ECE408 / CS483 / CSE408
Summer 2024

Applied Parallel Programming

Lecture 6: More on Tiling

What Will You Learn Today?

- to handle boundary conditions in tiled algorithms

How to Handle Matrices of Other Sizes?

- Slide deck 5's tiled kernel
  - assumed integral number of tiles (thread blocks)
  - in all matrix dimensions.

How can we avoid this assumption?

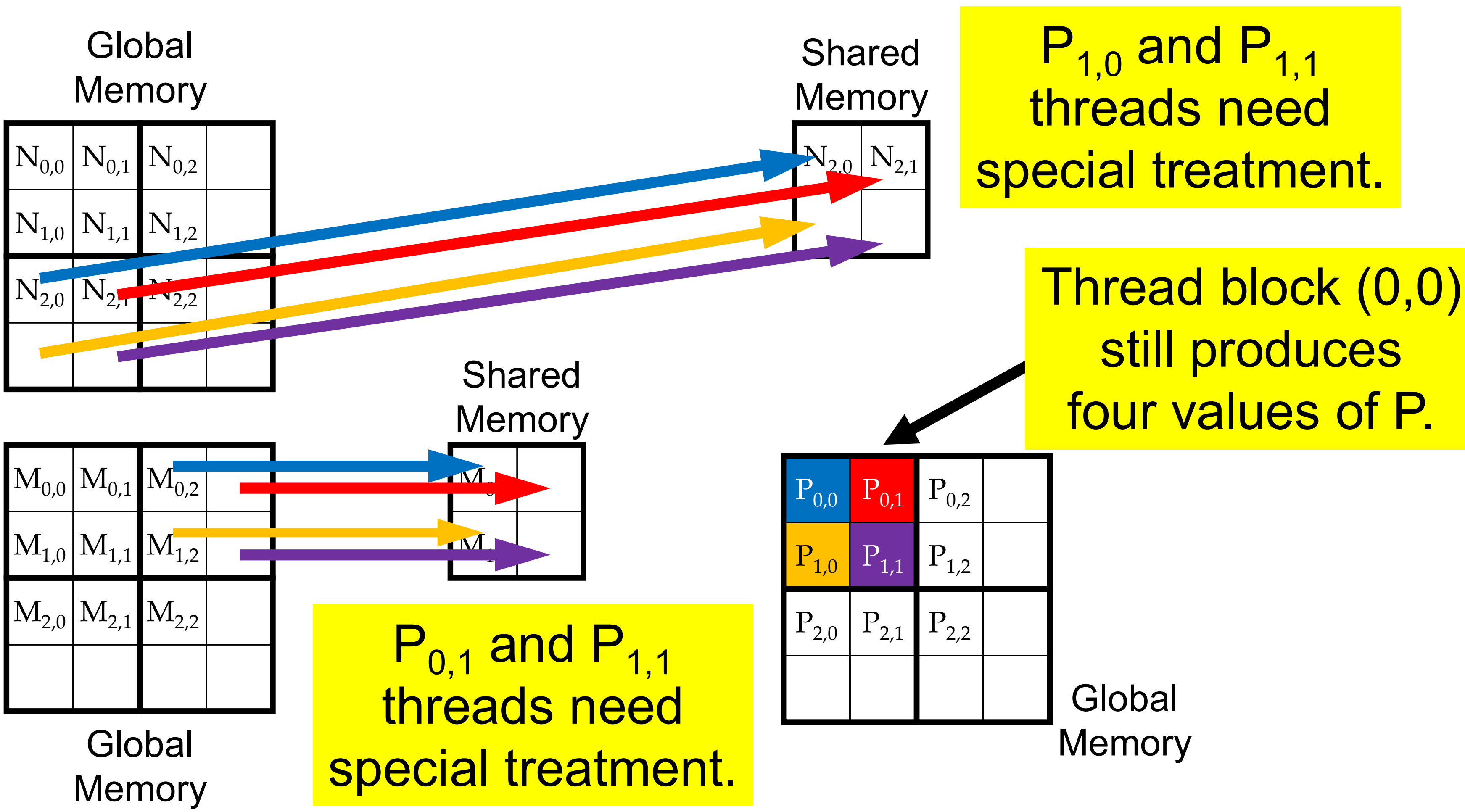
- One answer: add padding, but not easy to reformat data, and adds transfer time.

Other ideas?

