GPU Shared Memory

Lecture 17
April 29, 2025

To Dos

Program #7

Reading for next time

A Simple Matrix Multiplication Kernel

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

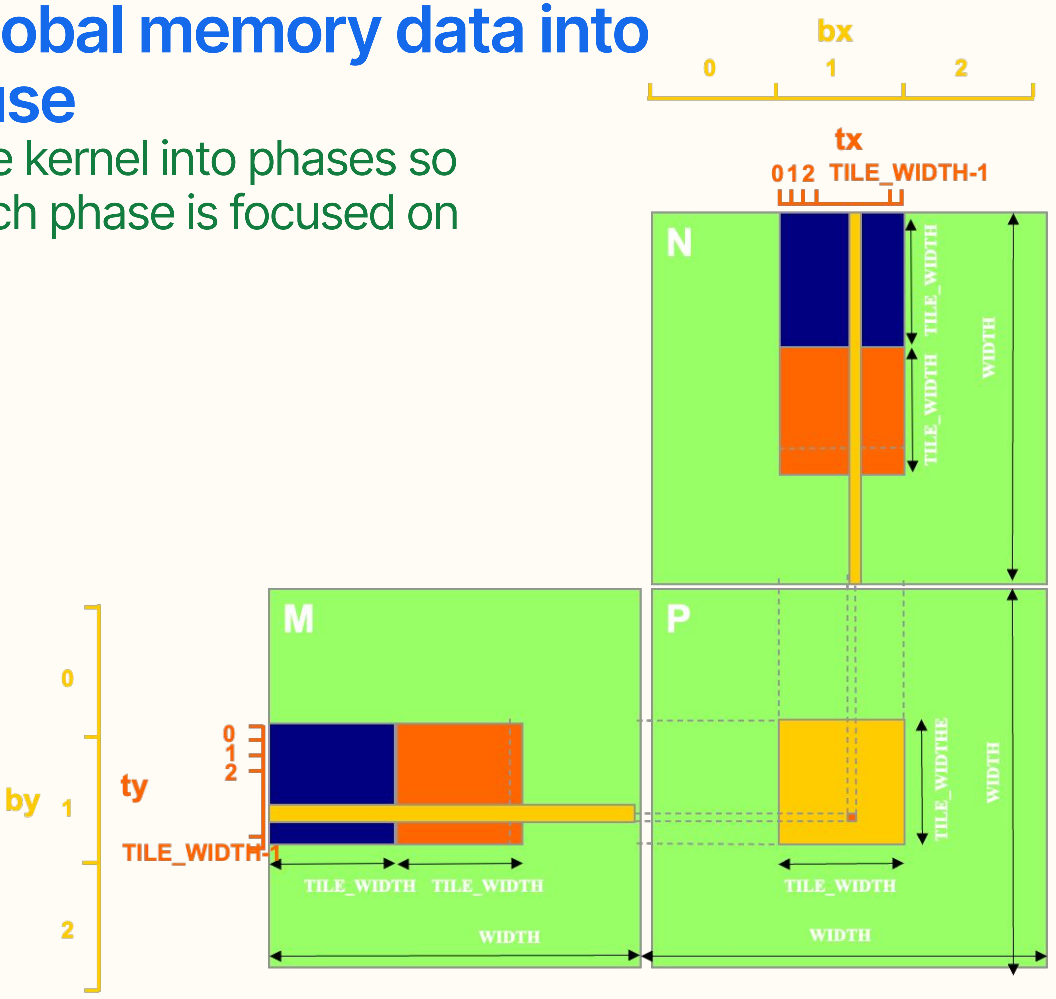
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{

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

Tiled Multiply: Place global memory data into Shared Memory for reuse

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

bx = blockId.x
tx = threadId.x



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

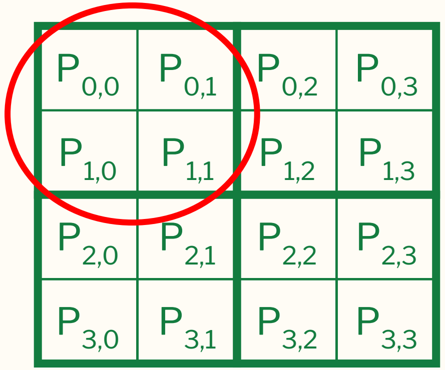
$$P_{0,1} = M_{0,0}*N_{0,1} + M_{0,1}*N_{1,1} + M_{0,2}*N_{2,1} + M_{0,3}*N_{3,1}$$

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

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

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

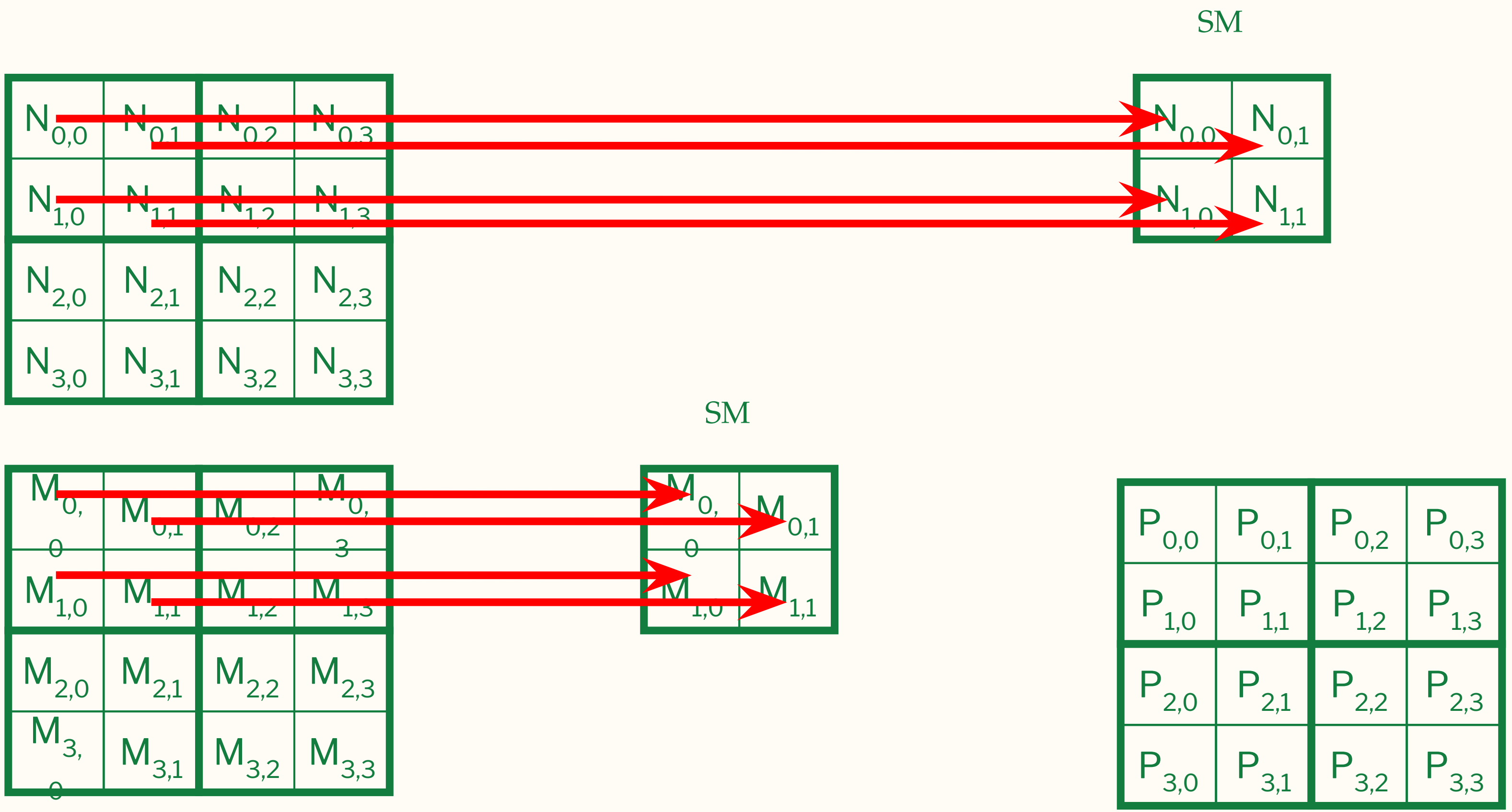
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



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

Work for Block (0,0)

