

Tiled Matrix Multiplication

Basic Matrix Multiplication Kernel

```
__global__
void MatrixMulKernel(int m, int n, int k, float* A, float*
B, float* C)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

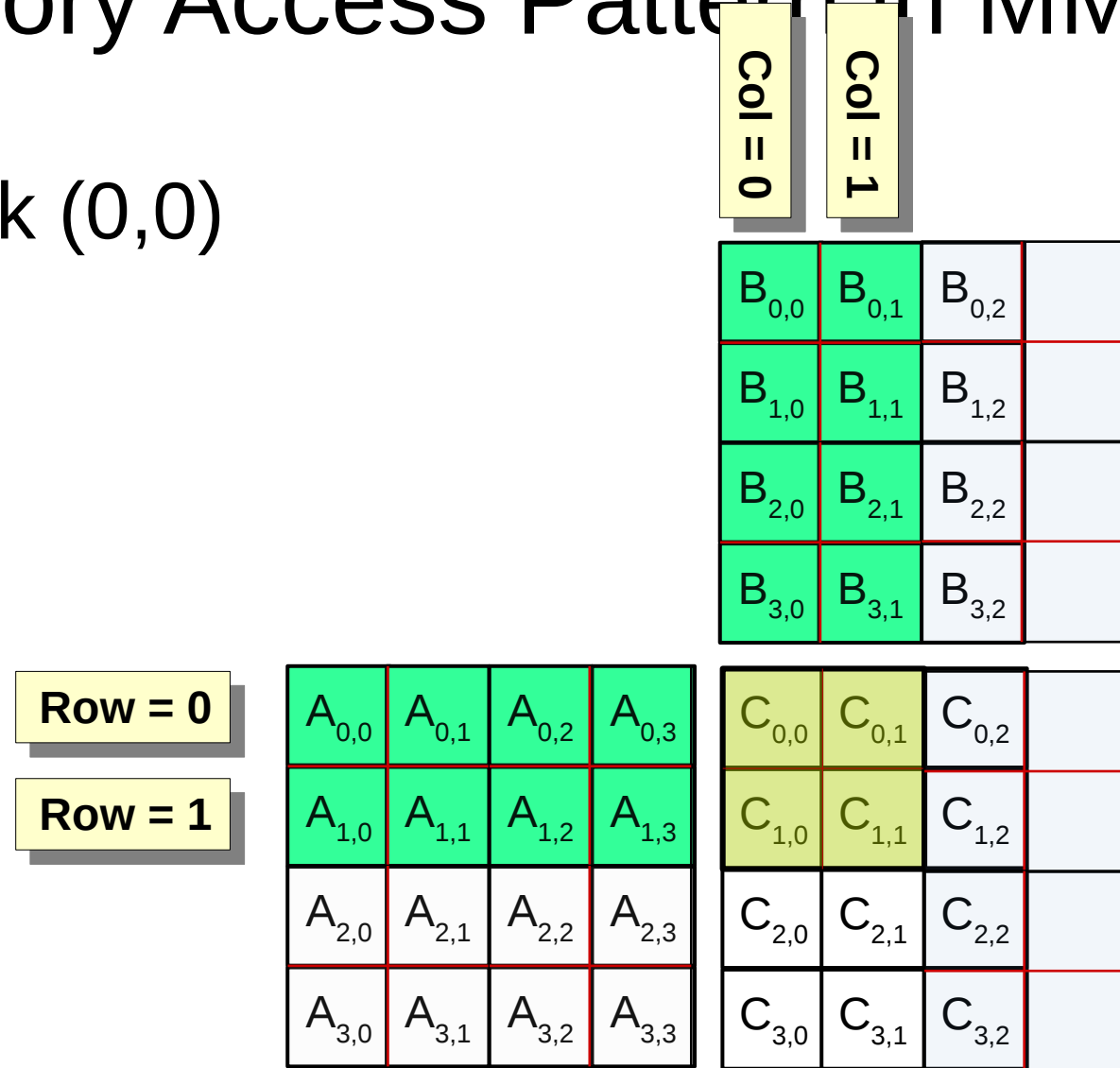
    if ((Row < m) && (Col < k)) {

        float Cvalue = 0.0;
        for (int i = 0; i < n; ++i)
            /* A[Row, i] and B[i, Col] */
            Cvalue += A[Row*n+i] * B[Col+i*k];

        C[Row*k+Col] = Cvalue;
    }
}
```

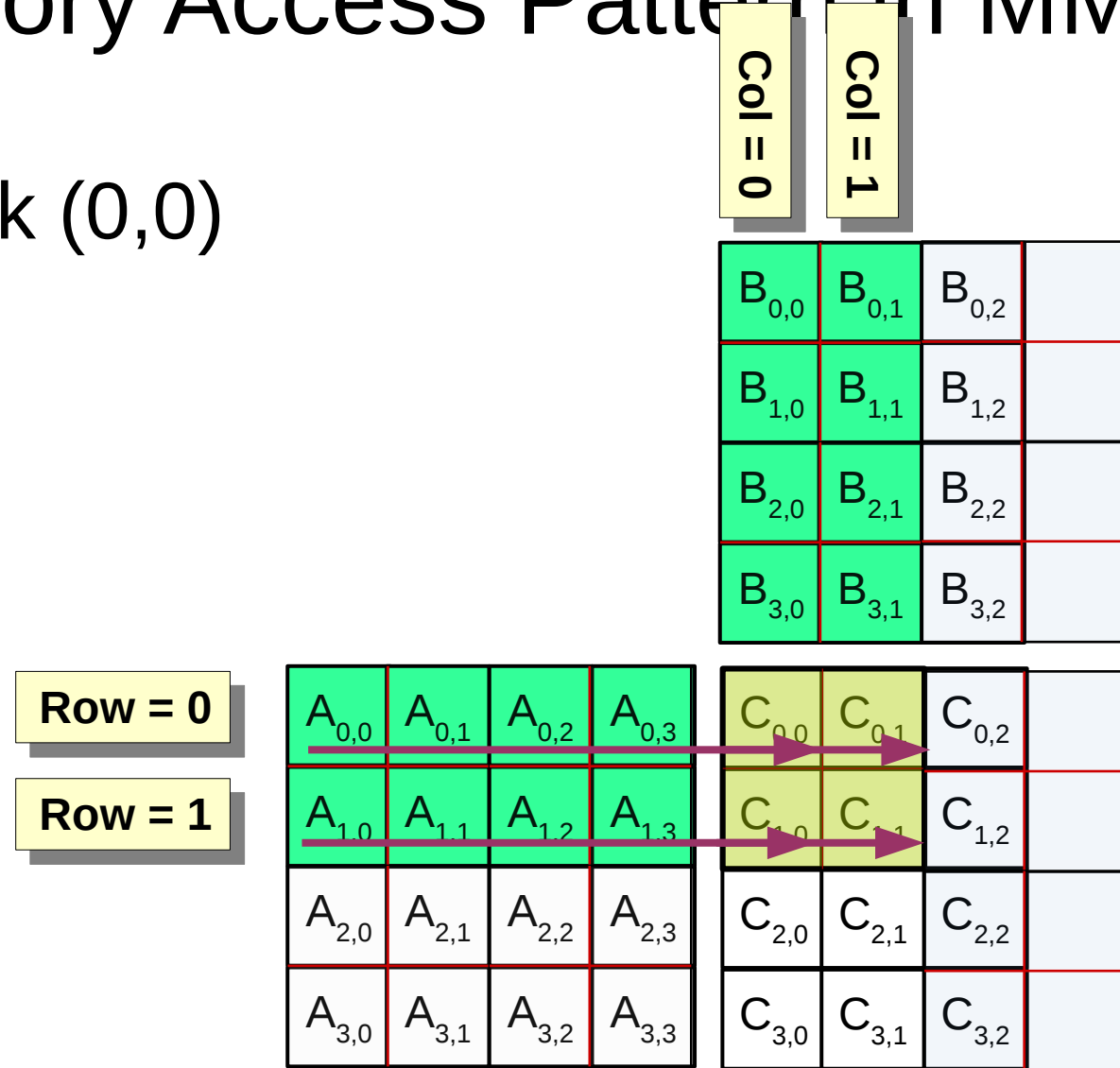
Global Memory Access Pattern in MM

- Work in Block (0,0)



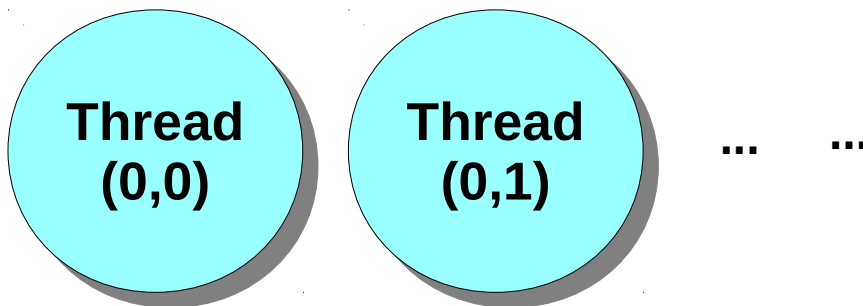
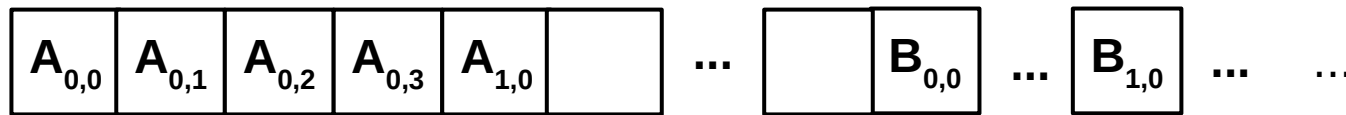
Global Memory Access Pattern in MM

- Work in Block (0,0)



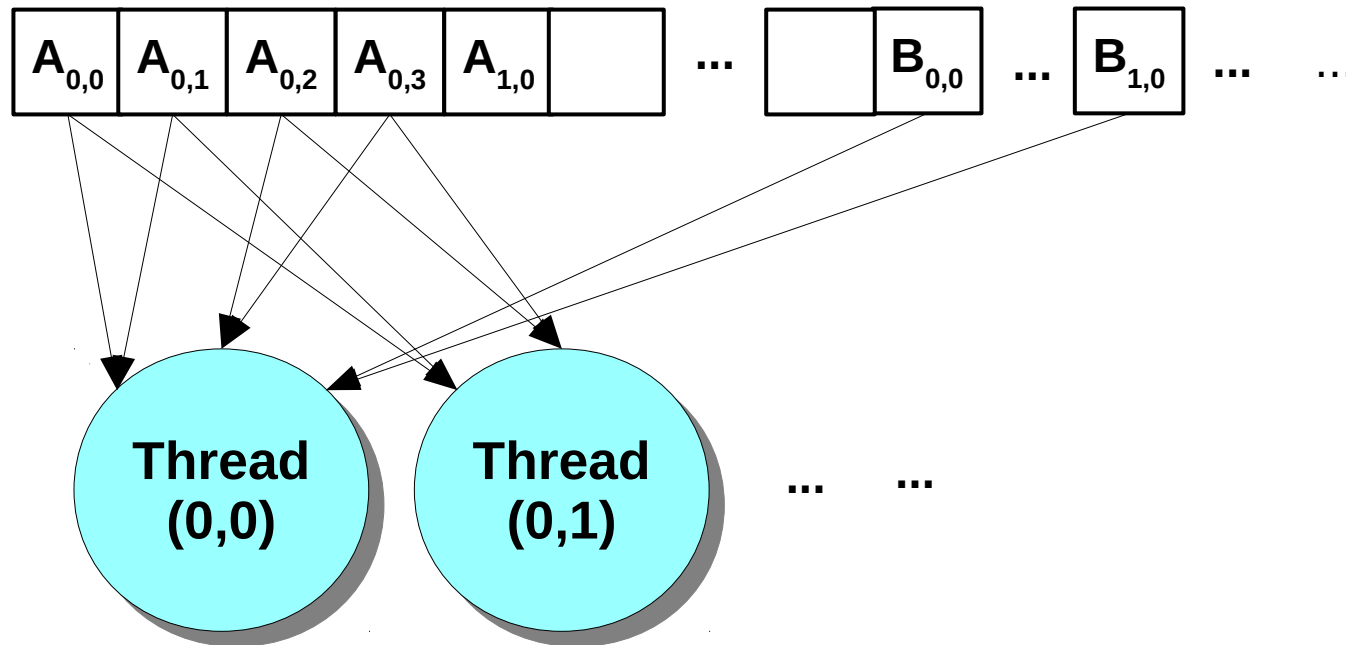
Global Memory Access Pattern in MM

- Consider threads (0,0) and (0,1)



Global Memory Access Pattern in MM

- Consider threads (0,0) and (0,1)



Performance on Fermi GPU

- All threads access global memory for their input matrix elements

Performance on Fermi GPU

- All threads access global memory for their input matrix elements
- Two memory accesses ($A_{0,0}, B_{0,0} = 8$ bytes) per floating point multiply-add (2 ops)

Performance on Fermi GPU

- All threads access global memory for their input matrix elements
- Two memory accesses ($A_{0,0}, B_{0,0} = 8$ bytes) per floating point multiply-add (2 ops)
 - 4B of memory read per FLOP

Performance on Fermi GPU

- All threads access global memory for their input matrix elements
- Two memory accesses ($A_{0,0}, B_{0,0} = 8$ bytes) per floating point multiply-add (2 ops)
 - 4B of memory read per FLOP
- Peak floating-point rate is 1TF = 1,000 GFLOPS
 - $4 * 1,000 = 4,000$ GB/s required to achieve peak FLOP rating

Performance on Fermi GPU

- Peak floating-point rate is 1TF = 1,000 GFLOPS
 - $4 * 1,000 = 4,000$ GB/s required to achieve peak FLOP rating
- Reality – Fermi supports 150 GB/s

