ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 6: Generalized Tiling // DRAM Bandwidth

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
    // The code assumes that the Width is a multiple of TILE_WIDTH!
    for (int q = 0; q < Width/TILE_WIDTH; ++q) {
       // Collaborative loading of M and N tiles into shared memory
       subTileM[ty][tx] = M[Row*Width + (q*TILE_WIDTH+tx)];
       subTileN[ty][tx] = N[(q*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;
}
```

Handling Matrix of Arbitrary Size

The tiled matrix multiplication kernel in Lecture 5 can only handle matrices whose dimensions are multiples of the tile dimensions, and are square.

However, a real application needs to handle arbitrary sized matrices.

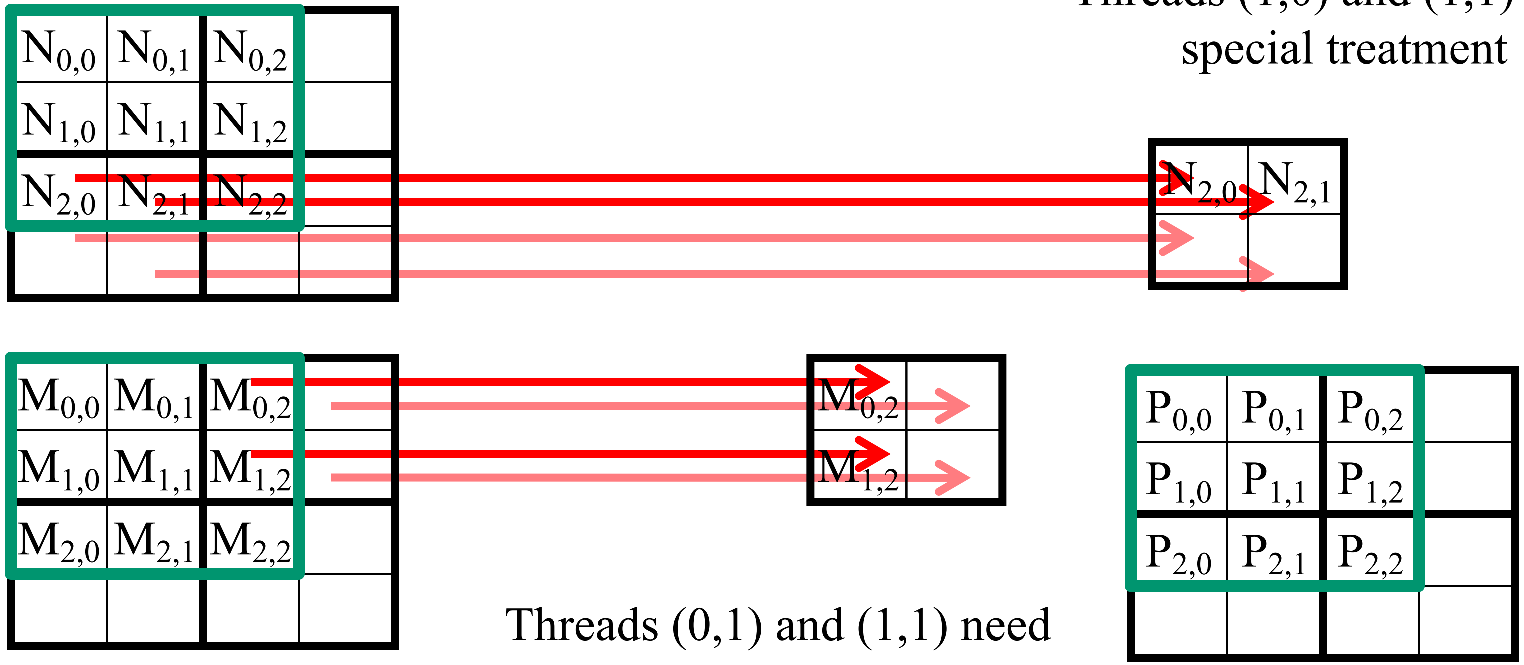
- We could pad (add elements to) the rows and columns into multiples of block size, but will have significant space and data transfer time overhead.
- We could add explicit checks in the code to handle boundaries

2x2 Tile on a 3x3 Multiply

`TILE_WIDTH = 2, width = 3`

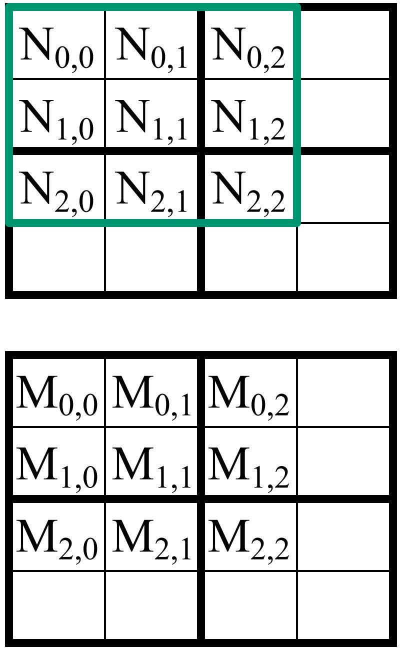
Load of 2nd tile of Block (0,0)

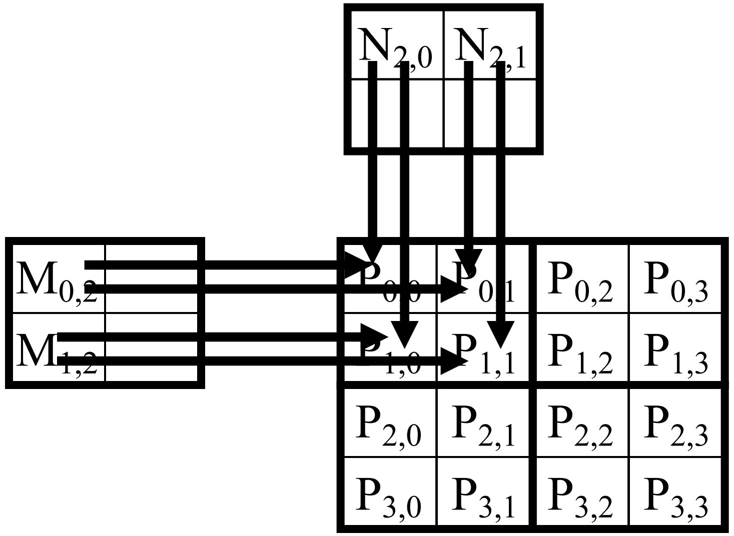
Threads (1,0) and (1,1) need special treatment



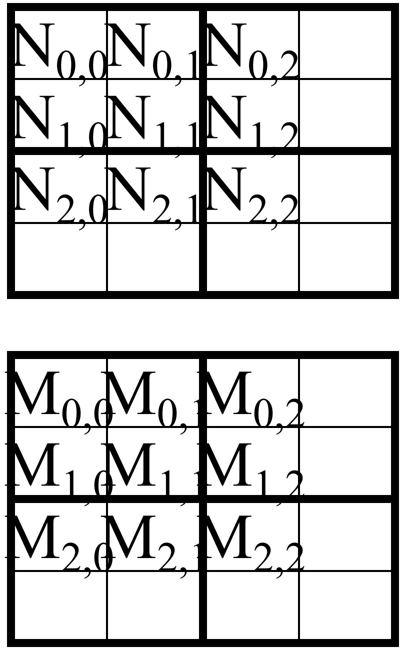
