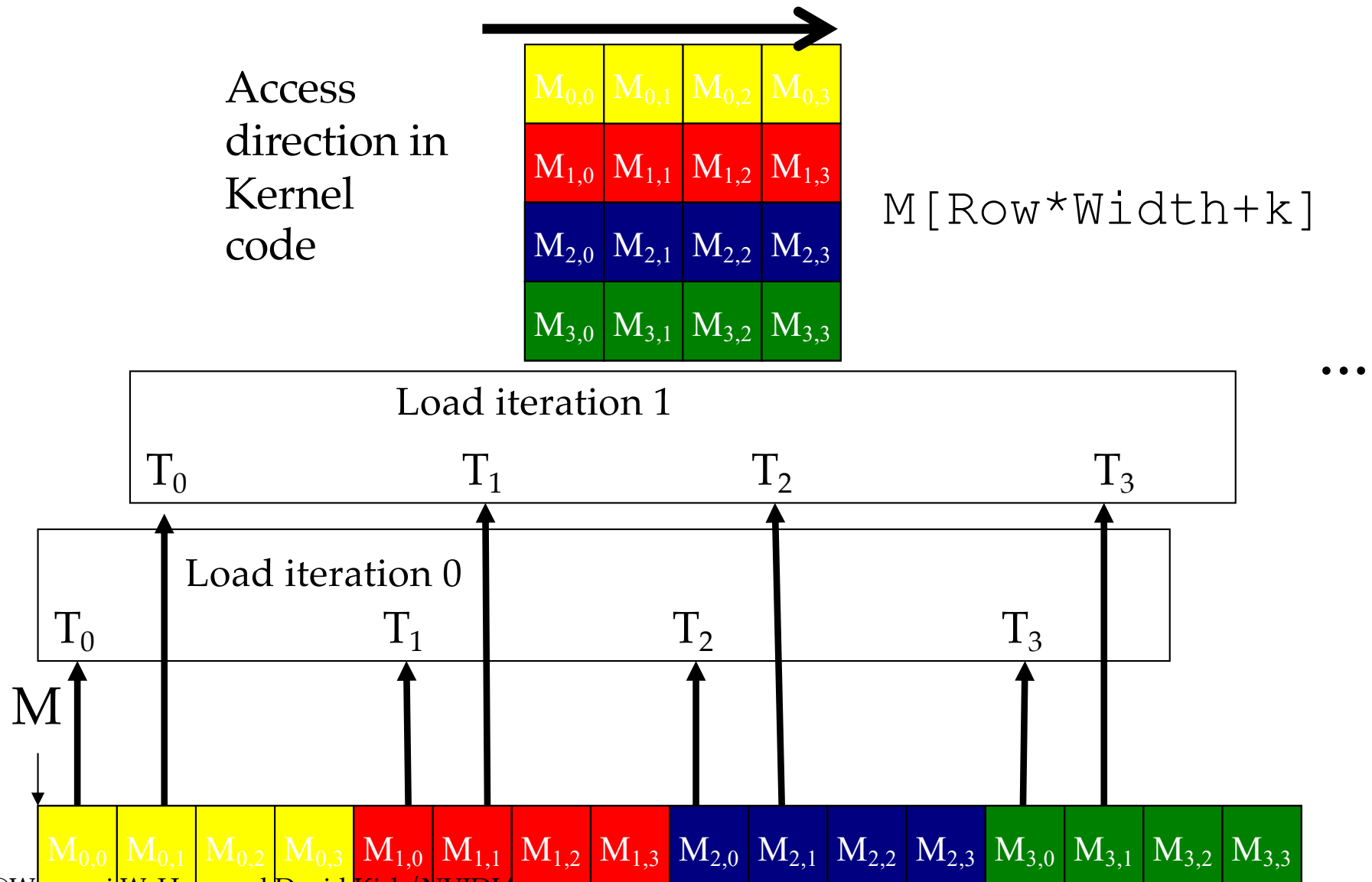
# N accesses are coalesced.



