ECE408/CS483/CSE408 Fall 2021 Applied Parallel Programming

Lecture 6: Generalized Tiling & DRAM Bandwidth

Course Reminders

- Lab 2 is due this Friday
- Lab 3 should be out, it is due next Friday

Objectives

- To learn to handle boundary conditions in tiled algorithms.
- To understand the organization of memory based on dynamic RAM (DRAM).
- To understand the use of burst mode and multiple banks (both sources of parallelism) to increase DRAM performance (data rate).
- To understand memory access coalescing, which connects GPU kernel performance to DRAM organization.

How to Handle Matrices of Other Sizes?

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

How can we avoid this assumption?

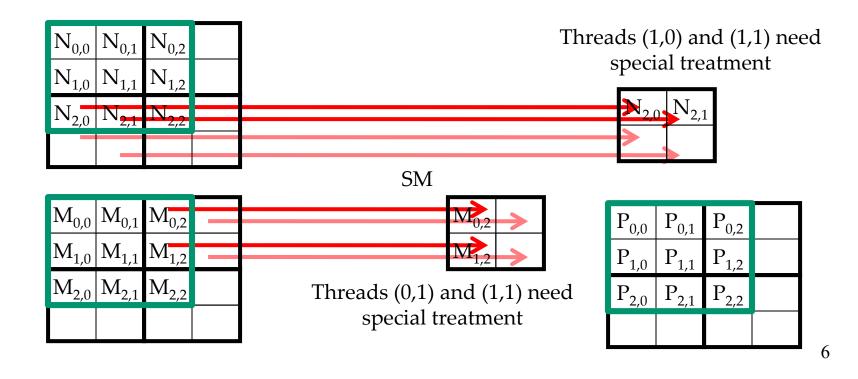
 One answer: add padding, but not easy to reformat data, and adds transfer time.

