ECE408 / CS483 / CSE408
Summer 2024

Applied Parallel Programming

Lecture 5: Locality and Tiled Matrix Multiplication

What Will You Learn Today?

- to evaluate the performance implications of global memory accesses
- prepare for MP-3: tiled matrix multiplication
- to assess the benefits of tiling

Kernel Invocation (Host-side Code) vk

```
// Setup the execution configuration
// BLOCK_WIDTH is a #define constant
dim3 dimGrid(ceil((1.0*Width)/BLOCK_WIDTH),
           ceil((1.0*Width)/BLOCK_WIDTH), 1);

dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH, 1);


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

The Problem: Accesses to Global Memory

```
__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 idenx 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*Width+k] *
               d_N[k*Width+Col];
   d_P[Row*Width+Col] = Pvalue;
  }
}
```

accesses to global memory

Review: 4B of Data per FLOP

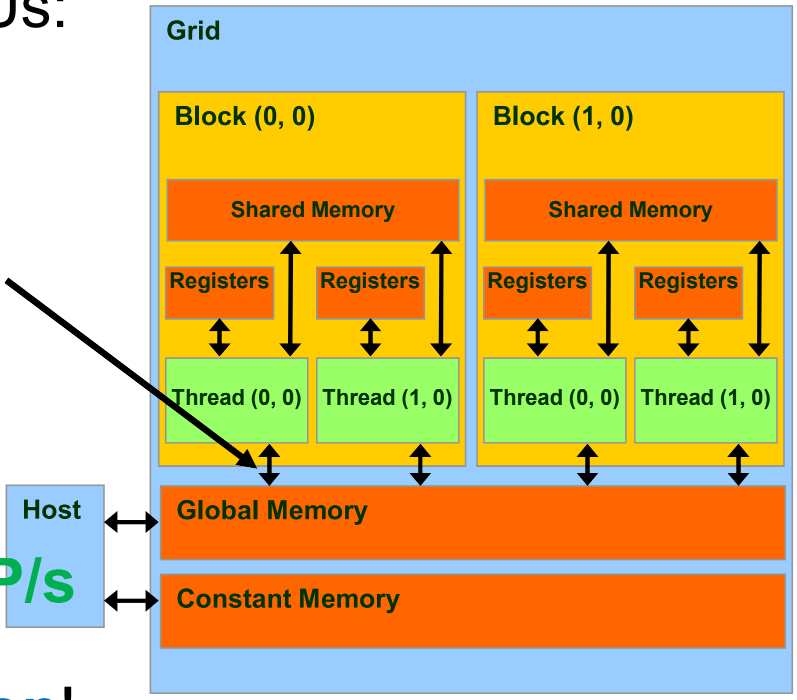
- Each threads access global memory
  - for elements of **M** and **N**:
  - **4B each**, or **8B per pair**.
  - (And once TOTAL to **P** per thread—ignore it.)
- With each pair of elements,
  - a thread does a single multiply-add,
  - **2 FLOP**—floating-point operations.
- So for every FLOP,
  - **a thread needs** 4B from memory:
  - **4B / FLOP**.

Review: Extremely Poor Performance

- One generation of GPUs:
  - –**1,000 GFLOP/s** of compute power, and
  - –**150 GB/s** of memory bandwidth.
- Dividing bandwidth by memory requirements:

$$\frac{150\ GB/s}{4\ B/FLOP} = 37.5\ \text{GFLOP/s}$$

which **limits computation**!



The Solution? Reuse Memory Accesses!

But **37.5 GFLOPs is a limit**.

In an **actual execution**,
- memory is not busy all the time, and
- the code **runs at** about **25 GFLOPs**.

To get closer to 1,000 GFLOPs
- we **need to** drastically **cut down**
- **accesses to global memory**.

But … how?

Tiled Matrix-Matrix Multiplication using Shared Memory

A Common Programming Strategy

- The dilemma:
  - Matrices **M** and **N** are large.
  - They **fit** easily **in global memory**, **but** that's **slow**.
  - **Shared memory** is **fast, but M and N don't fit.**
- The solution:
  - **Break M and N into tiles**
  - (called blocks in the much older CPU literature).
  - **Read a tile** into shared memory.
  - **Use the tile** from shared memory.
  - **Repeat** until done.

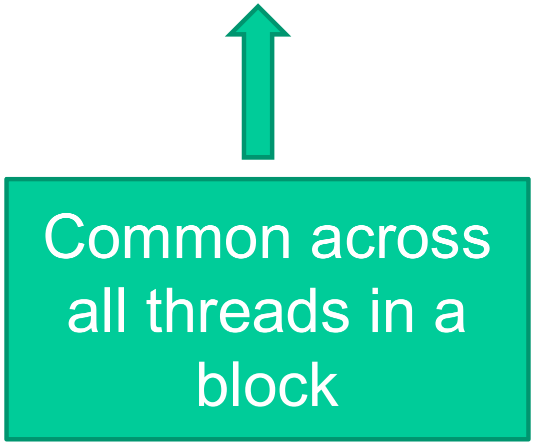
A Common Programming Strategy

- In a GPU, **only threads in a block can** use **share**d memory.
- Thus, each **block** operates on **separate tiles**:
  - **Read tile(s)** into shared memory **using multiple threads** to exploit memory-level parallelism.
  - **Compute** based on shared memory tiles.
  - **Repeat**.
  - **Write results** back **to global memory**.

Declaring Shared Memory Arrays

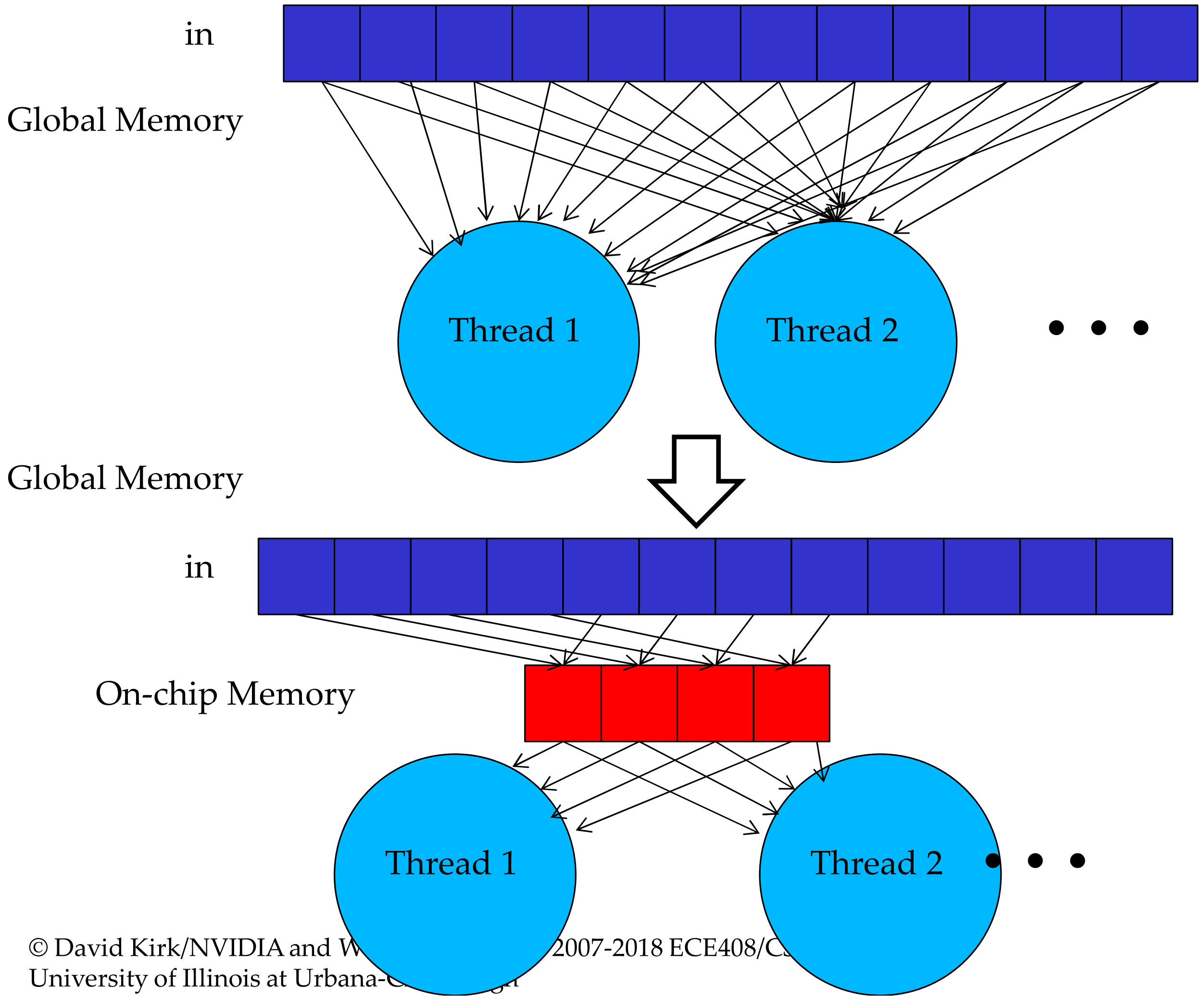
```
__global__
void MatrixMulKernel(float* M, float* N, float* P, int Width)
{