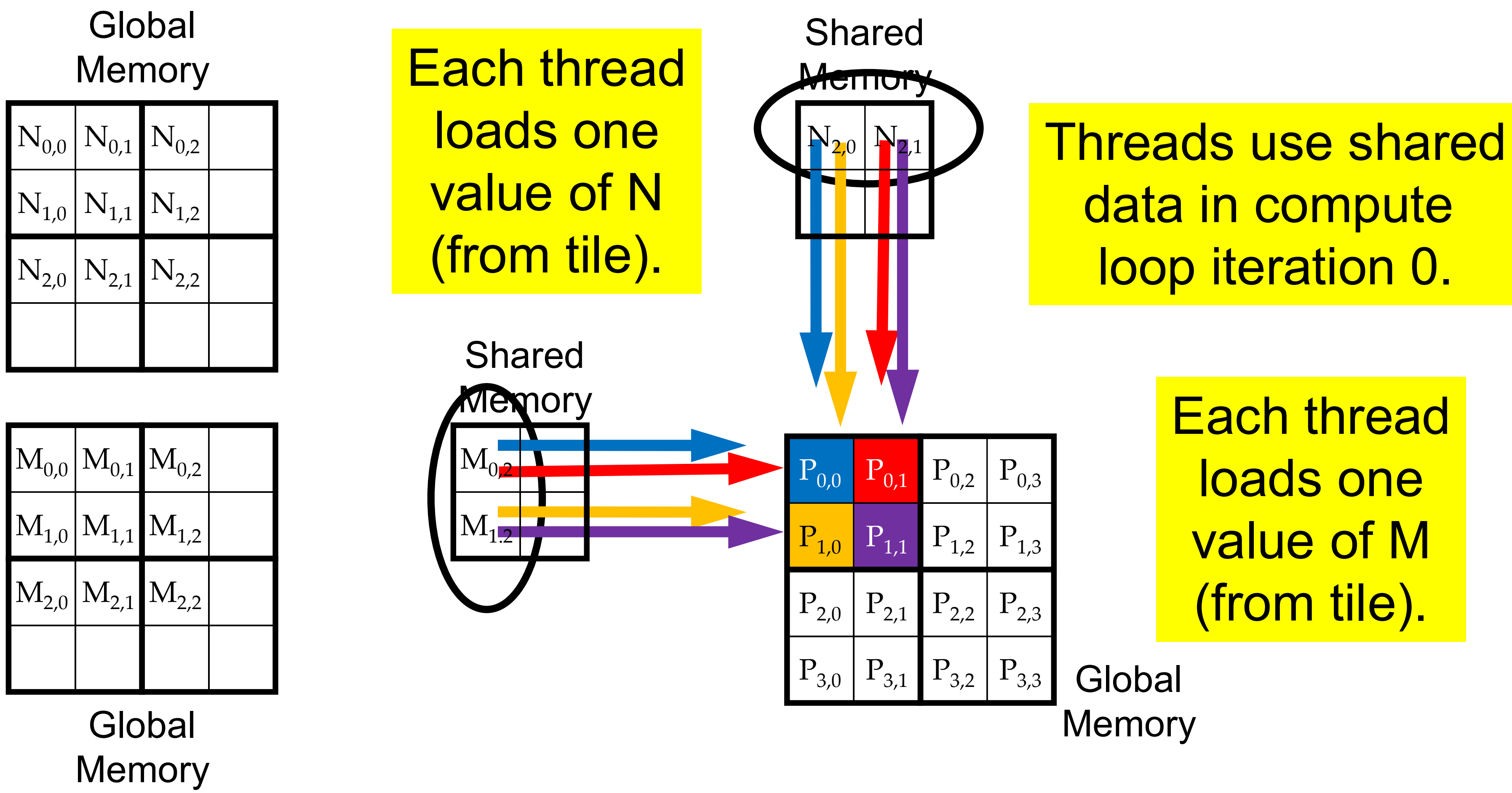
Let's Review Our Kernel

```
__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;
}
```