Threads (0,1) and (1,1) need special treatment

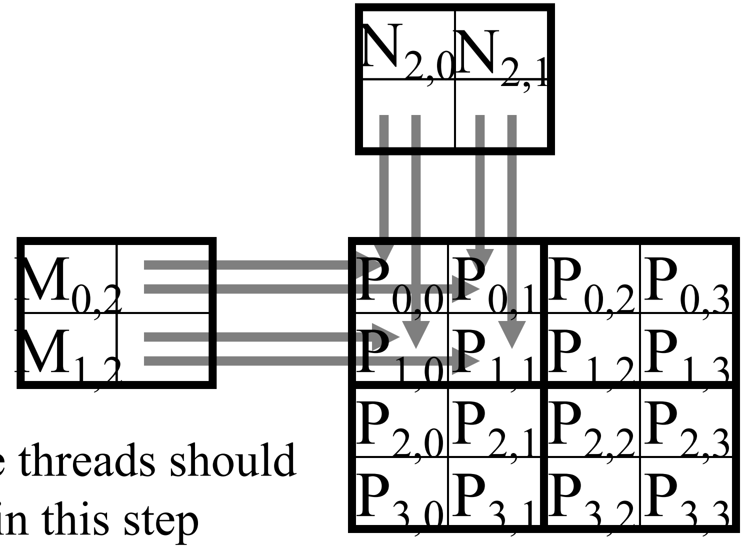
2x2 Tile on a 3x3 Multiply Block (0,0), 2nd tile





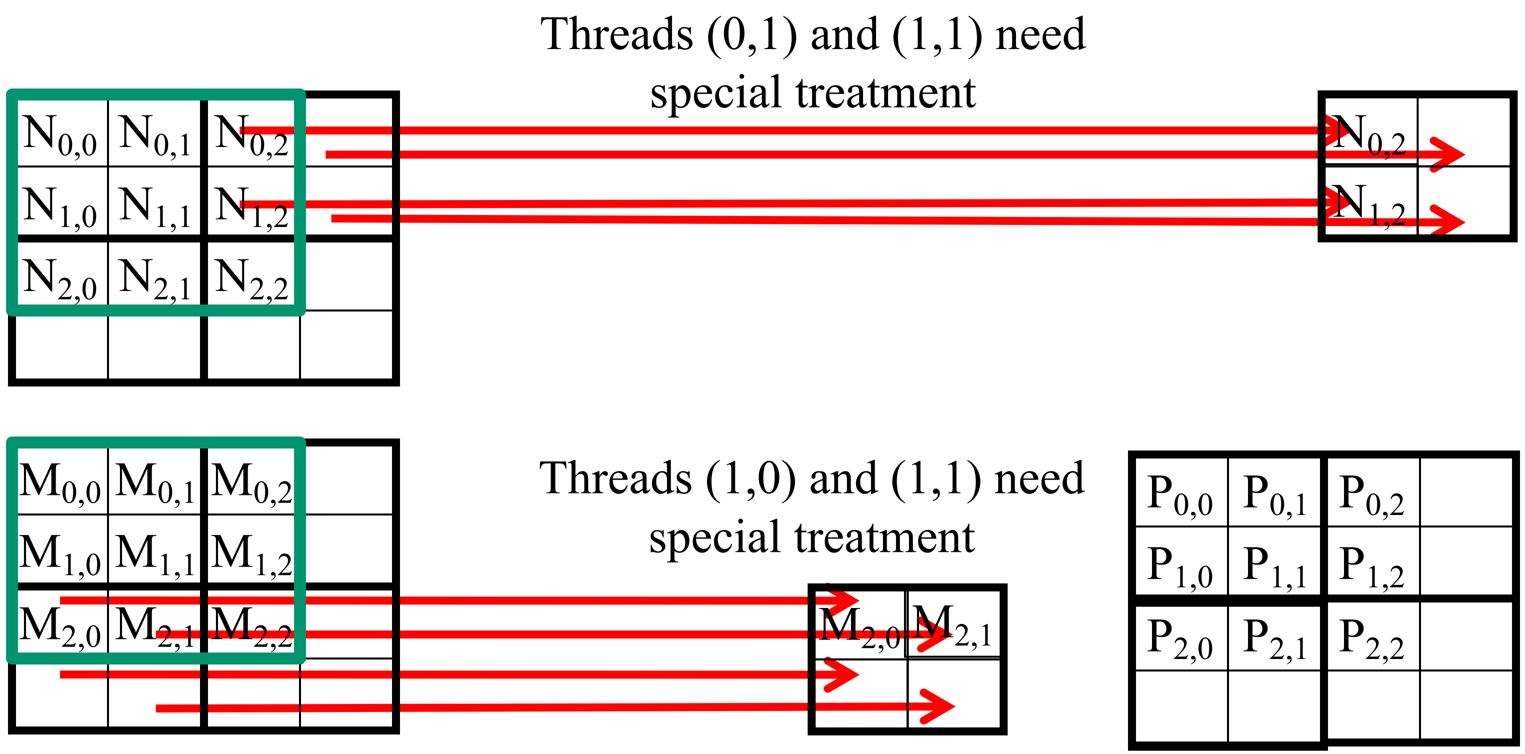
2x2 Tile on a 3x3 Multiply
Block (0,0), 2^{nd} tile



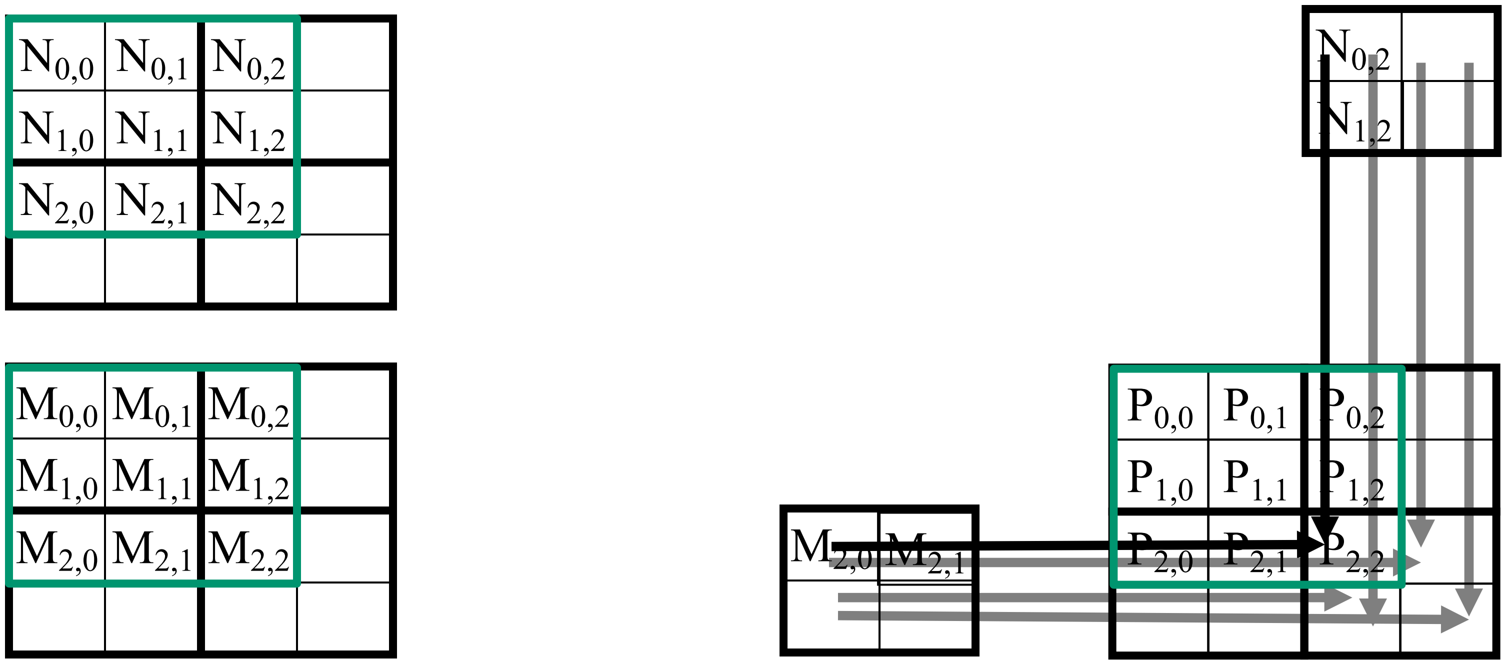


None of the threads should take effect in this step

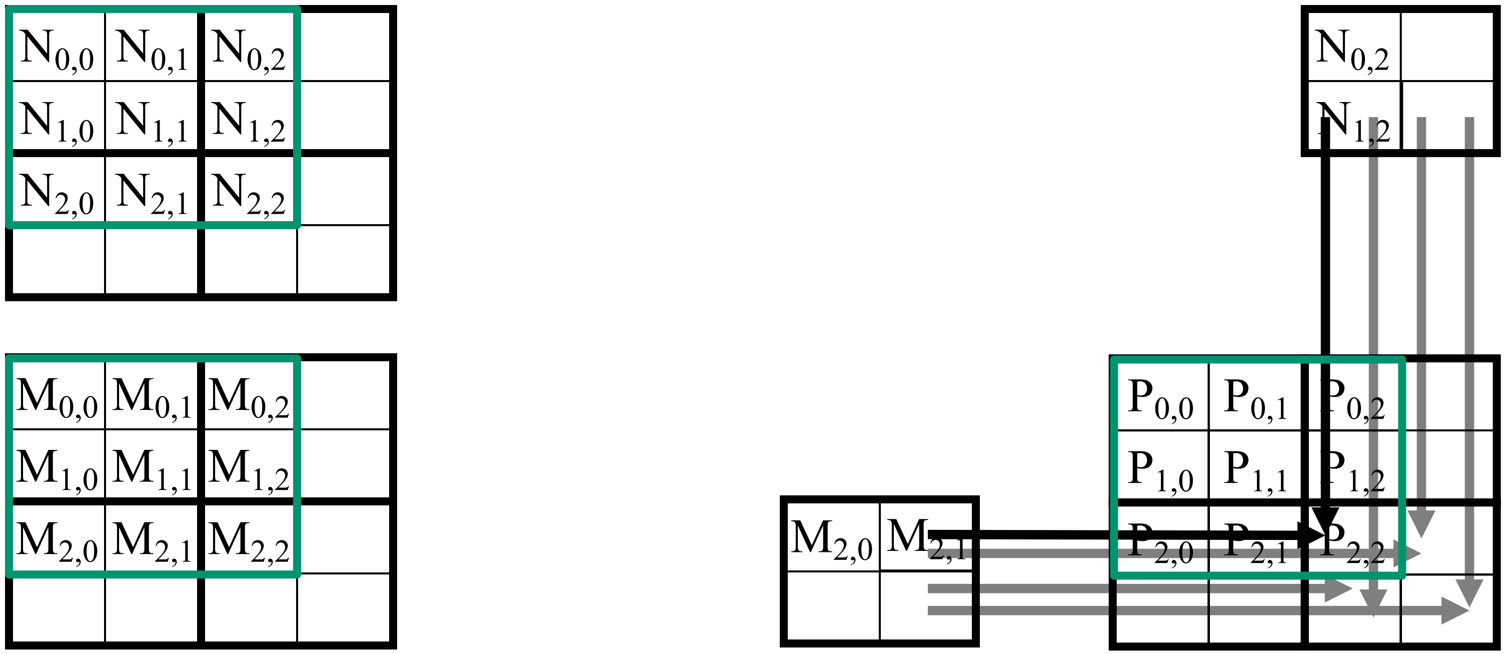
2x2 Tile on a 3x3 Multiply
Load 1st Tile of Block (1,1)



1st Tile for Block (1,1)



1st Tile for Block (1,1)



Major Cases in 2x2 Example

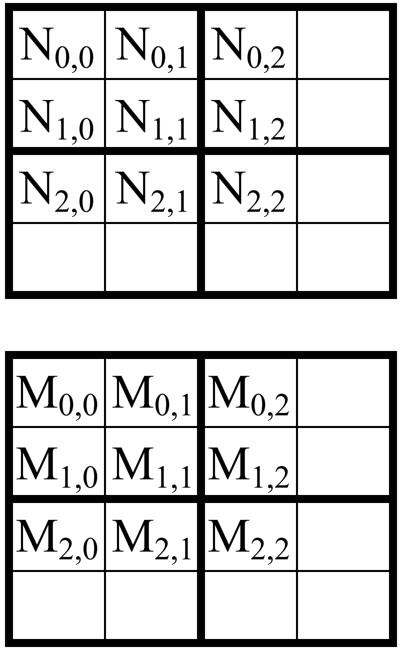
- Threads that calculate valid P elements, but use invalid input
  - 1st Tile of Block(0,0), 2nd step, all threads

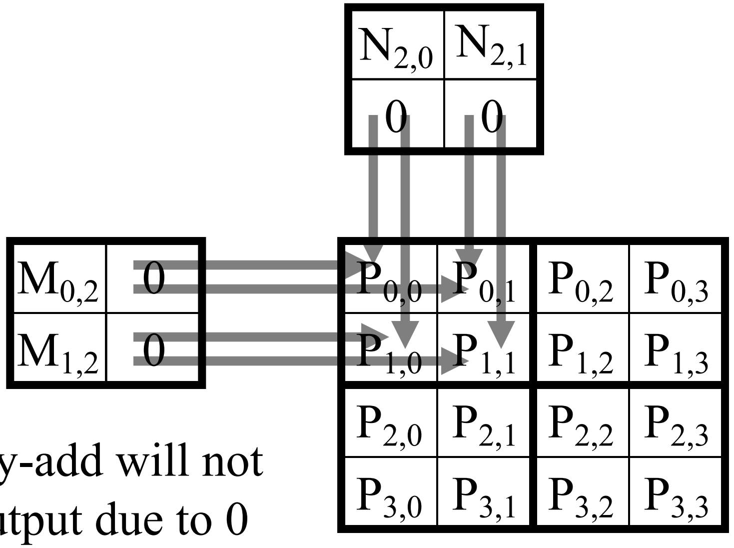
- Threads that calculate invalid P elements
  - Block(1,1), Thread(1,0), non-existent row
  - Block(1,1), Thread(0,1), non-existing column
  - Block(1,1), Thread(1,1), non-existing row/column

A “Simple” Solution

- Invalid input element, “load” a 0
  - Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- Invalid output element, don’t update global memory
  - Can still perform pvalue calculation (partial dot product), but doesn’t write to the global memory at the end of the kernel