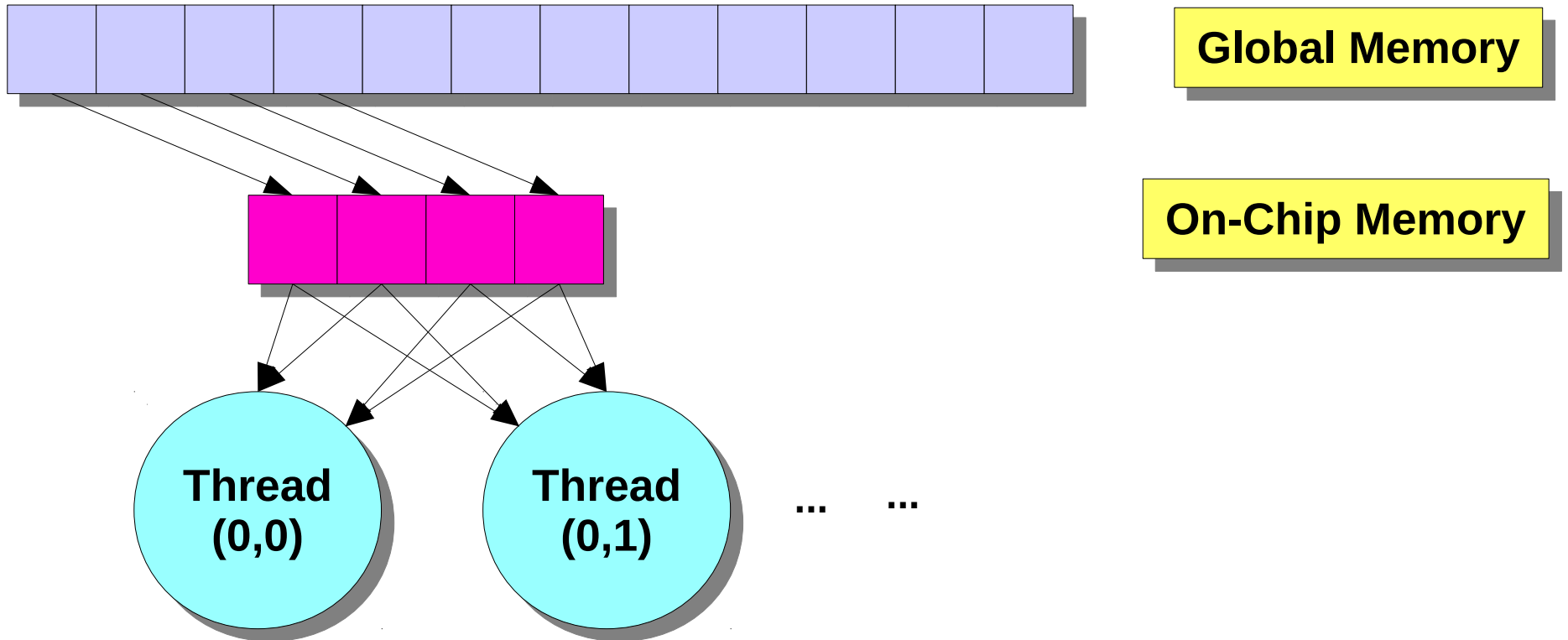
Performance on Fermi GPU

- Peak floating-point rate is 1TF = 1,000 GFLOPS
 - $4 \times 1,000 = 4,000$ GB/s required to achieve peak FLOP rating
- Reality – Fermi supports 150 GB/s
 - Upper FLOPs limit: 37.5 GFLOPS
 - The actual code runs at about 25 GFLOPS

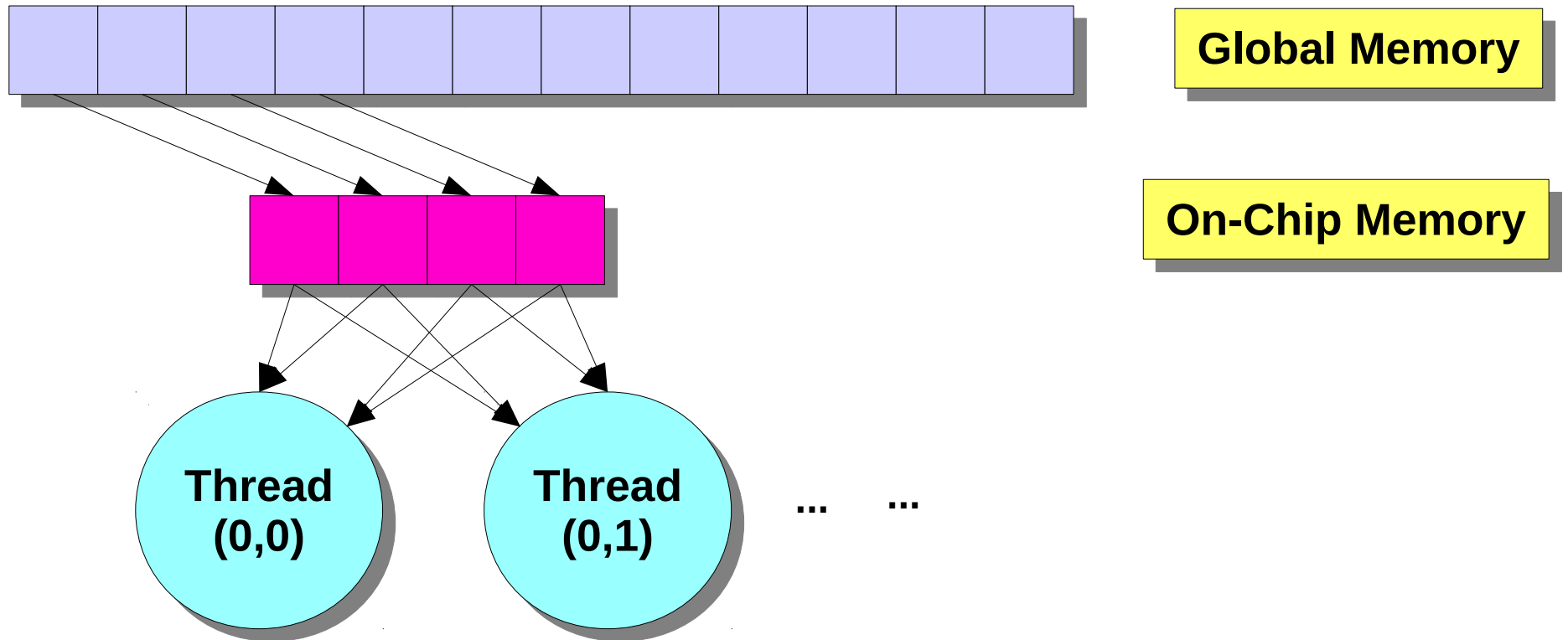
Performance on Fermi GPU

- Peak floating-point rate is 1TF = 1,000 GFLOPS
 - $4 \times 1,000 = 4,000$ GB/s required to achieve peak FLOP rating
- Reality – Fermi supports 150 GB/s
 - Upper FLOPs limit: 37.5 GFLOPS
 - The actual code runs at about 25 GFLOPS
- Need to cut down global memory accesses to get close to 1TF
 - Compute-to-Global-Memory-Access Ratio

Shared Memory Tiling/Blocking



Shared Memory Tiling/Blocking



**Divide the global memory content into Tiles.
Focus the computation of threads on one or a small number of tiles at each point in time.**

Basic Concept of Blocking/Tiling

- More efficient if tiled data exhibit good spatial locality

Basic Concept of Blocking/Tiling

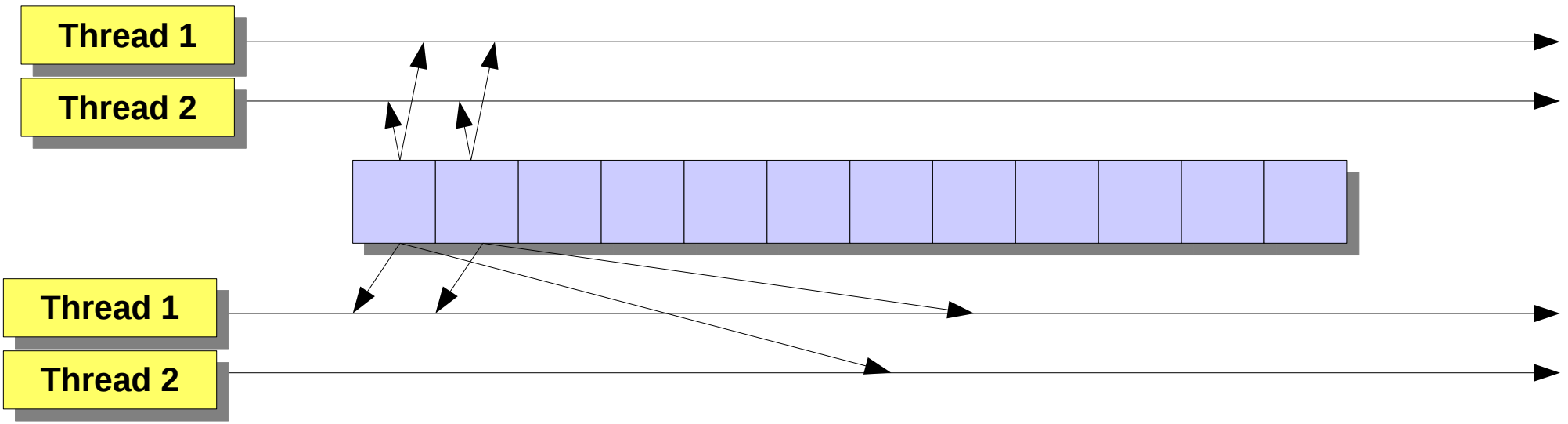
- More efficient if tiled data exhibit good spatial locality
- SM should be large enough to accommodate all the data

Basic Concept of Blocking/Tiling

- More efficient if tiled data exhibit good spatial locality
- SM should be large enough to accommodate all the data
- Needs Synchronization

Basic Concept of Blocking/Tiling

- More efficient if tiled data exhibit good spatial locality
- SM should be large enough to accommodate all the data
- Needs Synchronization



Tiling Technique

- Identify a tile of global memory content that are accessed by multiple threads

Tiling Technique

- Identify a tile of global memory content that are accessed by multiple threads
- Load the tile from global memory into on-chip memory

Tiling Technique

- Identify a tile of global memory content that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory

Tiling Technique

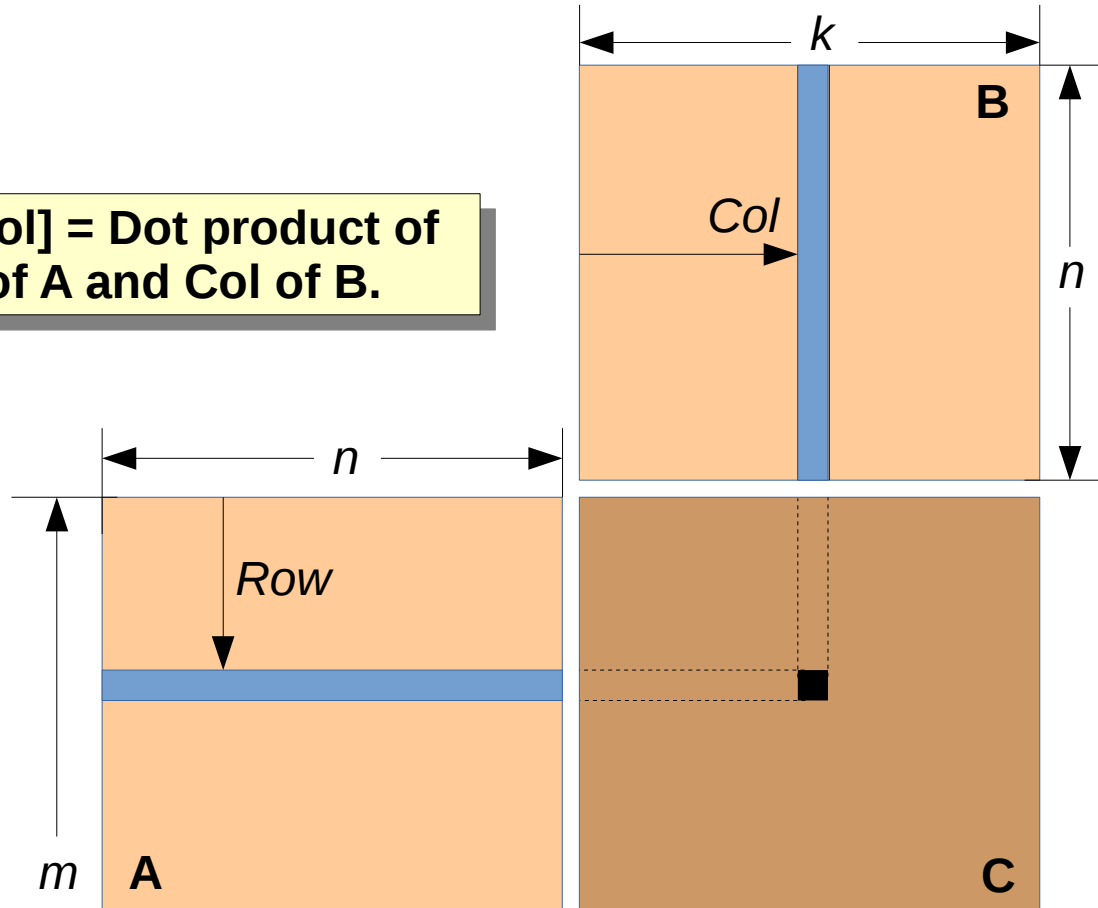
- Identify a tile of global memory content that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Have the multiple threads to access their data from the on-chip memory
- Move on to the next tile

Tiled Matrix Multiplication

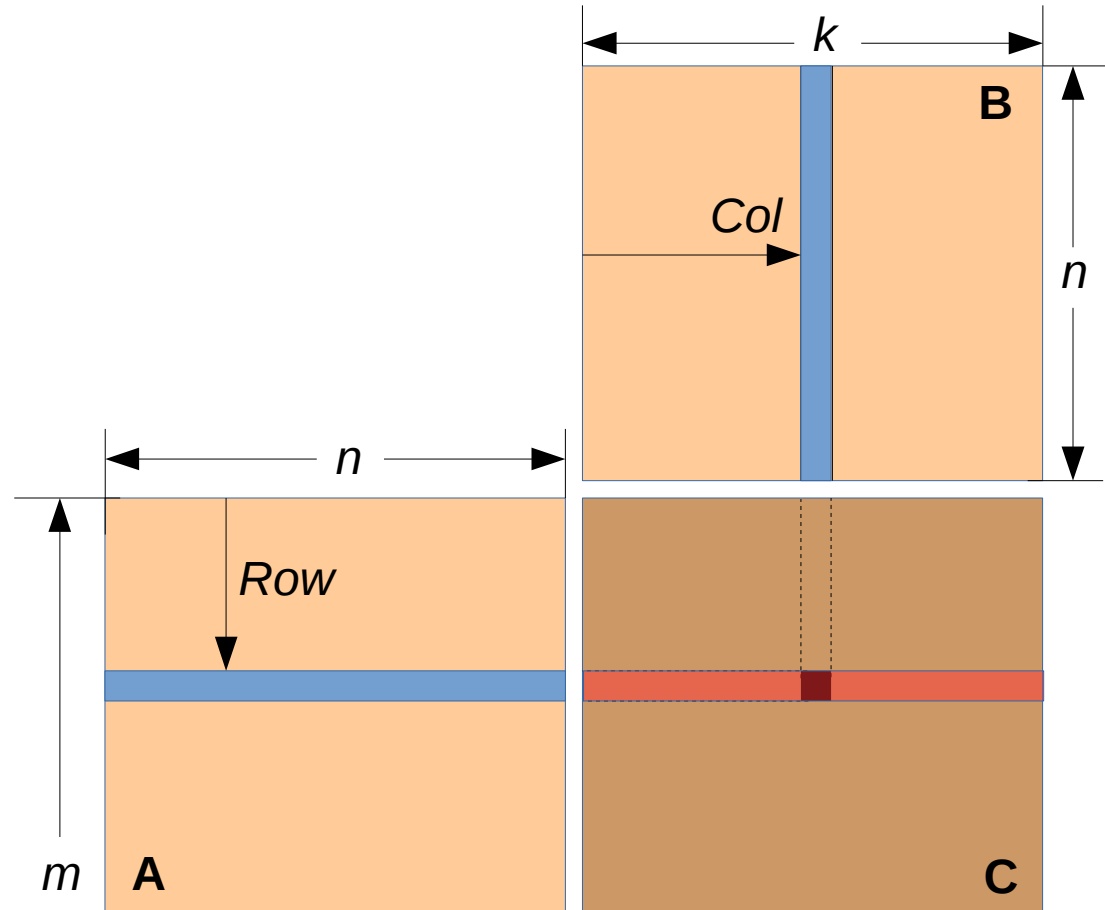
- Loading a tile
- Phased Execution
- Barrier Synchronization