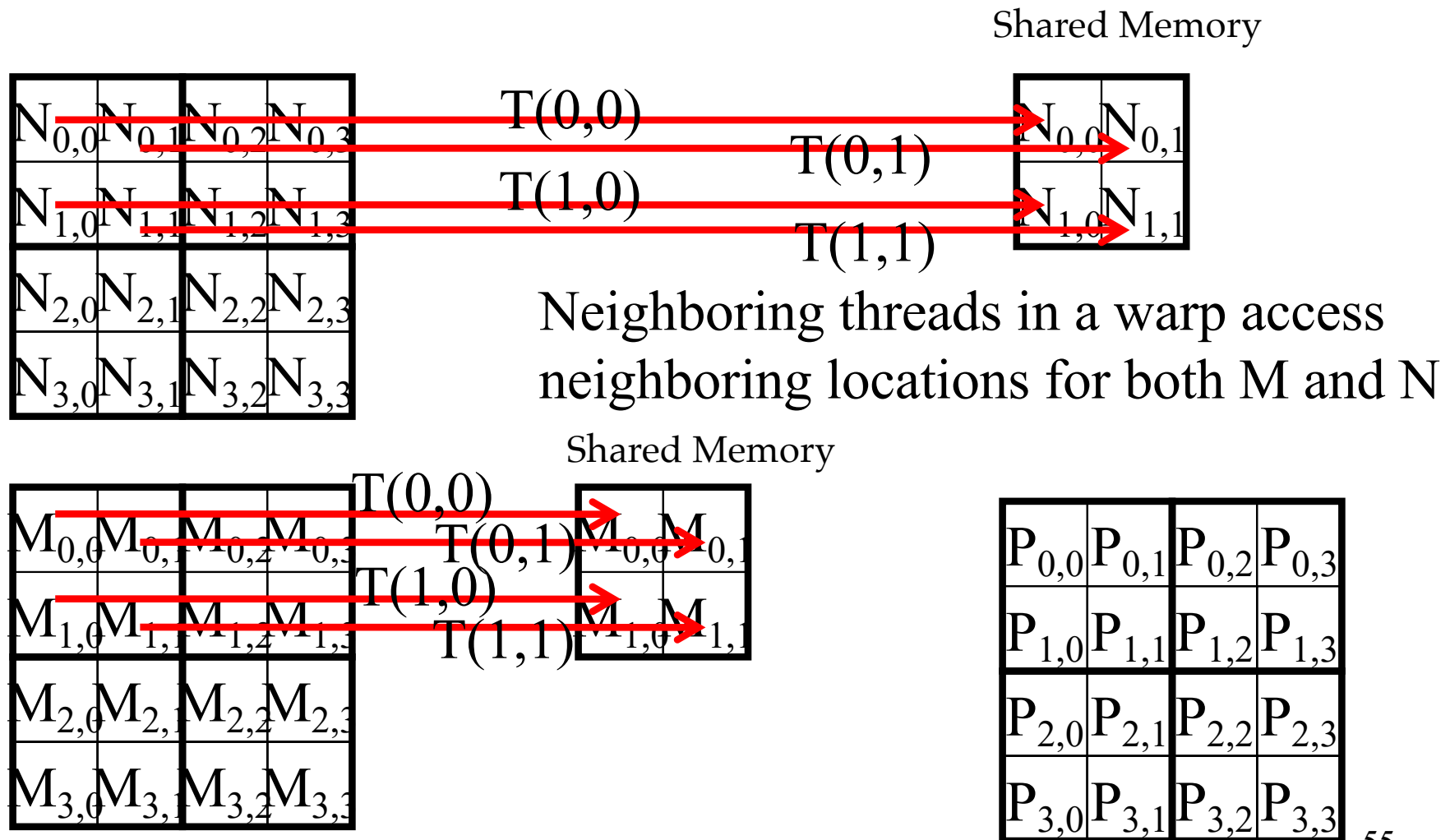
# M accesses are not exactly coalesced.