    __shared__  float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__  float subTileN[TILE_WIDTH][TILE_WIDTH];
```



Common across all threads in a block

Shared Memory Tiling Basic Idea



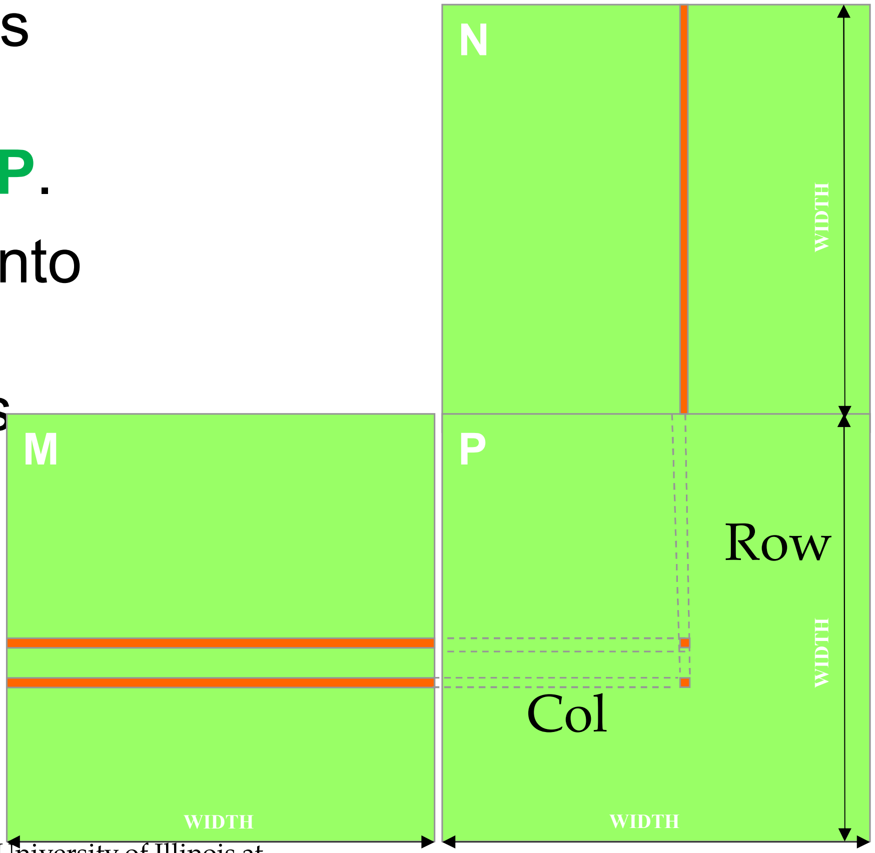
Transform Global Memory Accesses into On-Chip Accesses

- **Identify a tile of global data**
  - with each datum accessed by multiple threads
  - and/or accessed repeatedly.

- **Change access pattern in kernel**
  - Load the tile from global memory into on-chip memory
  - Have the threads access the data from on-chip memory
  - Move on to the next tile

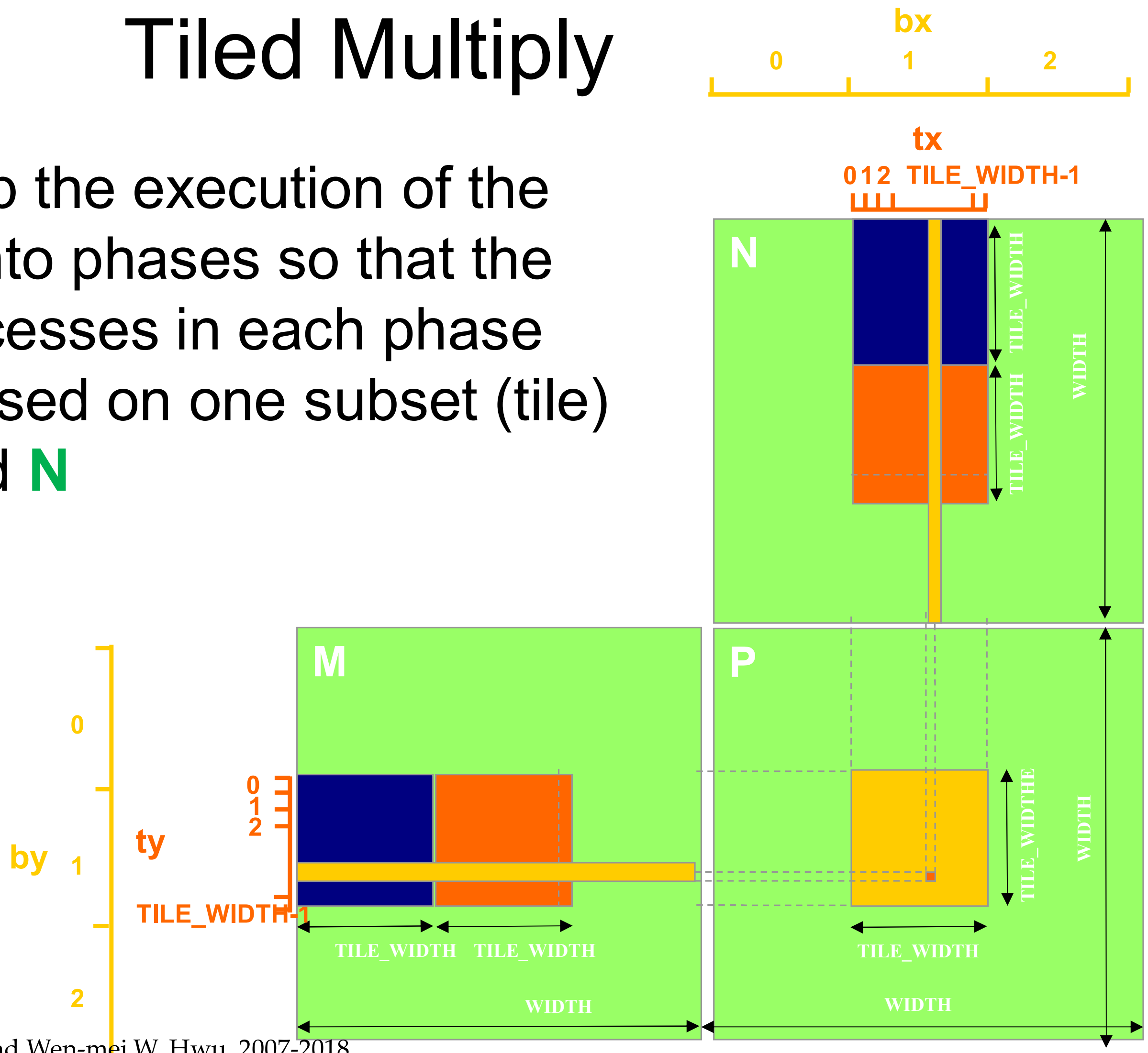
Idea: Place global memory data into Shared Memory for reuse

- Each input element is used to calculate WIDTH elements of P.
- Load each element into Shared Memory and have several threads use the local version to reduce 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**