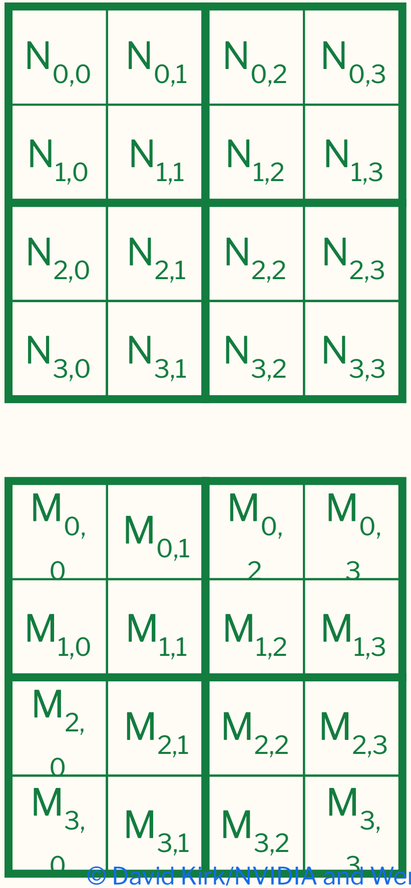
Read data into SM

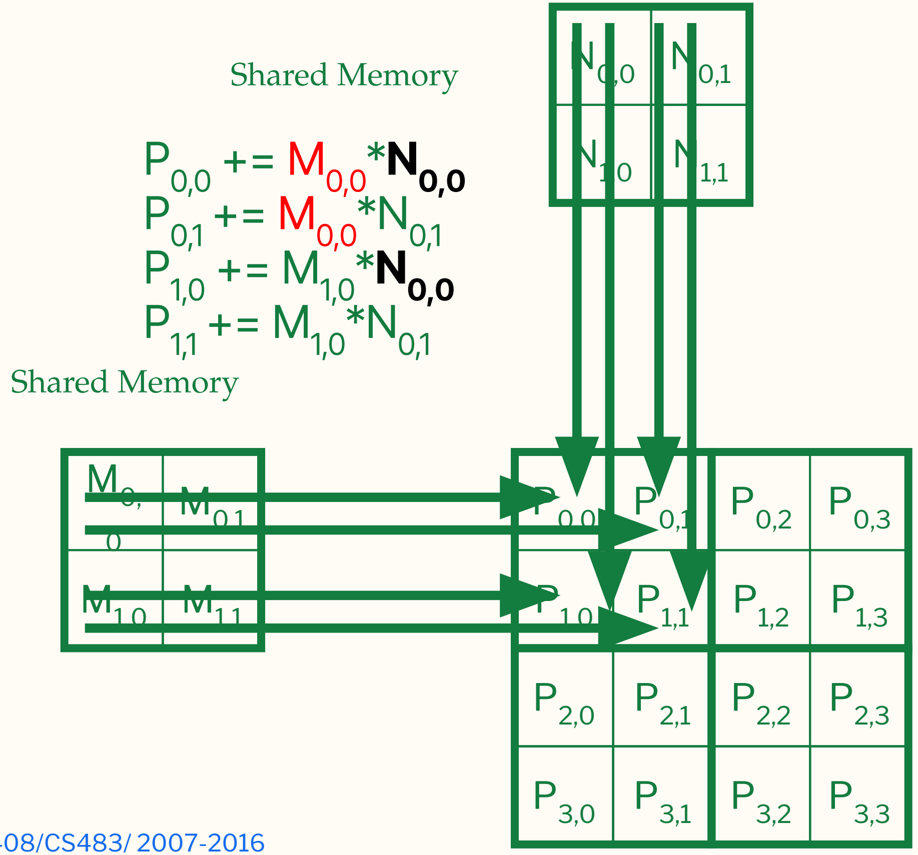


Work for Block (0,0)

Threads use shared memory data in step 0.

Use data in SM

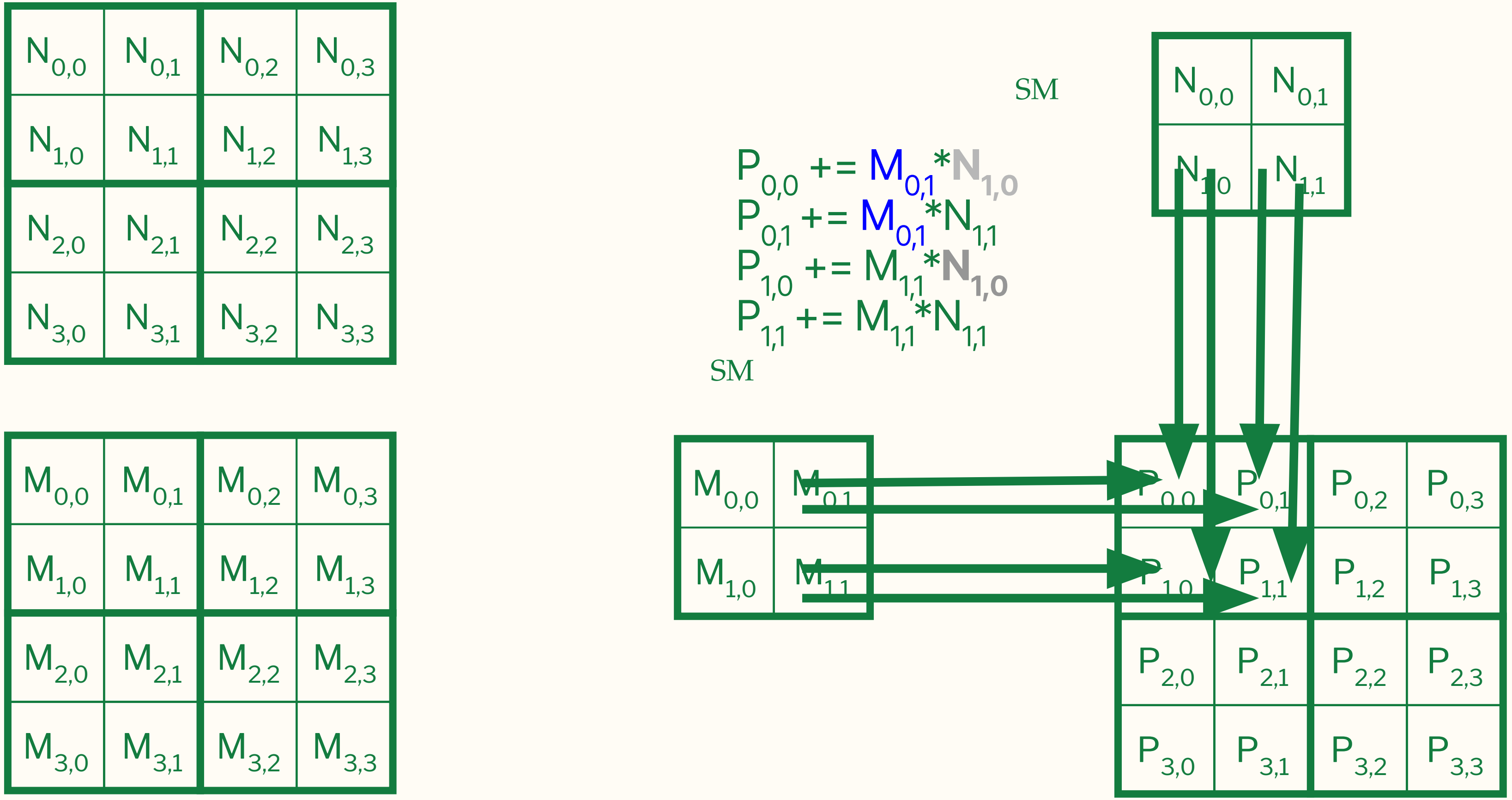

$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$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


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

$P_{0,0} += M_{0,0} * \mathbf{N_{0,0}}$
$P_{0,1} += M_{0,0} * N_{0,1}$
$P_{1,0} += M_{1,0} * \mathbf{N_{0,0}}$
$P_{1,1} += M_{1,0} * N_{0,1}$

Work for Block (0,0)

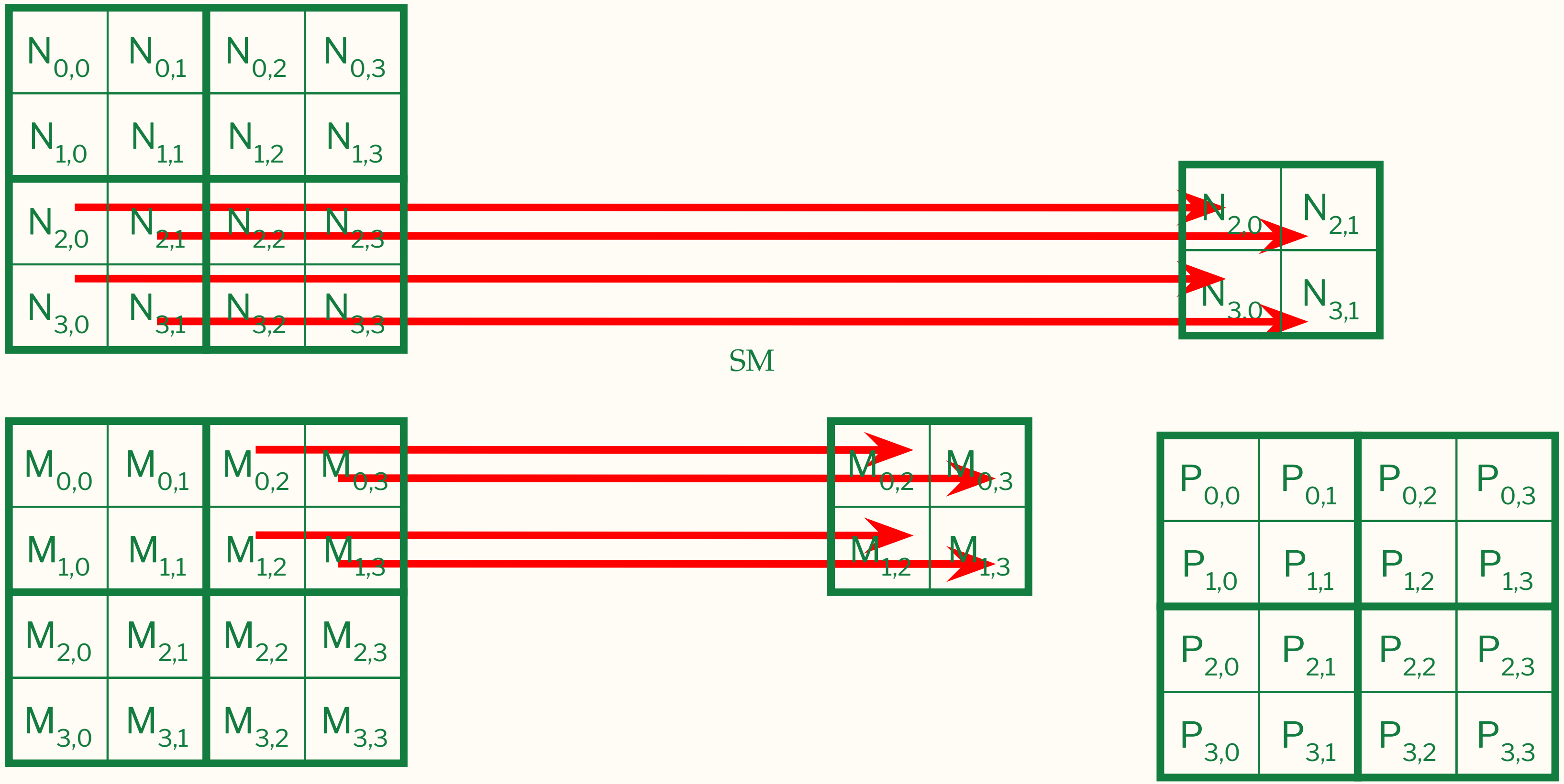
Threads use shared memory data in step 1.