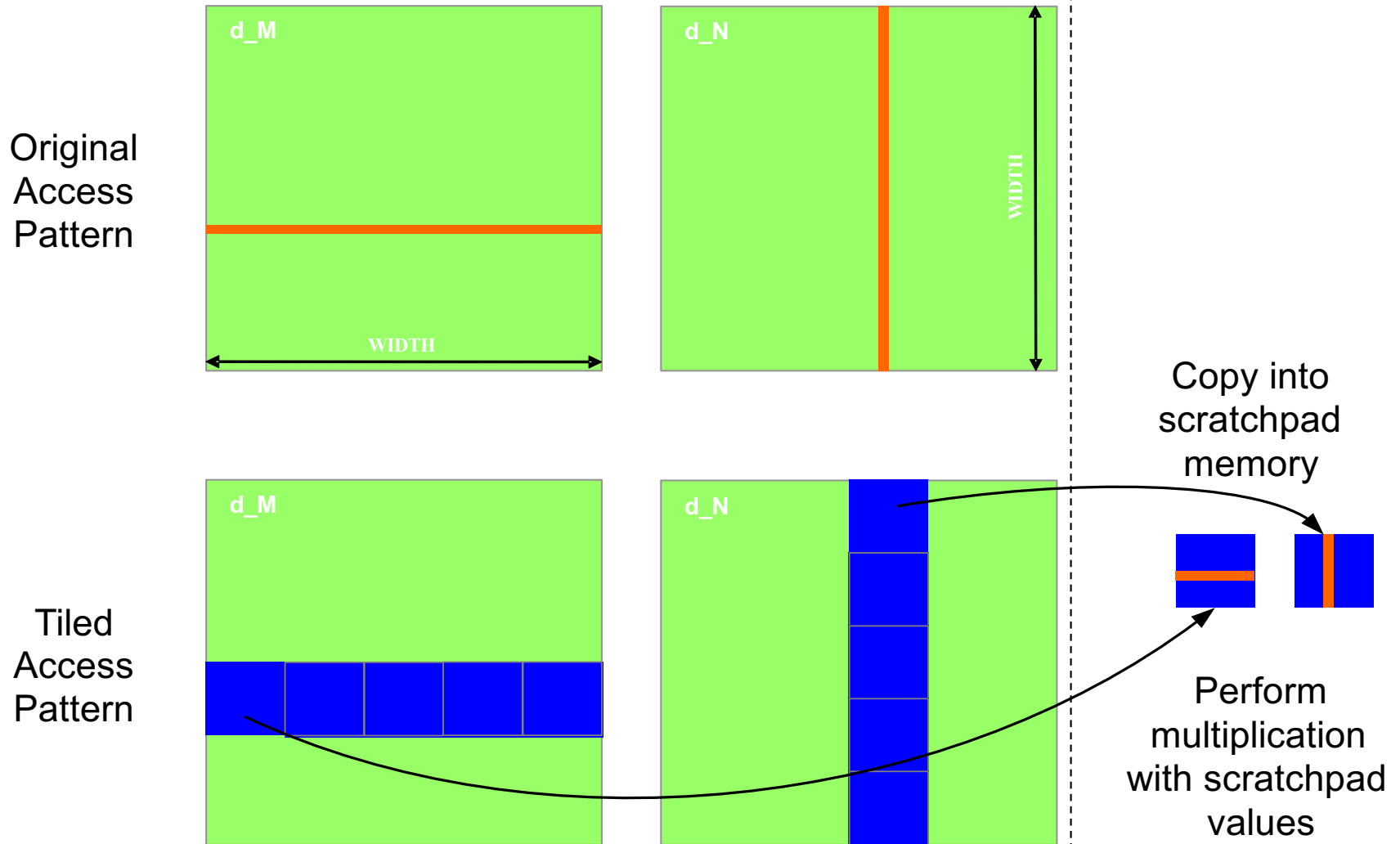
# In general, if neighboring threads access neighboring rows of M



# Loading Tiles for Block(0,0)



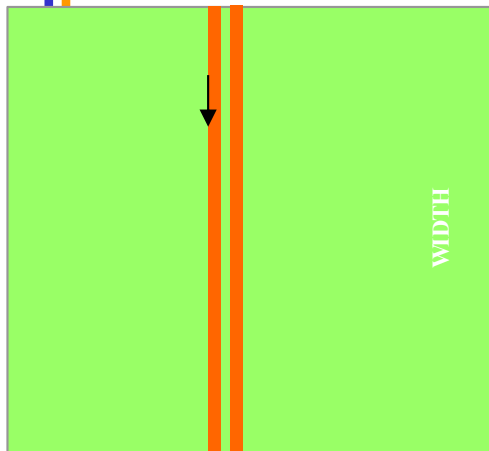
# Use shared memory to enable both data reuse and coalesced loading