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;
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
   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!
8. 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];
9.
       subTileN[ty][tx] = N[(m*TILE WIDTH+ty)*Width+Col];
10.
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;
```

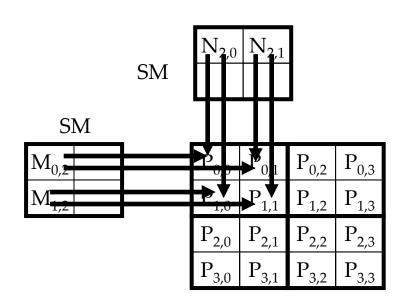
Second Tile Load for Block (0,0)



Second Tile Use for Block (0,0), k of 0

N _{0,0}	N _{0,1}	N _{0,2}	
N _{2,0}		N _{2,2}	

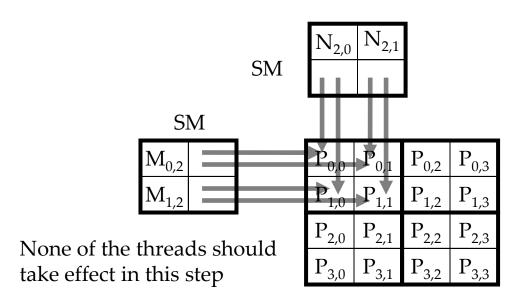
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



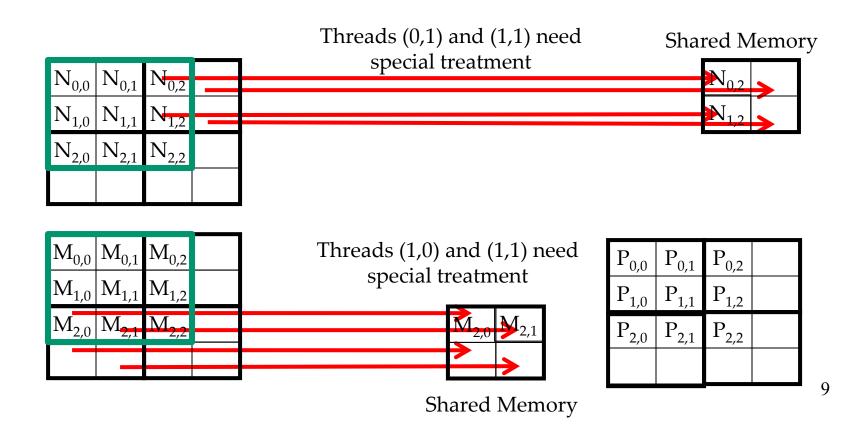
Second Tile Use for Block (0,0), k of 1

N _{0,0}	N _{0,1}	N _{0,2}	

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

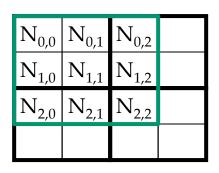


First Tile Load for Block (1,1)

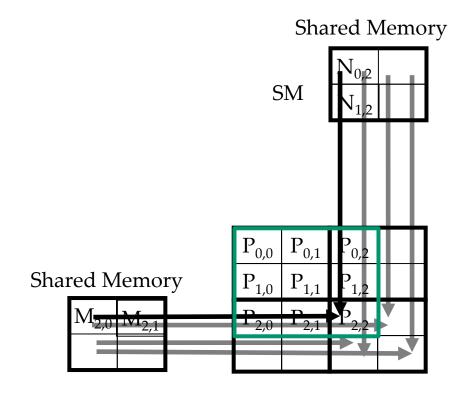


© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2018 ECE408/CS483/ University of Illinois at Urbana-Champaign

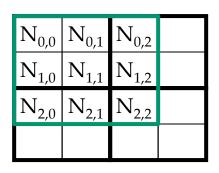
First Tile Use for Block (1,1), k of 0



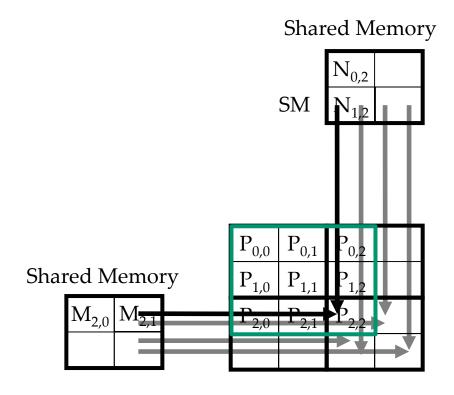
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



First Tile Use for Block (1,1), k of 1



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



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-existing column
 - Block(1,1), Thread(1,1), non-existing row/column

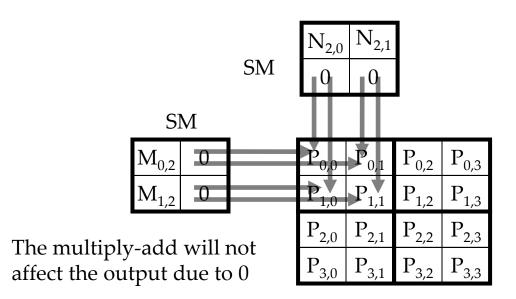
Solution: Write 0 for Missing Elements

- 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.

Second Tile Use for Block (0,0), k of 1

N _{0,0}	N _{0,1}	N _{0,2}	
N _{2,0}	N _{2,1}		

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

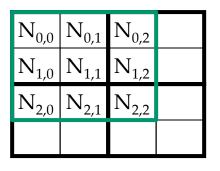


What About Threads Outside of P?

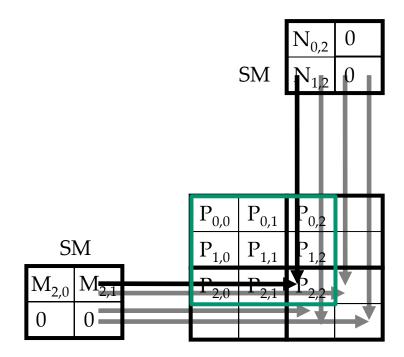
- If a thread is not within P,
 - All terms in sum are 0.
 - No harm in performing FLOPs.
 - No harm in writing to registers.
 - Must not be allowed to write to global memory!

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

First Tile Use for Block (1,1)



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



Modifying the Tile Count

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

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;
}</pre>
```

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;
}</pre>
```

Modifying the Tile Use Code

We had ...