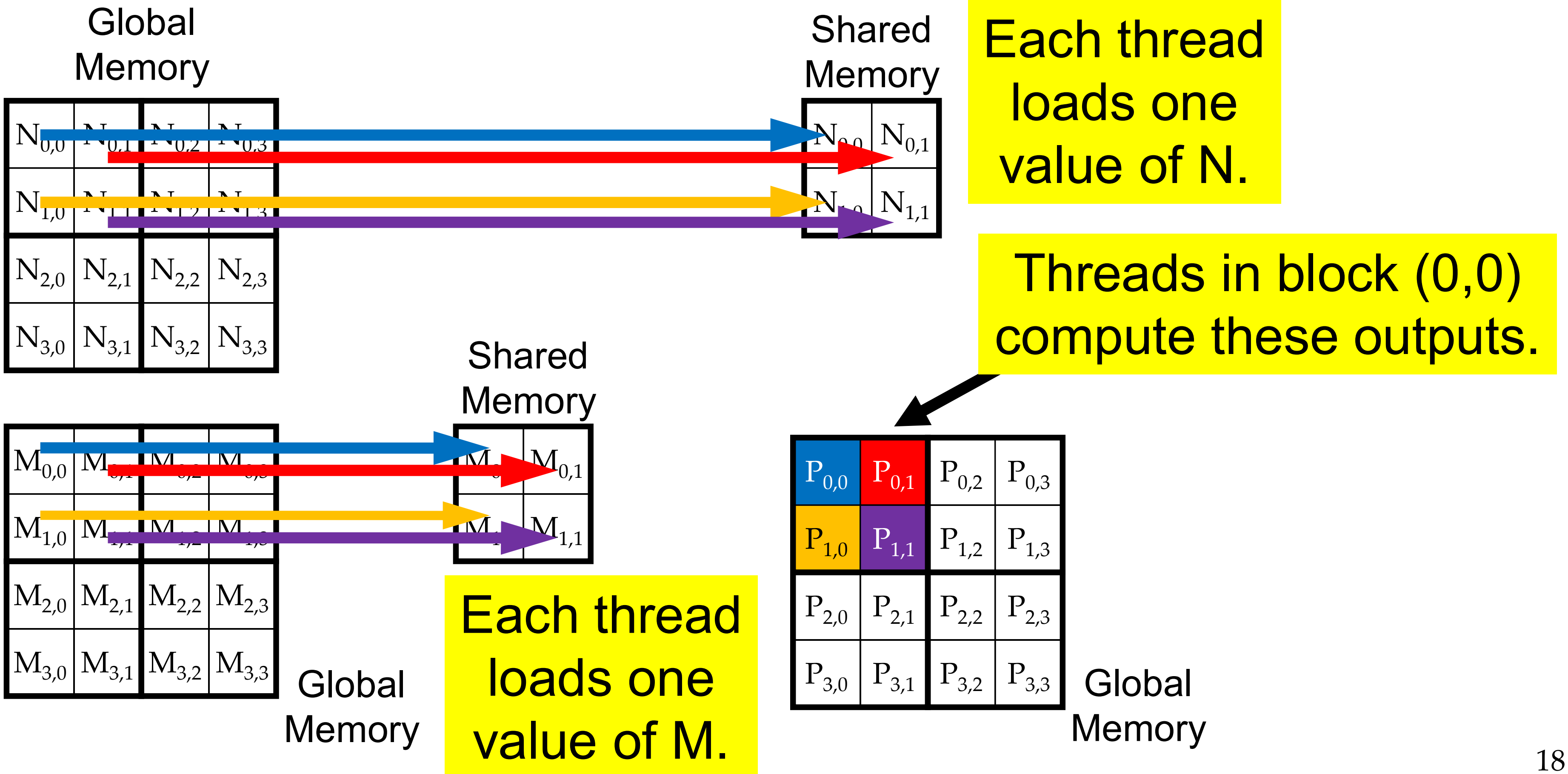


Loading a Tile

- 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).

Thread Block (0,0) Starts by Loading Two Tiles

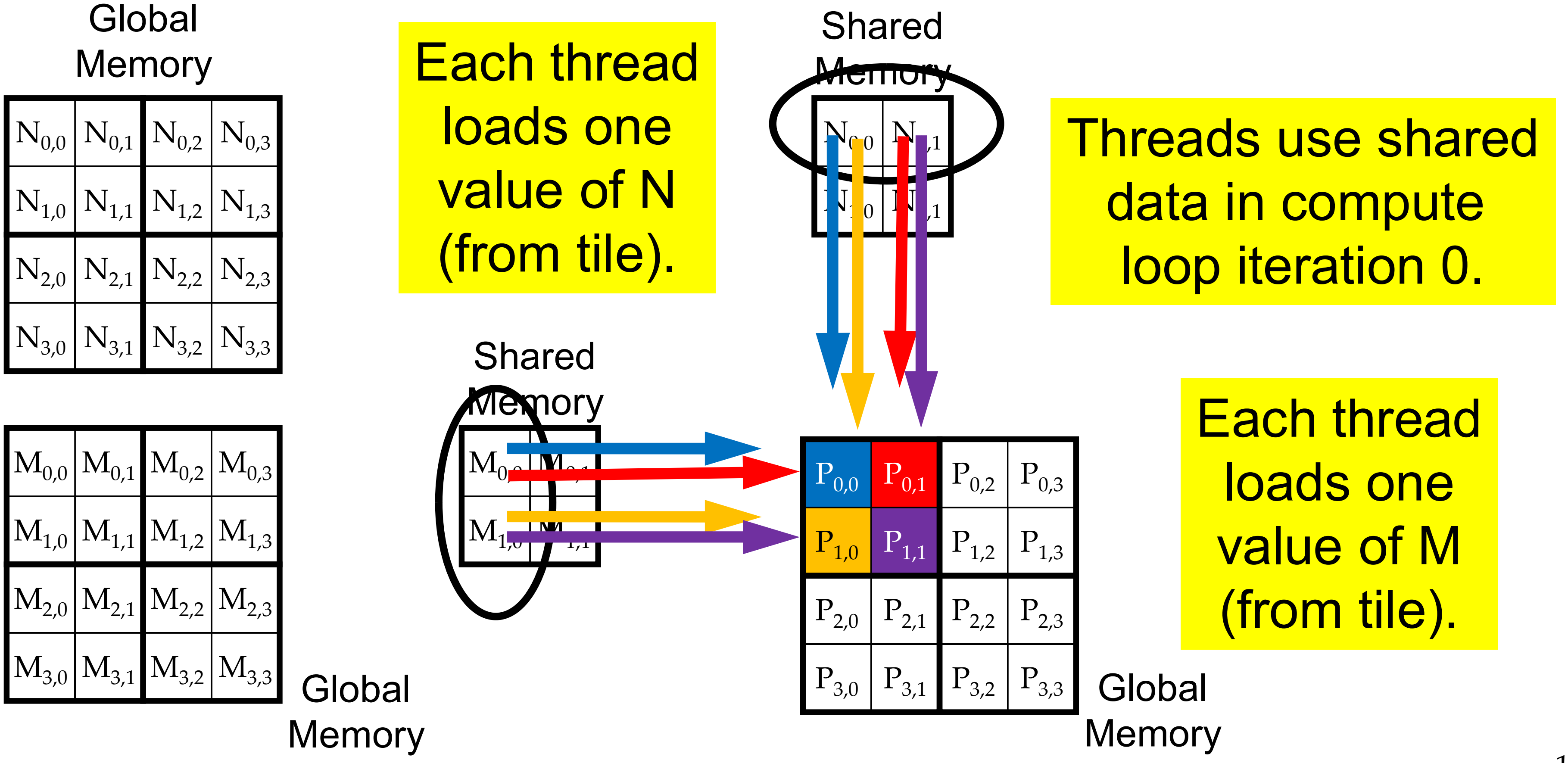

Global Memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Shared Memory
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Global Memory
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$
Global Memory

Each thread loads one value of N.

Threads in block (0,0) compute these outputs.

Each thread loads one value of M.

Thread Block (0,0) Computes on Shared Tiles (Iter 0)



Global Memory

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

Each thread loads one value of N (from tile).

Shared Memory

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Threads use shared data in compute loop iteration 0.