Use data in SM
SM
SM
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

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

Work for Block (0,0)

Read more data into SM

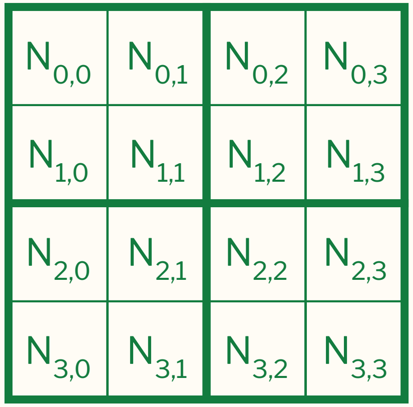


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
SM
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$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

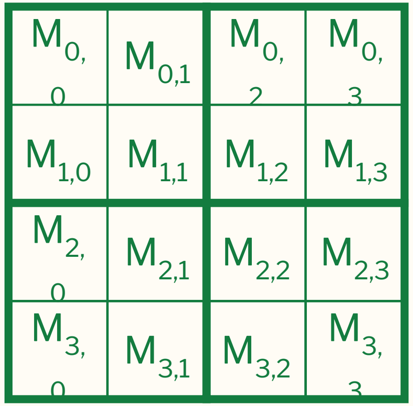
Work for Block (0,0)

Threads use shared memory data in step 2.

Use data in SM

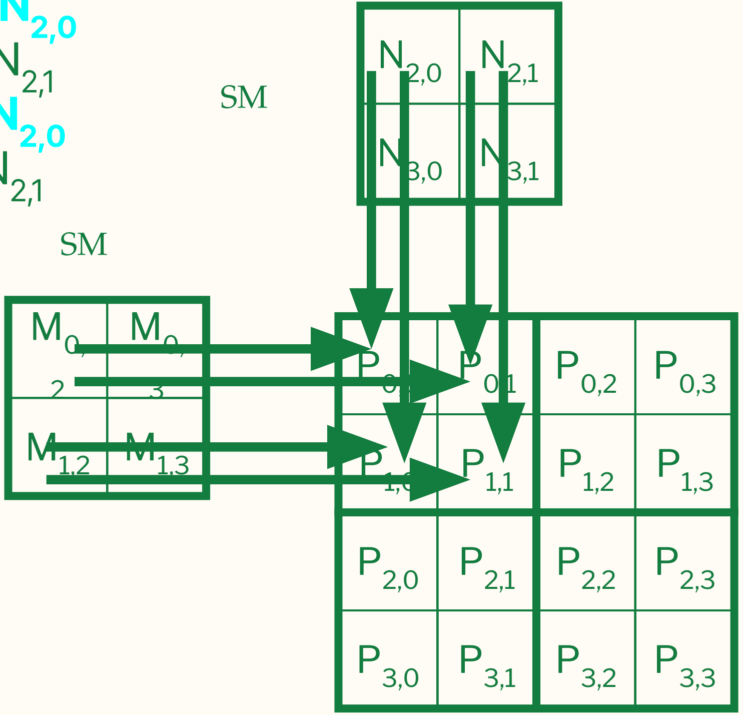


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$



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

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

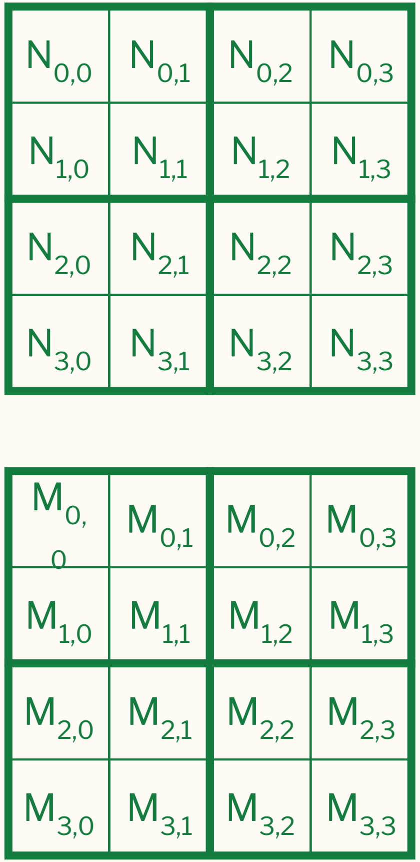


SM
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
SM
$M_{0,2}$ $M_{0,3}$
$M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Work for Block (0,0)

Threads use shared memory data in step 1.

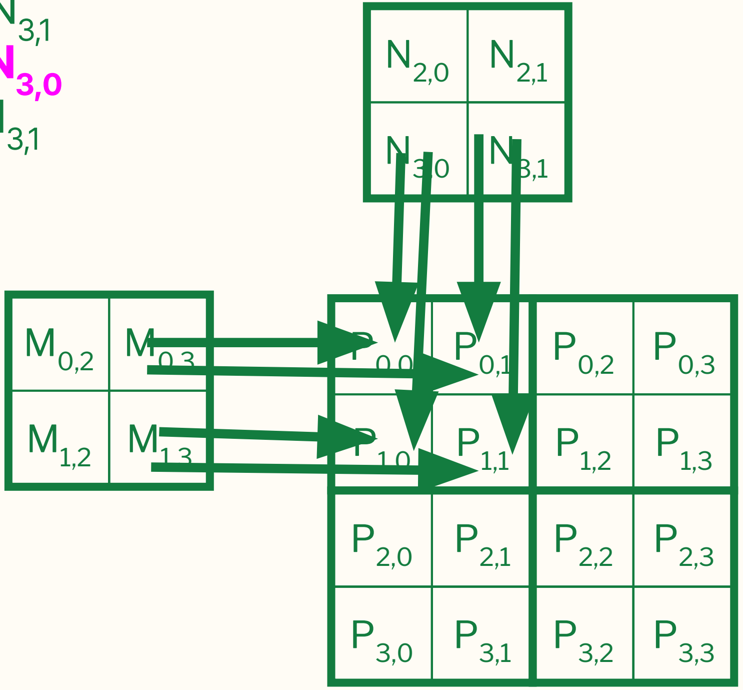
Use data in SM



SM

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

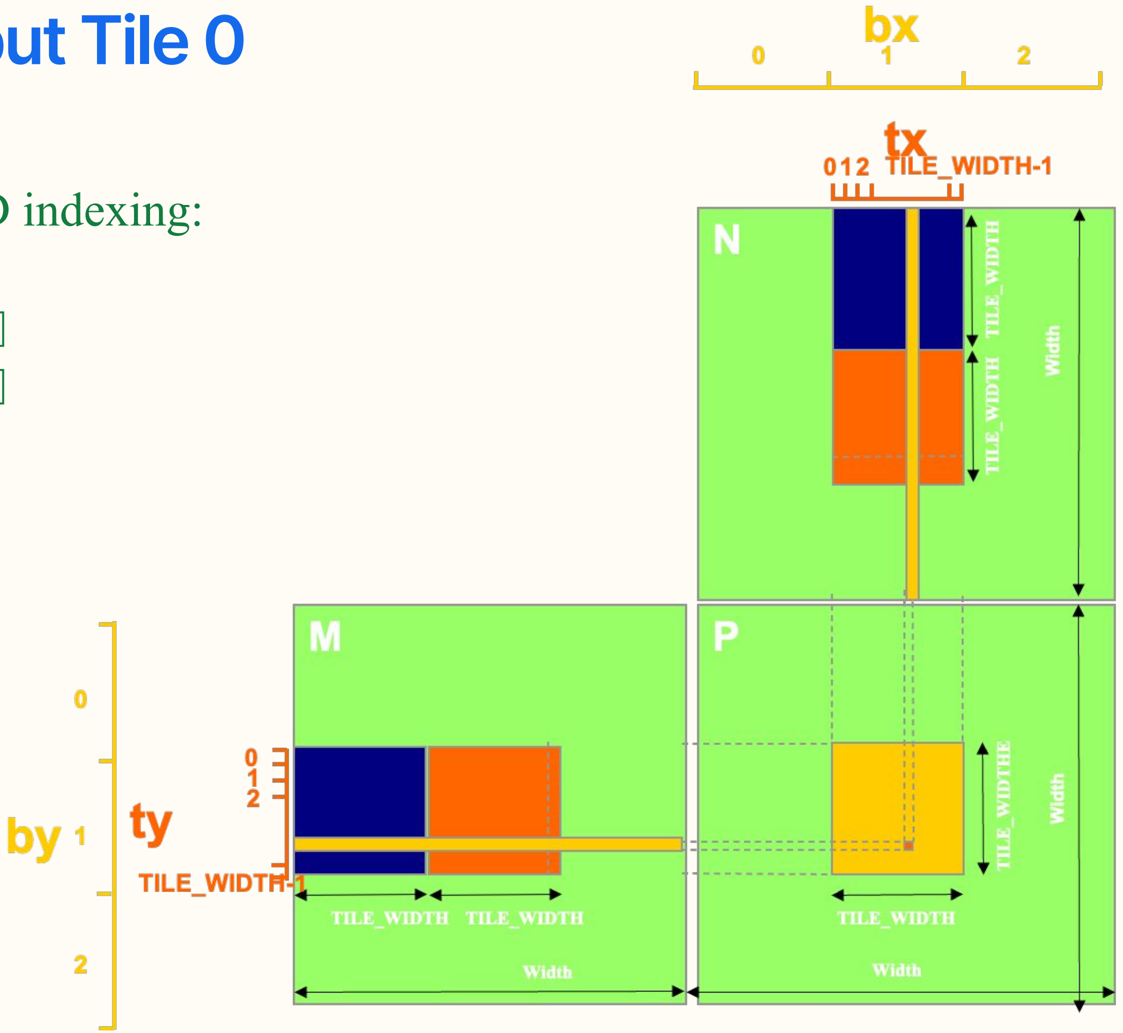
SM



Loading an Input Tile 0

Accessing tile 0 2D indexing:

