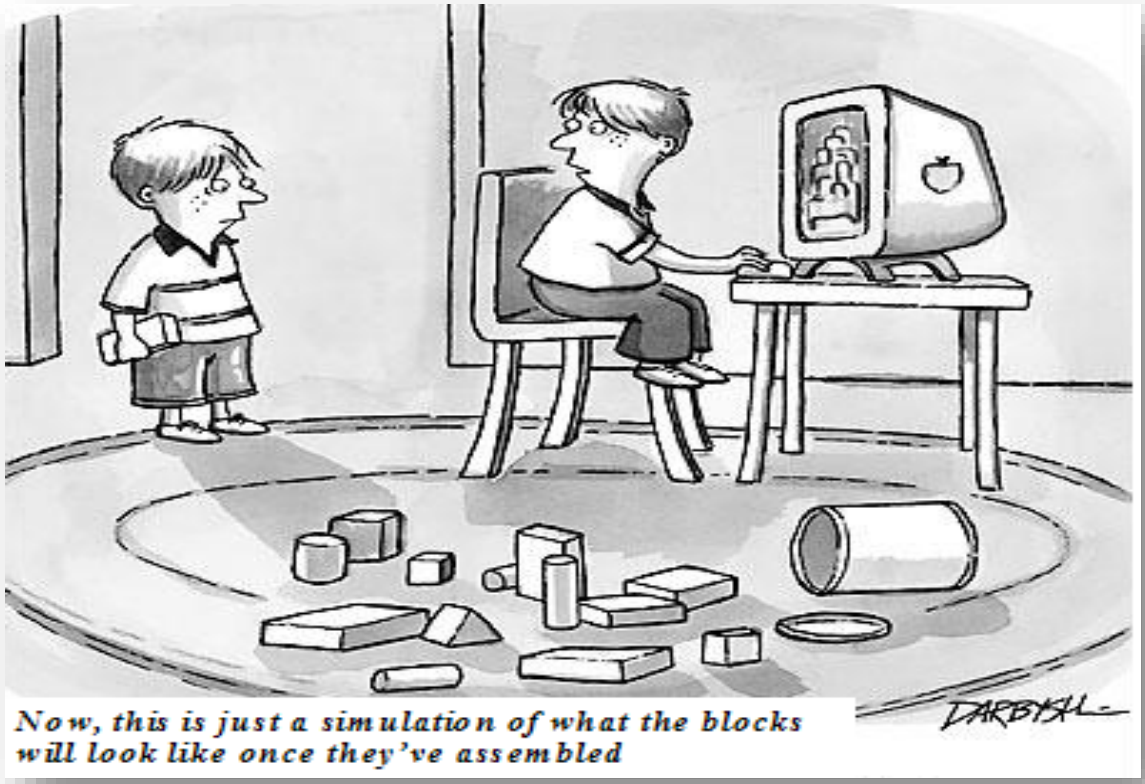
ECE569
Module 26

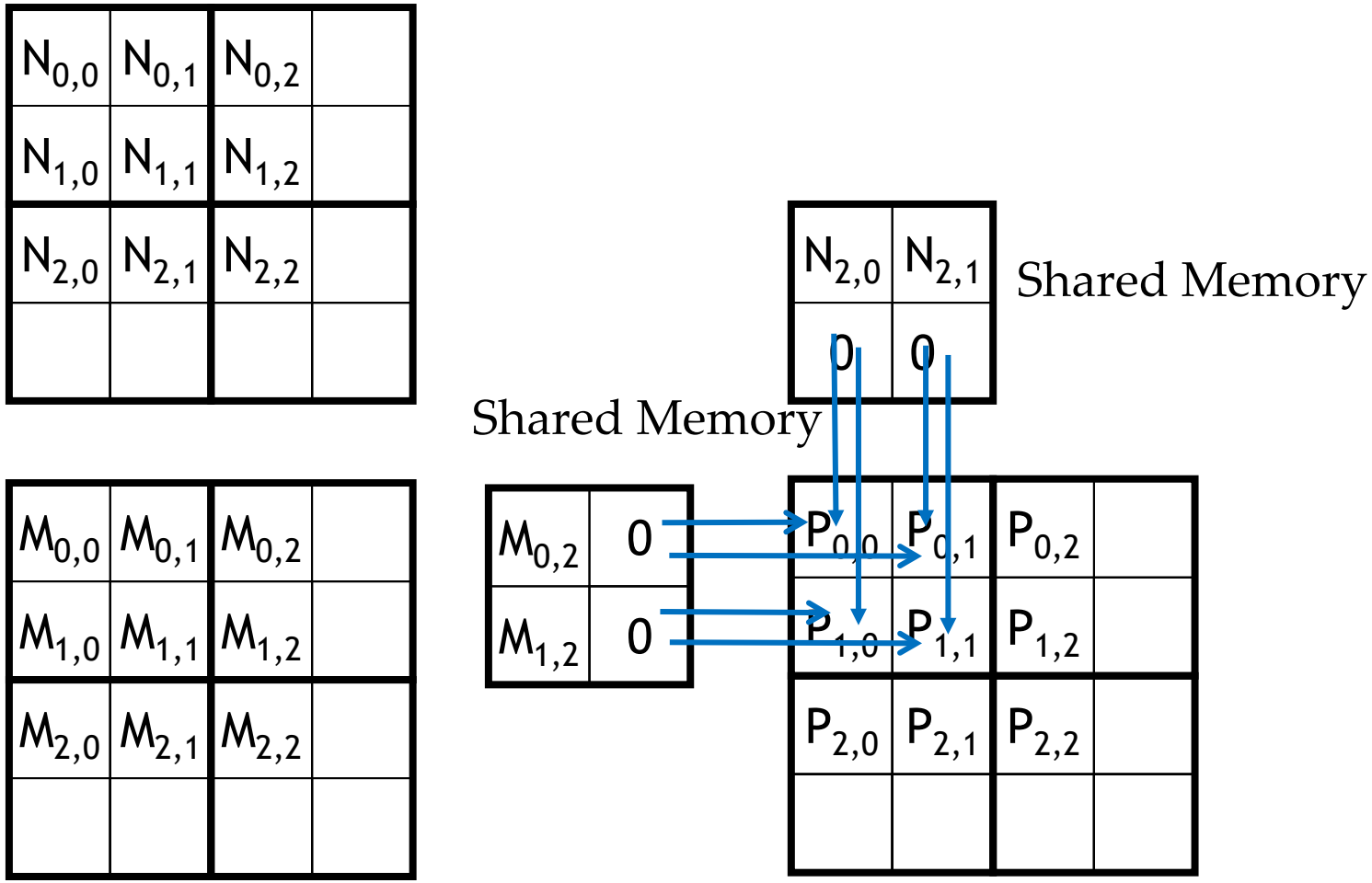


- Matrix Multiplication – Writing Boundary Conditions

A “Simple” Solution

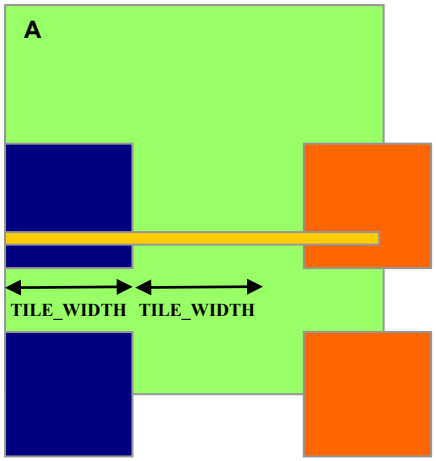
- **When a thread is to load any input element, test if it is in the valid index range**
  - If valid, proceed to load
  - Else, do not load, just write a 0
- **Rationale:** a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- **The condition tested for loading input elements is different from the test for calculating output P element**
  - A thread that does not calculate valid P element can still participate in loading input tile elements

Phase 1 Use for Block (0,0) (iteration 1)