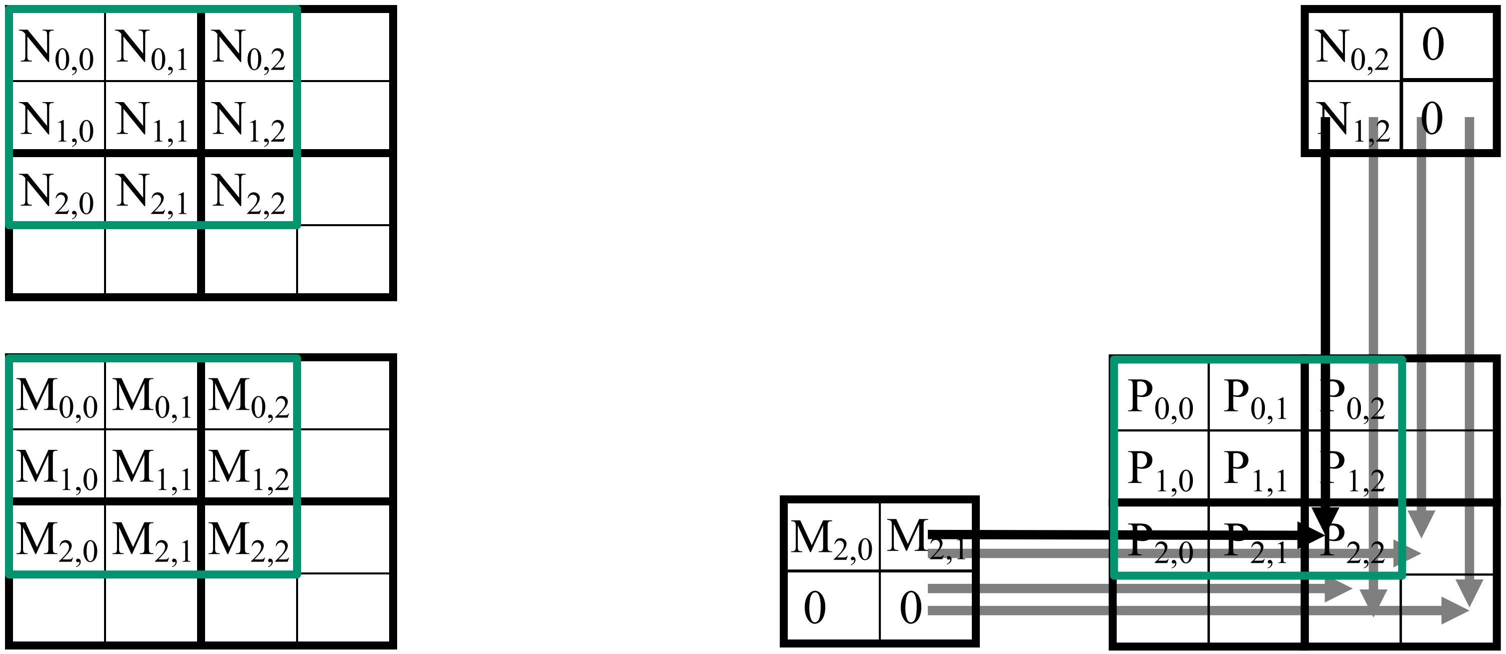
2nd Tile for Block (0,0)





The multiply-add will not affect the output due to 0

1^{st} Tile for Block (1,1)



Tiled Matrix Multiplication Kernel

```
    // Loop over the M and N tiles required to compute the P element

    for (int q = 0; q < (ceil((float)Width/TILE_WIDTH)); ++q) {
        // Collaborative loading of M and N tiles into shared memory
        if (Row < Width && (q*TILE_WIDTH+tx) < Width)
            subTileM[ty][tx] = M[Row*Width + q*TILE_WIDTH+tx];
        else
            subTileM[ty][tx] = 0;

        if (Col < Width && (q*TILE_WIDTH+ty) < Width)
            subTileN[ty][tx] = N[(q*TILE_WIDTH+ty)*Width+Col];
        else
            subTileN[ty][tx] = 0;
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k)
             Pvalue += subTileM[ty][k] * subTileN[k][tx];
        __syncthreads();
    }
    if (Row < Width && Col < Width)
        P[Row*Width+Col] = Pvalue;
}
```

Some Important Points

- For each thread the conditions are different for
  - Loading M element
  - Loading N element
  - Calculation/storing output elements

- The effect of control divergence should be small for large matrices

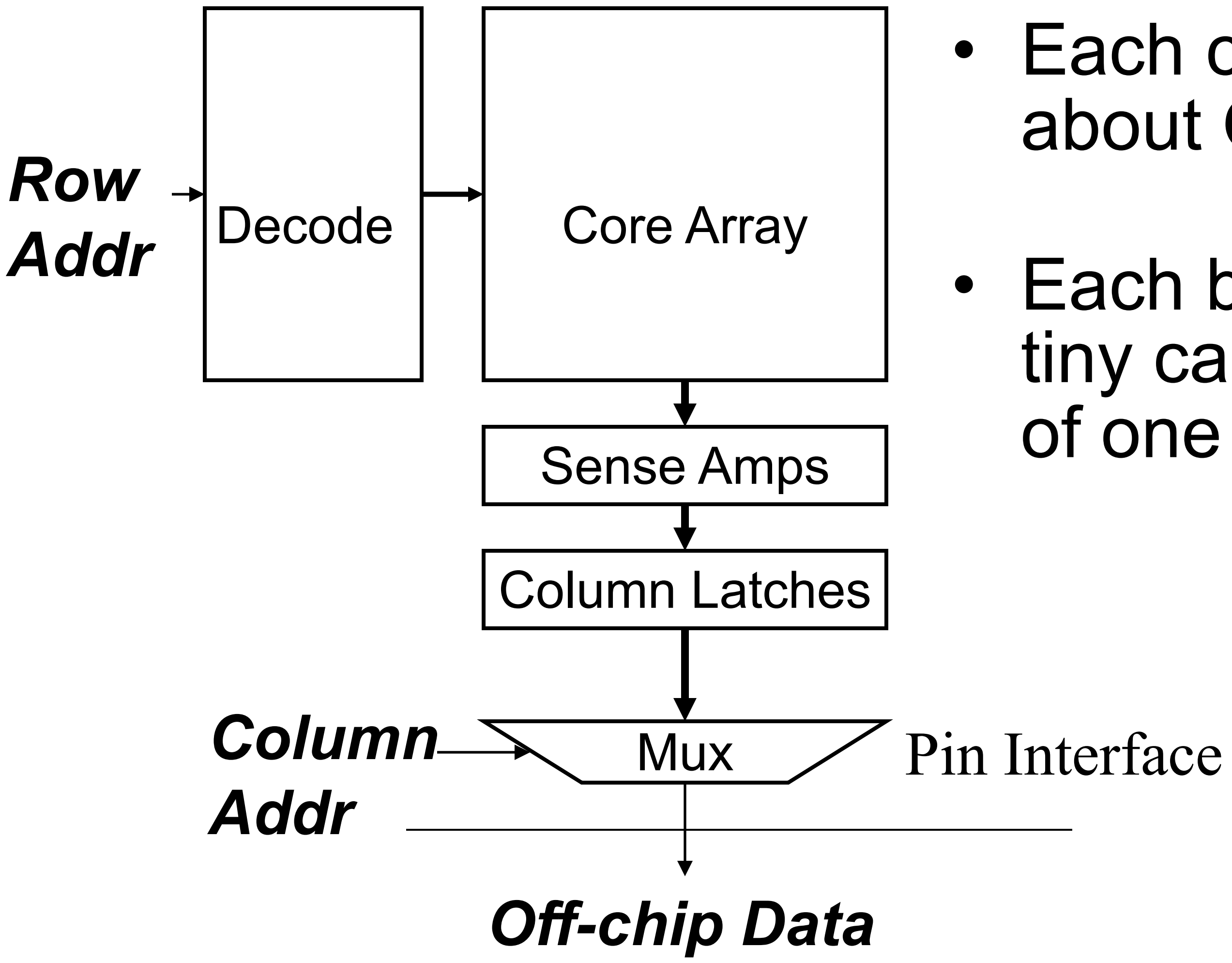
- How about rectangular matrices?