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

Global Memory

Shared Memory

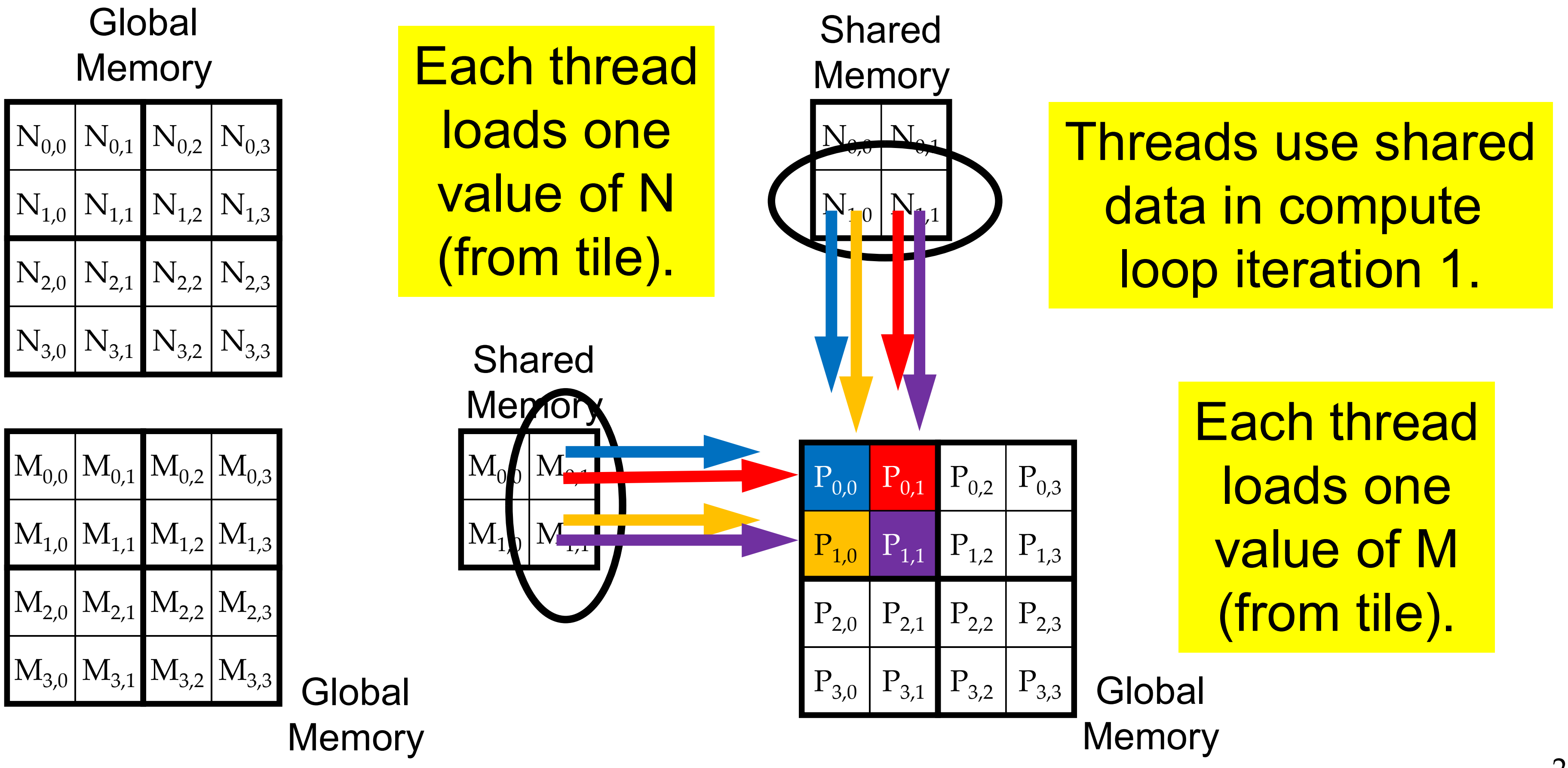
$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

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

Global Memory

Each thread loads one value of M (from tile).

Thread Block (0,0) Computes on Shared Tiles (Iter 1)

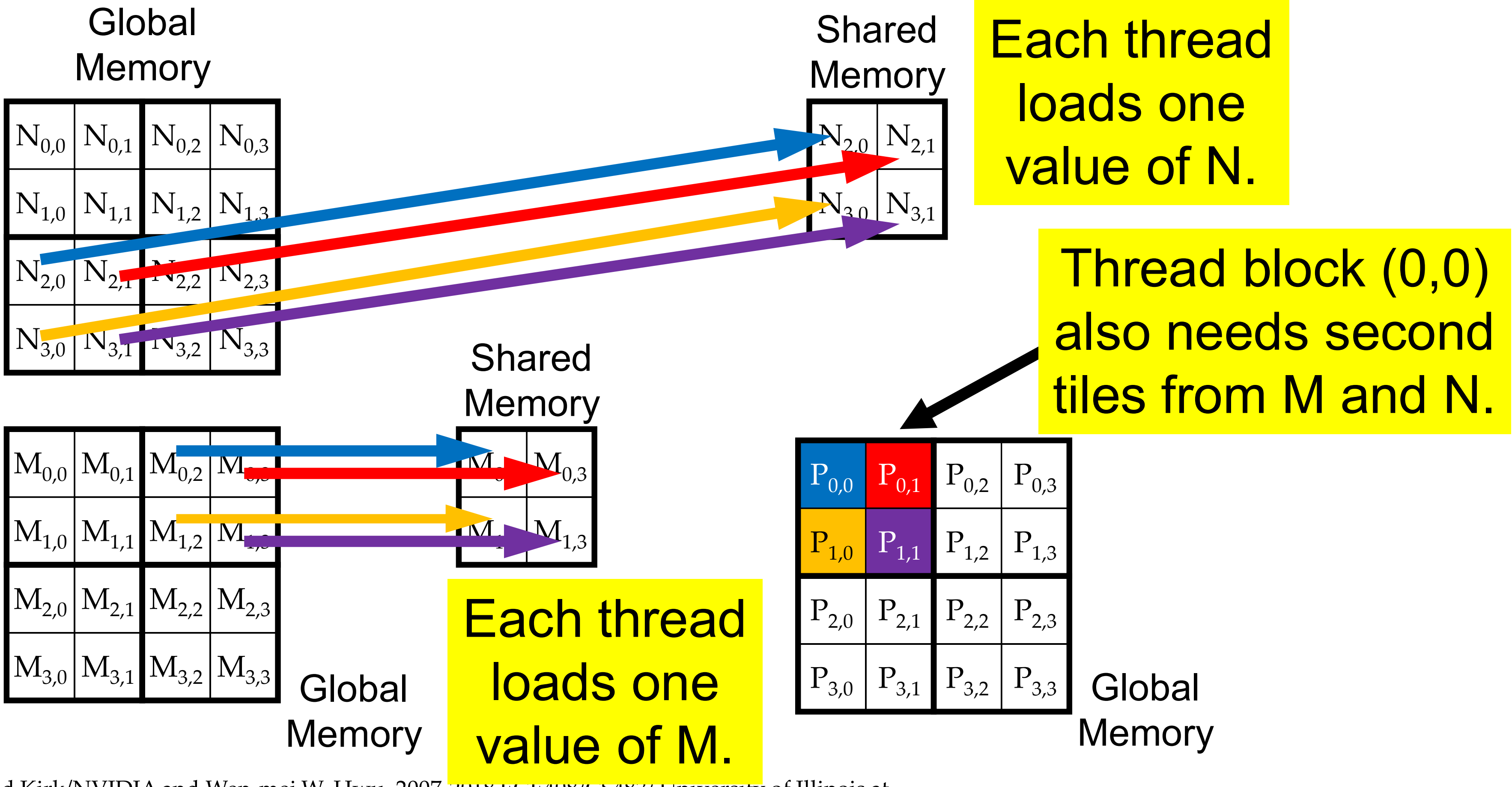

Global Memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Global Memory
Shared Memory
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$
Global Memory

Each thread loads one value of N (from tile).

Threads use shared data in compute loop iteration 1.

Each thread loads one value of M (from tile).

Then Thread Block (0,0) Loads Two New Tiles

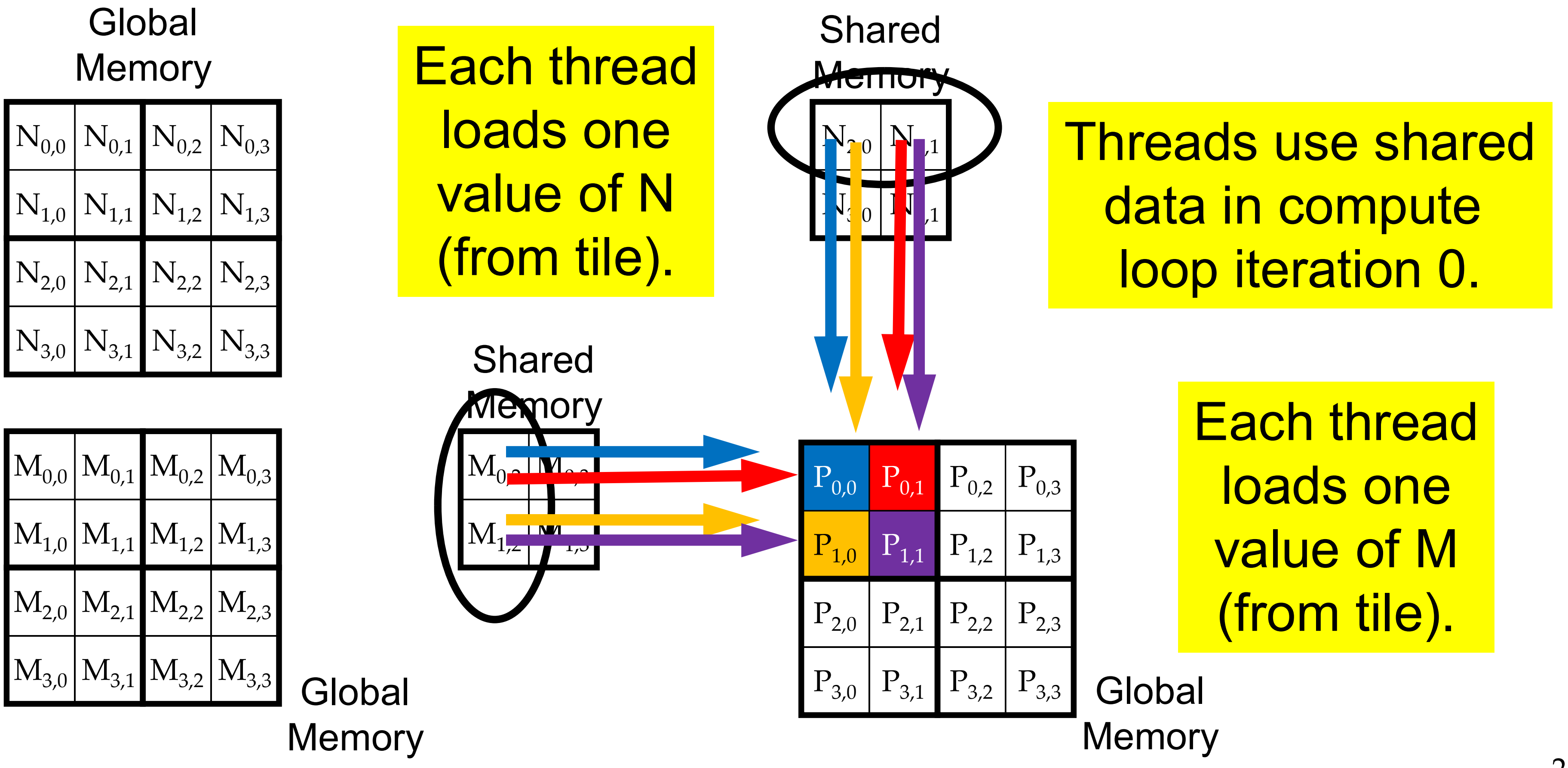

Global Memory
Shared Memory
Shared Memory
Global Memory
Global Memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$M_{0,3}$