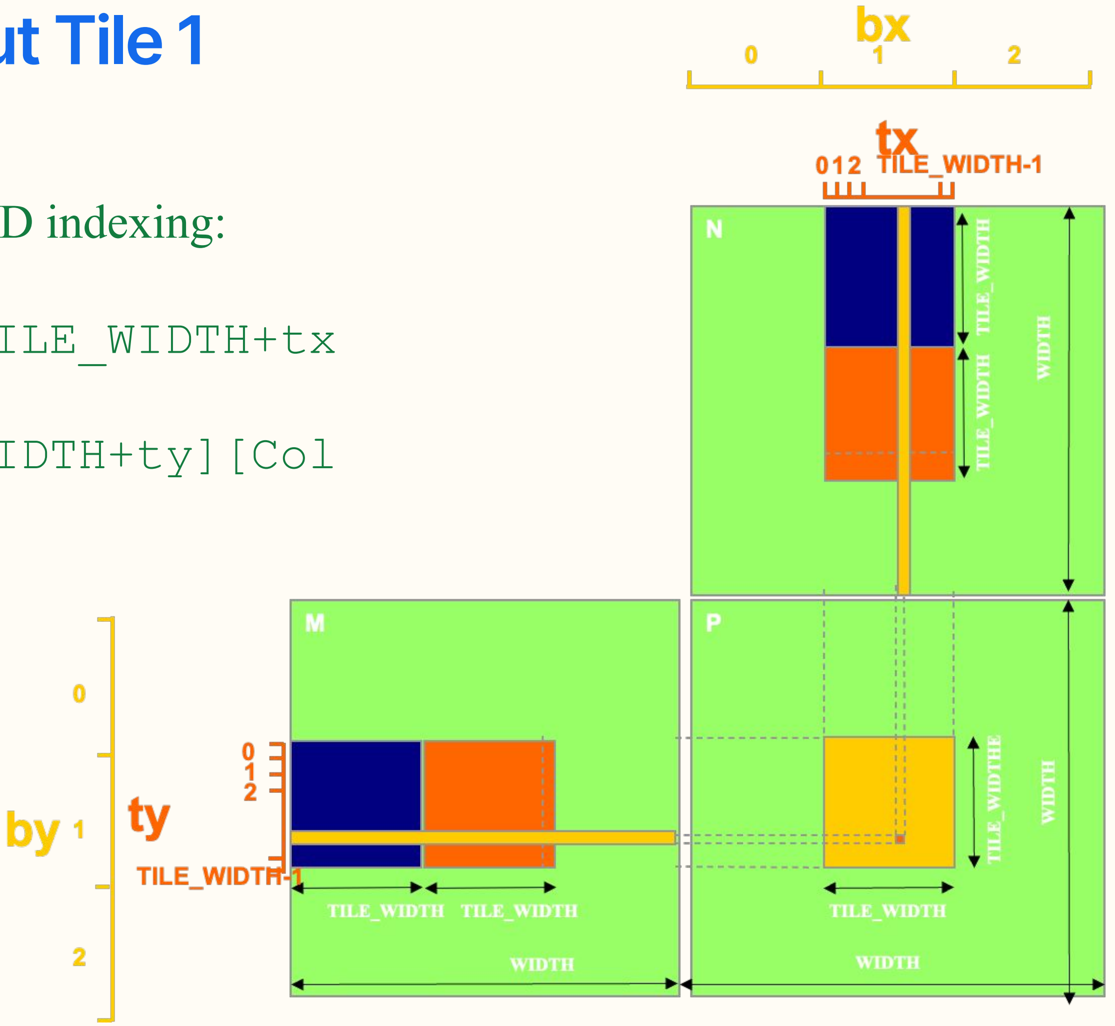
```
M[Row][tx]
N[ty][Col]
```



Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

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

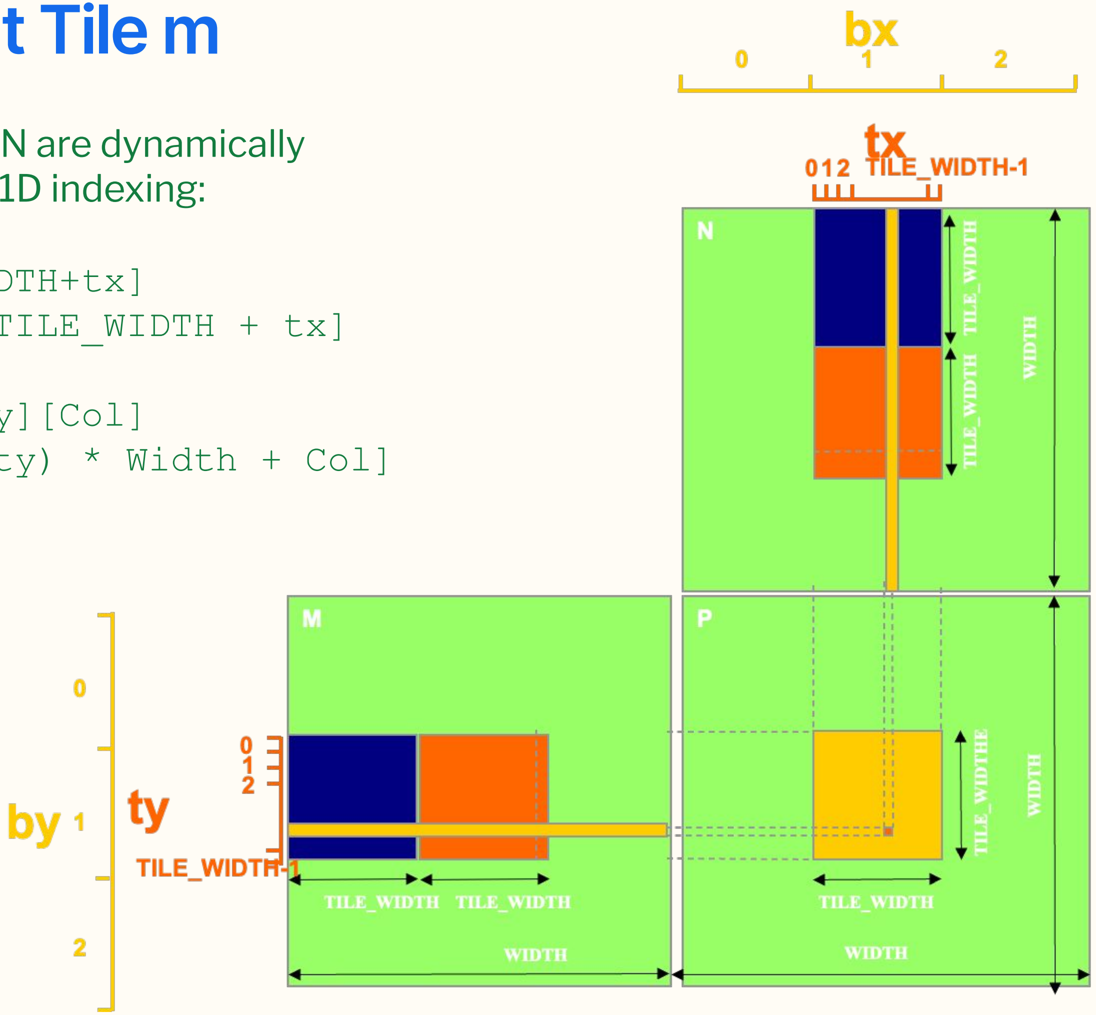


Loading an Input Tile m

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]
```



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

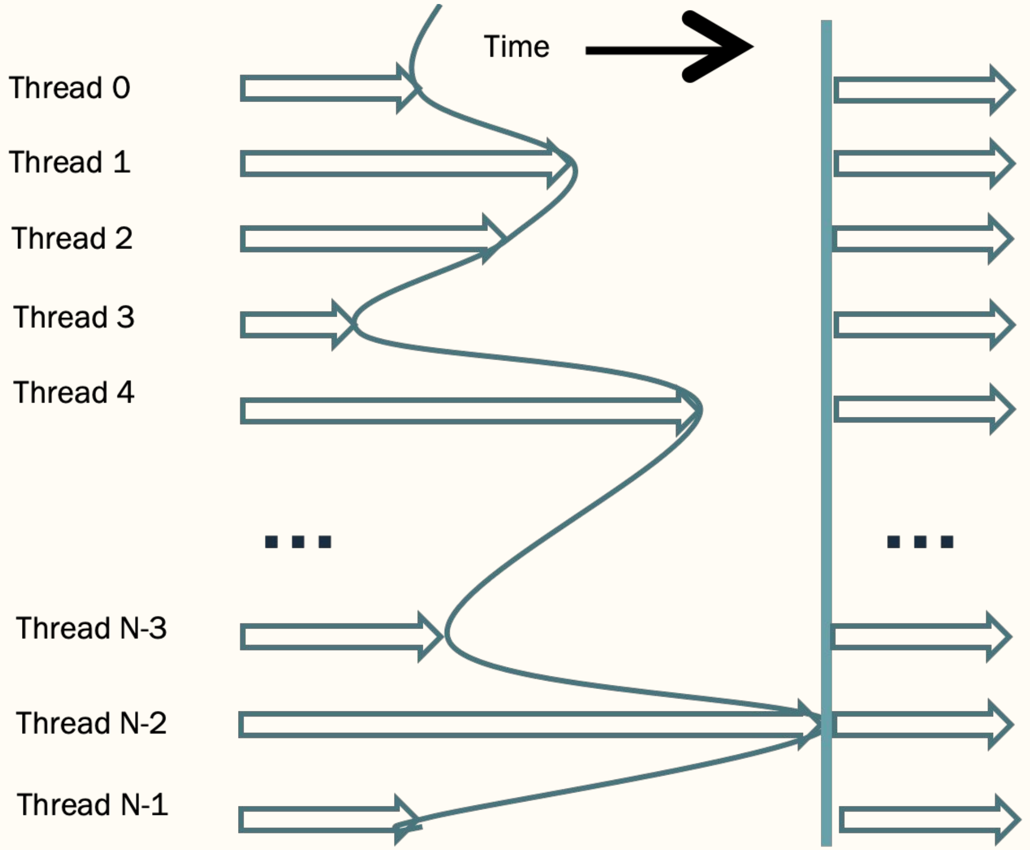


Figure 4.11 An example execution timing of barrier synchronization.

Tiled Matrix Multiplication Kernel

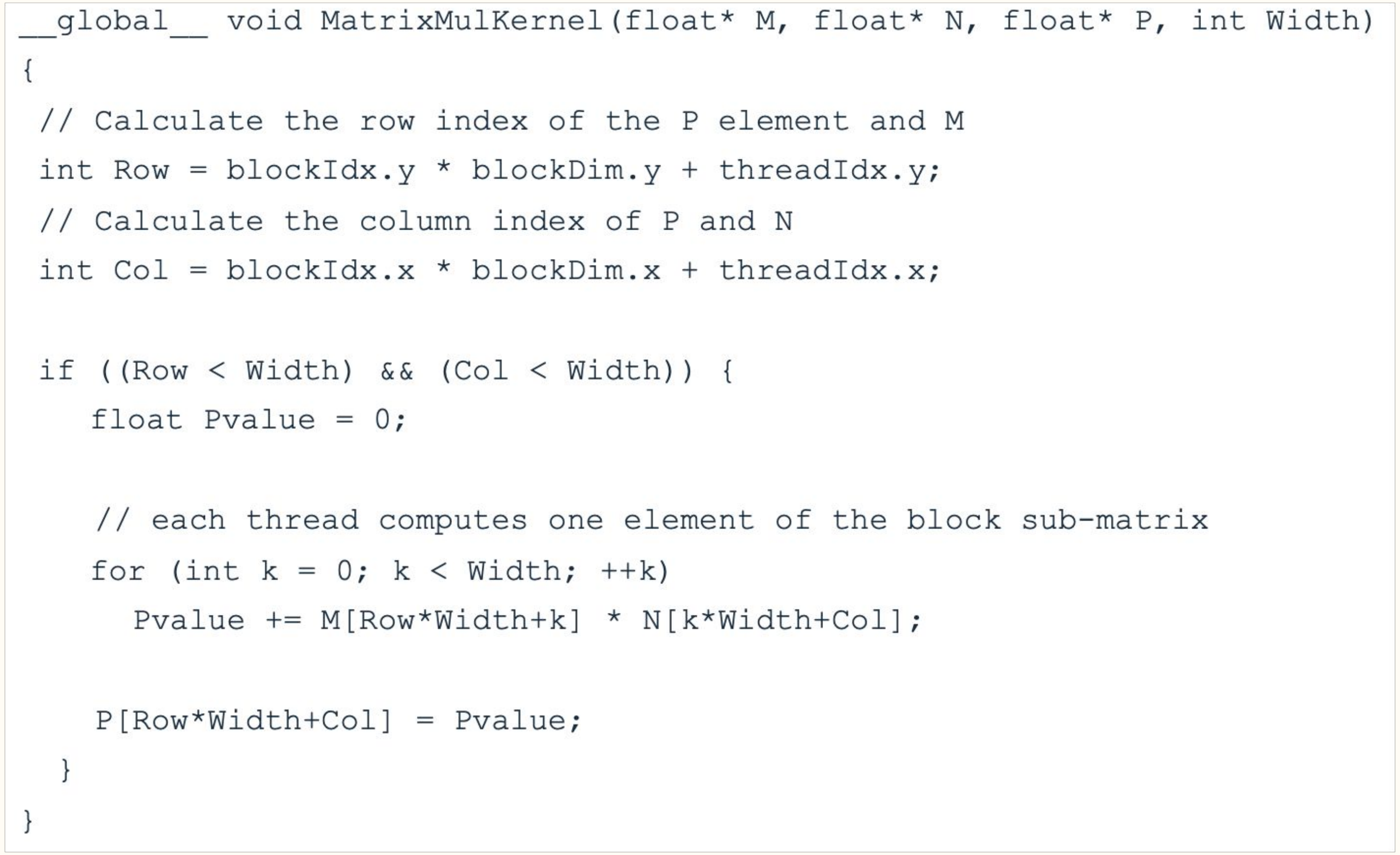
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

  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;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0;

  // Loop over the M and N tiles required to compute the P element