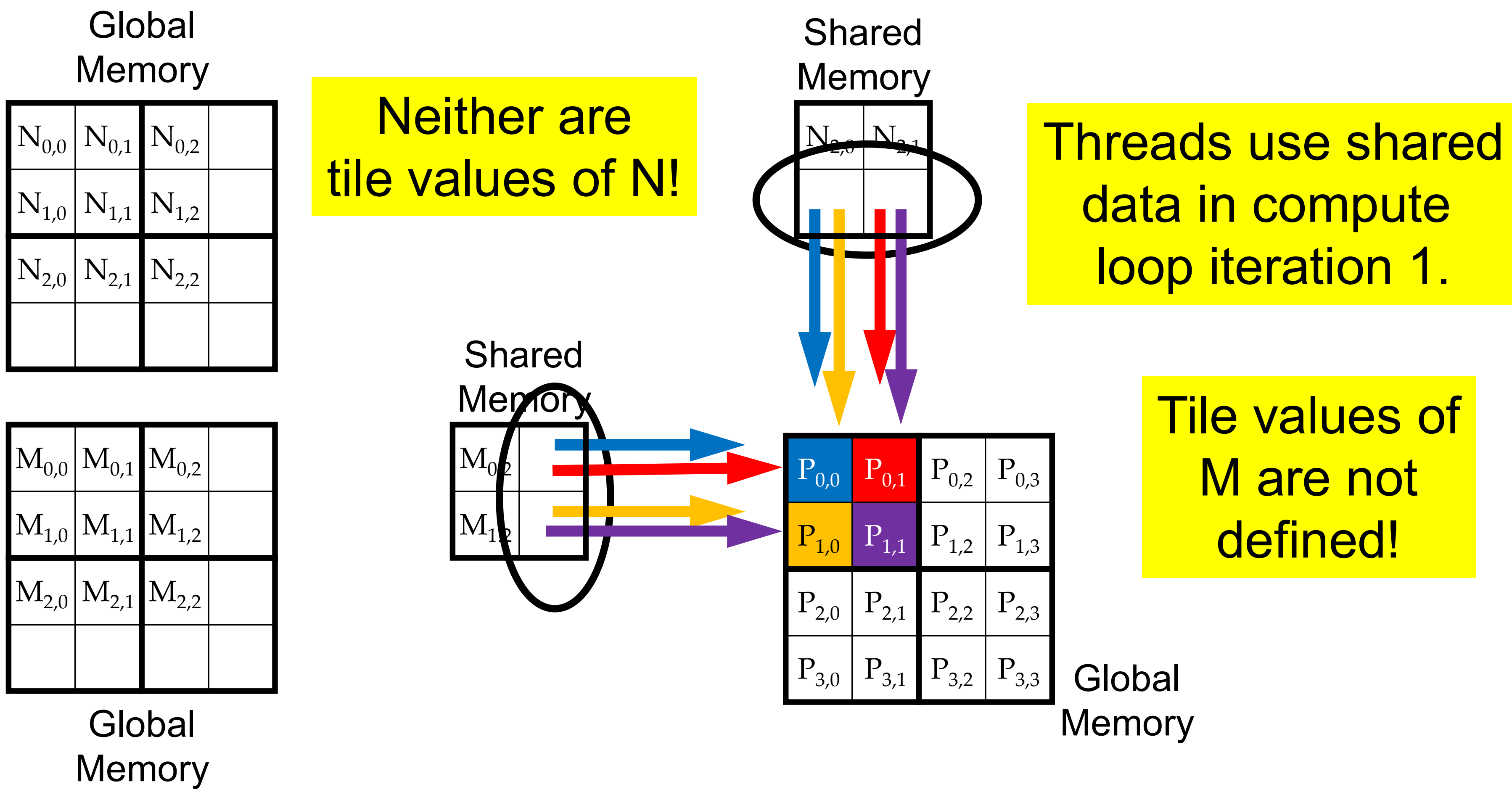
Recall Second Tiles Loaded for Thread Block (0,0)