Global Memory Bandwidth

Ideal

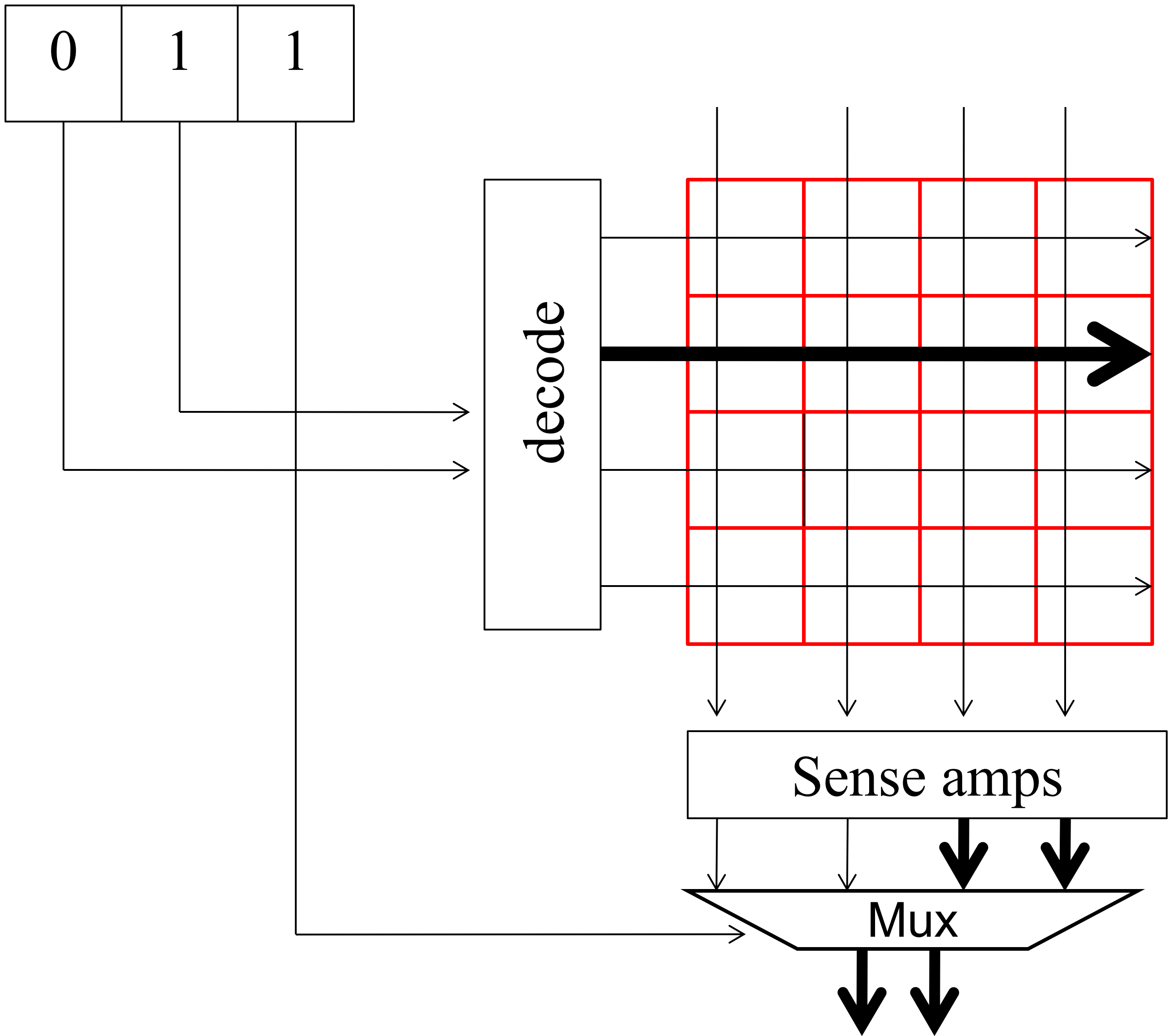
Reality

DRAM Bank Organization



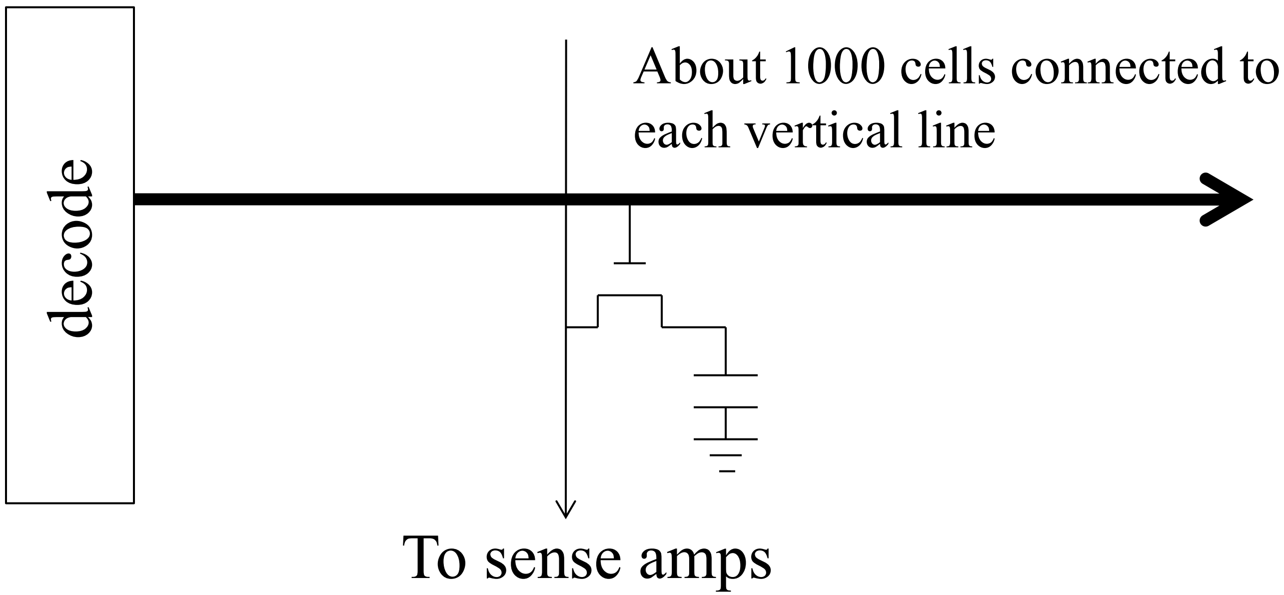
- Each core array has about O(1M) bits
- Each bit is stored in a tiny capacitor, made of one transistor

A very small (8x2 bit) DRAM Bank



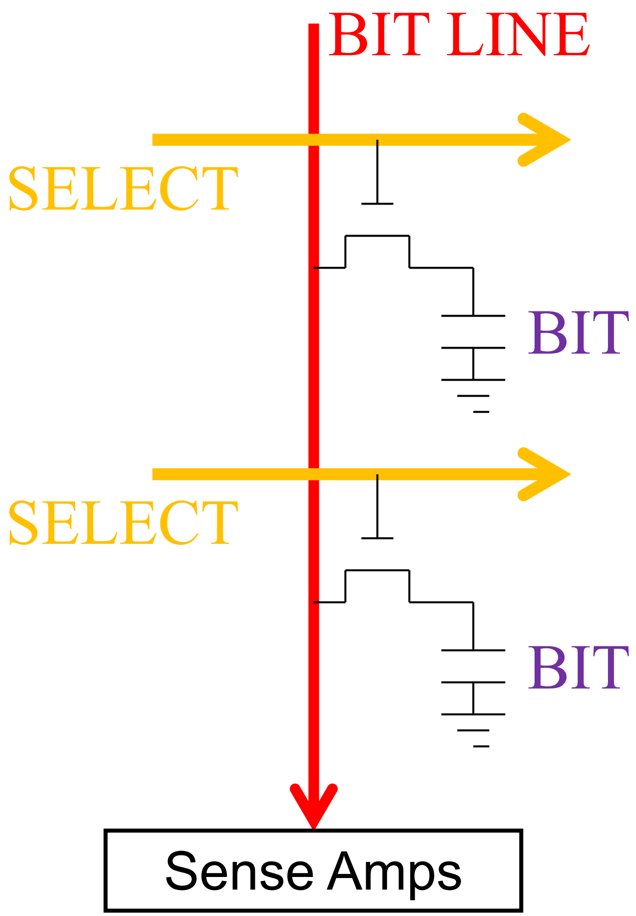
DRAM core arrays are slow.

