

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)
{
1.  __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the P element to work on
5.  int Row = by * TILE_WIDTH + ty; // note: blockDim.x == TILE_WIDTH
6.  int Col = bx * TILE_WIDTH + tx; //          blockDim.y == TILE_WIDTH
7.  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!
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of M and N tiles into shared memory
9.      subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.     subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
11.     __syncthreads();
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += subTileM[ty][k] * subTileN[k][tx];
14.     __syncthreads();
15. }
16. P[Row*Width+Col] = Pvalue;
}
```

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

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

Global Memory

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

Shared Memory

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

$P_{1,0}$  and  $P_{1,1}$  threads need special treatment.

Thread block (0,0) still produces four values of P.

Global Memory

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

Shared Memory

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

$P_{0,1}$  and  $P_{1,1}$  threads need special treatment.

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

Global Memory

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

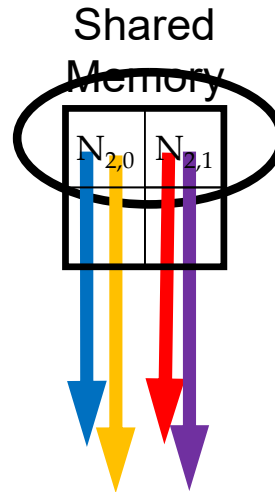
Global Memory

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

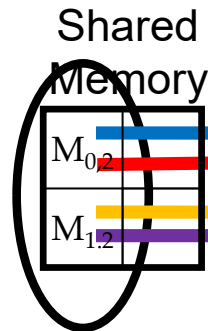
Global Memory

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

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).

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

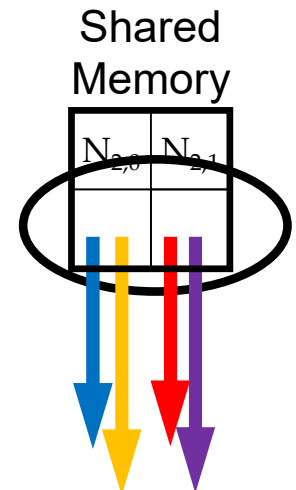
Global Memory

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

Global Memory

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

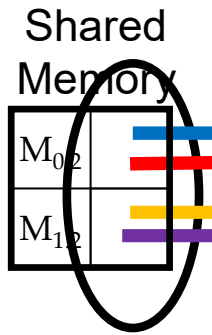
Neither are tile values of N!



Threads use shared data in compute loop iteration 1.

Global Memory

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



Tile values of M are not defined!

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

Global Memory

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

Global Memory

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

Shared Memory

$N_{1,1}$		
$N_{1,2}$		

**RED** and **PURPLE** threads need special treatment.

Global Memory

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

Shared Memory

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

**ORANGE** and **PURPLE** threads need special treatment.

Thread block (1,1) produces only one value of P.

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

Global Memory



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