Basic Matrix Multiplication

$C[\text{Row}, \text{Col}] = \text{Dot product of Row of A and Col of B.}$

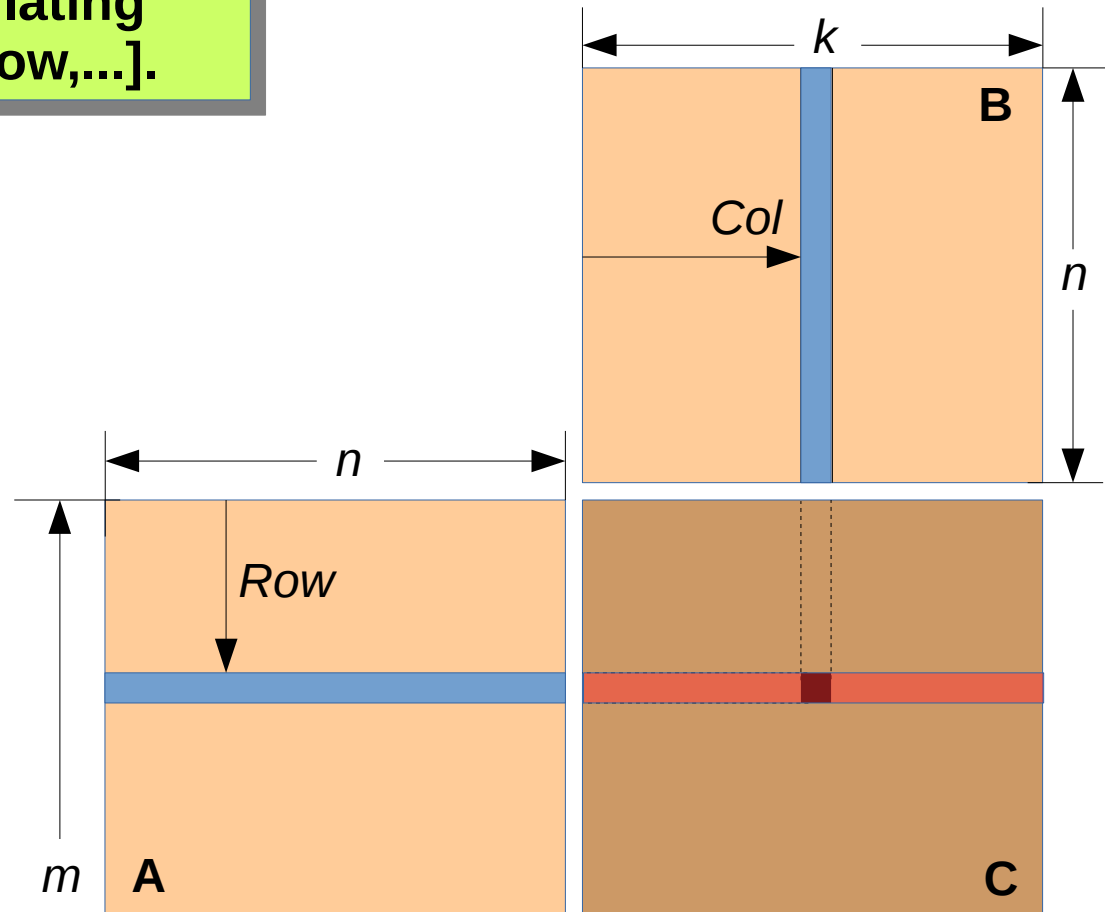


Matrix Multiplication



Matrix Multiplication

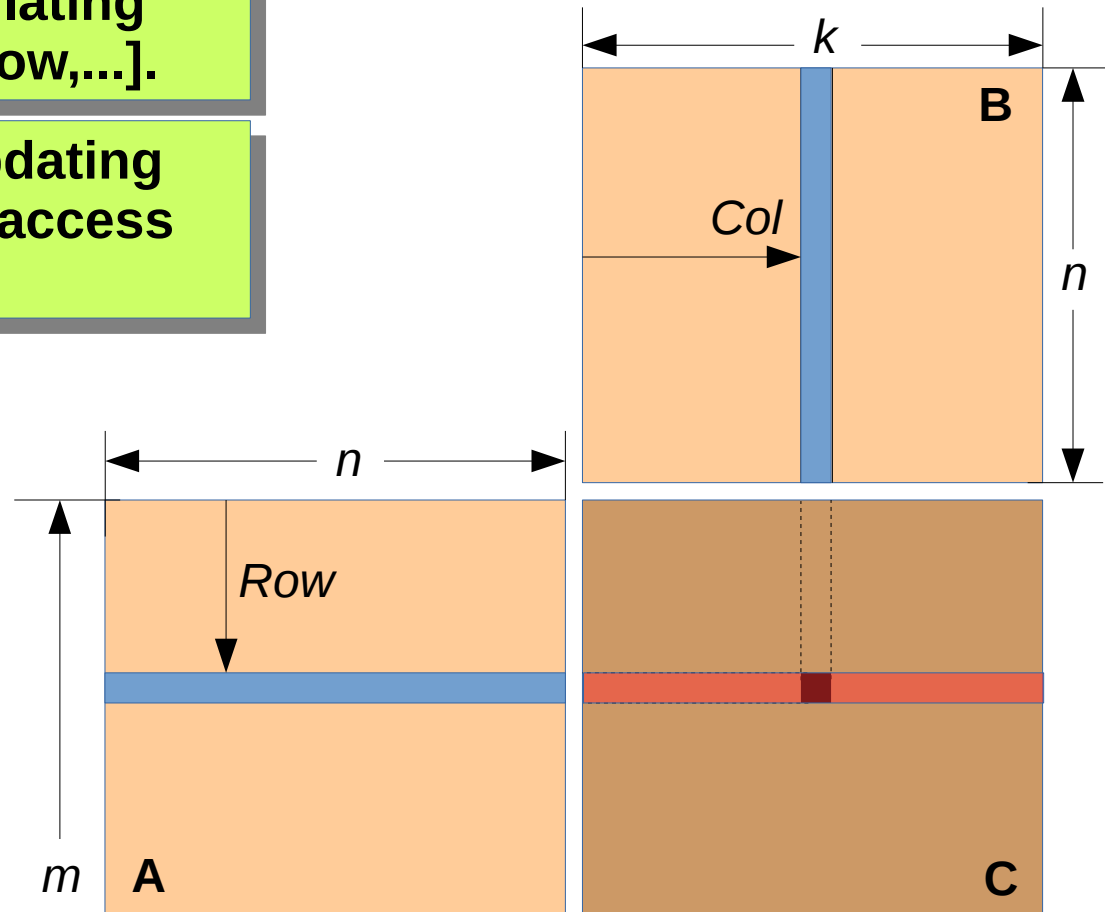
Elements $A[\text{Row}, \dots]$ are used in calculating elements $C[\text{Row}, \dots]$.



Matrix Multiplication

Elements $A[\text{Row}, \dots]$ are used in calculating elements $C[\text{Row}, \dots]$.

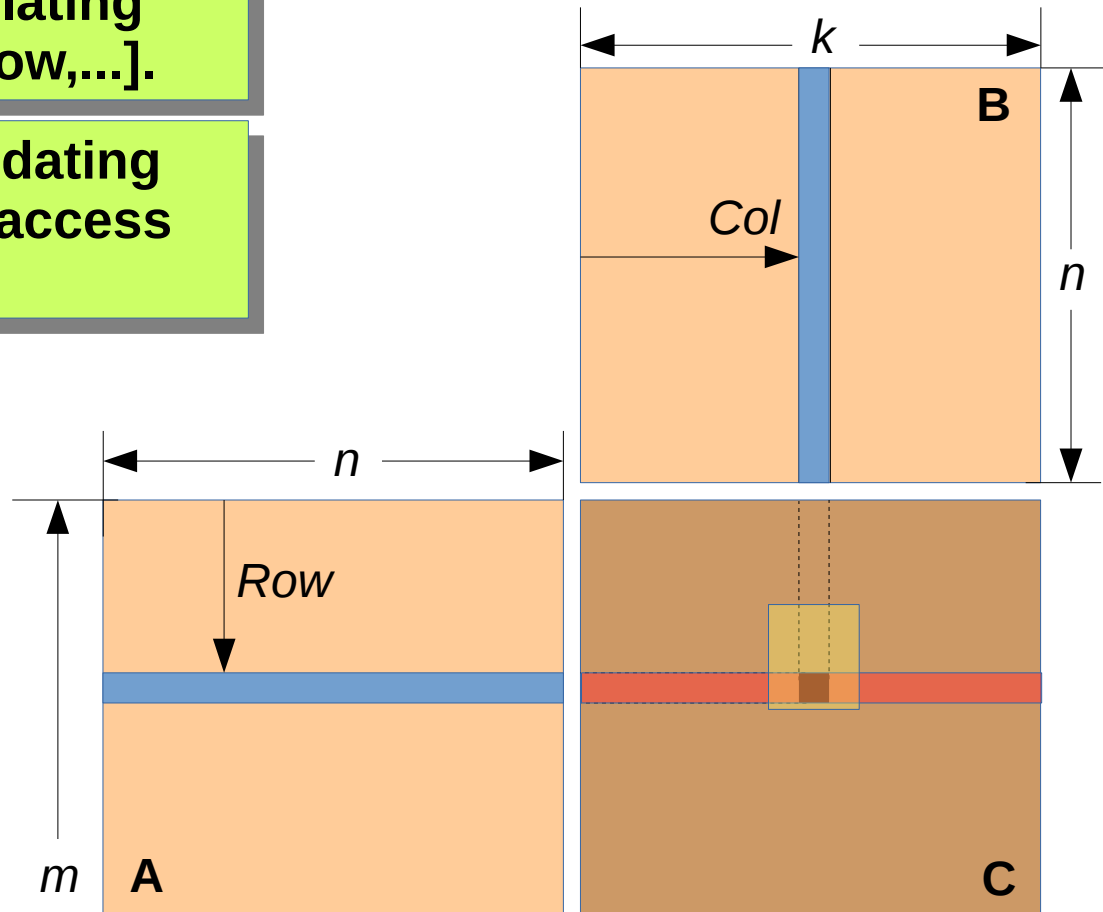
All threads updating $C[\text{Row}, \dots]$ will access $A[\text{Row}]$



Matrix Multiplication

Elements $A[\text{Row}, \dots]$ are used in calculating elements $C[\text{Row}, \dots]$.

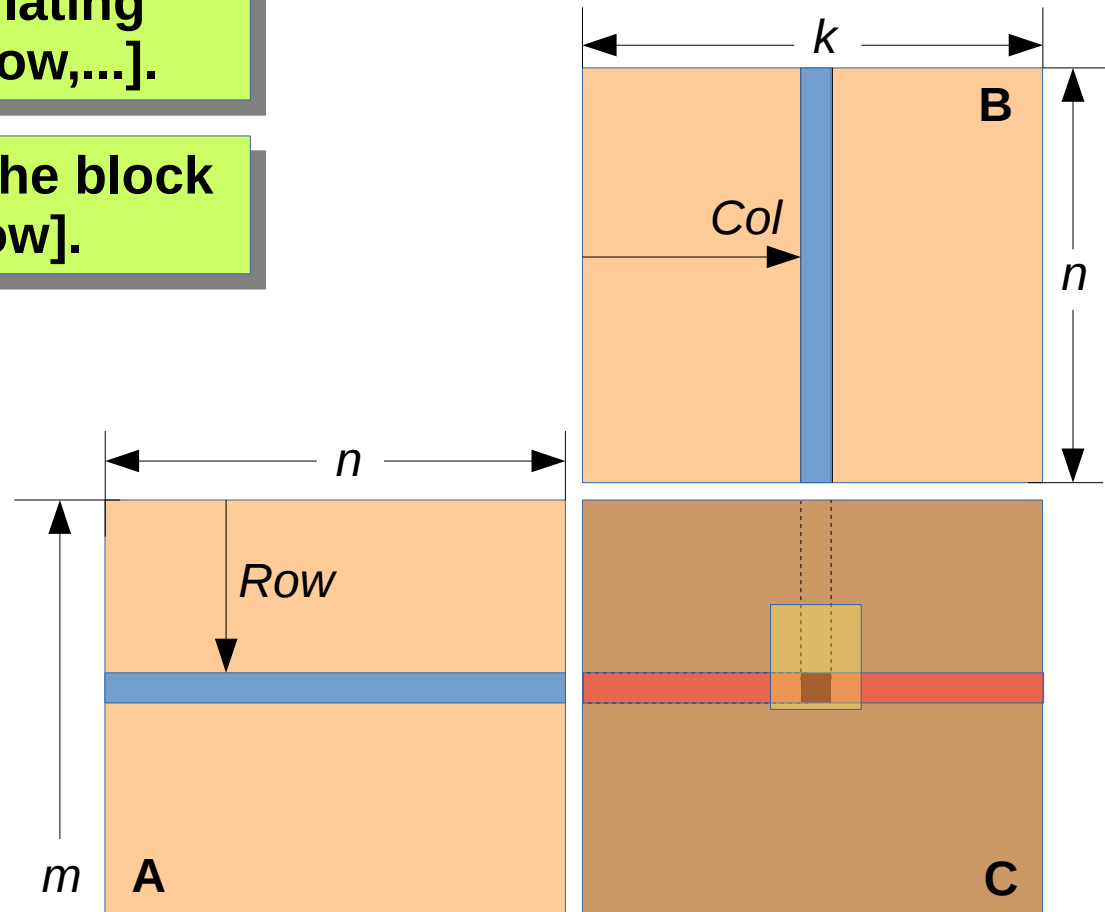
All threads updating $C[\text{Row}, \dots]$ will access $A[\text{Row}]$



Matrix Multiplication

Elements $A[\text{Row}, \dots]$ are used in calculating elements $C[\text{Row}, \dots]$.