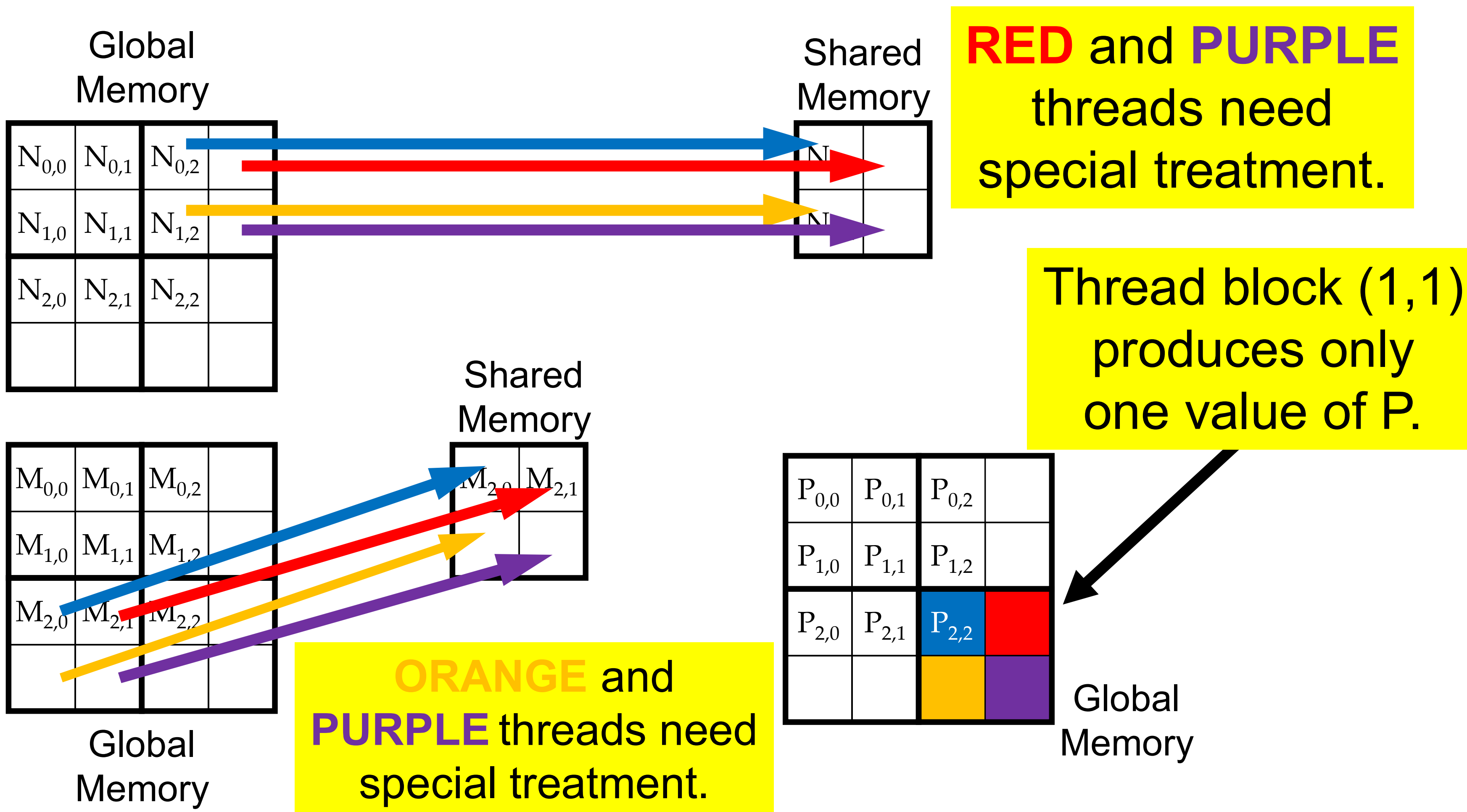
Thread Block (0,0) Computes on Shared Tiles (Iter 0)