```
12. for (int k = 0; k < TILE_WIDTH; ++k)
13. Pvalue += subTileM[ty][k] * subTileN[k][tx];</pre>
```

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];
}</pre>
```

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;
}</pre>
```

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?

Global Memory (DRAM) Bandwidth

Ideal



Reality



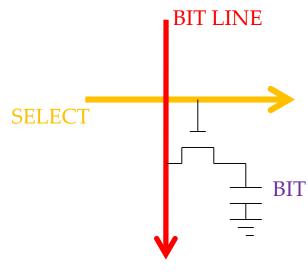
Most Large Memories Use DRAM

 Random Access Memory (RAM): same time needed to read/write any address

Dynamic RAM (DRAM):

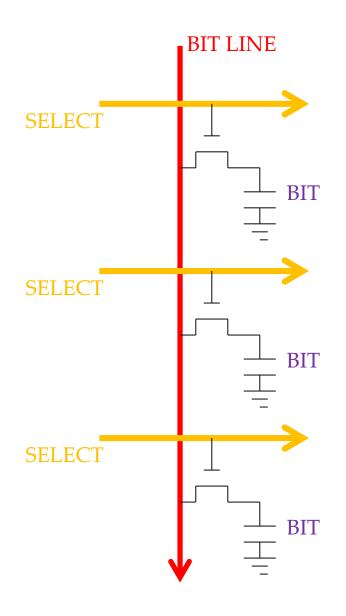
- bit stored on a capacitor
- connected via transistor to
 bit line for read/write
- bits disappear after a while