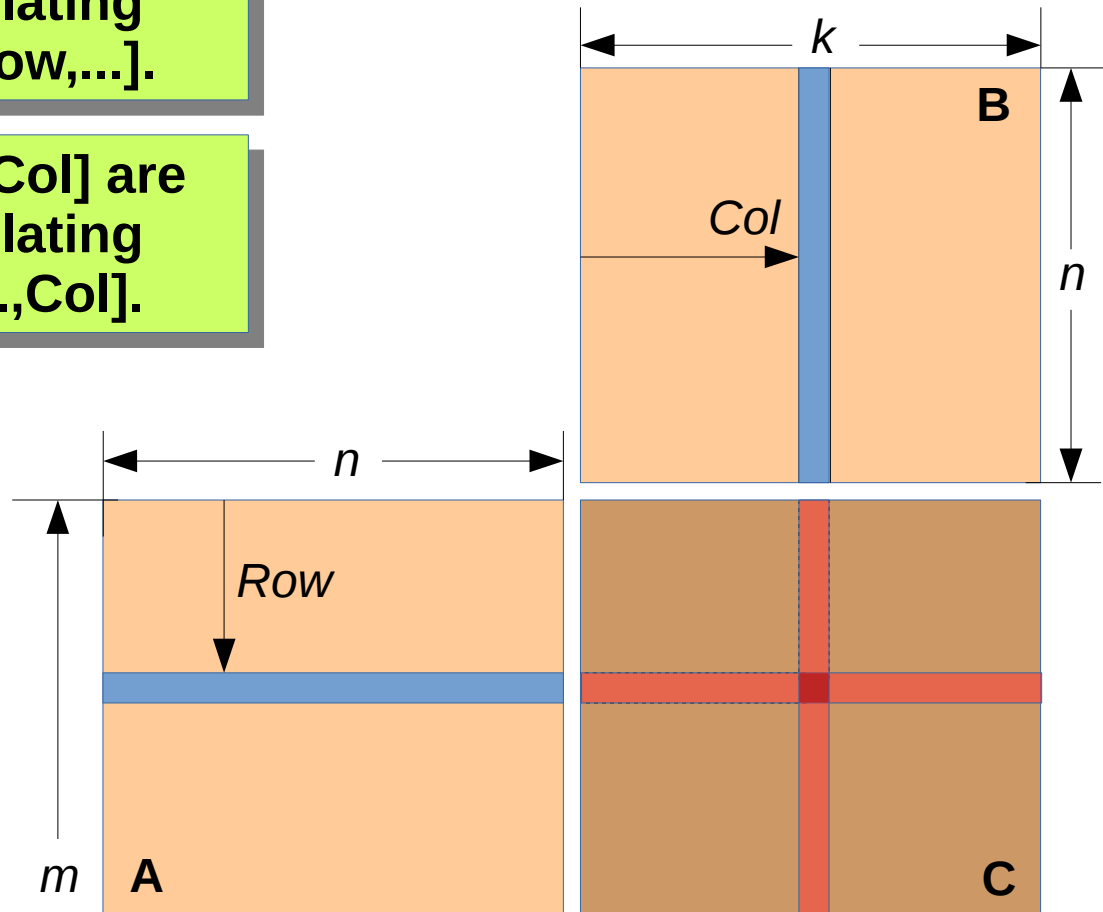
Each thread in the block loads $A[\text{Row}]$.



Matrix Multiplication

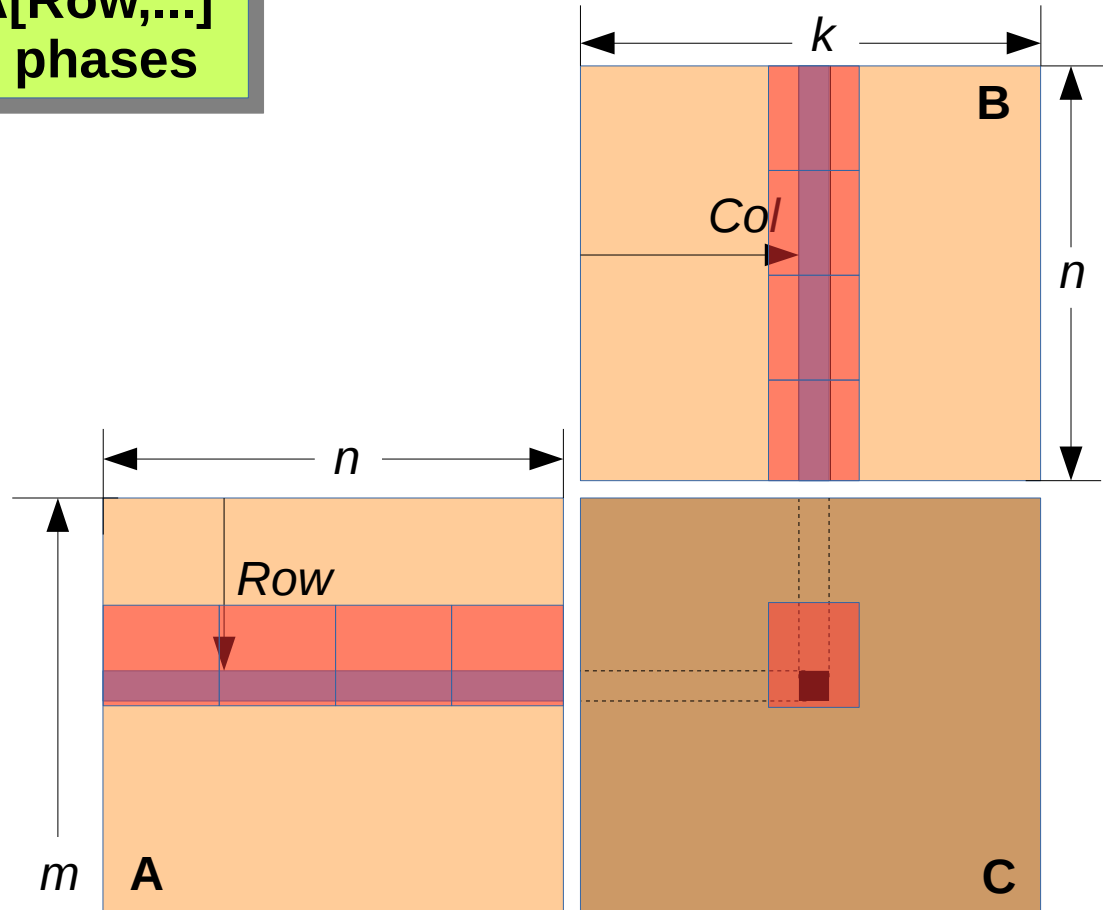
Elements $A[\text{Row}, \dots]$ are used in calculating elements $C[\text{Row}, \dots]$.

Elements $B[\dots, \text{Col}]$ are used in calculating elements $C[\dots, \text{Col}]$.



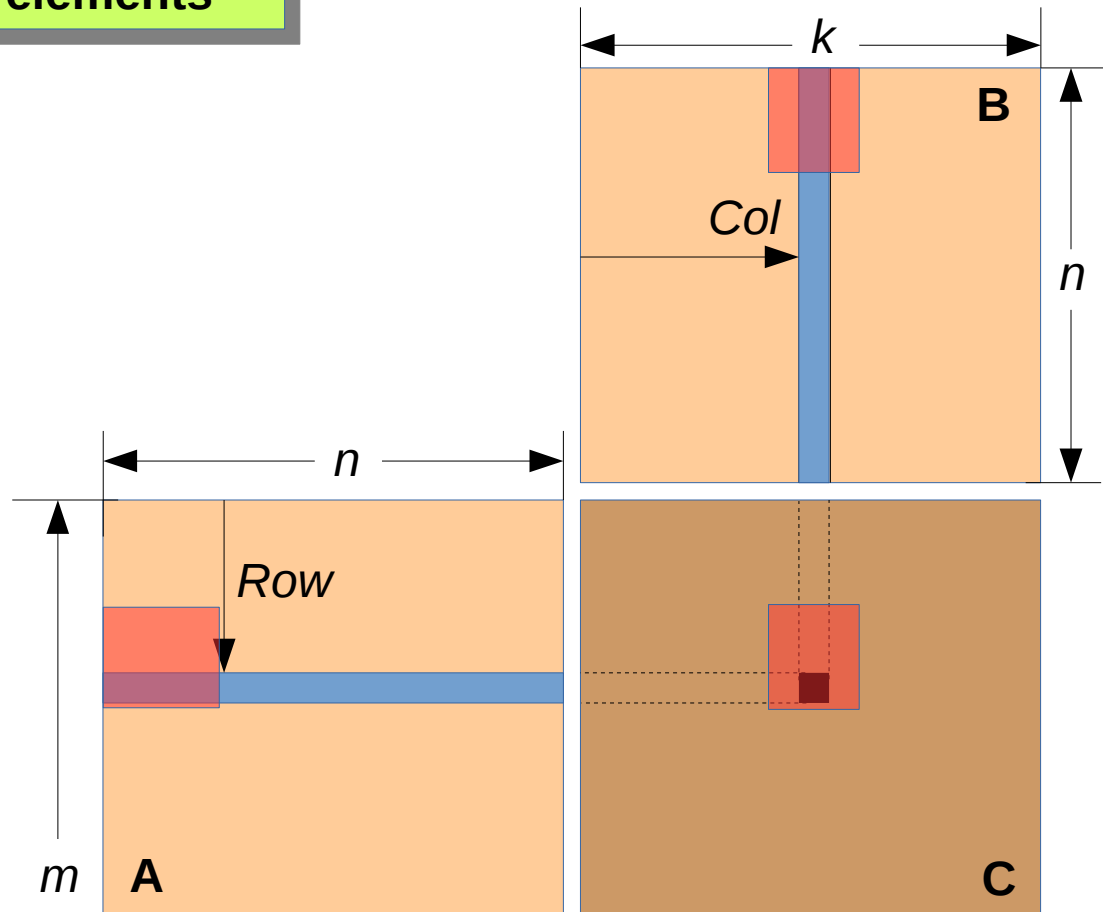
Tiled Matrix Multiplication

Compromise: Load elements from $A[\text{Row}, \dots]$ and $B[\dots, \text{Col}]$ in phases



Tiled Matrix Multiplication

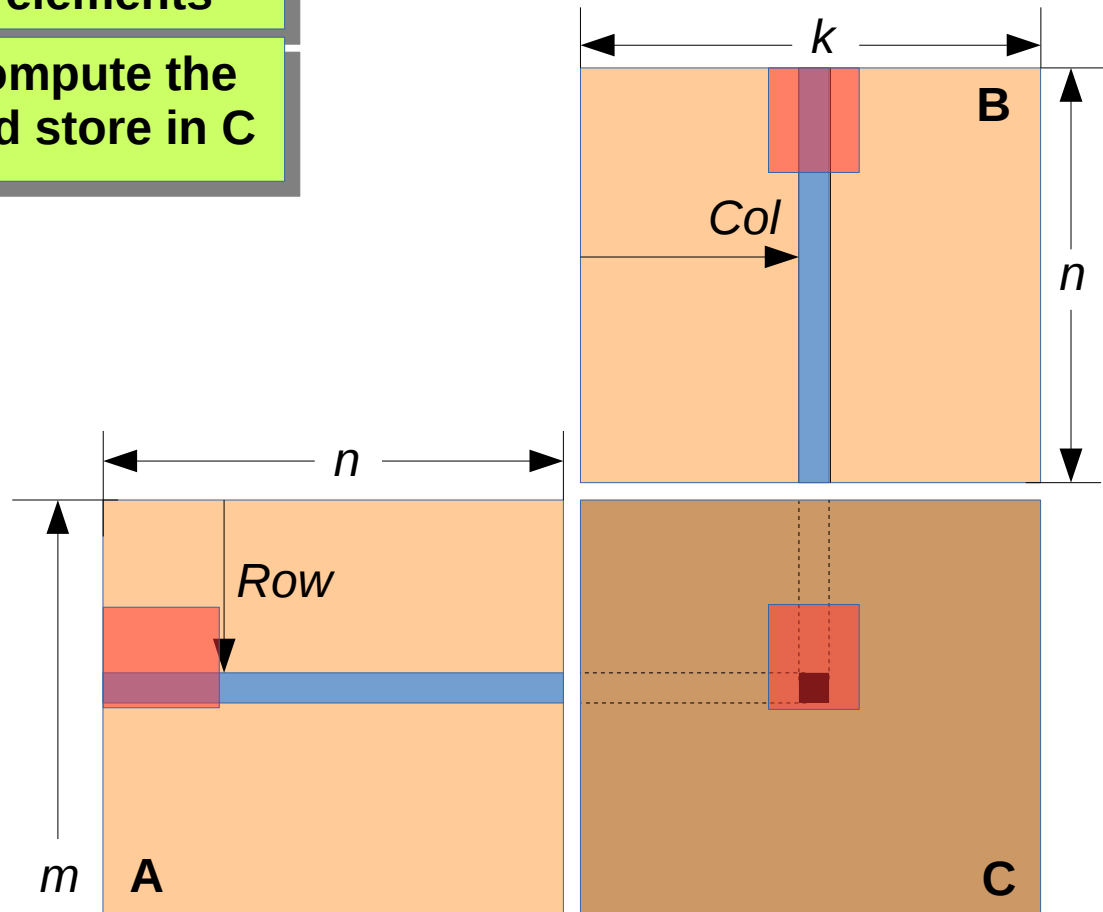
All the threads participate in loading A and B elements



Tiled Matrix Multiplication

All the threads participate in loading A and B elements

All the threads compute the partial product and store in C

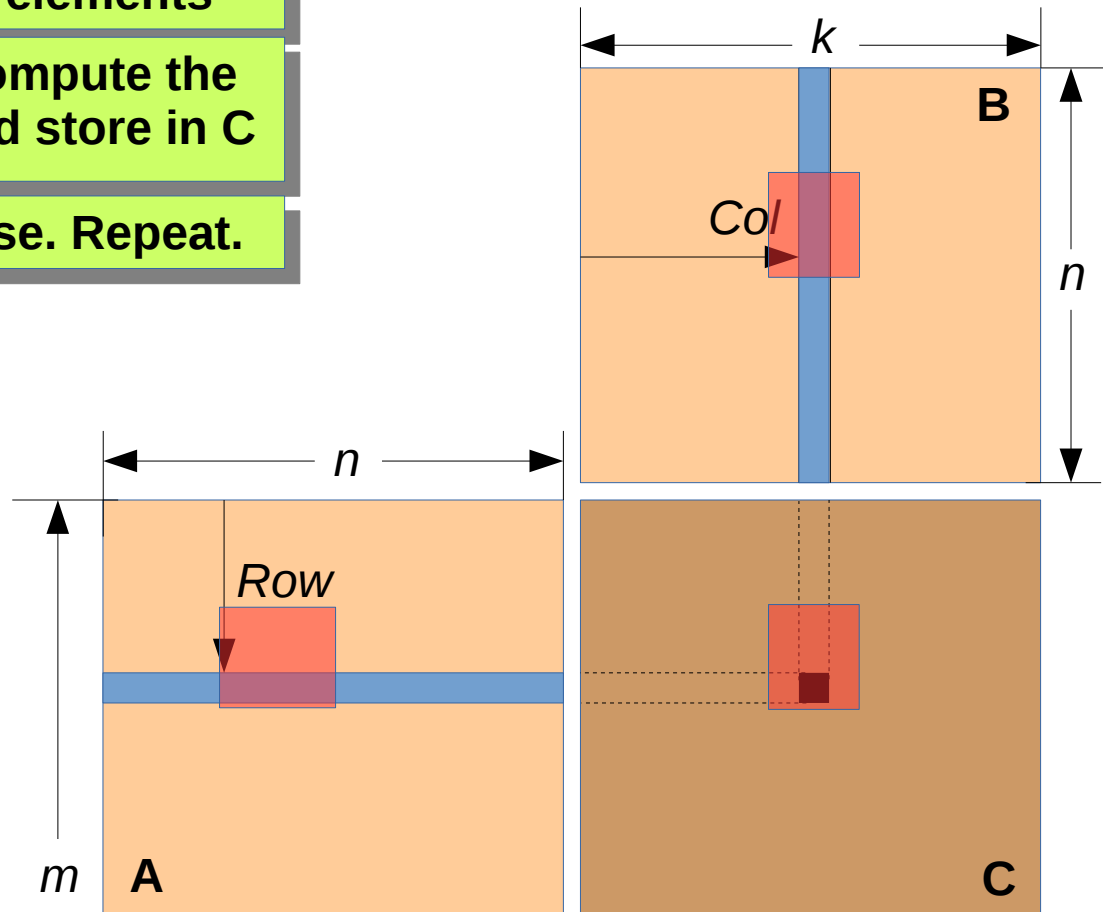


Tiled Matrix Multiplication

All the threads participate in loading A and B elements

All the threads compute the partial product and store in C

Move to next phase. Repeat.