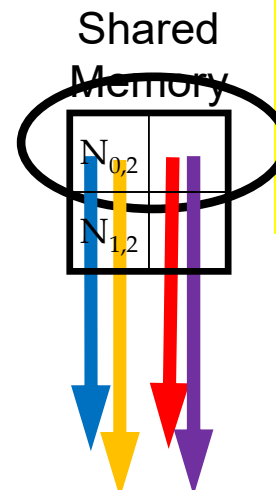
Global Memory

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

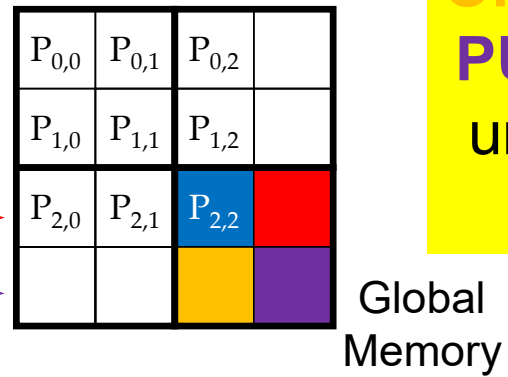
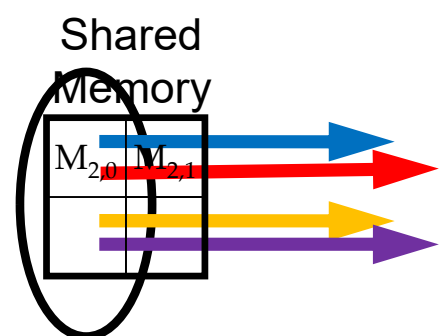
Global Memory

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

**RED** and **PURPLE** use undefined N value!



Threads use shared data in compute loop iteration 0.



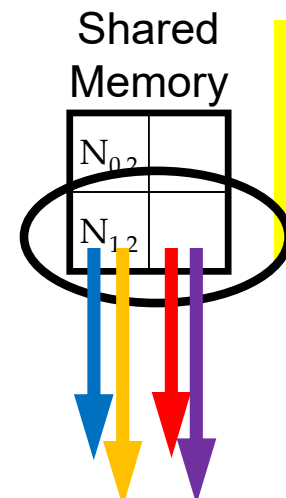
**ORANGE** and **PURPLE** use undefined M value!

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

Global Memory

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

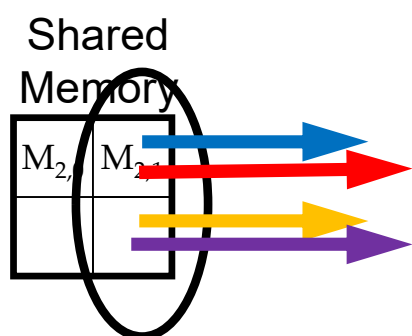
**RED** and **PURPLE** use undefined N value!



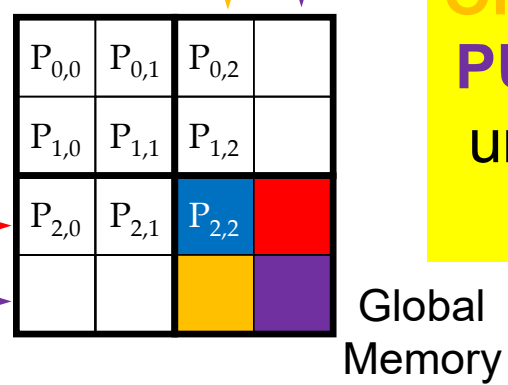
Threads use shared data in compute loop iteration 1.

Global Memory

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



**ORANGE** and **PURPLE** use undefined M value!



# 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

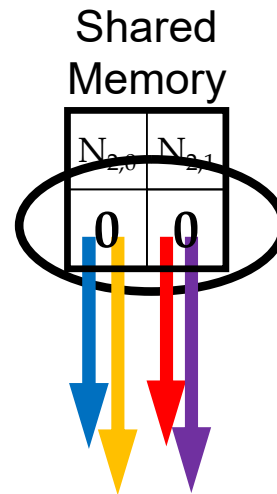
- Test during tile load:  
is **target within input matrix**?
  - **If yes**, proceed to **load**;
  - **otherwise**, just **write 0** to shared memory.
- The **benefit**?
  - **No specialization during tile use!**
  - Multiplying by 0 guarantees that unwanted terms do not contribute to the inner product.

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

Global Memory

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

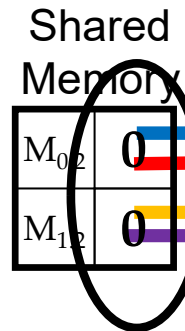
Tile values of 0 have no effect on sum.



Threads use shared data in compute loop iteration 1.

Global Memory

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



Tile values of 0 have no effect on sum.

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

Global Memory

# 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)

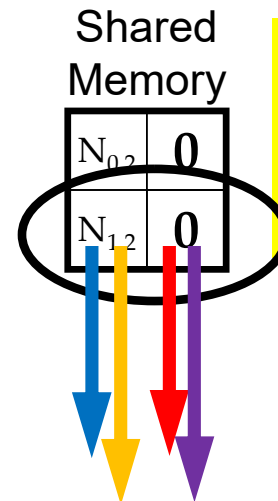
Global Memory

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

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

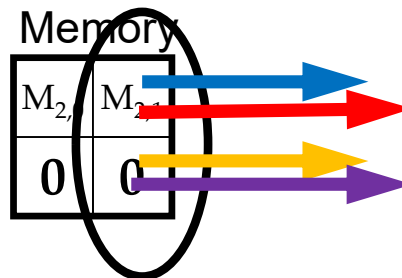
Global Memory

All but  $P_{2,2}$  computed as 0.



Threads use shared data in compute loop iteration 1.

Shared Memory



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

All but  $P_{2,2}$  computed as 0.

Global Memory

# 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
9.   subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.  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
9.   subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
10.  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 ...

```
12. for (int k = 0; k < TILE_WIDTH; ++k)
13.     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?

A decorative graphic on the left side of the slide consisting of two vertical lines: a blue line on the left and an orange line on the right, both extending from the top to the bottom of the slide.

**QUESTIONS?**

**READ CHAPTER 4!**