$M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Each thread loads one value of N.

Thread block (0,0) also needs second tiles from M and N.

Each thread loads one value of M.

Inner Loop Iteration is Now Identical (Iter 0)!



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

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

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

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

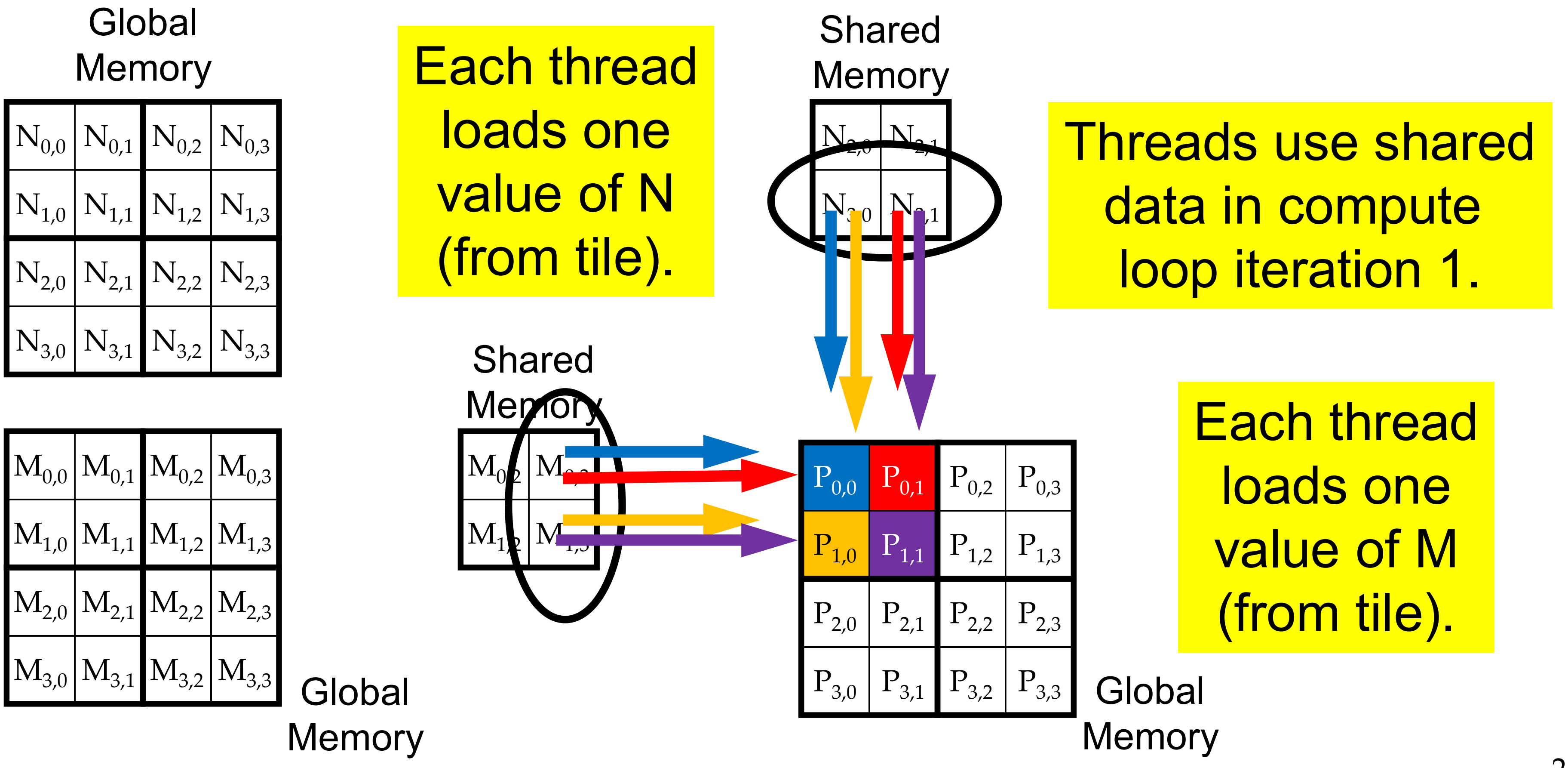
$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Each thread loads one value of N (from tile).

Threads use shared data in compute loop iteration 0.

Each thread loads one value of M (from tile).

Inner Loop Iteration is Now Identical (Iter 1)!



Global Memory

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

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