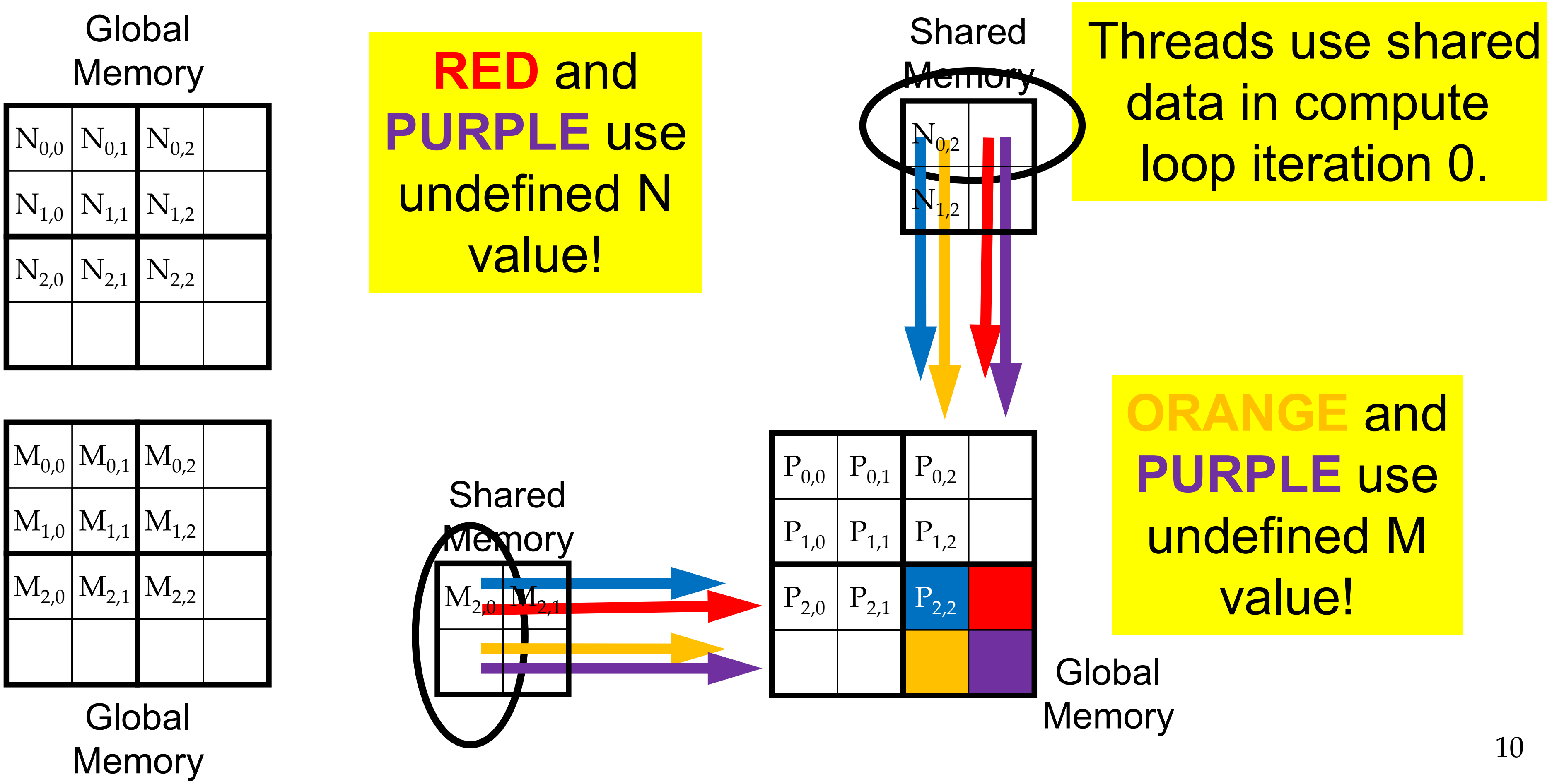
Thread Block (0,0) Computes on Shared Tiles (Iter 1)