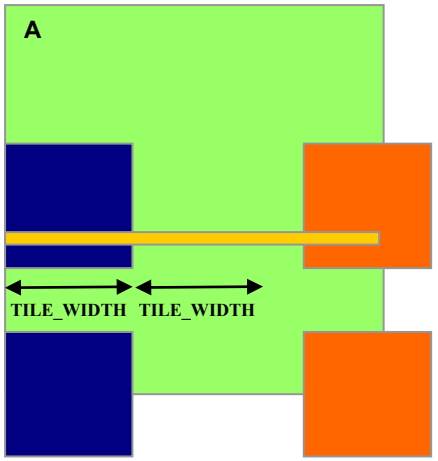
Boundary Condition for Input M Tile

- **Each thread loads**
  - M[Row][p*TILE_WIDTH+tx]
  - M[Row*Width + p*TILE_WIDTH+tx]
- **Need to test**
  - (**???**)
  - If true, load M element
  - Else , load 0



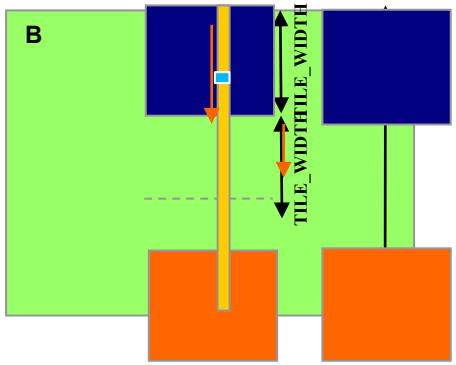
Boundary Condition for Input M Tile

- **Each thread loads**
  - M[Row][p*TILE_WIDTH+tx]
  - M[Row*Width + p*TILE_WIDTH+tx]
- **Need to test**
  - (Row < Width) && (p*TILE_WIDTH+tx < Width)
  - If true, load M element
  - Else , load 0



Boundary Condition for Input N Tile

- **Each thread loads**
  - N[p*TILE_WIDTH+ty][Col]
  - N[(p*TILE_WIDTH+ty)*Width+ Col]
- **Need to test**
  - (p*TILE_WIDTH+ty < Width) && (Col< Width)
  - If true, load N element
  - Else , load 0



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];
  int bx = blockIdx.x;  int by = blockIdx.y;
  int tx = threadIdx.x; int ty = threadIdx.y;

  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 p = 0; p < (Width-1)/TILE_WIDTH+1; ++p) {
    // Collaborative loading of M and N tiles into shared memory
    ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
    ds_N[ty][tx] = N[(p*TILE_WIDTH+ty)*Width + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
      Pvalue += ds_M[ty][i] * ds_N[i][tx];
    __synchthreads();
  }
  P[Row*Width+Col] = Pvalue;
}
```

Loading Elements – with boundary check

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

    if(Row < Width && p * TILE_WIDTH+tx < Width) {
       ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
    } else {
       ds_M[ty][tx] = 0.0;
    }
    if (p*TILE_WIDTH+ty < Width && Col < Width) {
      ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
    } else {
      ds_N[ty][tx] = 0.0;
    }
    __syncthreads();
```

Inner Product – Before and After

```
   if(_____________________________________) {
      for (int i = 0; i < TILE_WIDTH; ++i) {
        Pvalue += ds_M[ty][i] * ds_N[i][tx];
      }
      __syncthreads();
   } /* end of outer for loop */
   if (_____________________________________)
      P[Row*Width + Col] = Pvalue;
} /* end of kernel */
```

Inner Product – Before and After

```
  if(Row < Width && Col < Width) {
     for (int i = 0; i < TILE_WIDTH; ++i) {
       Pvalue += ds_M[ty][i] * ds_N[i][tx];
     }
     __syncthreads();
  } /* end of outer for loop */
  if (Row < Width && Col < Width)
     P[Row*Width + Col] = Pvalue;
} /* end of kernel */
```

When accessing the shared memory, since we initialized contents to 0 we can let all threads in a tile participate in computation.

Some Important Points

- **For each thread the conditions are different for**
  - Loading M element
  - Loading N element
  - Calculating and storing output elements
- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - **The concern is that these checks may cause significant performance degradation**
- **The effect of control divergence should be small for large matrices**

Next

- **Divergence in matrix multiplication**