- Reading from a cell in the core array is a very slow process
  - Current GDDR: Core speed = ⅛ interface speed
  - … likely to be worse in the future

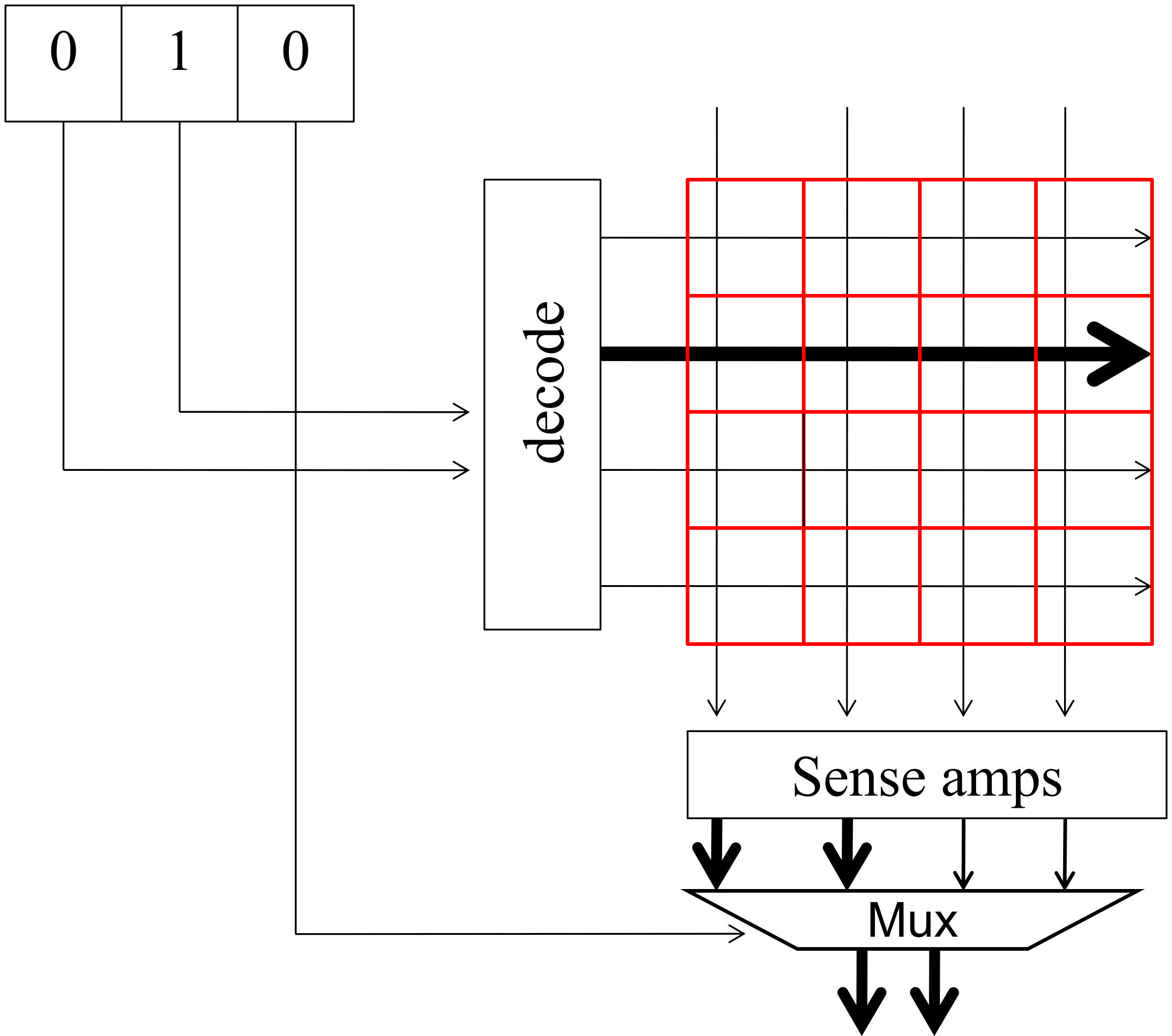


DRAM is Slow But Dense

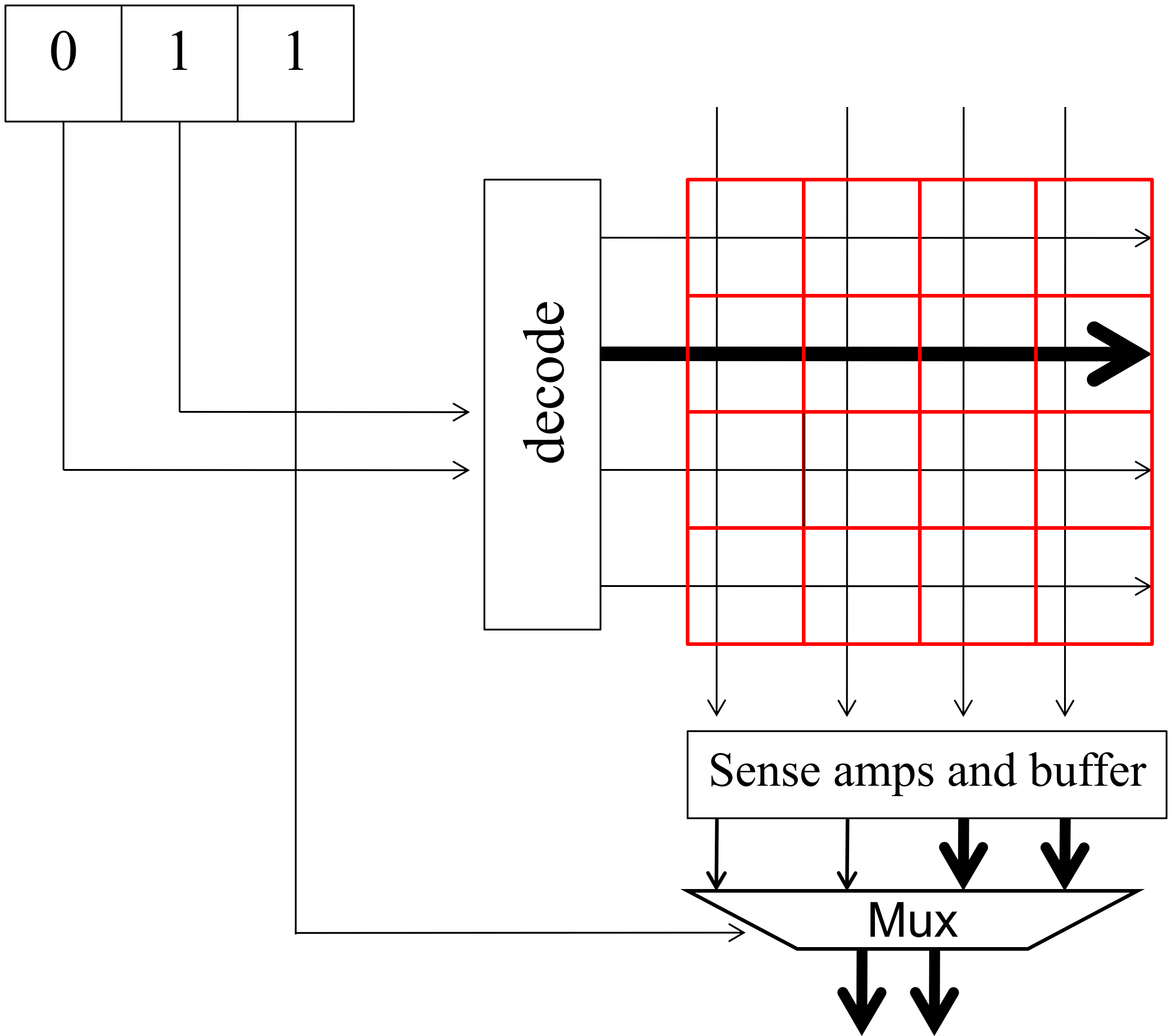
- Capacitance…
  - tiny for the **BIT**, but
  - huge for the **BIT LINE**
- Use an amplifier for higher speed!
- Still **slow**…
- But only need **1 transistor per bit**.