 (around 50 msec, due to tiny
 leakage currents through transistor),
 and must be rewritten (hence dynamic)



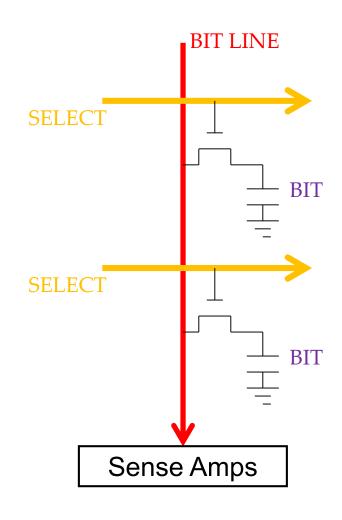
Many Cells (Bits) per Bit Line

- About 1,000 cells connect to each BIT LINE.
- Connection/disconnection depends on SELECT line.
- Some address bits decoded to connect exactly one cell to the BIT LINE.

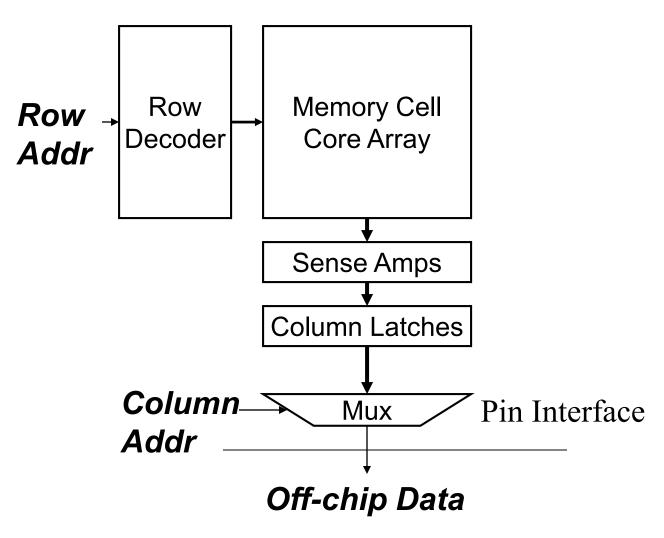


DRAM is Slow But Dense

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



DRAM Bank Organization

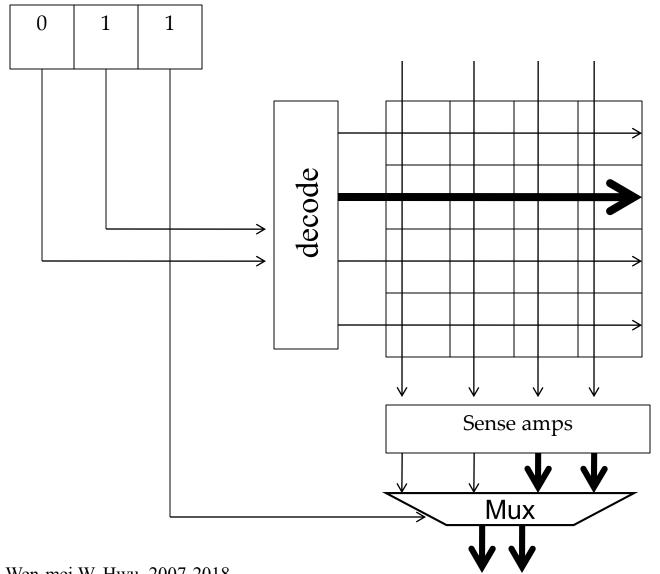


- SELECT lines connect to about 1,000 bit lines.
- Core array has about O(1M) bits