  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 Base 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 has 64KB shared memory (48KB max per block)
    - Shared memory size is implementation dependent!
    - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
        - Shared memory can potentially support up to 32 thread blocks actively executing, but only 8 blocks allowed
            - In reality, if only 1536 threads allowed on SM, only 1536/256 = 6 blocks allowed
        - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
- Using 16×16 tiling, we reduce the accesses to the global memory by a factor of 16
    - The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!

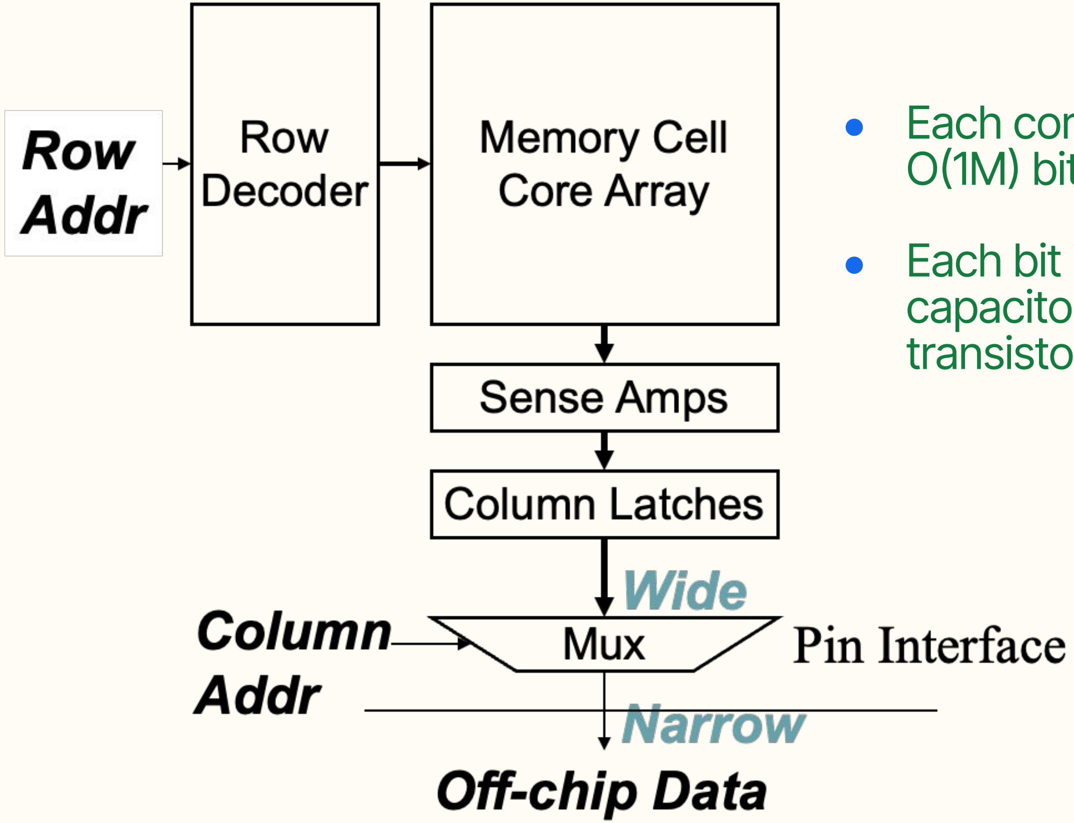
DRAM CHARACTERISTICS

Global Memory (DRAM) Bandwidth

Ideal

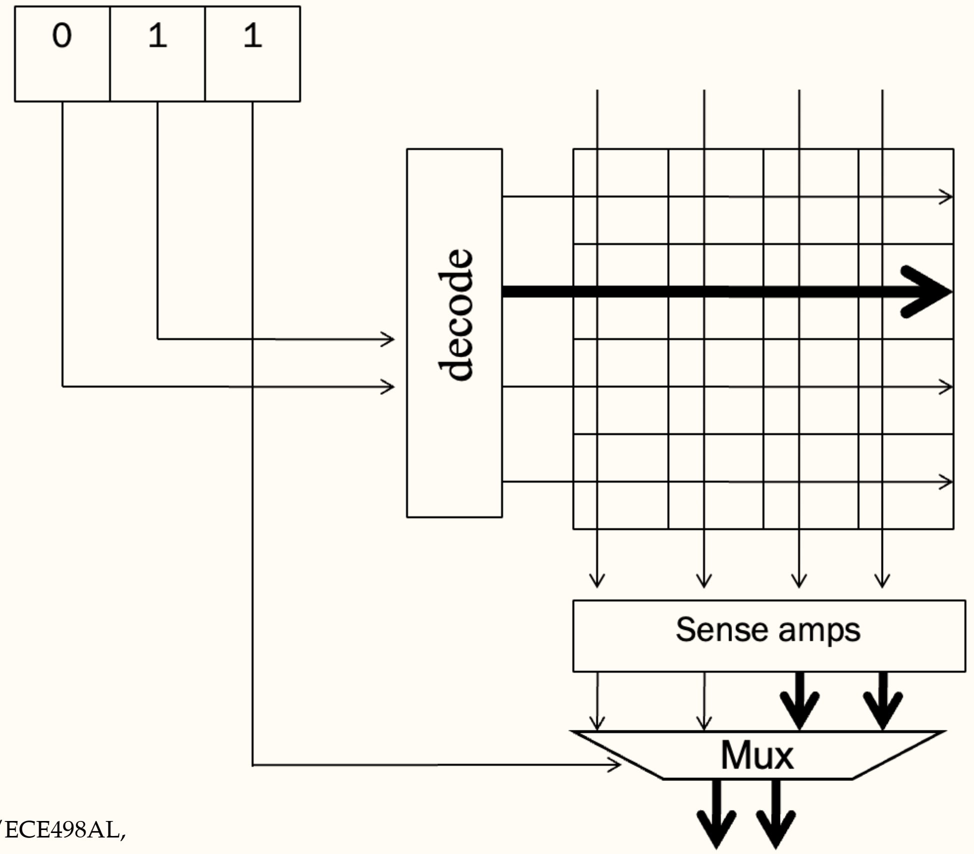