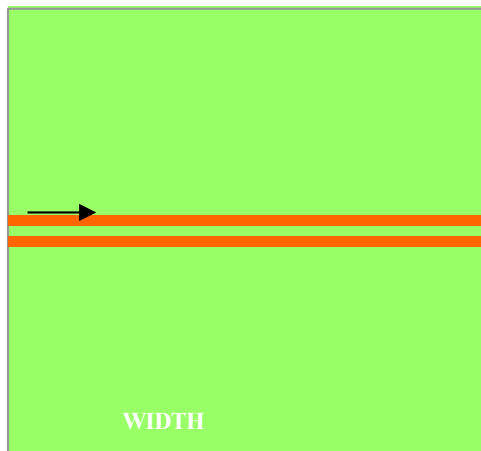


# In General, Four Important Access Patterns

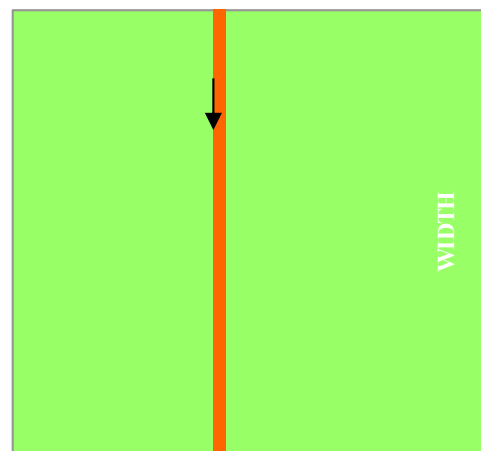
Neighboring  
Threads access



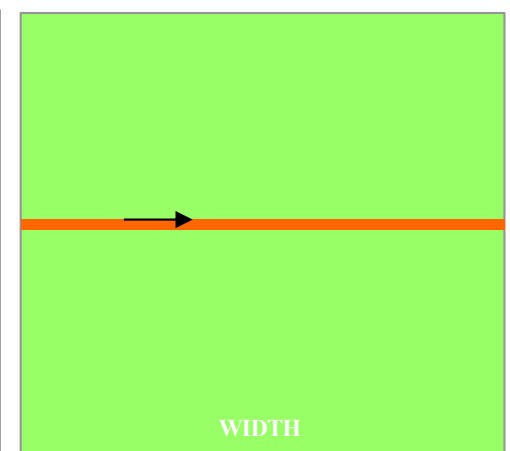
neighboring  
columns



neighboring  