- Use more address bits to choose bit line(s).

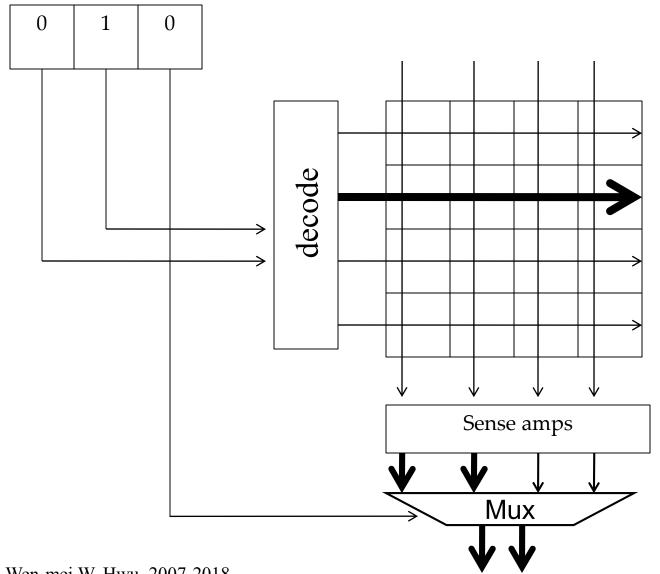
DRAM Interfaces are Clocked

- DRAM cells are not clocked (clocking requires transistors).
- DRAM interfaces are clocked.
 - DDR: Core speed = ½ interface speed
 - DDR2/GDDR3: Core speed = ¼ interface speed
 - DDR3/GDDR4: Core speed = ½ interface speed
 - likely to be worse in the future

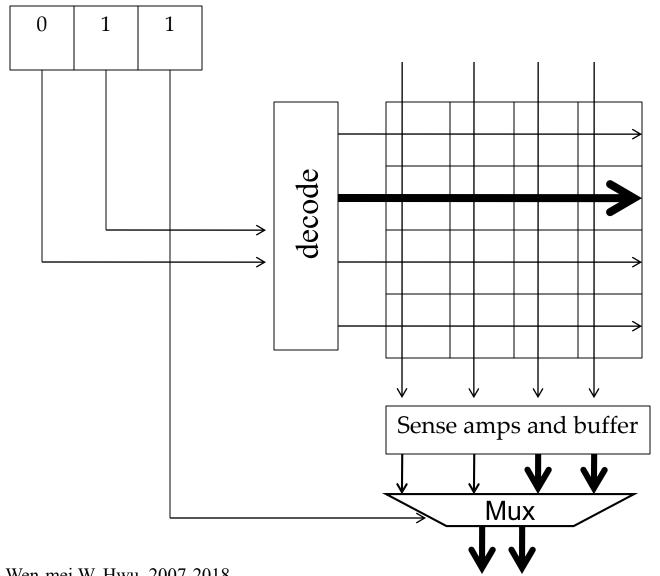
A very small (8x2 bit) DRAM Bank



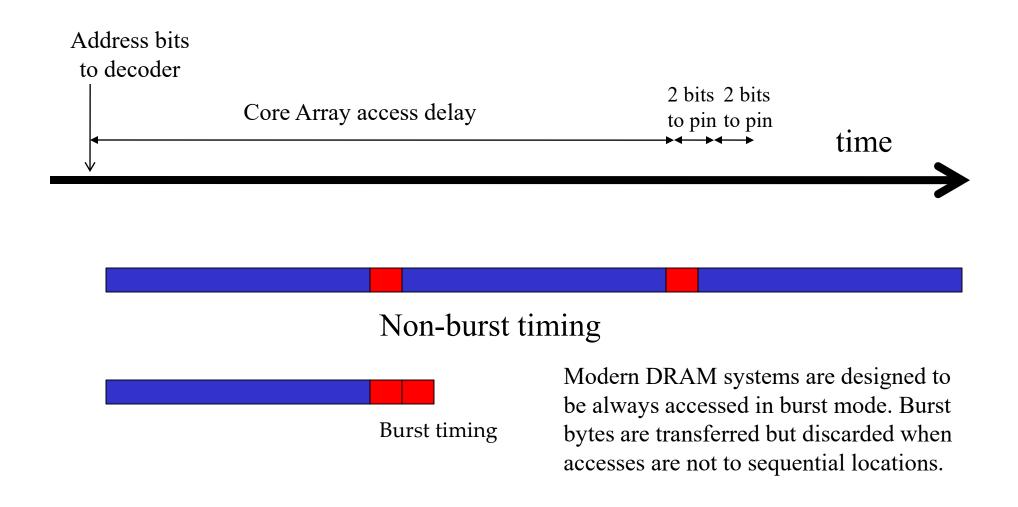
DRAM Bursting (burst size = 4 bits)



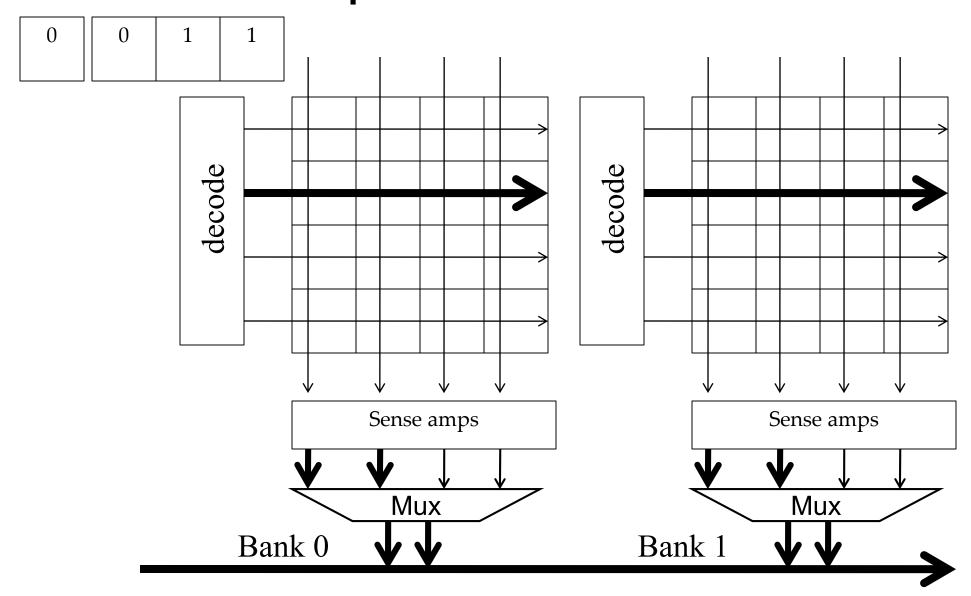
Second part of the burst



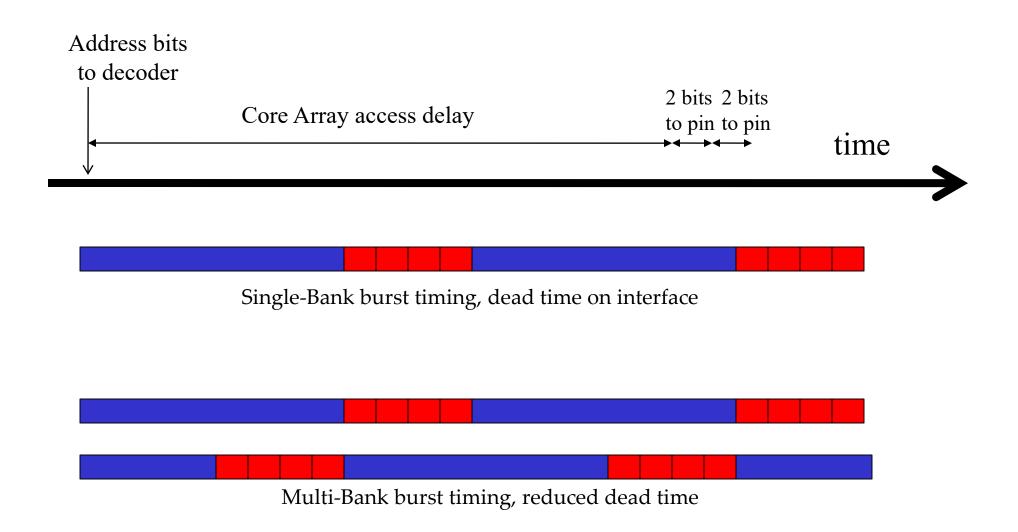
DRAM Bursting for the 8x2 Bank