Global Memory

Each thread loads one value of N (from tile).

Shared Memory

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

Threads use shared data in compute loop iteration 1.

Shared Memory

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

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

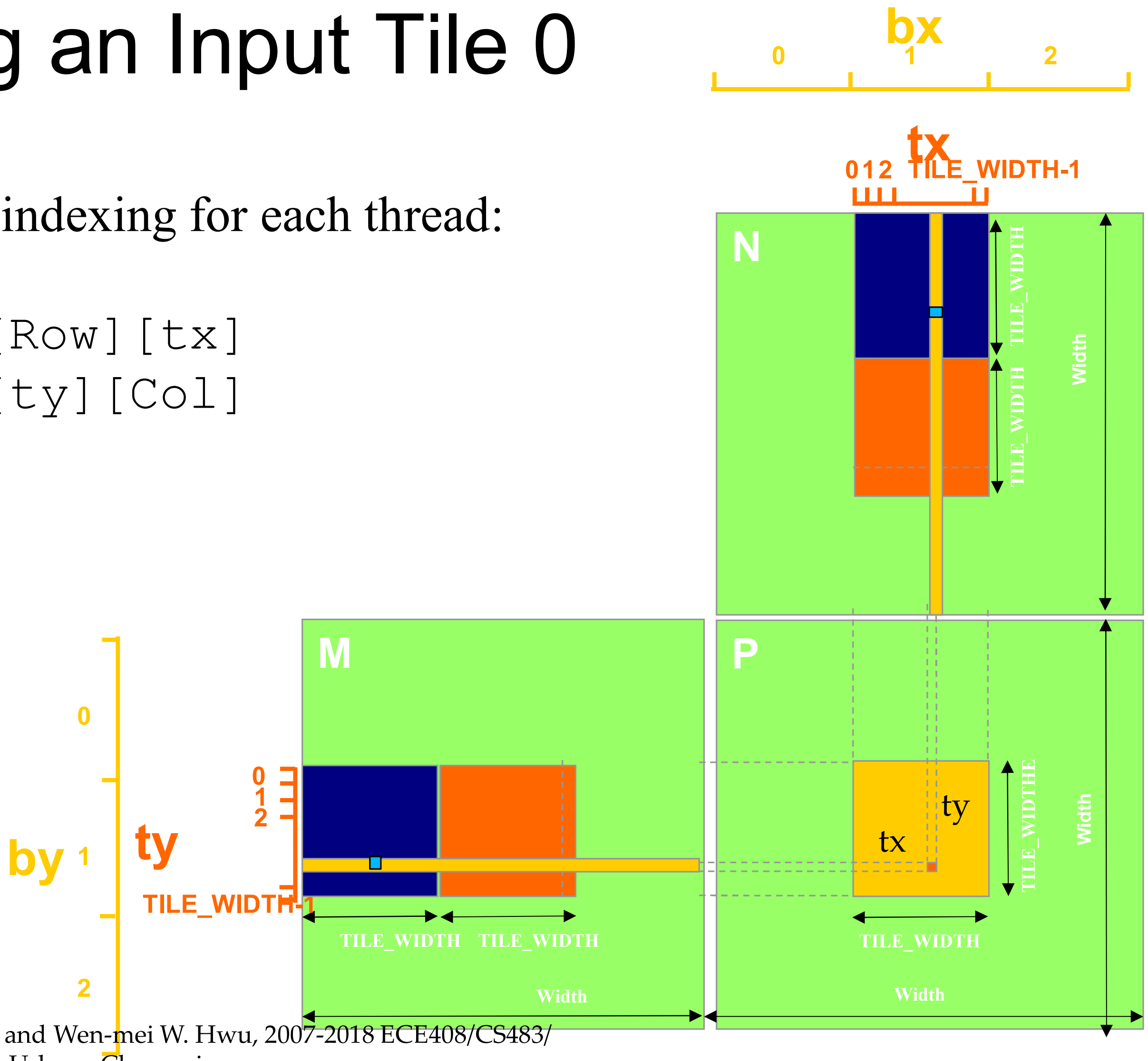
Each thread loads one value of M (from tile).

Global Memory

Loading an Input Tile 0

Tile 0 2D indexing for each thread:

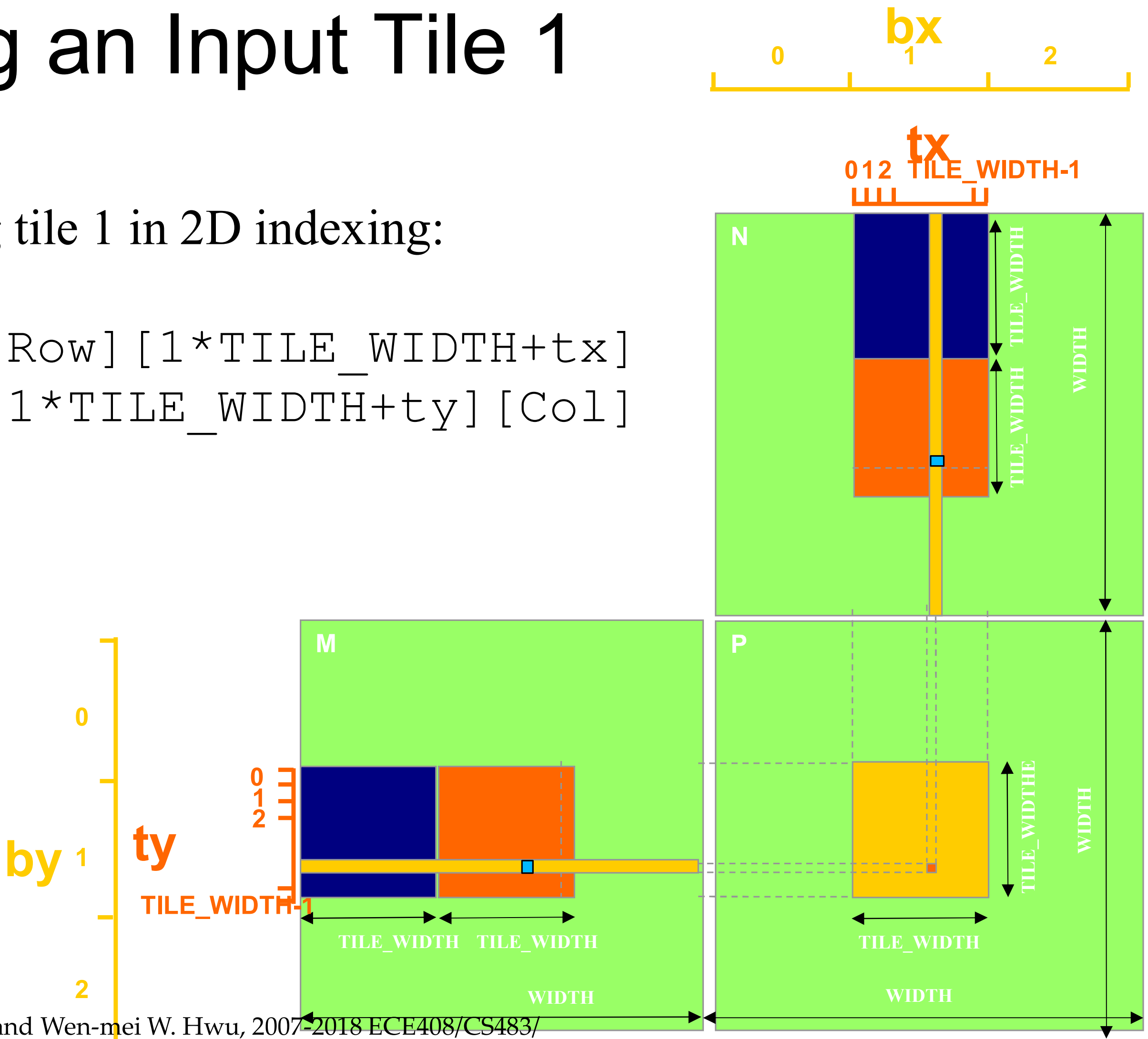
```
M[Row][tx]
N[ty][Col]
```



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

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



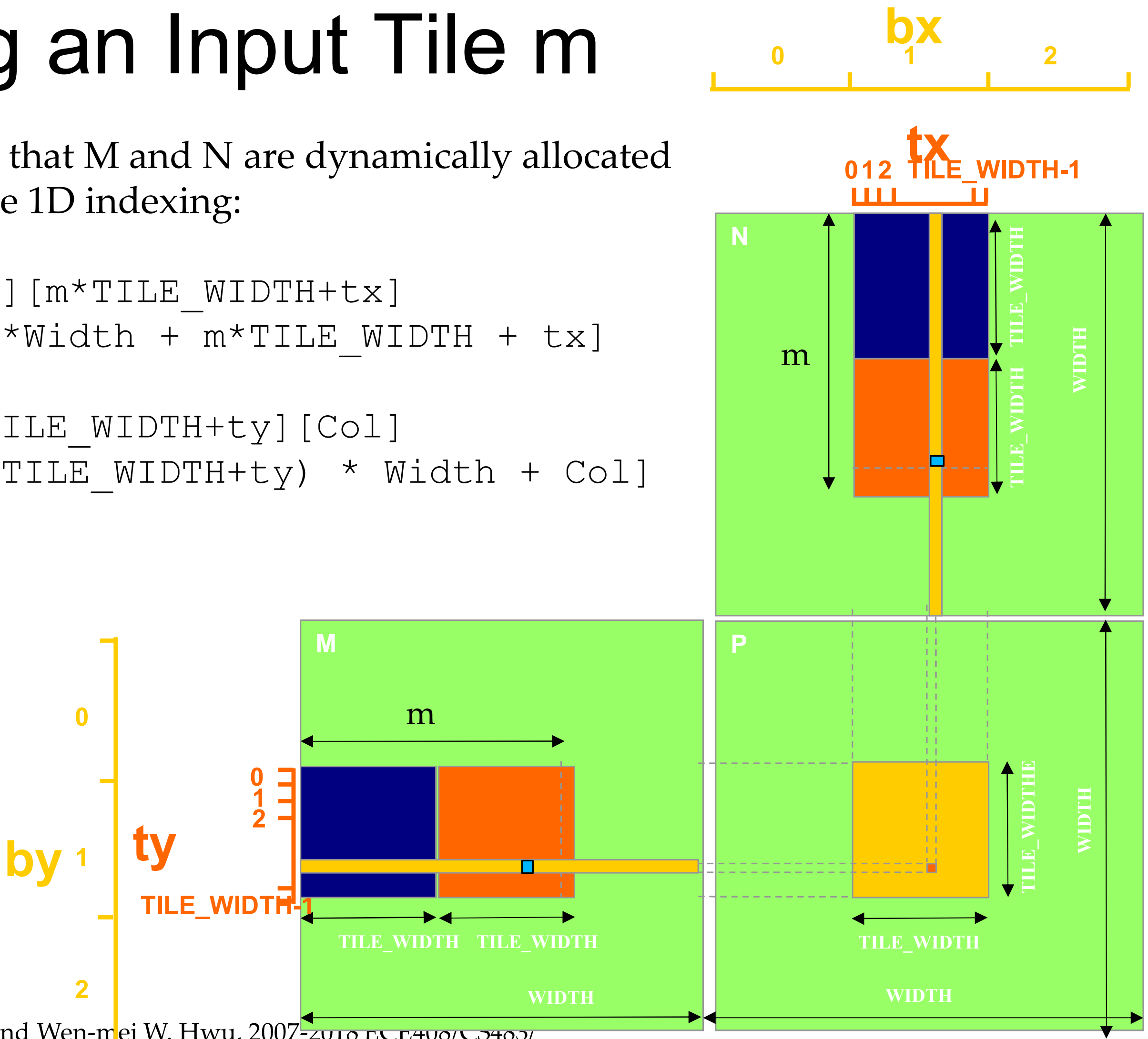
© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2018 ECE408/CS483/
University of Illinois at Urbana-Champaign

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]
```