rows

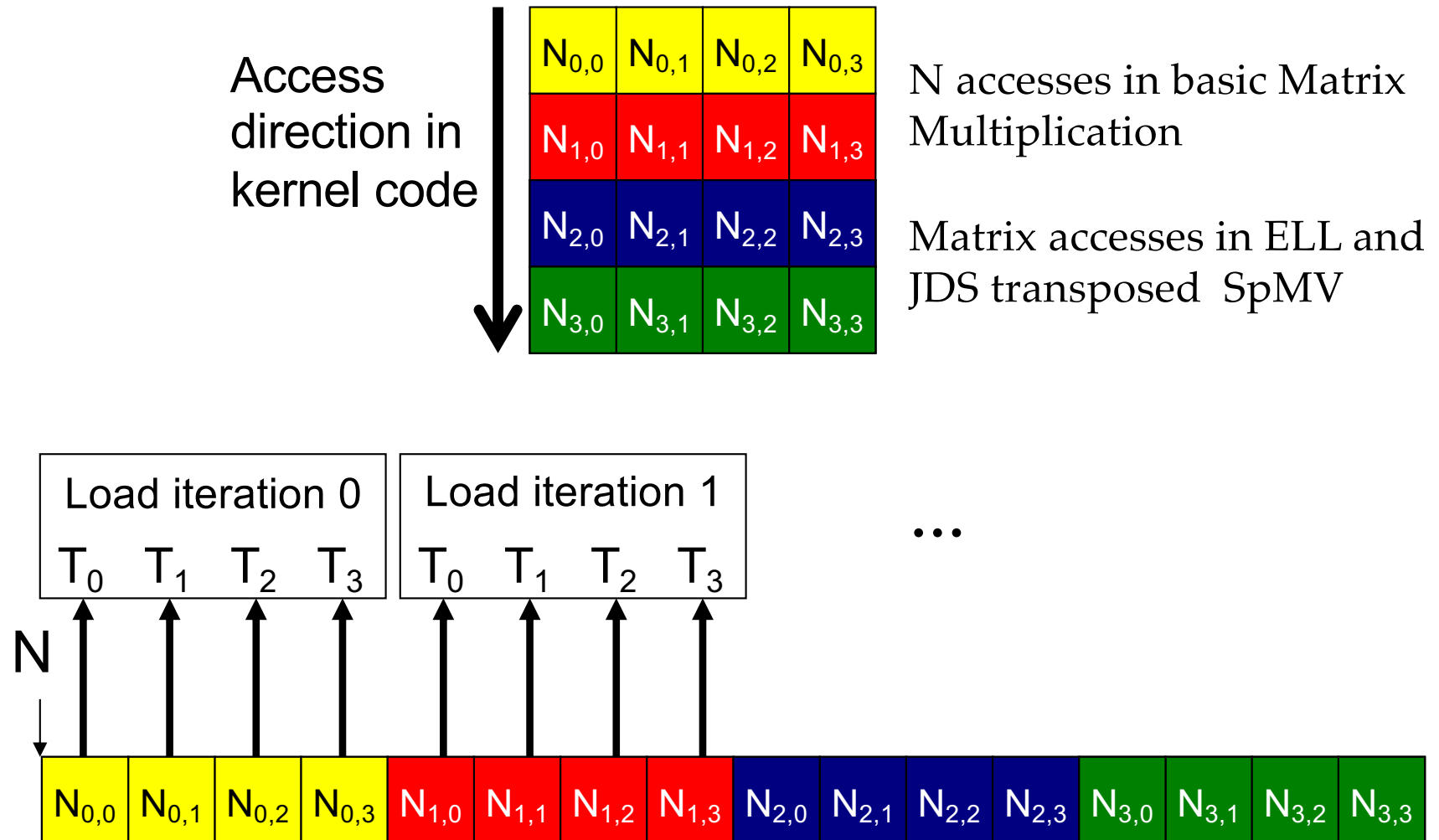


same  
columns



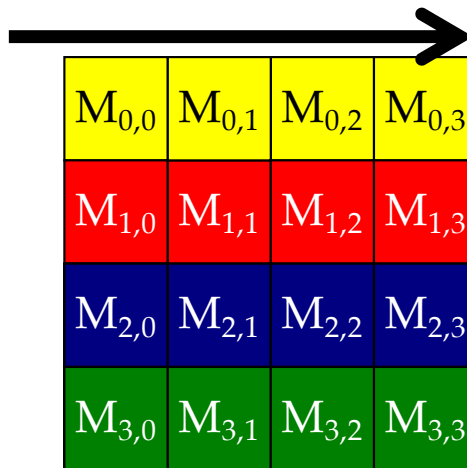
Same rows

# Neighboring Columns (Coalesced)



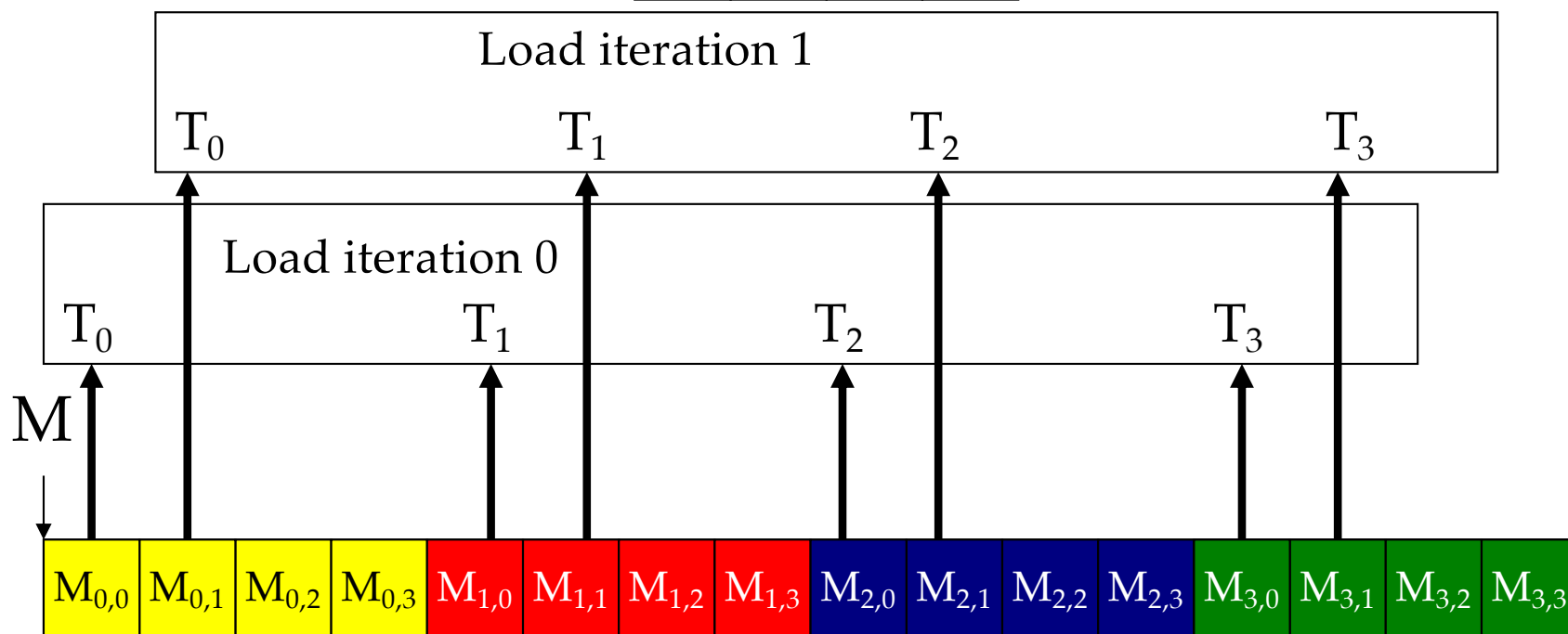
# Neighboring Rows (Not Coalesced)