Tiled Matrix Multiplication

- All threads in a block participate
 - Each thread loads one A element and one B element in tiled code

Tiled Matrix Multiplication

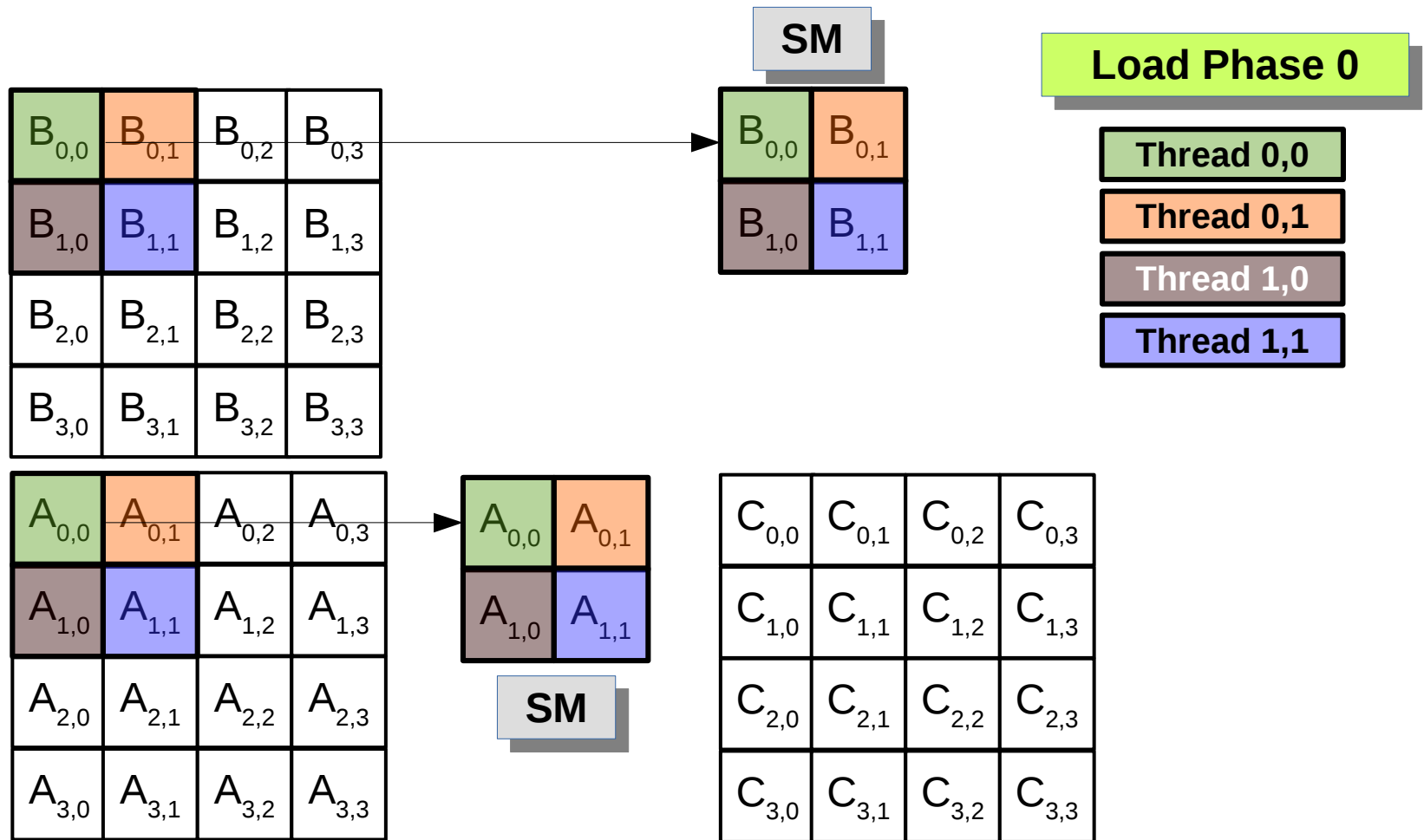
Rows in A, and Cols in B needed for threads in Block (0,0)

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

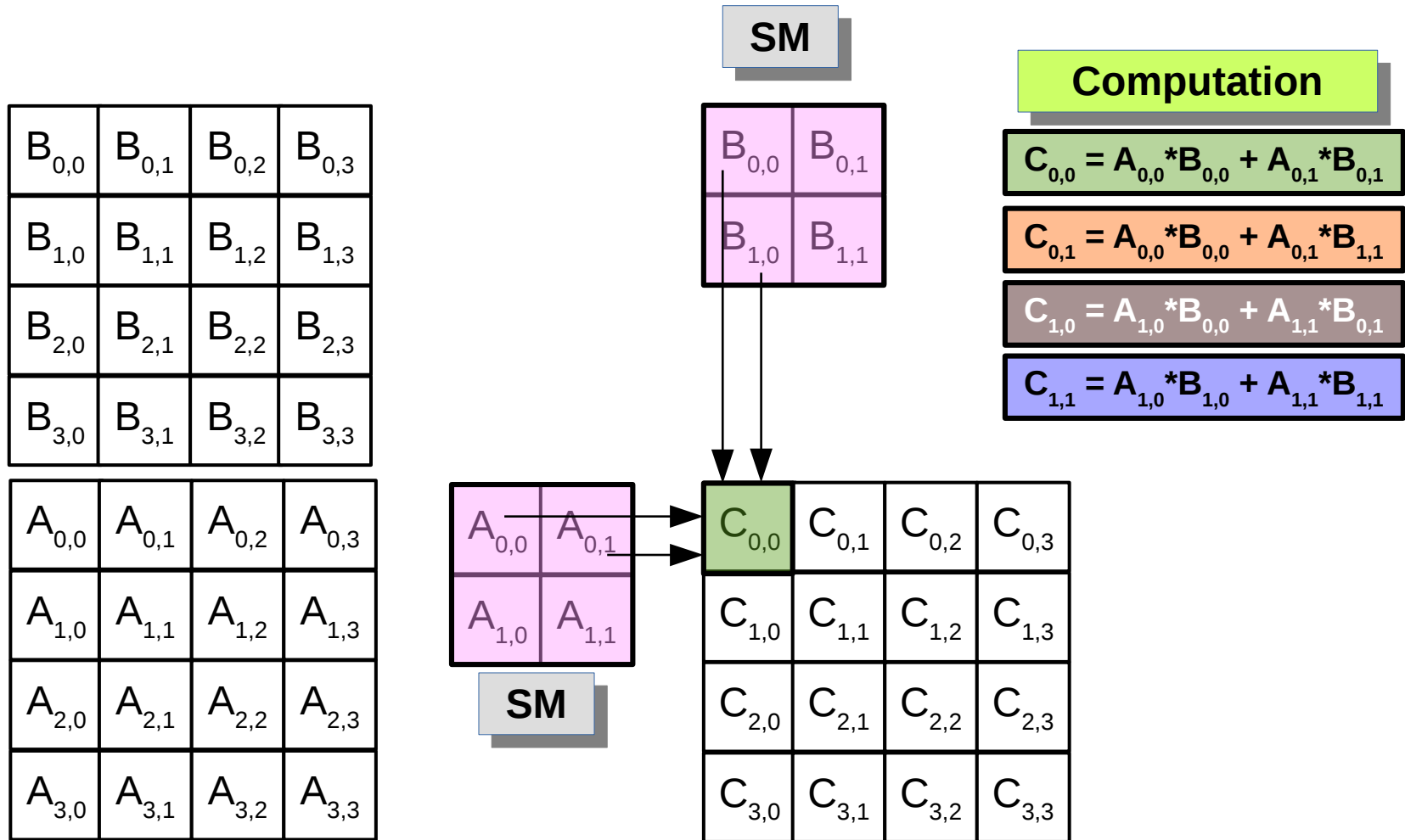
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

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

Tiled Matrix Multiplication



Tiled Matrix Multiplication



Tiled Matrix Multiplication

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

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

SM

$B_{0,0}$	$B_{0,1}$
$B_{1,0}$	$B_{1,1}$

$A_{0,0}$	$A_{0,1}$
$A_{1,0}$	$A_{1,1}$

SM

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

Computation

$$C_{0,0} = A_{0,0} * B_{0,0} + A_{0,1} * B_{0,1}$$

$$C_{0,1} = A_{0,0} * B_{0,0} + A_{0,1} * B_{1,1}$$

$$C_{1,0} = A_{1,0} * B_{0,0} + A_{1,1} * B_{0,1}$$

$$C_{1,1} = A_{1,0} * B_{1,0} + A_{1,1} * B_{1,1}$$

Needs 2 iterations

Tiled Matrix Multiplication

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

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

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

SM

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

SM

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

Load Phase 1

Thread 0,0

Thread 0,1

Thread 1,0

Thread 1,1

Tiled Matrix Multiplication

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

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

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

SM

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

SM

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

Computation

$$C_{0,0} += A_{0,2} * B_{2,0} + A_{0,3} * B_{3,0}$$

$$C_{0,1} += A_{0,2} * B_{2,1} + A_{0,3} * B_{3,1}$$

$$C_{1,0} += A_{1,2} * B_{2,0} + A_{1,3} * B_{3,0}$$

$$C_{1,1} += A_{1,2} * B_{2,1} + A_{1,3} * B_{3,1}$$

Matrix Multiplication Kernel

Shared Memory Variable Declaration

```
__global__  
void MatrixMulKernel(int m, int n, int k, float* A,  
float* B, float* C)  
{  
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];  
}
```

Tiled Matrix Multiplication

- How many memory accesses are reduced?

Tiled Matrix Multiplication

- How many memory accesses are reduced?
 - In the example, each value from A and B is loaded once and used twice

Tiled Matrix Multiplication

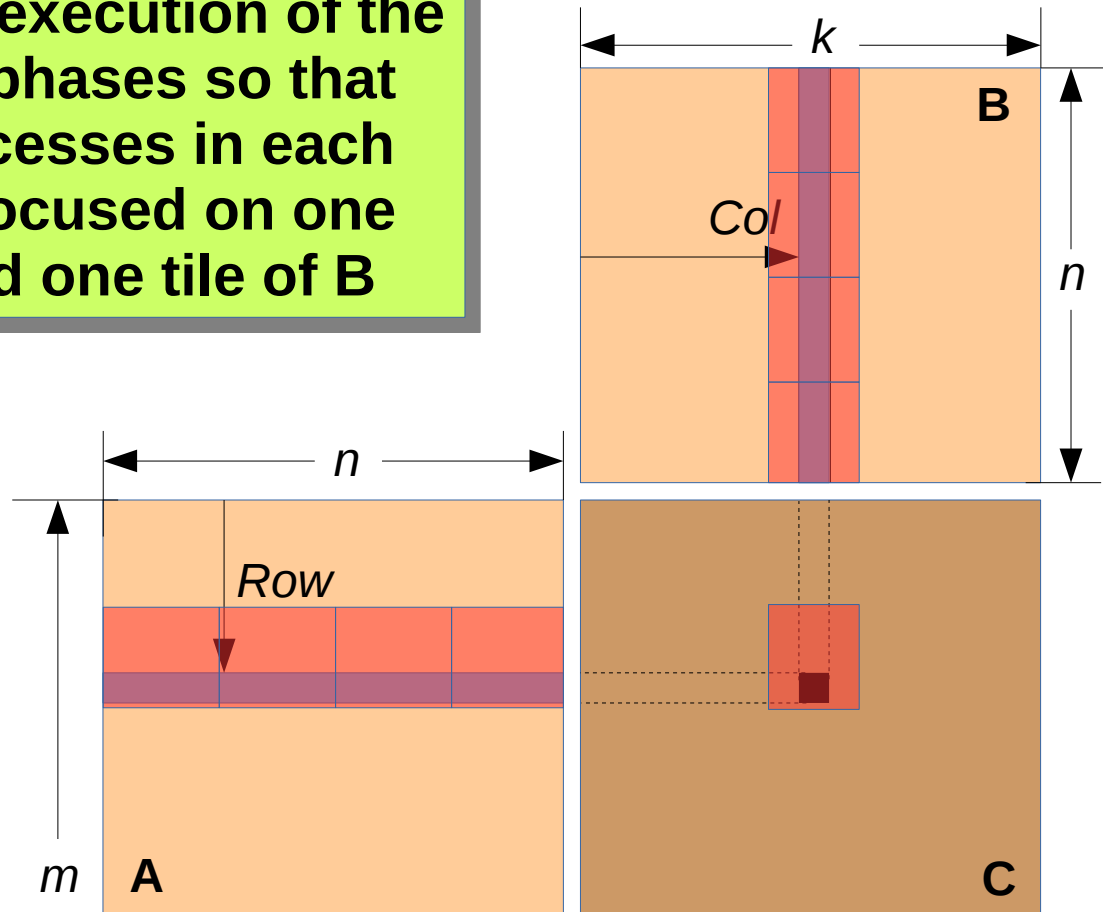
- How many memory accesses are reduced?
 - In the example, each value from A and B is loaded once and used twice
 - In the basic implementation, each value from A and B is loaded once and used once