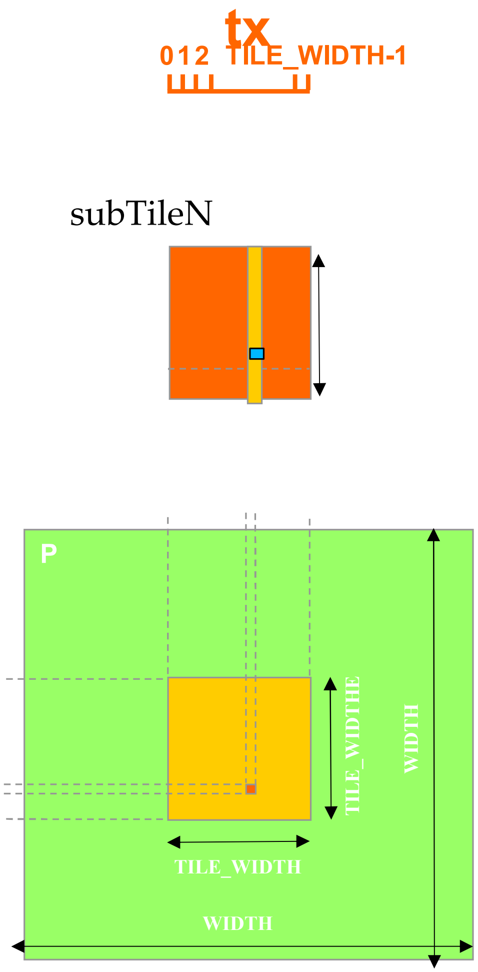


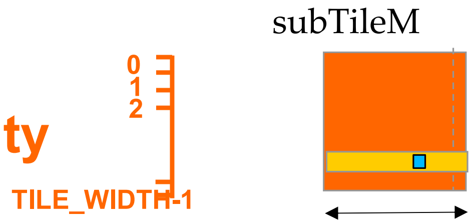
Accessing a Tile

To perform the kth step of the product within the tile:

```
subTileM[ty][k]
```

```
subTileN[k][tx]
```





We’re Not There Yet!

- But …

- **How can a thread know** …
  - **That another thread has finished** its part of the tile?
  - Or that another thread has finished using the previous tile?

We need to synchronize!