Access  
direction in  
Kernel  
code

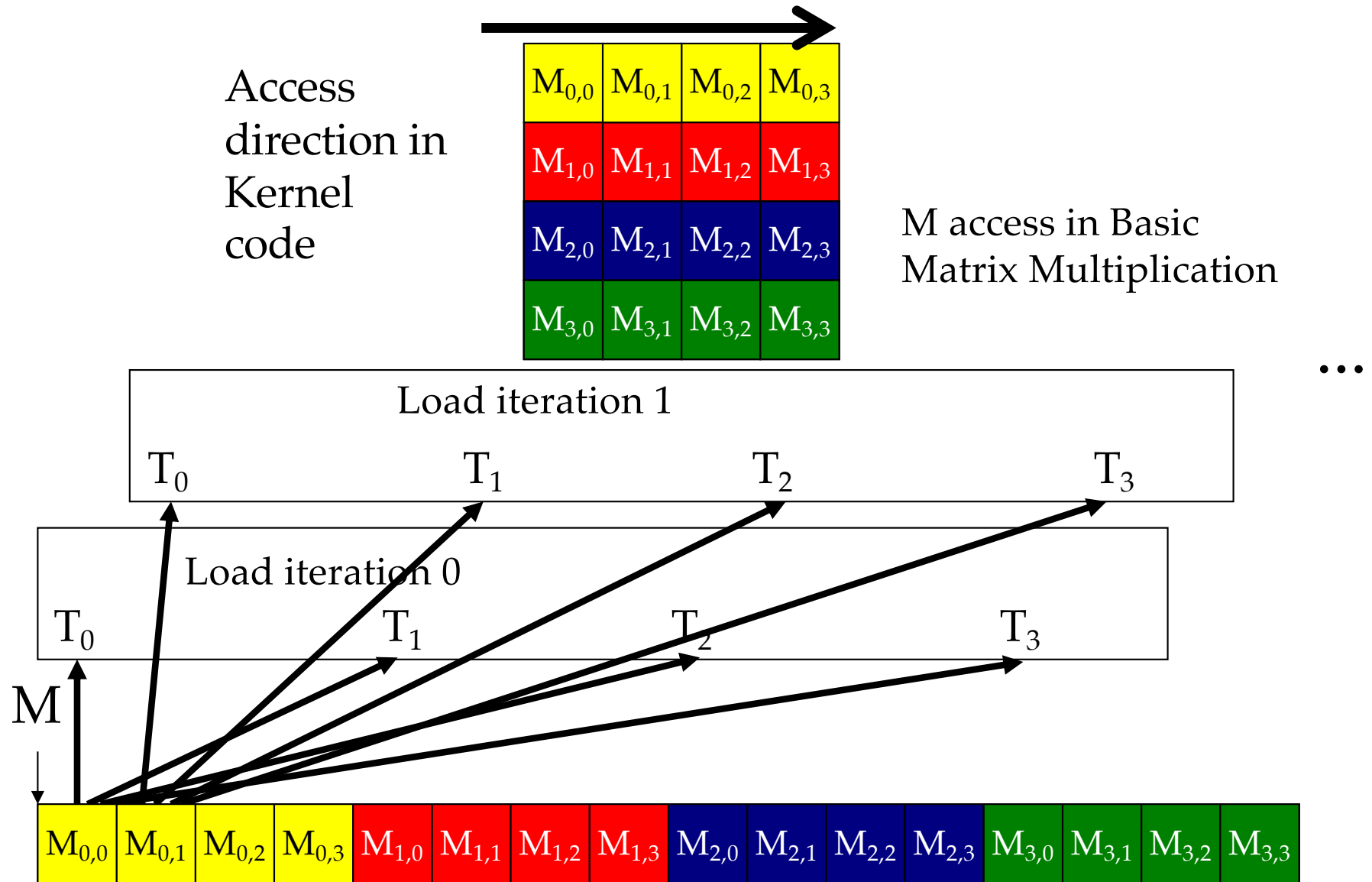


$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Matrix accesses in  
CSR and JDS SpMV



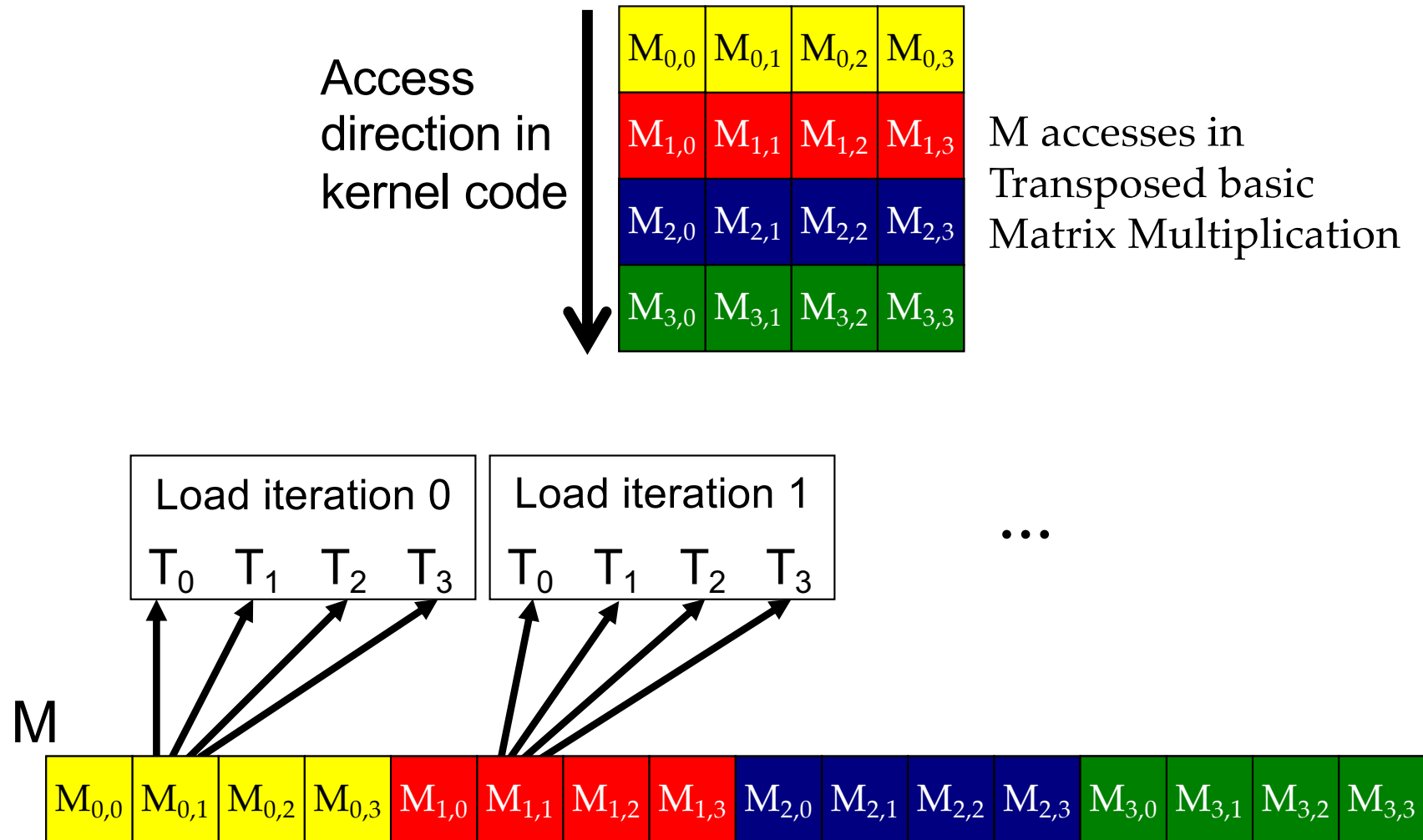
# Same Row (Coalesced, not efficient)