Reality

DRAM Bank Organization



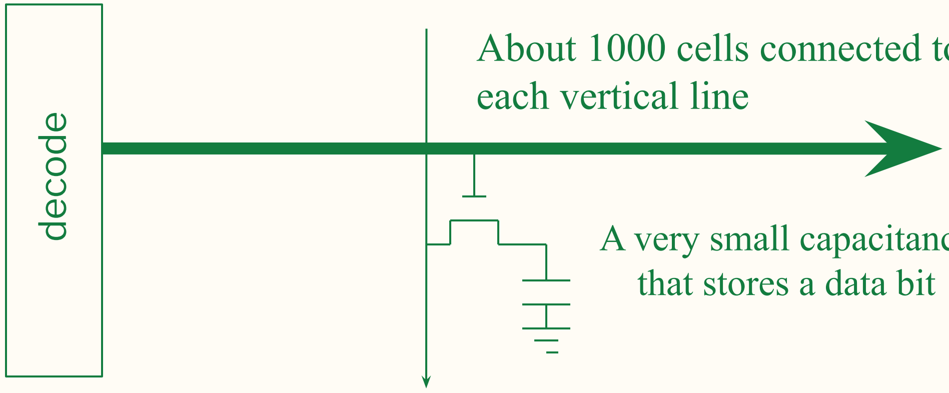
- Each core array has about O(1M) bits
- Each bit is stored in a tiny capacitor, made of one transistor

A very small DRAM Bank



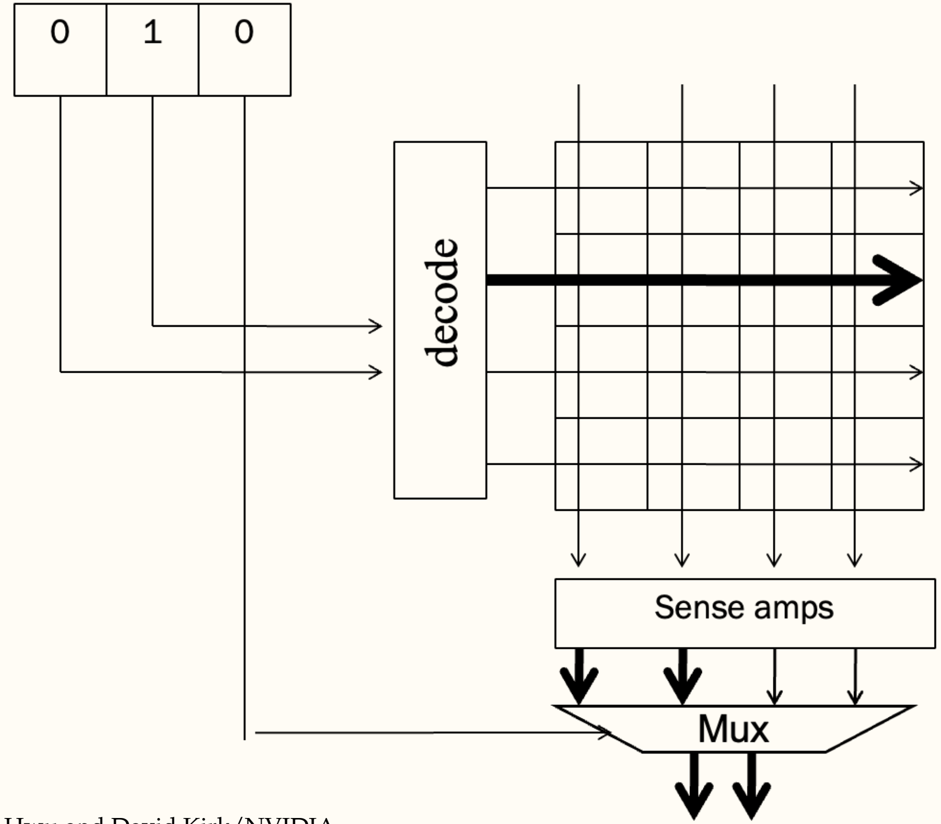
DRAM core arrays are slow.

- Reading from a cell in the core array is a very slow process

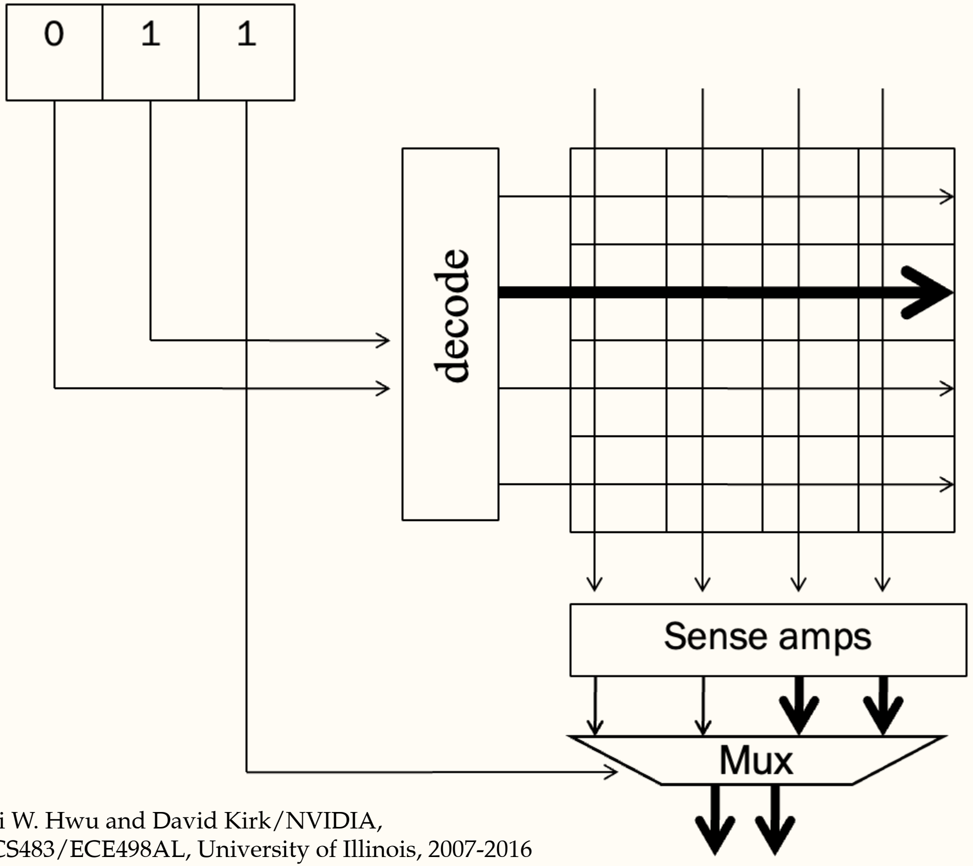

decode
About 1000 cells connected to each vertical line
A very small capacitance that stores a data bit

To sense amps

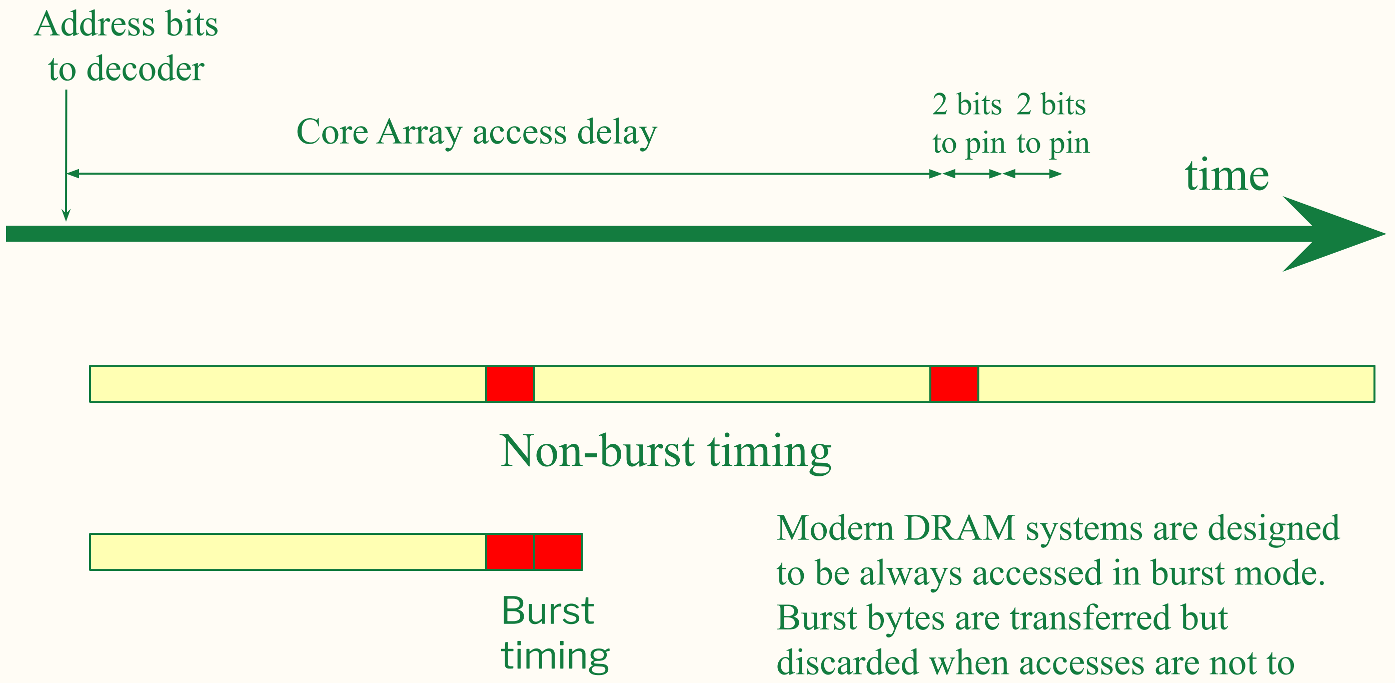
DRAM Bursting (burst size = 4 bits)



DRAM Bursting (cont.)
second part of the burst

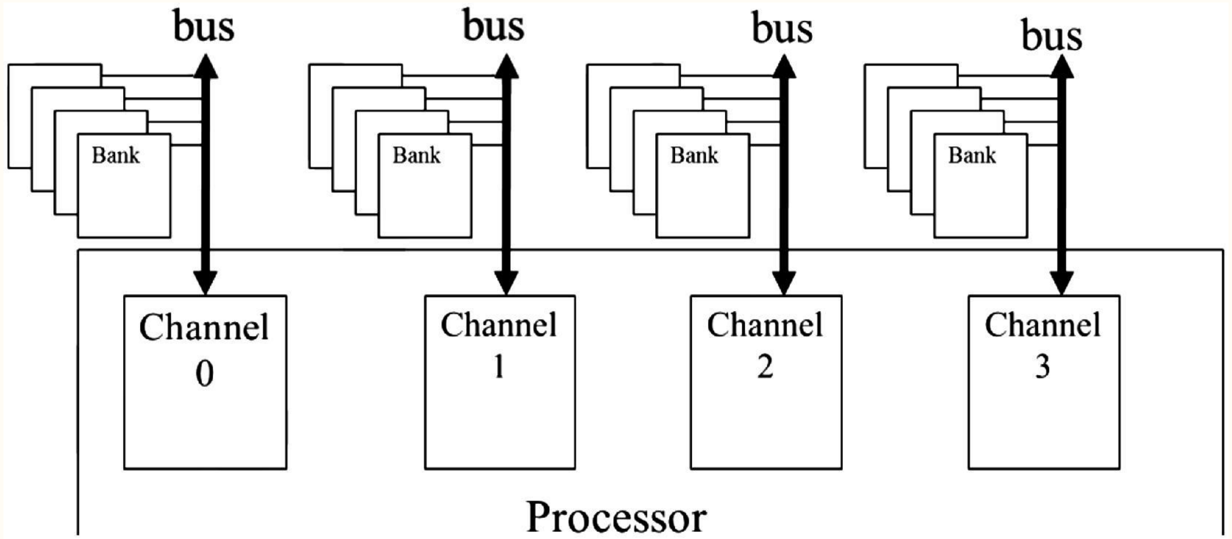


DRAM Bursting for our Example Bank



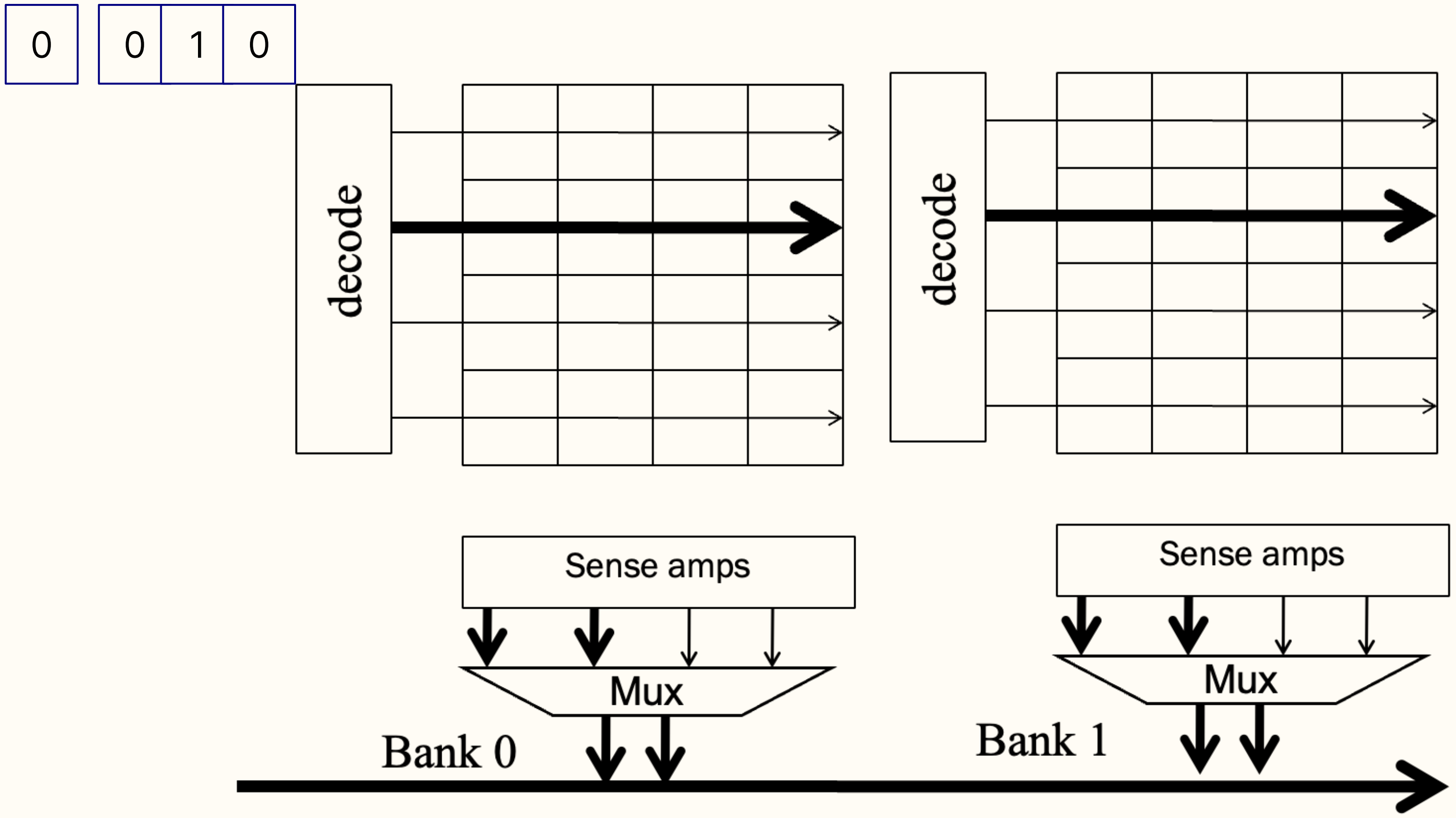
Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

Multiple DRAM Banks