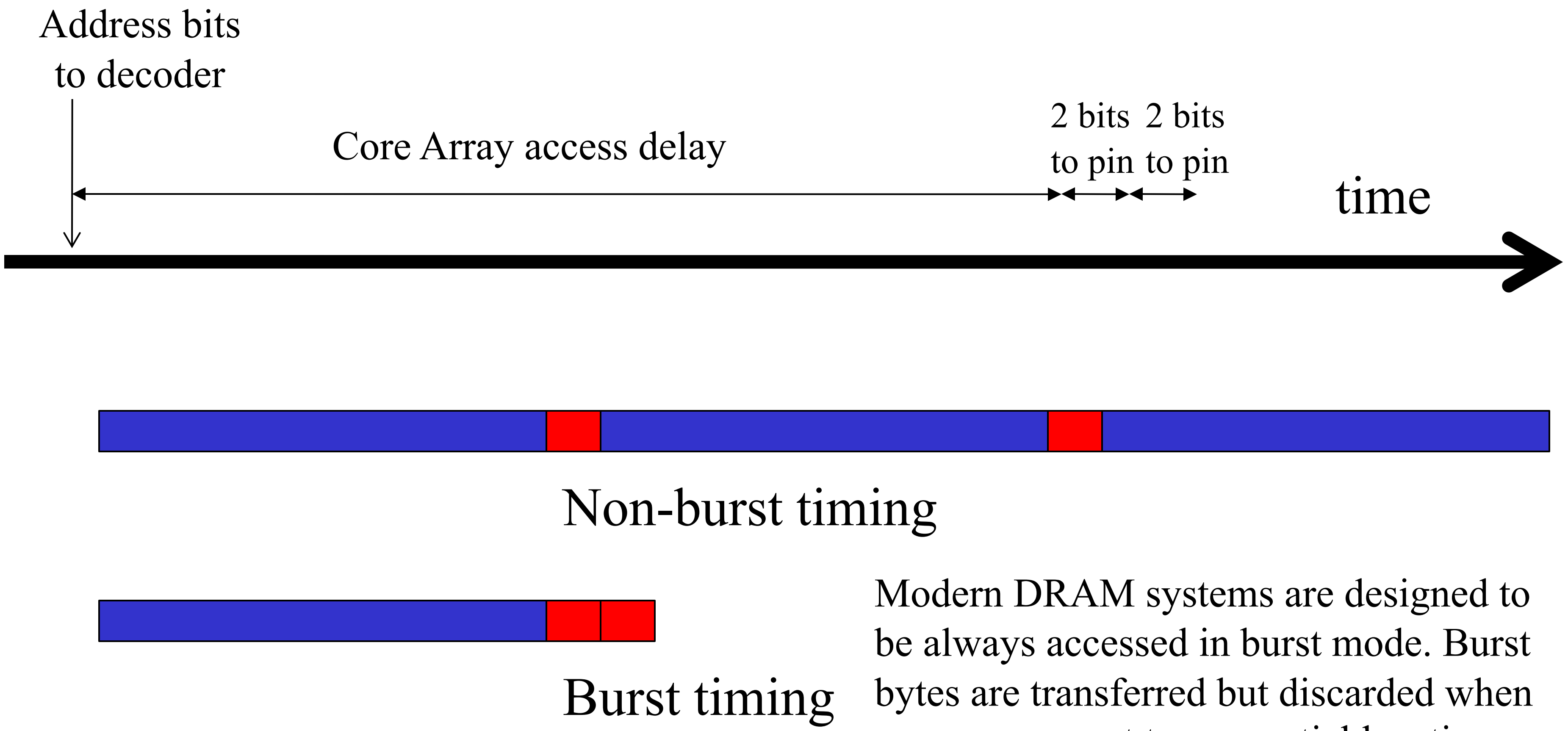
DRAM Bursting (burst size = 4 bits)



DRAM Bursting (cont.)
second part of the burst

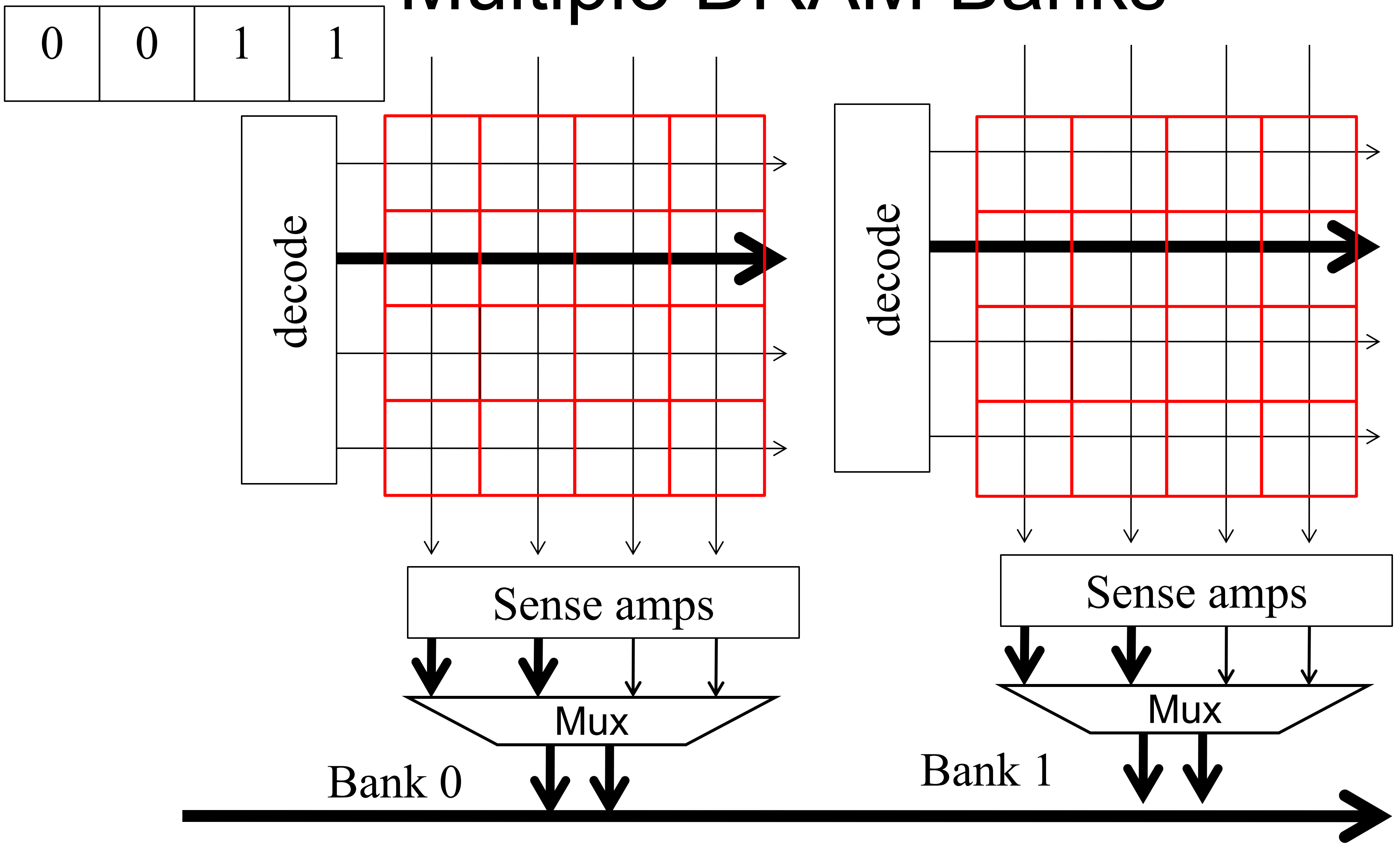


DRAM Bursting for the 8x2 Bank

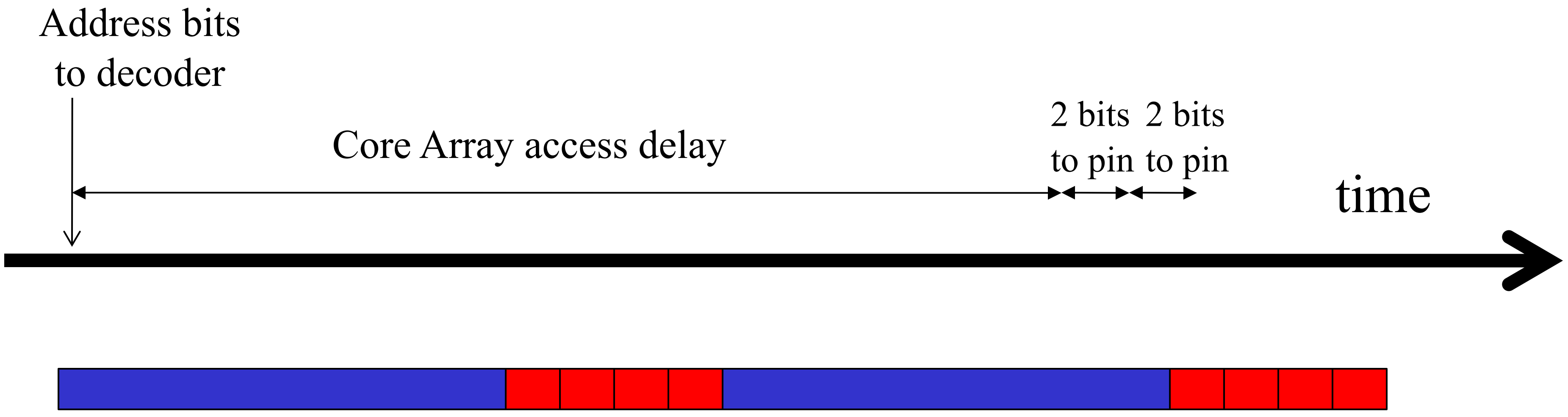


Modern DRAM systems are designed to be always accessed in burst mode. Burst bytes are transferred but discarded when accesses are not to sequential locations.