Coalesced since only one access will be done for all threads

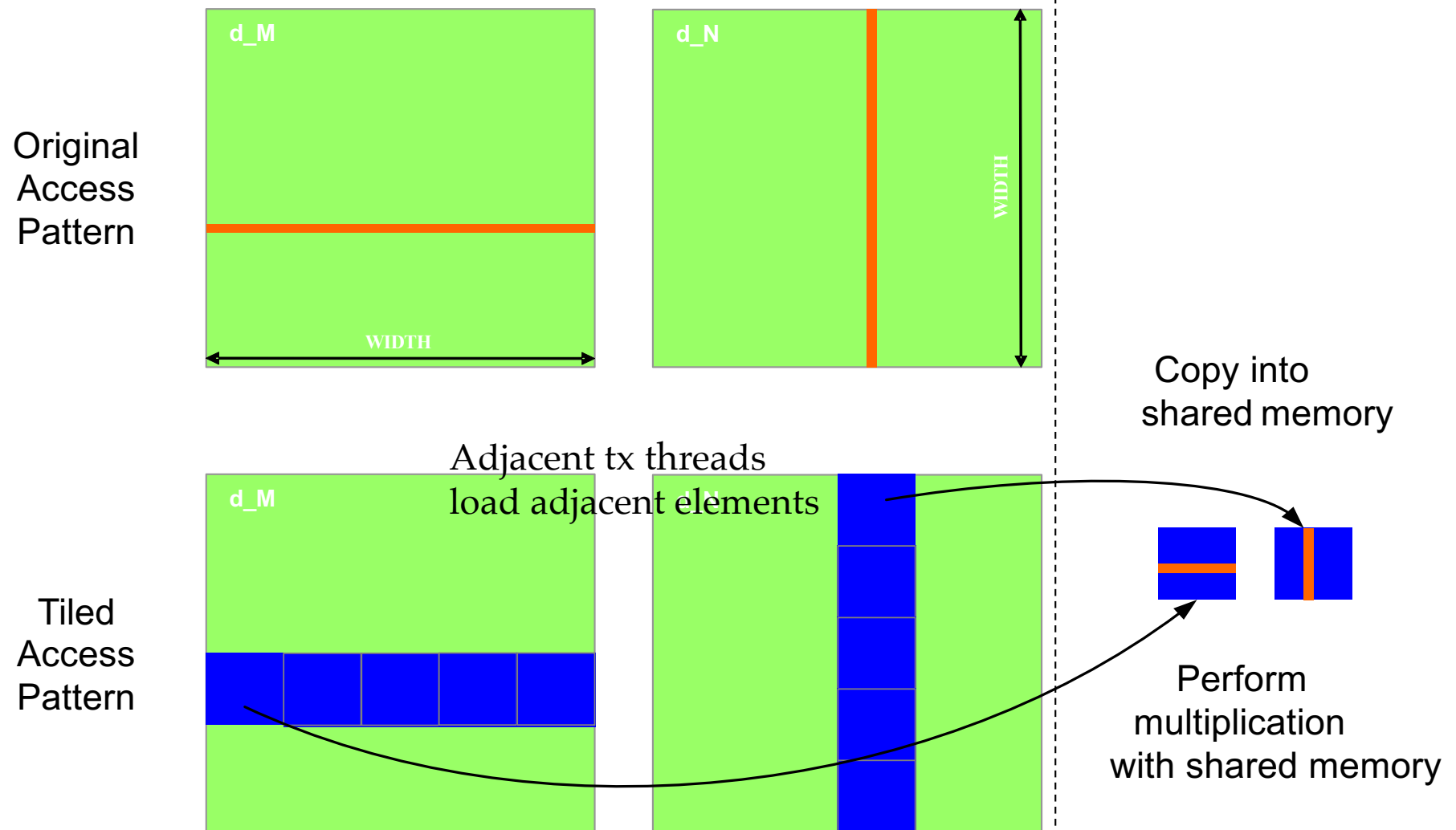
Inefficient since most of the data returned will be wasted.