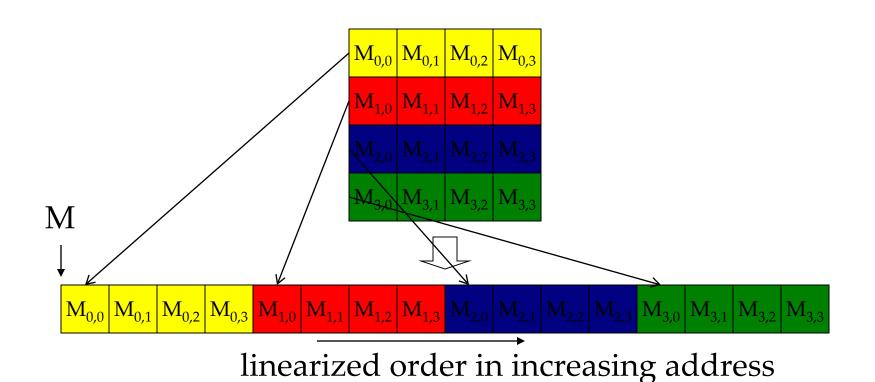
Multiple DRAM Banks



DRAM Bursting for the 8x2 Bank



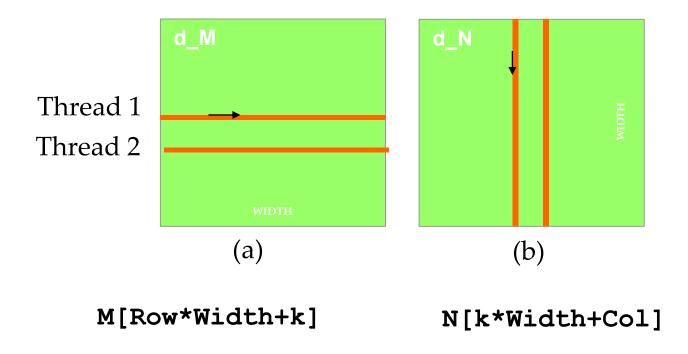
Placing a 2D C array into linear memory space (review)



A Simple Matrix Multiplication Kernel (review)

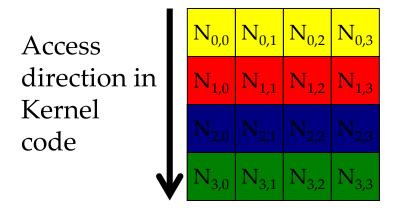
```
global void MatrixMulKernel(float* M, float* N, float* P, int Width)
// Calculate the row index of the P element and M
int Row = blockIdx.y * blockDim.y + threadIdx.y;
// Calculate the column index of P and N
int Col = blockIdx.x * blockDim.x + threadIdx.x;
if ((Row < Width) && (Col < Width)) {
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