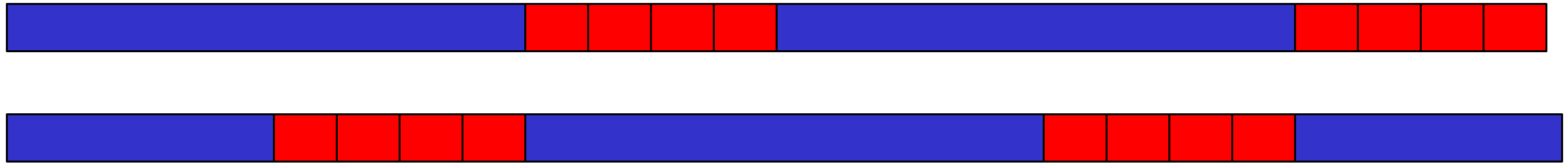
Multiple DRAM Banks



DRAM Bursting for the 8x2 Bank

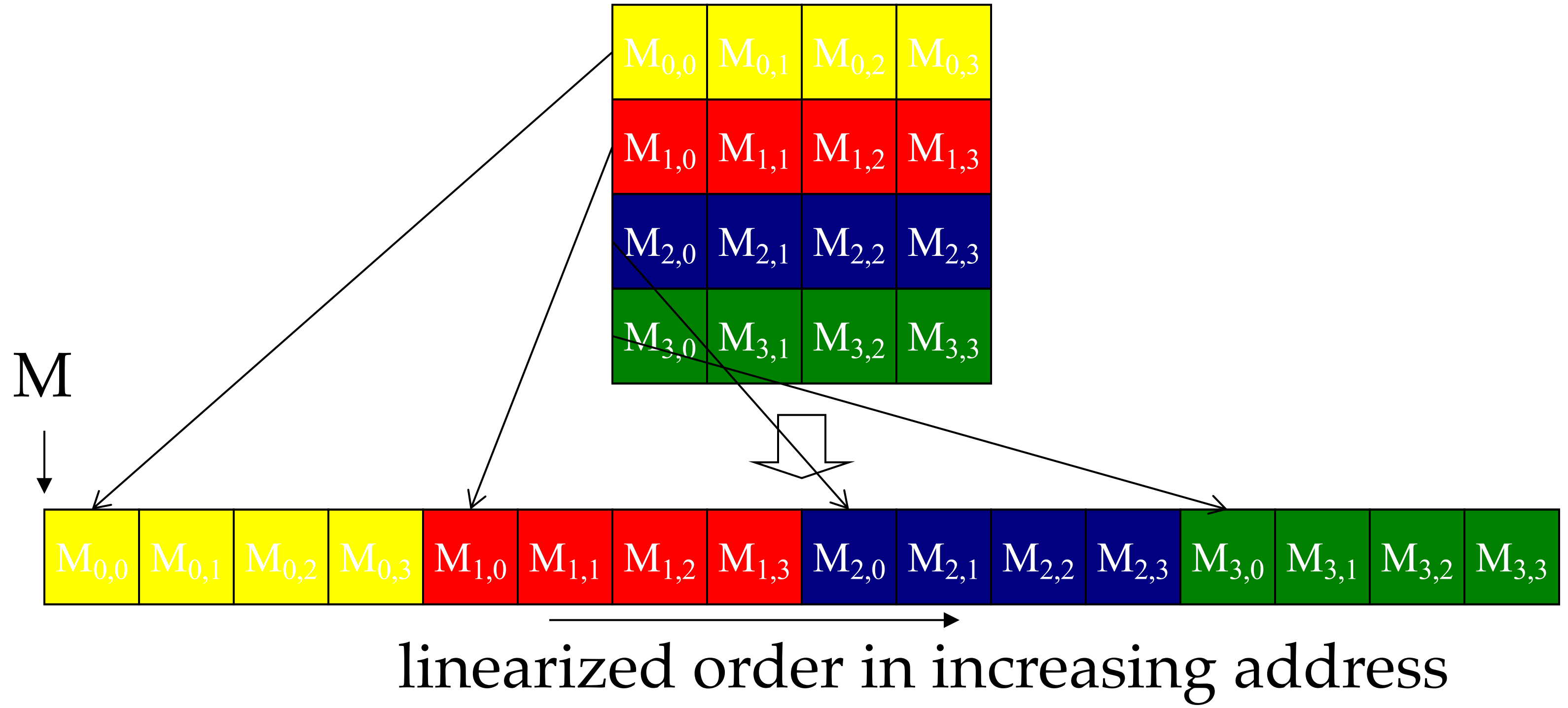


Single-Bank burst timing, dead time on interface



Multi-Bank burst timing, reduced dead time

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



A Simple Matrix Multiplication Kernel

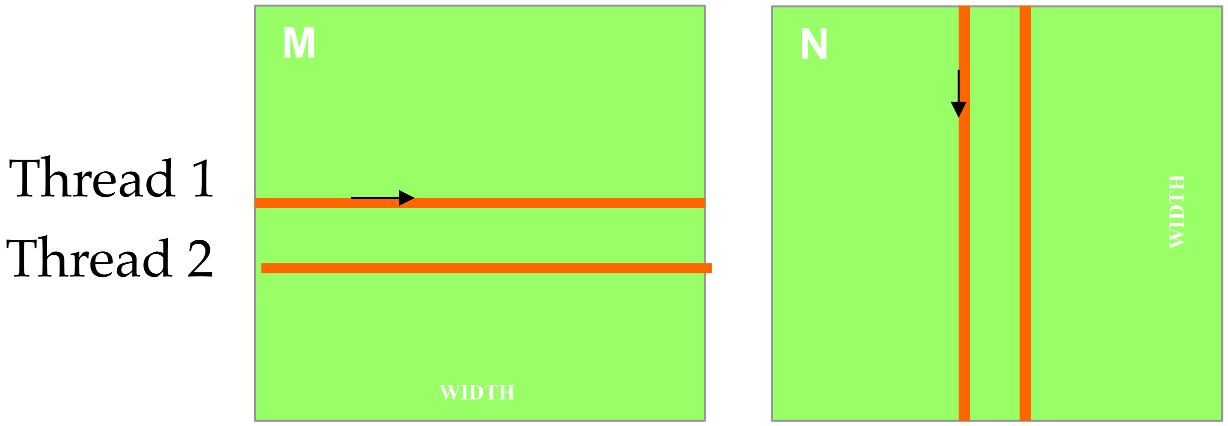
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 // Calculate the row index of the P element and M
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 // Calculate the column index of P and N