# Same Column (Coalesced, not efficient)

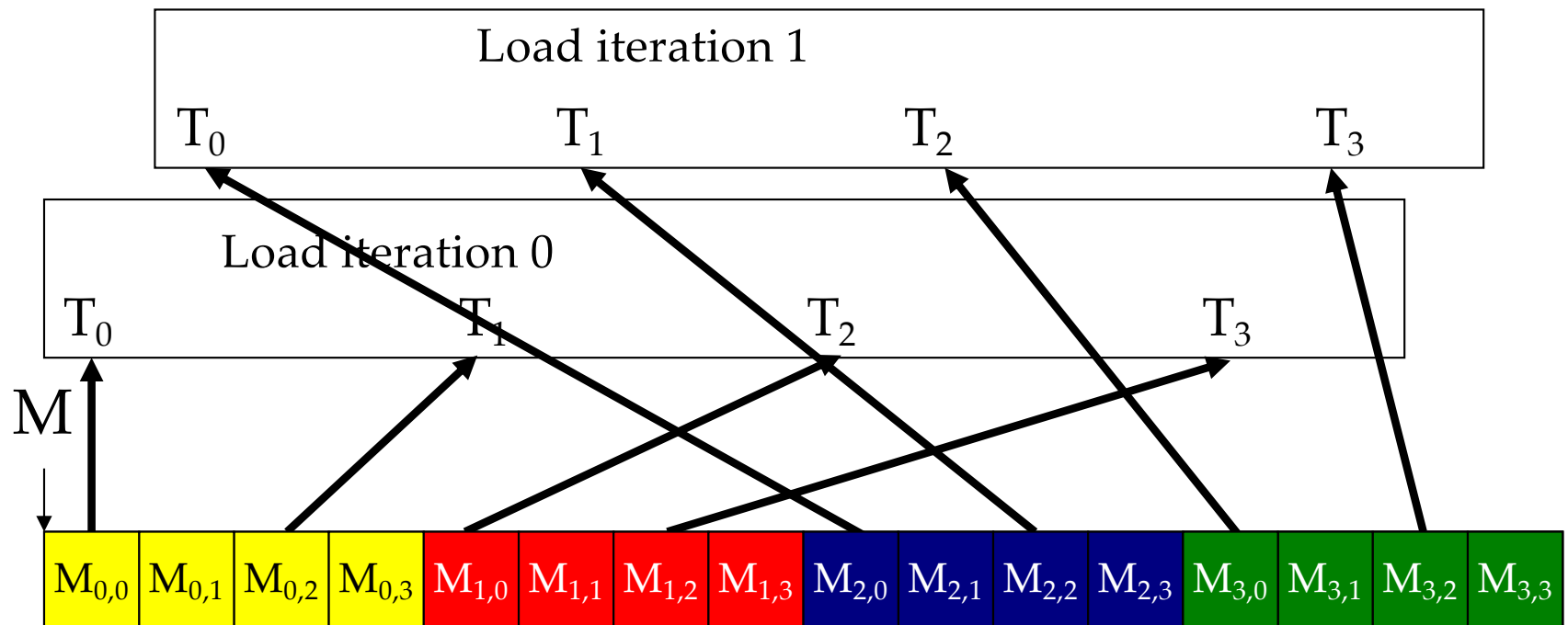


Coalesced since only one access will be done for all threads  
Inefficient since most of the data returned will be wasted.

# Enabling Coalescing with Shared Memory



# Stride Access Example (Stride = 1)



Adjacent threads in a warp access elements that are one way from each other.

Inefficient since half of the data returned will be wasted.

# Memory Coalescing - Summary

- After row-major linearization, adjacent threads in each warp access adjacent memory locations
  - Accesses for threads in the warp consolidated into one (or two) DRAM access (burst)
- The point is efficiency in using DRAM bursts
  - Current DRAM burst size is 64-128 bytes
  - Any distance between locations accessed by adjacent threads in a warp reduces DRAM efficiency
- When adjacent threads in a warp access identical locations, accesses for threads in the warp consolidated into one DRAM access



# Acknowledgement

---

- ◆ **Most slides for the rest of the lectures are from Prof. Wen-Mei Hwu**
  - **Used in UIUC ECE 408 in Fall 2019**
  - **© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2019**