Tiled Matrix Multiplication

- How many memory accesses are reduced?
 - In the example, each value from A and B is loaded once and used twice
 - In the basic implementation, each value from A and B is loaded once and used once
 - Memory bandwidth reduction by 50%

Tiled Matrix Multiplication

Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one tile of A and one tile of B



Barrier Synchronization

- API call: `__syncthreads ()`

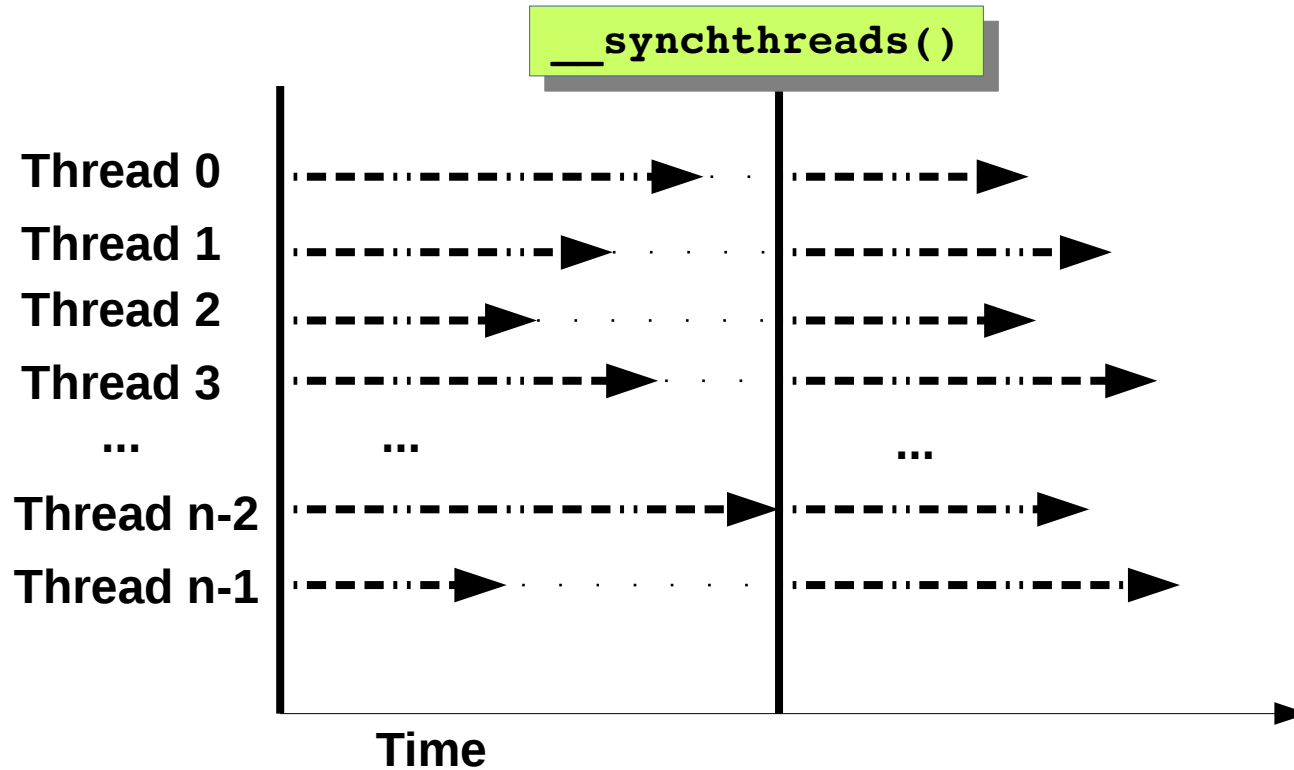
Barrier Synchronization

- API call: `__syncthreads ()`
- All threads in the same block must reach the `__syncthreads()` before any can move on

Barrier Synchronization

- API call: `__syncthreads ()`
- All threads in the same block must reach the `__syncthreads()` before any can move on
- Best used to coordinate tiled algorithms
 - To ensure that all elements of a tile are loaded
 - To ensure that all elements of a tile are consumed

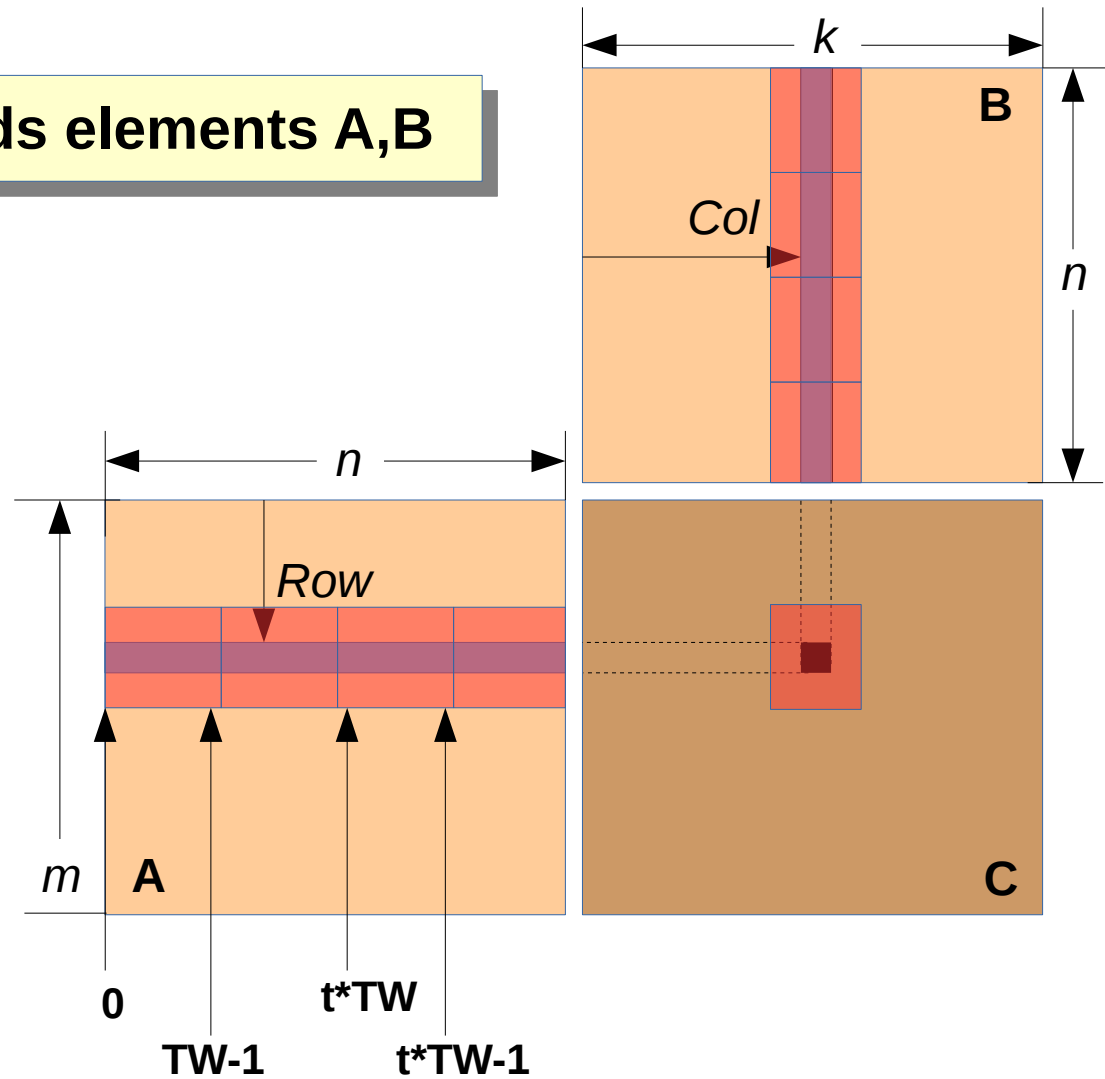
Barrier Synchronization



Barriers can significantly reduce active threads in a block

Tiled Matrix Multiplication

Tx, Ty loads elements A,B

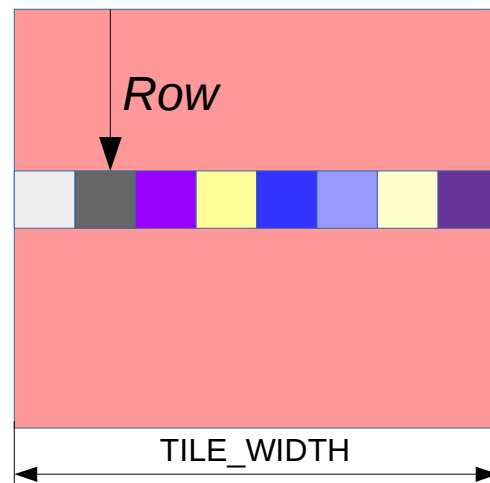
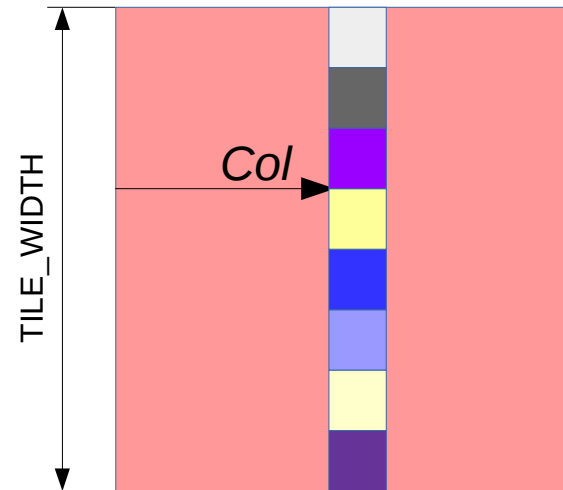


Load Phase 0 of a Thread

```
Row = by * blockDim.y + ty;
```

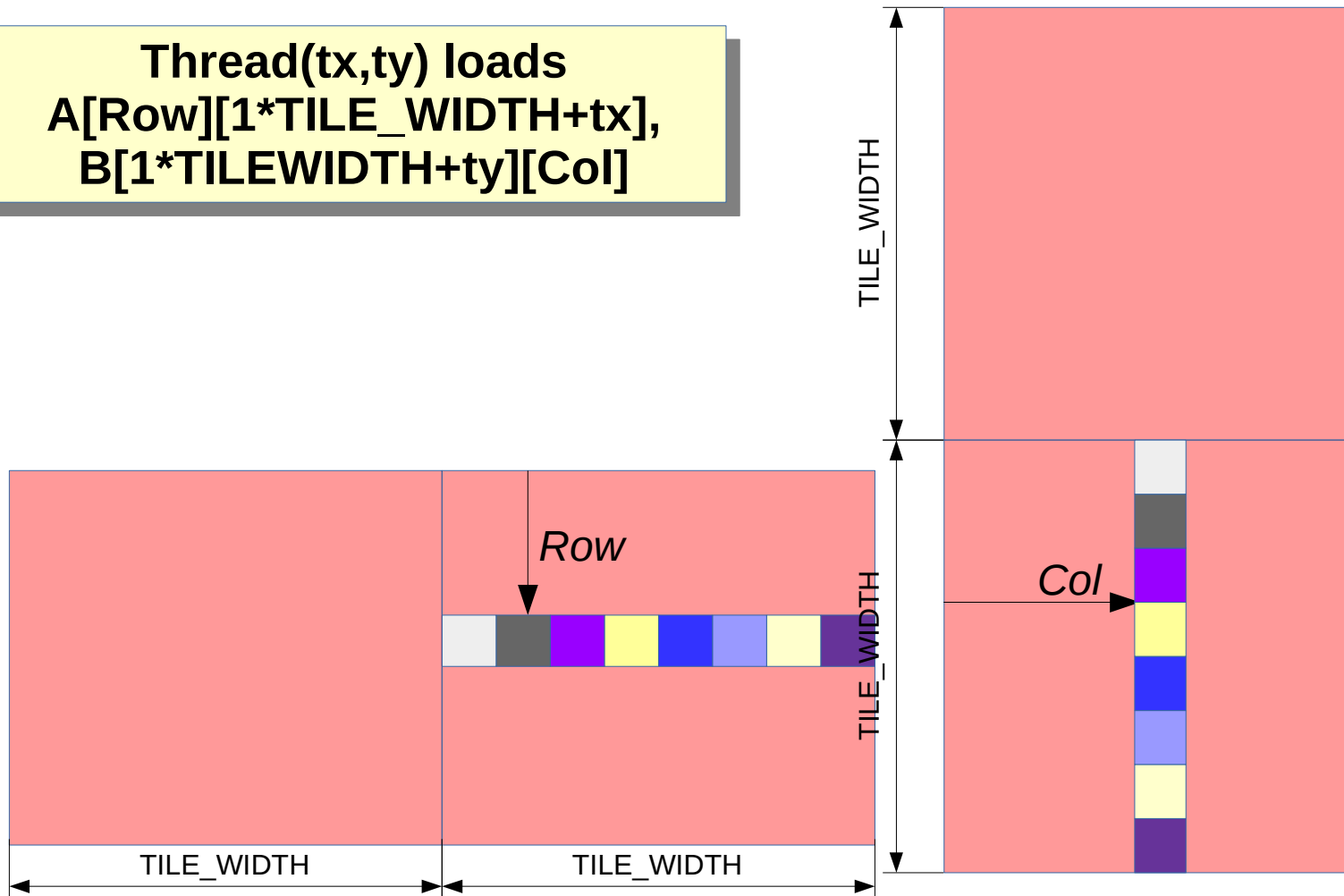
```
Col = bx * blockDim.x + tx;
```

```
Thread(tx,ty) loads  
A[Row][tx], B[ty][Col]
```



Load Phase 1 of a Thread

Thread(tx,ty) loads
 $A[\text{Row}][1*\text{TILE_WIDTH}+\text{tx}]$,
 $B[1*\text{TILEWIDTH}+\text{ty}][\text{Col}]$