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 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;
  }
}
```

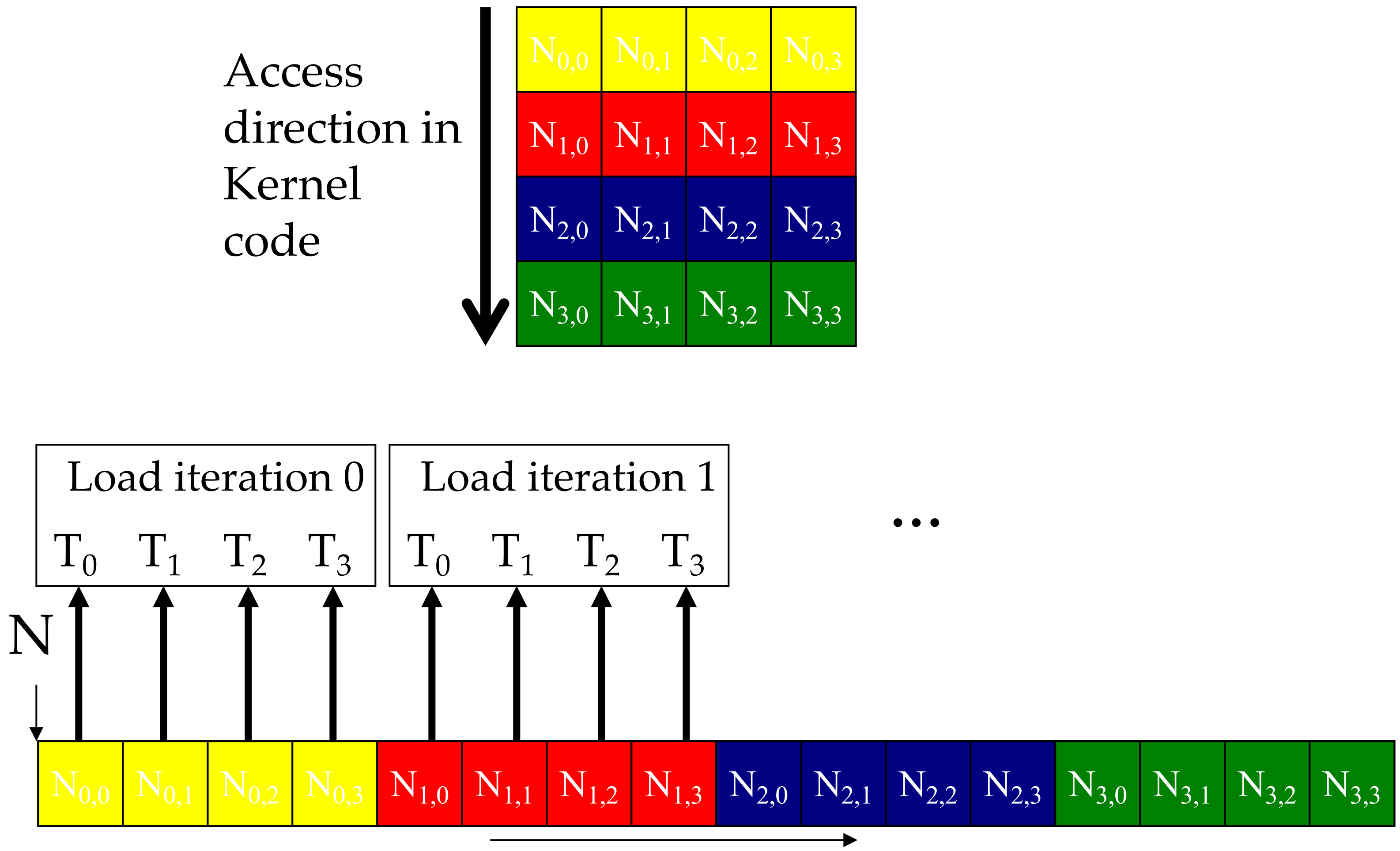
Two Access Patterns



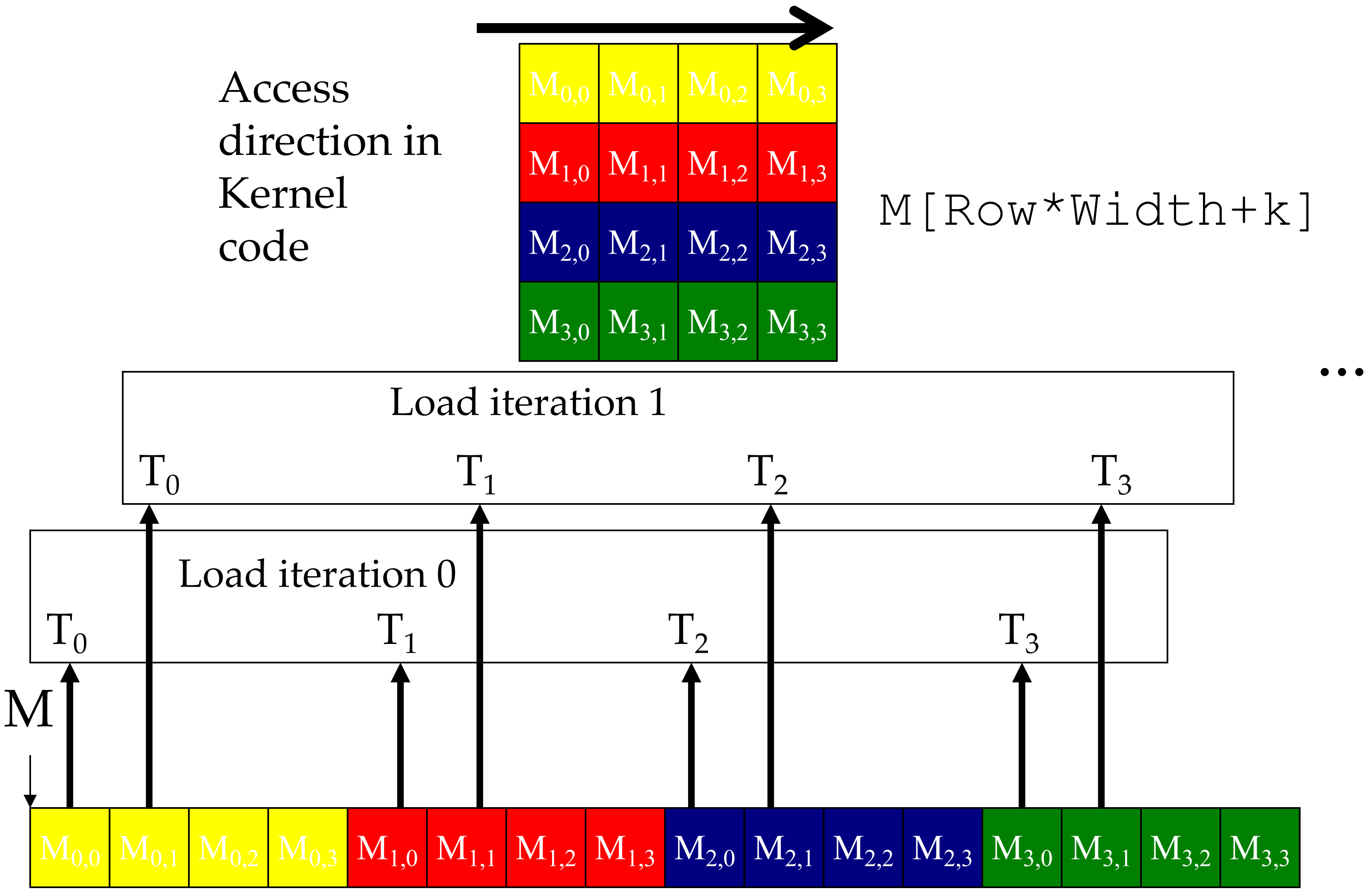
`M[Row*Width+k]`   `N[k*Width+Col]`

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

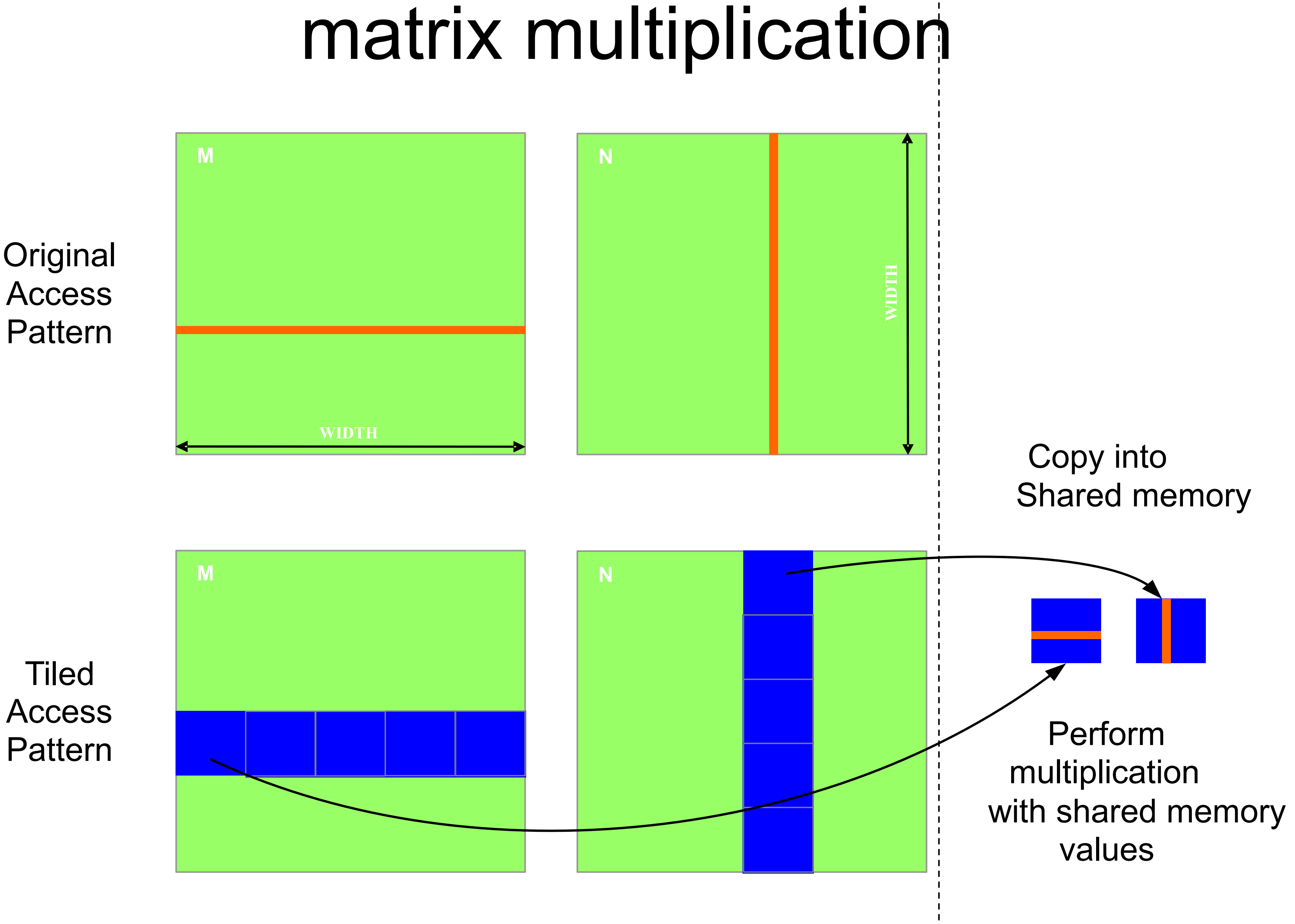
N accesses are coalesced.



M accesses are not coalesced.



Use shared memory to enable coalescing in tiled matrix multiplication



ANY MORE QUESTIONS?
READ CHAPTER 5