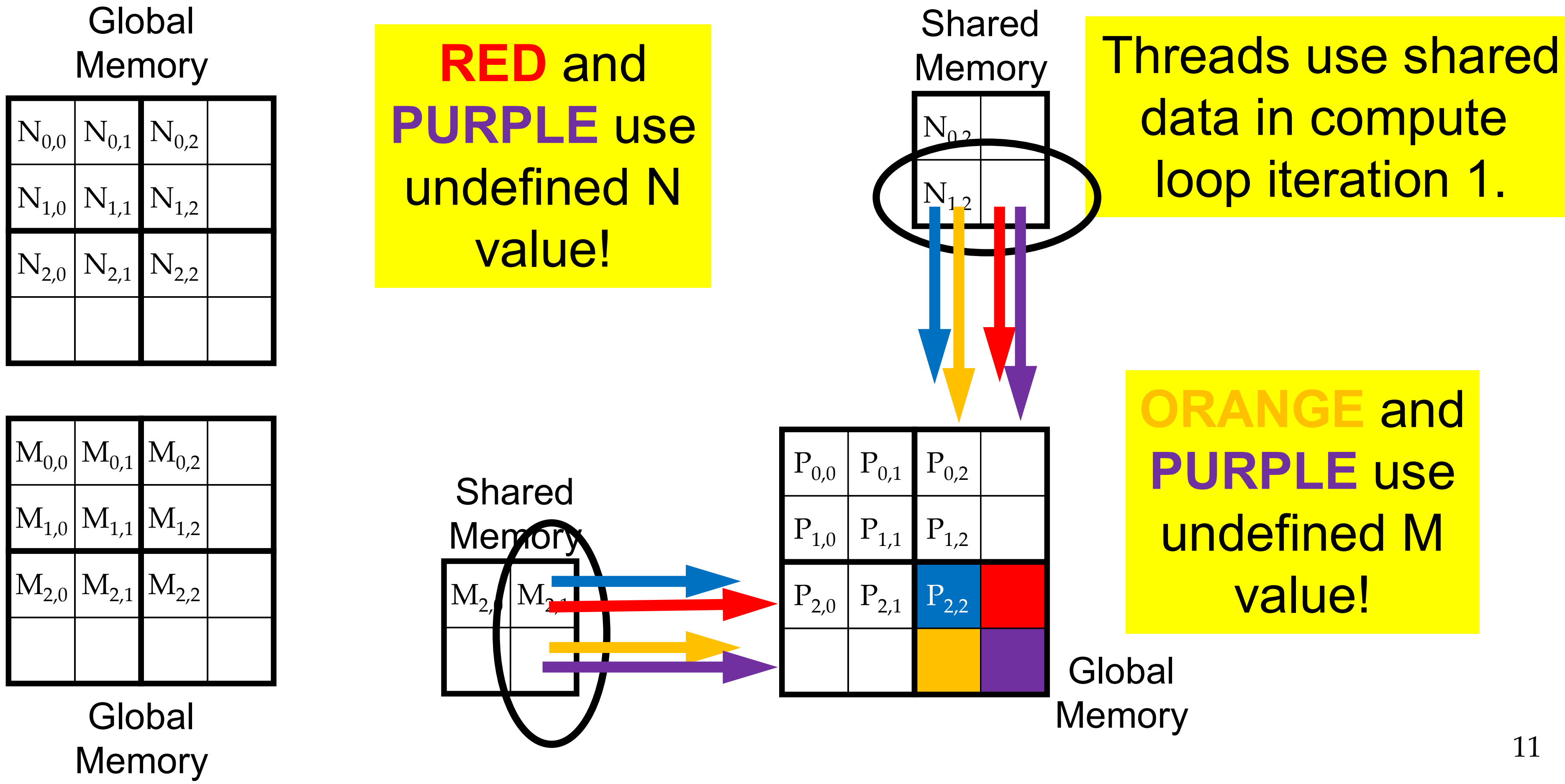
Let’s Look at the First Tile for Block(1,1) Next