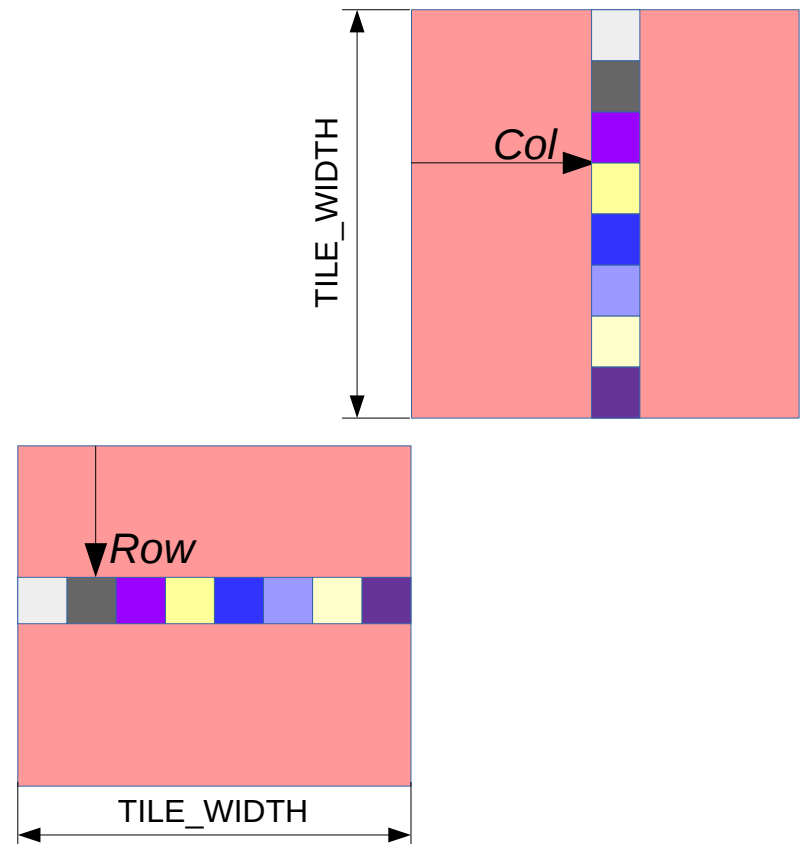


Linear Address

$$A[\text{Row}][t \cdot \text{TILE_WIDTH} + tx] = A[\text{Row} \cdot n + t \cdot \text{TILE_WIDTH} + tx]$$

$$B[t \cdot \text{TILE_WIDTH} + ty][\text{Col}] = B[(t \cdot \text{TILE_WIDTH} + ty) \cdot k + \text{Col}]$$

t is the tile sequence
number of the current phase



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(int m, int n, int k,  
float* A, float* B, float* C)  
{  
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];  
  
    ...  
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(int m, int n, int k,  
float* A, float* B, float* C)  
{  
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];  
  
    int bx = blockIdx.x; int by = blockIdx.y;  
    int tx = threadIdx.x; int ty = threadIdx.y;  
  
    ...
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(int m, int n, int k,  
float* A, float* B, float* C)  
{  
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];  
  
    int bx = blockIdx.x; int by = blockIdx.y;  
    int tx = threadIdx.x; int ty = threadIdx.y;  
  
    int Row = by * blockDim.y + ty;  
    int Col = bx * blockDim.x + tx;  
  
    ...
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(int m, int n, int k,  
float* A, float* B, float* C)  
{  
    __shared__ float ds_A[TILE_WIDTH][TILE_WIDTH];  
    __shared__ float ds_B[TILE_WIDTH][TILE_WIDTH];  
  
    int bx = blockIdx.x; int by = blockIdx.y;  
    int tx = threadIdx.x; int ty = threadIdx.y;  
  
    int Row = by * blockDim.y + ty;  
    int Col = bx * blockDim.x + tx;  
  
    float Cvalue = 0;  
  
    ...
```

Tiled Matrix Multiplication Kernel

```
...
// Loop over the A and B tiles required
// to compute the C element
for (int t = 0; t < n/TILE_WIDTH; ++t) {
    // Collaborative loading of A and B tiles
    // into shared memory

    ...
}

}
```

Tiled Matrix Multiplication Kernel

```
...
// Loop over the A and B tiles required
// to compute the C element
for (int t = 0; t < n/TILE_WIDTH; ++t) {
    // Collaborative loading of A and B tiles
    // into shared memory
    ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH+tx];
    ds_B[ty][tx] = B[(t*TILE_WIDTH+ty)*k + Col];
    __syncthreads();

    ...
}

...
}
```

Tiled Matrix Multiplication Kernel

```
...
// Loop over the A and B tiles required
// to compute the C element
for (int t = 0; t < n/TILE_WIDTH; ++t) {
    // Collaborative loading of A and B tiles
    // into shared memory
    ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH+tx];
    ds_B[ty][tx] = B[(t*TILE_WIDTH+ty)*k + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Cvalue += ds_A[ty][i] * ds_B[i][tx];
    __syncthreads();
}

...
}
```

Tiled Matrix Multiplication Kernel

```
...
// Loop over the A and B tiles required
// to compute the C element
for (int t = 0; t < n/TILE_WIDTH; ++t) {
    // Collaborative loading of A and B tiles
    // into shared memory
    ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH+tx];
    ds_B[ty][tx] = B[(t*TILE_WIDTH+ty)*k + Col];
    __syncthreads();

    for (int i = 0; i < TILE_WIDTH; ++i)
        Cvalue += ds_A[ty][i] * ds_B[i][tx];
    __syncthreads();
}