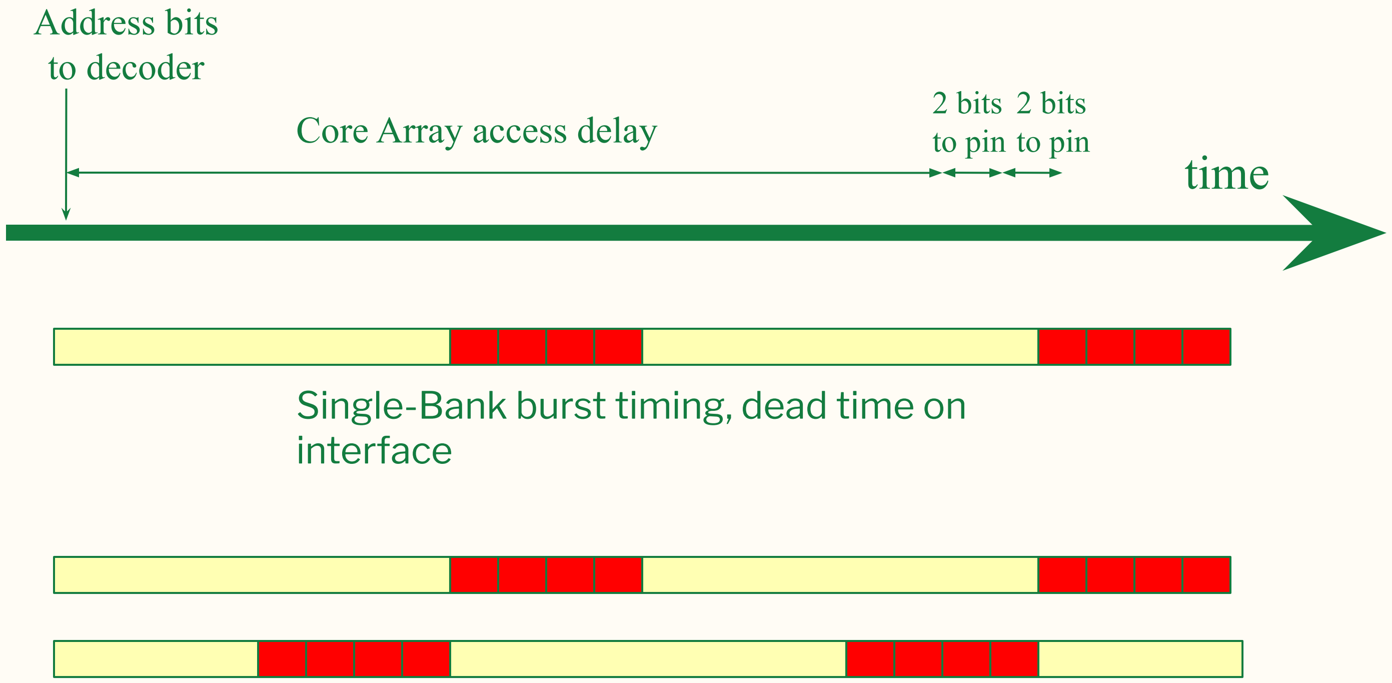


Channel: memory controller w/ bus that connects set of DRAM banks to processor

Multiple DRAM Banks



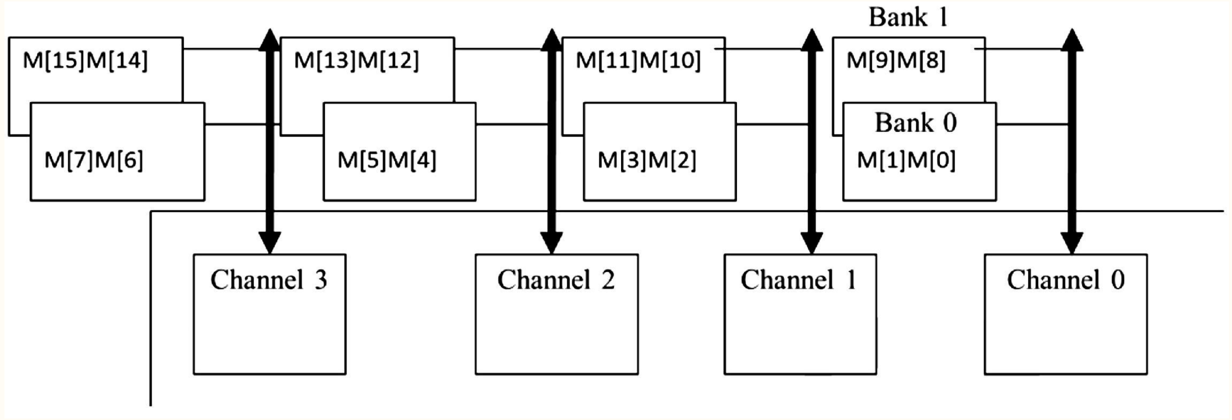
DRAM Bursting for the 8×2 Bank



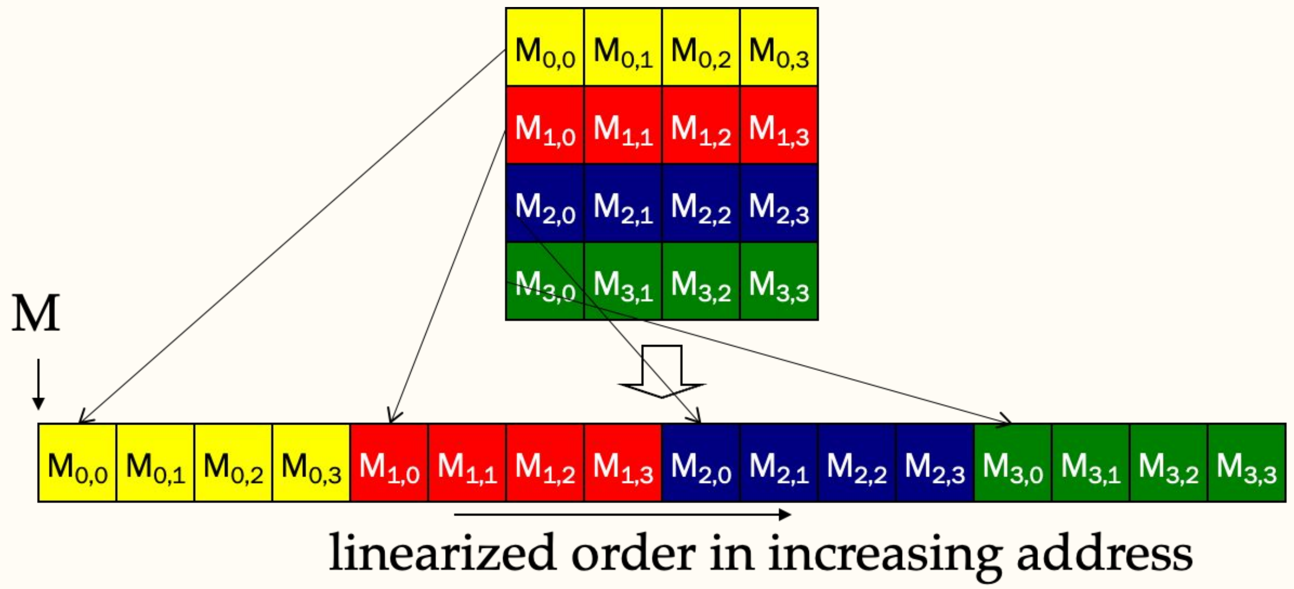
Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

Bank Interleaving



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



$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
linearized order in increasing address

A Simple Matrix Multiplication Kernel (review)

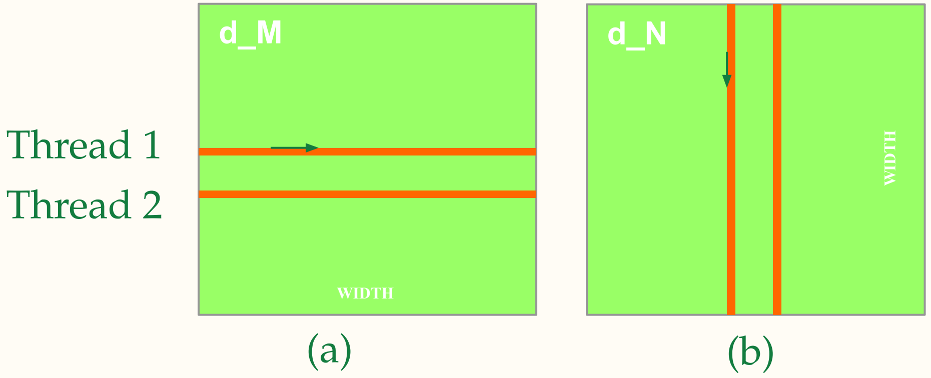
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 // Calculate the row index of the P element and M
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 // Calculate the column index of P and N