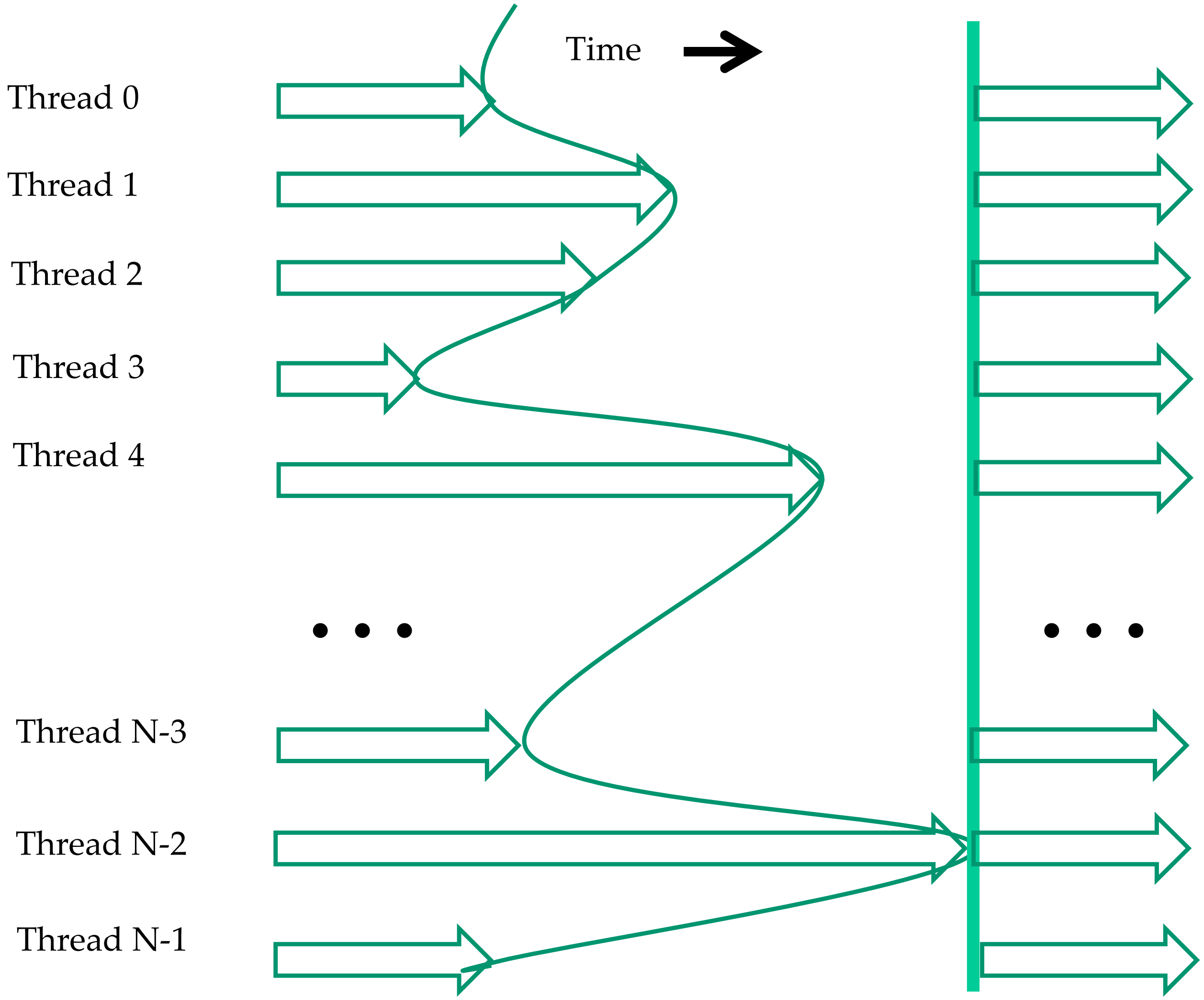
Leveraging Parallel Strategies

- **Bulk synchronous execution**:
  threads execute roughly in unison
  1. Do some work
  2. Wait for others to catch up
  3. Repeat
- **Much easier programming model**
  – Threads only parallel within a section
  – Debug lots of little programs
  – Instead of one large one.
- **Dominates high-performance applications**

Bulk Synchronous Steps Based on Barriers

- **How does it work?**
  Use a barrier to wait for thread to ‘catch up.’

- A barrier is a synchronization point:
  - **each thread calls a function** to enter barrier;
  - **threads block** (sleep) in barrier function **until all threads have called**;
  - **after last thread calls** function, **all threads continue** past the barrier.



Time

Thread 0

Thread 1

Thread 2

Thread 3

Thread 4

• • •

Thread N-3

Thread N-2

Thread N-1

• • •

Use __syncthreads for CUDA Blocks

- **How does it work in CUDA?**
  Only **within thread blocks**!

- The function: **`void __syncthreads(void);`**

- N.B.
  - **All threads** in block **must enter** (no subsets).
  - All threads must enter the **SAME static call**
    (not the same as all threads calling function!).

Barrier Trauma: What's Actually Done?

- **What exactly is guaranteed** to have finished?
  - Are **shared memory** operations before a barrier (e.g., stores) guaranteed to have completed?
  - What about **global memory** ops?
  - What about **atomic ops** with no return values?
  - What about I/O operations?
- CUDA manual: all global and shared memory ops (which presumably includes atomic variants) have completed.
- **Avoid assumptions about I/O** (such as printf).

Tiled Matrix Multiplication Kernel

```
__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];

    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 Row = by * TILE_WIDTH + ty; // note: blockDim.x == TILE_WIDTH
    int Col = bx * TILE_WIDTH + tx; //       blockDim.y == TILE_WIDTH
    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!
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
       // Collaborative loading of M and N tiles into shared memory
       subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
       subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
       __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
           Pvalue += subTileM[ty][k] * subTileN[k][tx];
       __syncthreads();
    }
    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;
  }
}
```

Use of Large Tiles Shifts Bottleneck

- Recall our example GPU: **1,000 GFLOP/s**, **150 GB/s**
- **16x16 tiles** use each operand for 16 operations
  - **reduce global** memory **accesses by** a factor of **16**
  - **150GB/s** bandwidth supports
    (150/4)*16 = **600 GFLOPS**!
- **32x32 tiles** use each operand for 32 operations
  - **reduce global** memory **accesses by** a factor of **32**
  - **150 GB/s** bandwidth supports
    (150/4)*32 = **1,200 GFLOPS**!
  - **Memory bandwidth is no longer the bottleneck!**

Also Need Parallel Accesses to Memory

- Shared memory size
  - implementation dependent
  - **64kB** per SM in Maxwell (48kB max per block)
- Given **TILE_WIDTH of 16** (256 threads / block),
  - each thread block uses
    2*256*4B = 2kB of shared memory,
  - which limits active blocks to 32;
  - max. of 2048 threads per SM,
  - which limits blocks to 8.
  - Thus up to 8*512 = **4,096 pending loads**
    (2 per thread, 256 threads per block)

Another Good Choice: 32x32 Tiles

- Given **TILE_WIDTH of 32** (1,024 threads / block),
  - each thread block uses
    2*1024*4B = 8kB of shared memory,
  - which limits active blocks to 8;
  - max. of 2,048 threads per SM,
  - which limits blocks to 2.
  - Thus up to 2*2,048 = **4,096 pending loads**
    (2 per thread, 1,024 threads per block)

(same memory parallelism exposed)

Current GPU? Use 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
  - …

QUESTIONS?

READ CHAPTER 4!

Problem Solving

Barriers are needed to ensure that one thread's writes are visible to other threads—for communication between threads in a block. Consider some code…

```
int32_t tx = threadIdx.x;
int32_t ty = threadIdx.y;
__shared__ float tile[TW][TW];
tile[ty][tx] = myNumber;

otherNumber = tile[ty][tx];
```

Q: Do we need to add a call to **`__syncthreads`** here?

A: No __syncthreads call needed: each thread reads only the value that it wrote itself!

Problem Solving

Here’s a slightly different code…

```
int32_t tx = threadIdx.x;
int32_t ty = threadIdx.y;
__shared__ float tile[TW][TW];
tile[ty][tx] = myNumber;

otherNumber = tile[tx][ty];
```

Q: Do we need to add a call to **`__syncthreads`** here?

A: Yes! Otherwise, the values copied into each thread’s **`otherNumber`** variable may not be deterministic.

Problem Solving

Let’s extend that last code…

```
int32_t tx = threadIdx.x;
int32_t ty = threadIdx.y;
__shared__ float tile[TW][TW];
tile[ty][tx] = myNumber;
__syncthreads();
otherNumber = tile[tx][ty];

thirdNumber = tile[TW - tx - 1][ty];
```

Q: Do we need to add a call to **`__syncthreads`** here?

A: No: no thread modifies **`tile`** after the first barrier.

Problem Solving

And one final extension…

```
int32_t tx = threadIdx.x;
int32_t ty = threadIdx.y;
__shared__ float tile[TW][TW];
tile[ty][tx] = myNumber;
__syncthreads();
otherNumber = tile[tx][ty];
thirdNumber = tile[TW - tx - 1][ty];

tile[ty][tx] = myNumber * 2.0;
```

Q: Do we need to add a call to **`__syncthreads`** here?

A: Yes! Reads for **`thirdNumber`** must finish before other threads change **`tile`**.