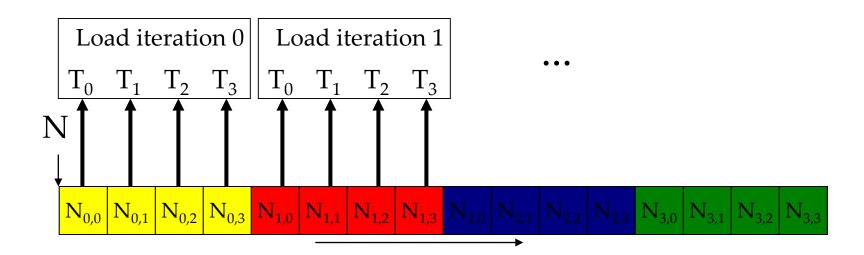
Two Access Patterns



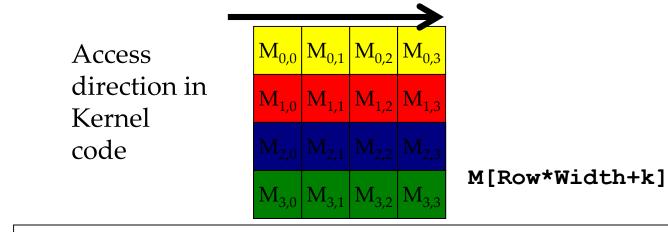
k is loop counter in the inner product loop of the kernel code

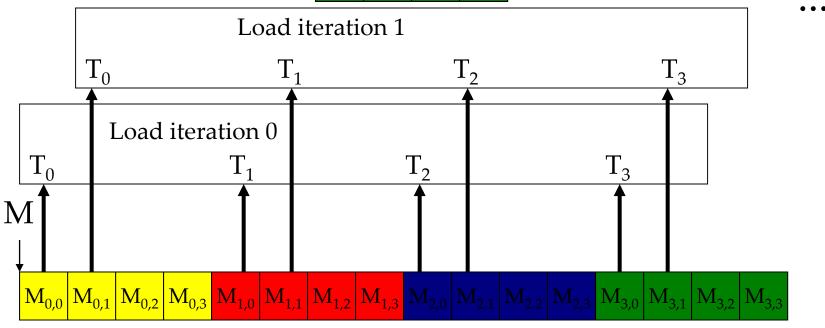
N accesses are coalesced





M accesses are not coalesced



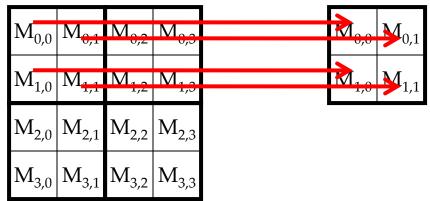


Work for Block (0,0) Step 0

Shared Memory

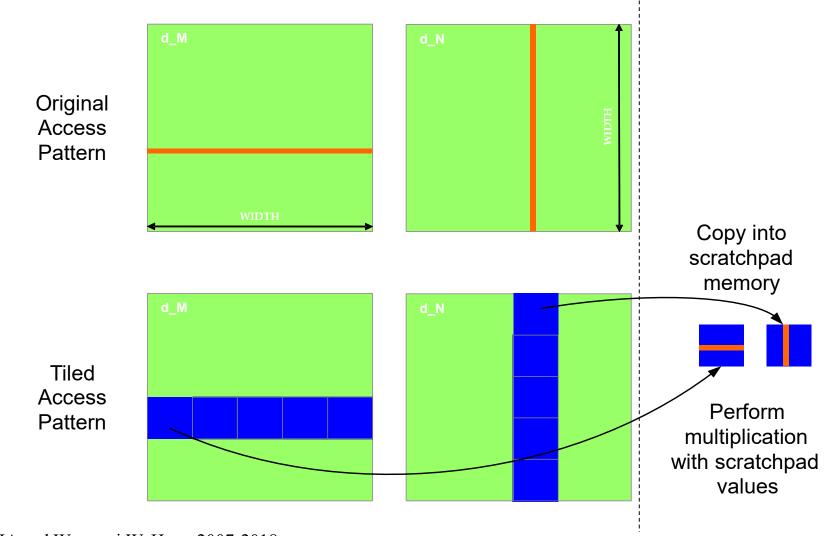


Shared Memory



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}

Use shared memory to enable coalescing in tiled matrix multiplication



ANY MORE QUESTIONS? READ CHAPTER 5