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 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

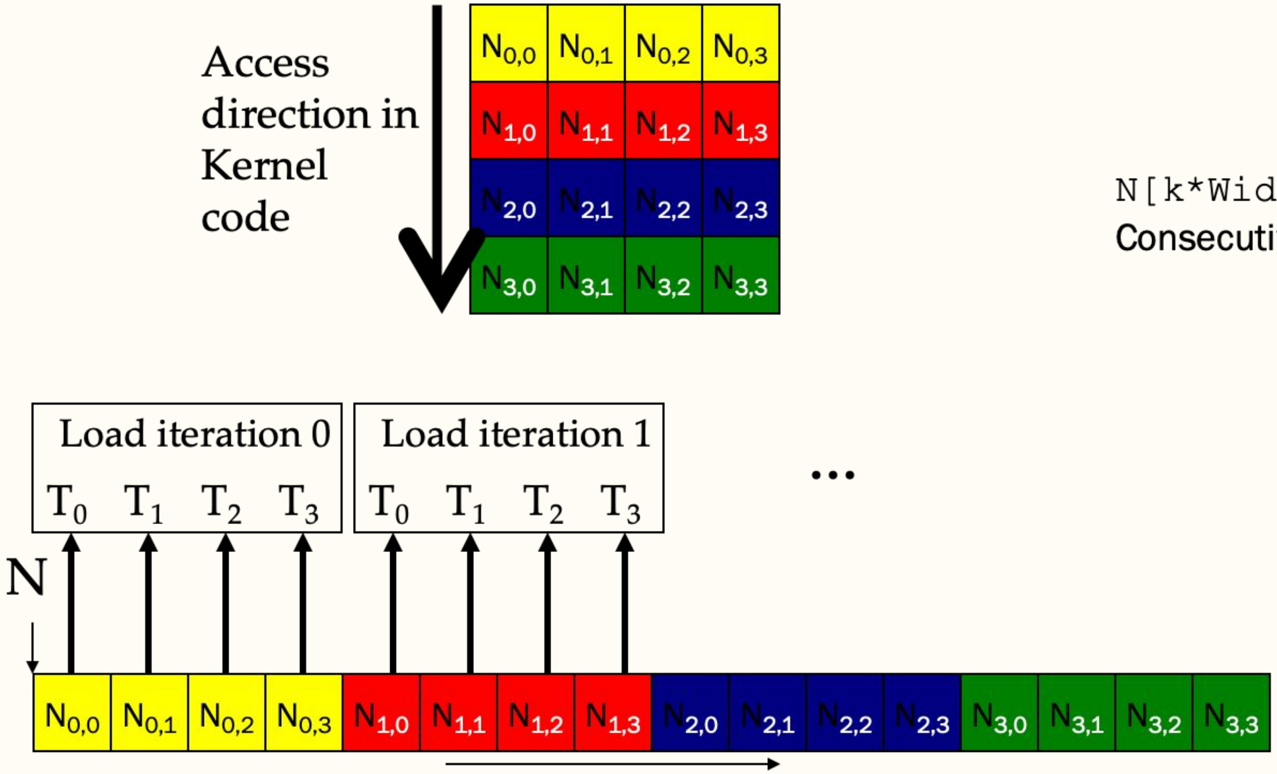


`M[Row*Width+k]`

`N[k*Width+Col]`

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

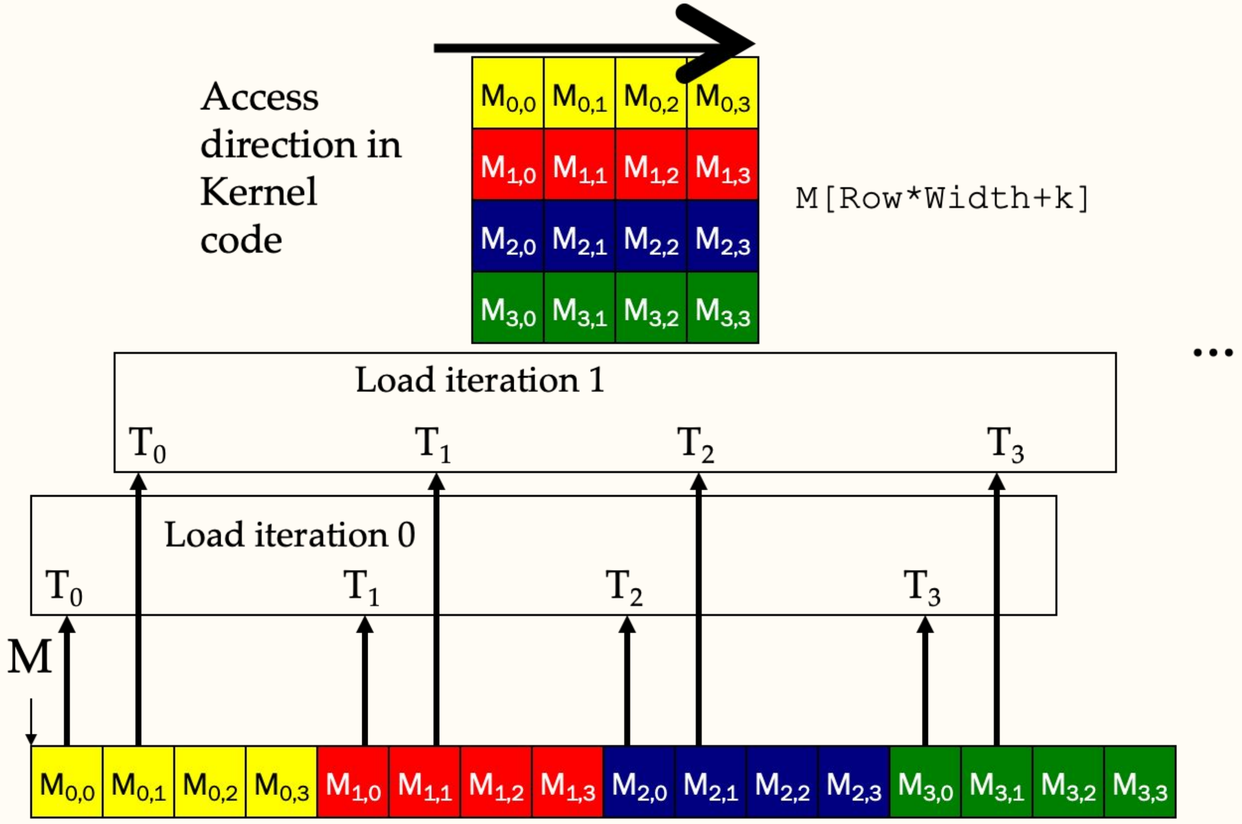
N accesses are coalesced.


Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

`N[k*Width+Col]`

Consecutive columns

M accesses are not coalesced.



Access direction in Kernel code
M[Row*Width+k]
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
...
Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Coalescing

- On load request, all accesses from a warp reduced to the smallest number of DRAM accesses
    - Perfect coalescing (all consecutive accesses) reduce number of DRAM accesses the most
    - No coalescing results in 32 different DRAM accesses