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



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



Major Cases in Toy Example

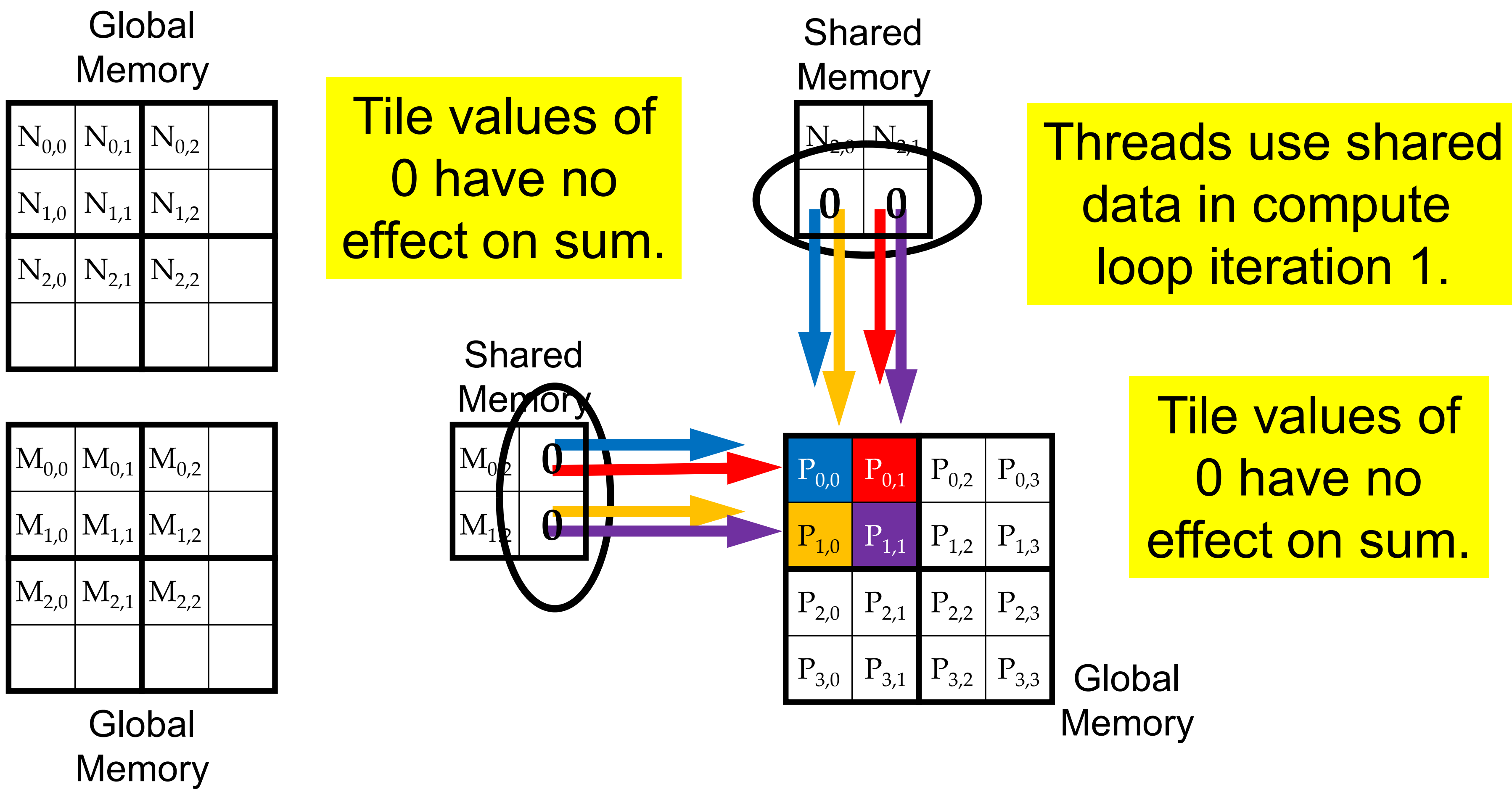
- Threads that calculate valid P elements but can step outside valid input
  – Second tile of Block(0,0), all threads when k is 1
- Threads that do not calculate valid P elements
  – Block(1,1), Thread(1,0), non-existent row
  – Block(1,1), Thread(0,1), non-existent column
  – Block(1,1), Thread(1,1), non-existent row and column

Solution: Write 0 for Missing Elements

- Test during tile load:
  is **target within input matrix**?
  - **If yes**, proceed to **load**;
  - **otherwise**, just **write 0** to shared memory.

- The **benefit**?
  - **No specialization during tile use**!
  - Multiplying by 0 guarantees that unwanted terms do not contribute to the inner product.

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

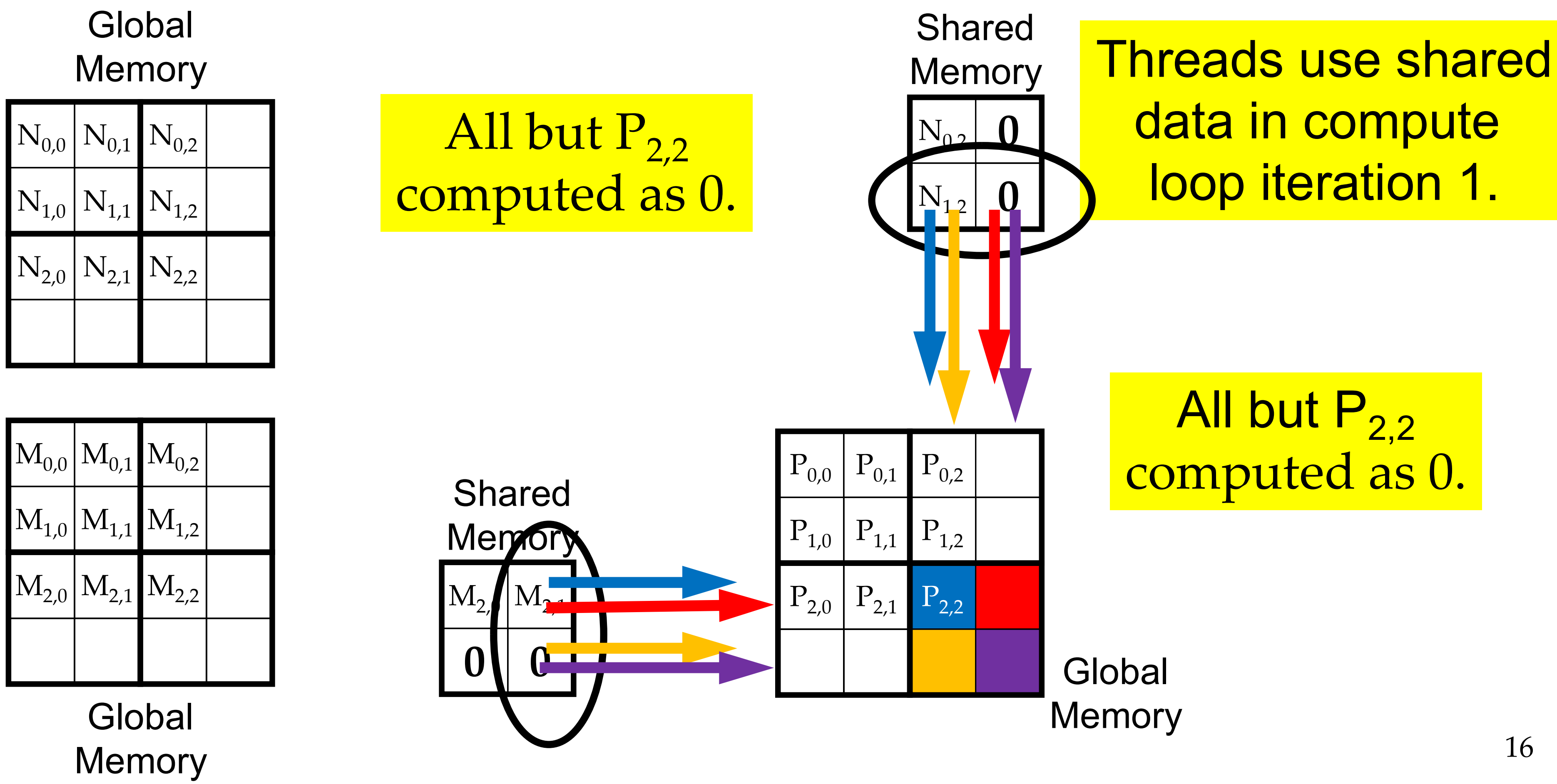


What About Threads Outside of P?

- If a **thread is not within P**,
  - All terms in sum are 0.
  - No harm in performing FLOPs.
  - No harm in writing to registers.
  - **Must not be allowed to write to global memory!**

So: **Threads outside of P calculate 0, but store nothing.**

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



Modifying the Tile Count

```
8.   for (int m = 0; m < Width/TILE_WIDTH; ++m) {
```

The bound for **m** implicitly assumes that Width is a multiple of **TILE_WIDTH**. We need to round up.

```
for (int m = 0; m < (Width - 1)/TILE_WIDTH + 1; ++m) {
```

For non-multiples of **TILE_WIDTH**:
- quotient is unchanged;
- add one to round up.

For multiples of **TILE_WIDTH**:
- quotient is now one smaller,
- but we add 1.

Modifying the Tile Loading Code

We had …

```
// 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];
```

Note: the tests for M and N tiles are NOT the same.

```
if (Row < Width && m*TILE_WIDTH+tx < Width) {
    // as before
    subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
} else {
    subTileM[ty][tx] = 0;
}
```

And for Loading N…

We had …

```
// 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];
```

Note: the tests for M and N tiles are NOT the same.

```
if (m*TILE_WIDTH+ty < Width && Col < Width ) {
    // as before
    subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
} else {
    subTileN[ty][tx] = 0;
}
```

Modifying the Tile Use Code

We had …

```
for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += subTileM[ty][k] * subTileN[k][tx];
```

Note: **no changes are needed**, but we might save a little energy (fewer floating-point ops)?

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

Modifying the Write to P

We had …

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

We must test for threads outside of P:

```
if (Row < Width && Col < Width) {
    // as before
    P[Row*Width+Col] = Pvalue;
}
```

Some Important Points

- For each thread, conditions are different for
  - Loading M element
  - Loading N element
  - Calculation/storing output elements
- Branch divergence
  - affects only blocks on boundaries, and
  - should be small for large matrices.
- What about rectangular matrices?

QUESTIONS?

READ CHAPTER 4!