C[Row*k+Col] = Cvalue;
}
```


First Order Considerations

- Each thread block should have many threads
 - TILE_WIDTH of 16 gives $16*16 = 256$ threads
 - TILE_WIDTH of 32 gives $32*32 = 1024$ threads

First Order Considerations

- Each thread block should have many threads
 - TILE_WIDTH of 16 gives $16*16 = 256$ threads
 - TILE_WIDTH of 32 gives $32*32 = 1024$ threads
- For 16, each block performs $2*256 = 512$ float loads from global memory for $256 * (2*16) = 8,192$ mul/add operations.
 - memory traffic reduced by a factor of 16

First Order Considerations

- Each thread block should have many threads
 - TILE_WIDTH of 16 gives $16*16 = 256$ threads
 - TILE_WIDTH of 32 gives $32*32 = 1024$ threads
- For 16, each block performs $2*256 = 512$ float loads from global memory for $256 * (2*16) = 8,192$ mul/add operations.
 - memory traffic reduced by a factor of 16
- For 32, each block performs $2*1024 = 2048$ float loads from global memory for $1024 * (2*32) = 65,536$ mul/add operations.
 - memory traffic reduced by a factor of 32

Shared Memory and Threading

- Fermi SM has 16KB or 48KB shared memory (configurable vs. L1 cache)

Shared Memory and Threading

- Fermi SM has 16KB or 48KB shared memory (configurable vs. L1 cache)
- $TILE_WIDTH = 16$. Thread Block uses $2 * 256 * 4B = 2KB$ of shared memory.

Shared Memory and Threading

- Fermi SM has 16KB or 48KB shared memory (configurable vs. L1 cache)
- `TILE_WIDTH = 16`. Thread Block uses $2 * 256 * 4B = 2KB$ of shared memory.
- 16K SM, can have up to 8 thread blocks executing
 - Pending Loads: $8 * 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)

Shared Memory and Threading

- Fermi SM has 16KB or 48KB shared memory (configurable vs. L1 cache)
- $TILE_WIDTH = 16$. Thread Block uses $2 * 256 * 4B = 2KB$ of shared memory.
- 16K SM, can have up to 8 thread blocks executing
 - Pending Loads: $8 * 512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
- $TILE_WIDTH = 32$. TB uses $2 * 32 * 32 * 4 \text{ Byte} = 8KB$ SM. 2 thread blocks can be active

Shared Memory and Threading

- `TILE_WIDTH = 16` reduces accesses to global memory by a factor of 16

Shared Memory and Threading

- `TILE_WIDTH = 16` reduces accesses to global memory by a factor of 16
- The 150 GB/s bandwidth can now support $(150/4)*16 = 600$ GFLOPS!

Tiled MM – Arbitrary Matrix Dimensions

- Real applications need to handle arbitrary sized matrices

Tiled MM – Arbitrary Matrix Dimensions

- Real applications need to handle arbitrary sized matrices
- Pad (add elements to) the rows and columns into multiples of the tile size
 - Significant space and data transfer time overhead!

Loads for Block (0,0) – Phase 0

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	

SM	
$B_{0,0}$	$B_{0,1}$
$B_{1,0}$	$B_{1,1}$

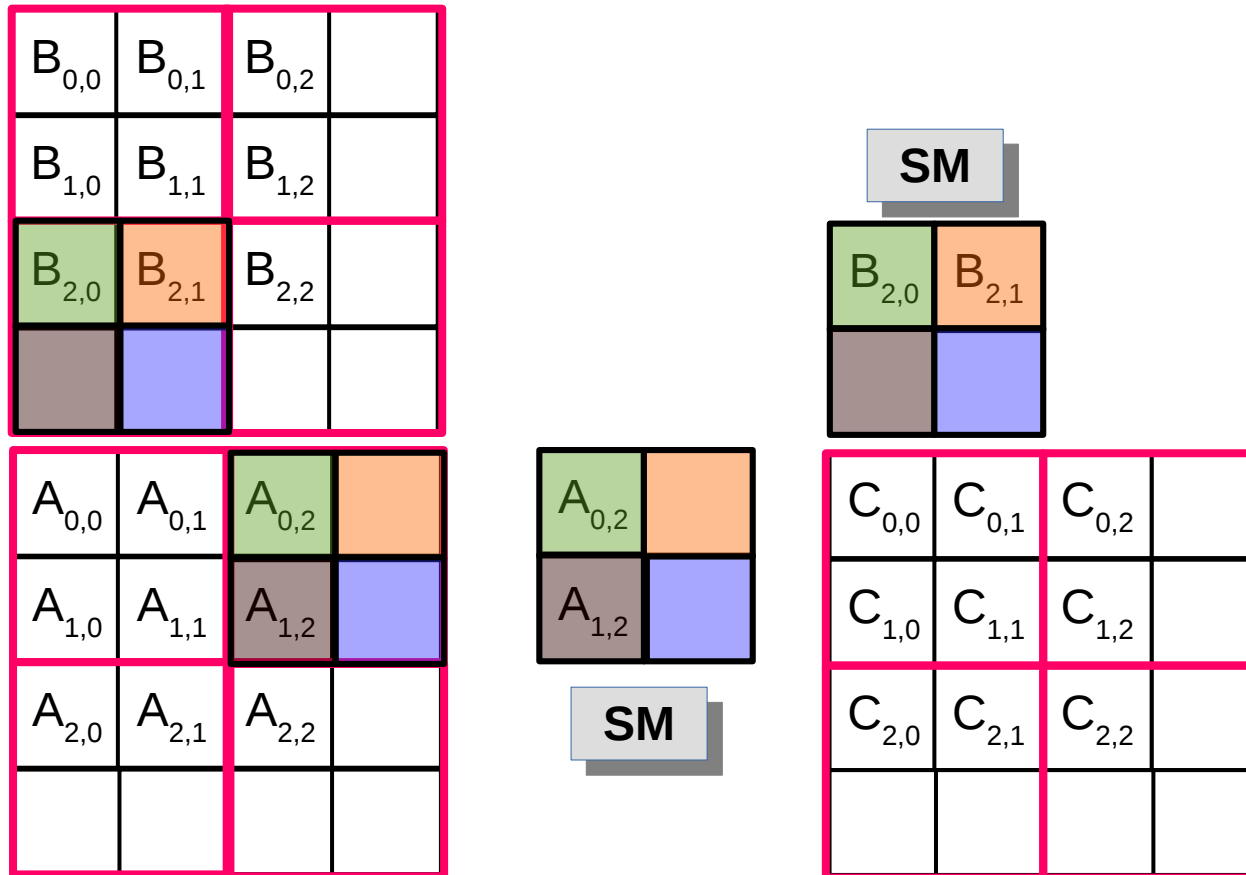
$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	

$A_{0,0}$	$A_{0,1}$
$A_{1,0}$	$A_{1,1}$

SM

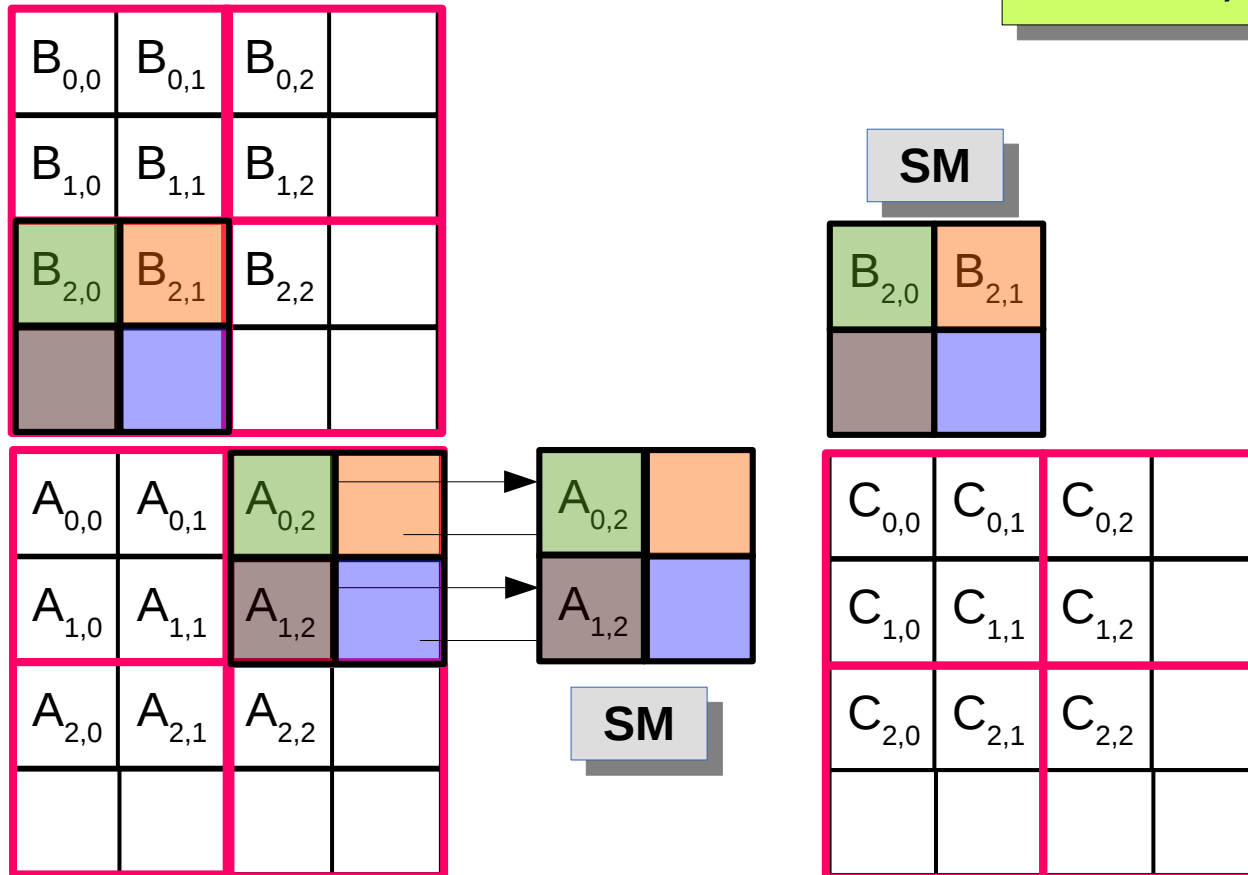
$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	

Loads for Block (0,0) – Phase 1



Loads for Block (0,0) – Phase 1

Thread 0,1 attempts to load $A_{0,3}$
Thread 1,1 attempts to load $A_{1,3}$



Loads for Block (0,0) – Phase 1

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	

$A_{0,2}$	
$A_{1,2}$	

SM

Thread 0,1 attempts to load $A_{0,3}$
Thread 1,1 attempts to load $A_{1,3}$

The threads load $A_{1,0}$ and $A_{2,0}$

$B_{2,0}$	$B_{2,1}$

$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	

Loads for Block (0,0) – Phase 1

$B_{0,0}$	$B_{0,1}$	$B_{0,2}$	
$B_{1,0}$	$B_{1,1}$	$B_{1,2}$	
$B_{2,0}$	$B_{2,1}$	$B_{2,2}$	

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	

$A_{0,2}$	$A_{1,0}$
$A_{1,2}$	$A_{2,0}$

SM

Thread 0,1 attempts to load $A_{0,3}$
Thread 1,1 attempts to load $A_{1,3}$

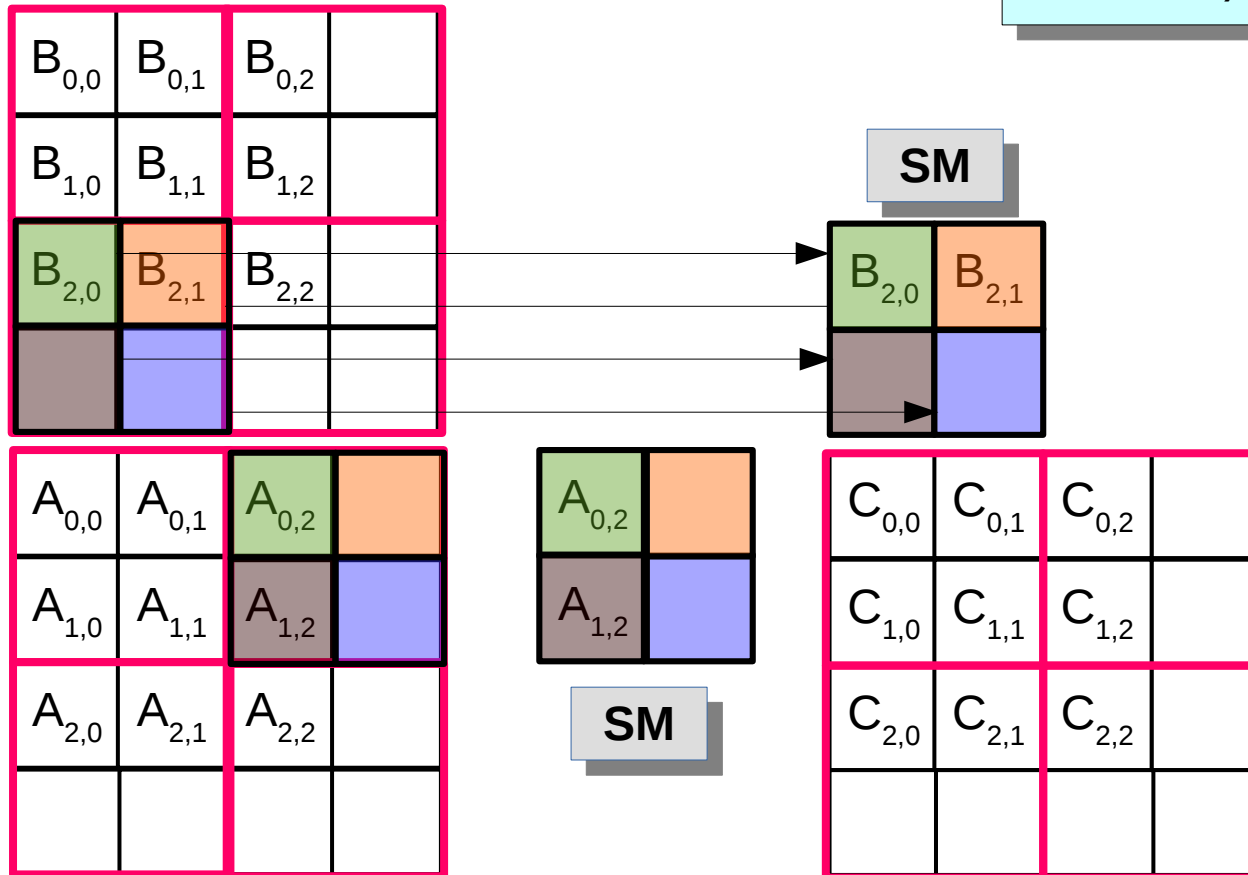
The threads load $A_{1,0}$ and $A_{2,0}$

$B_{2,0}$	$B_{2,1}$

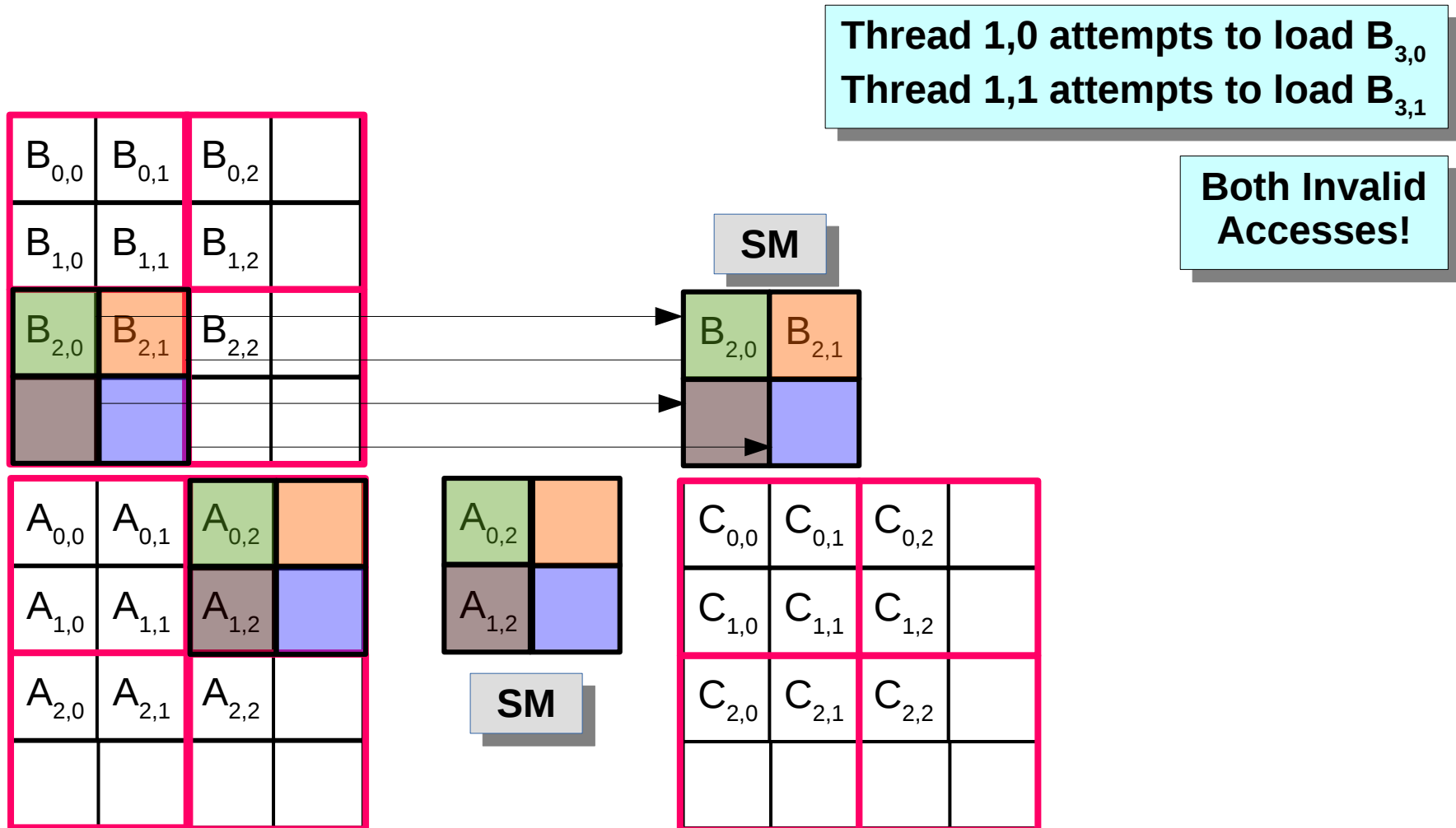
$C_{0,0}$	$C_{0,1}$	$C_{0,2}$	
$C_{1,0}$	$C_{1,1}$	$C_{1,2}$	
$C_{2,0}$	$C_{2,1}$	$C_{2,2}$	

Loads for Block (0,0) – Phase 1

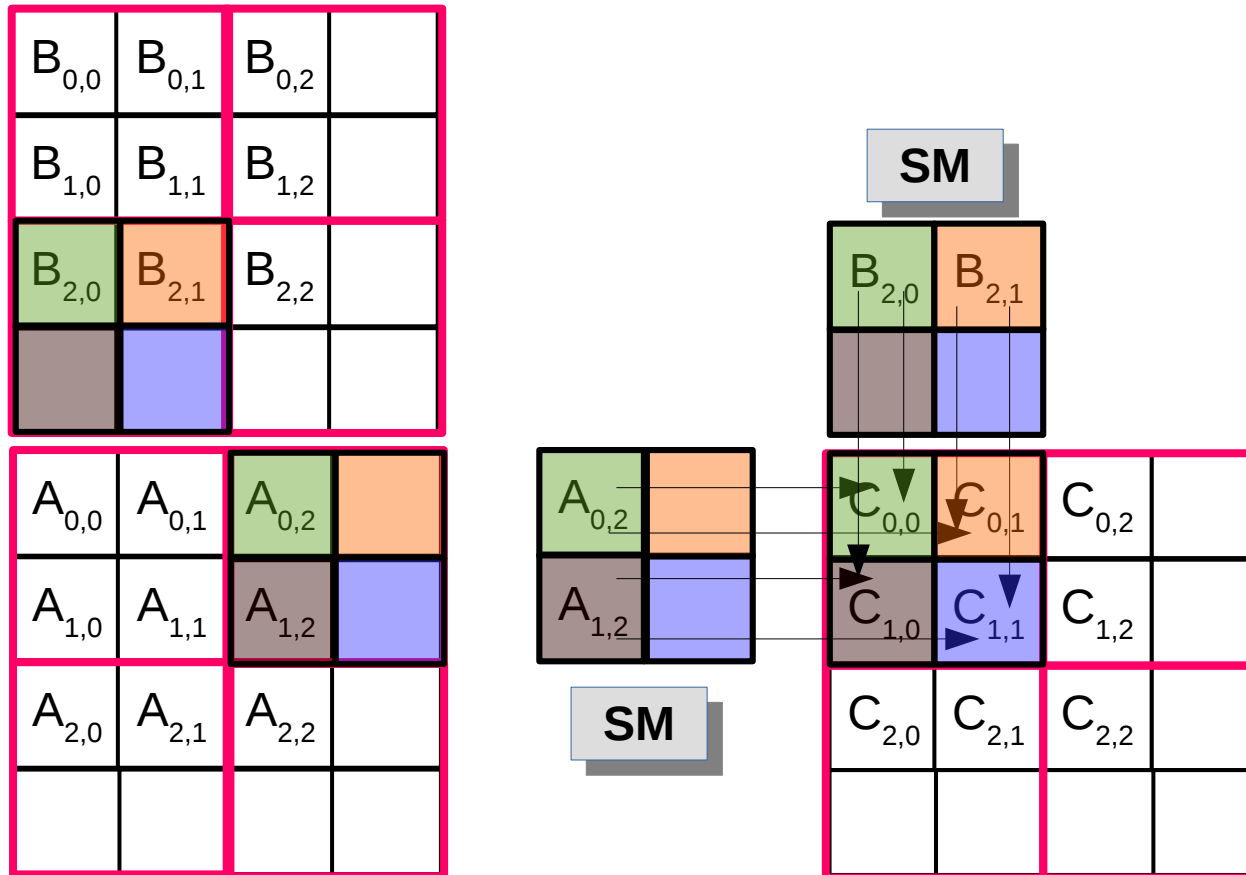
Thread 1,0 attempts to load $B_{3,0}$
Thread 1,1 attempts to load $B_{3,1}$



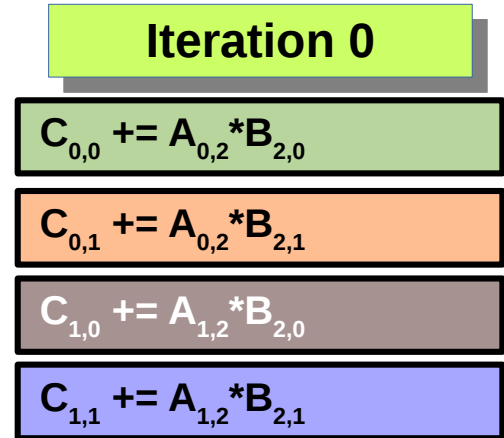
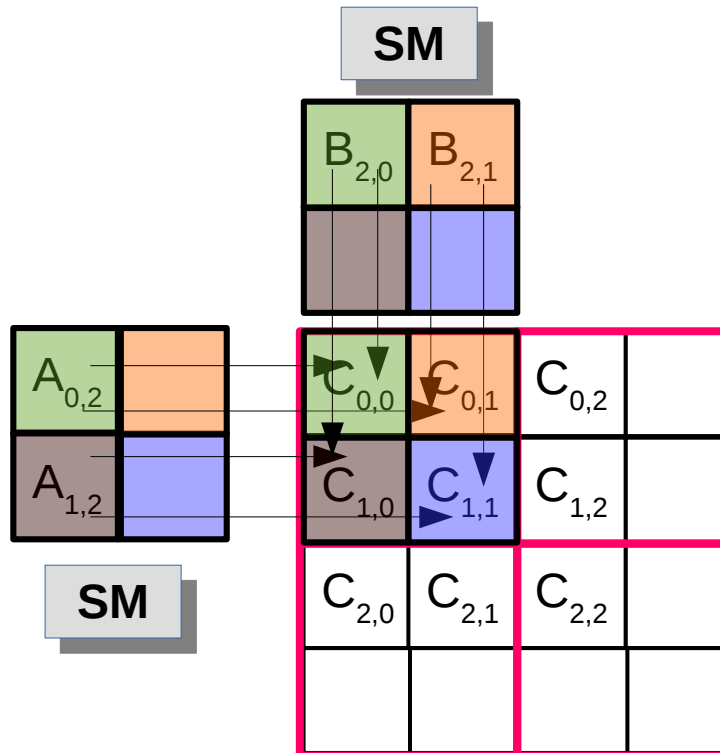
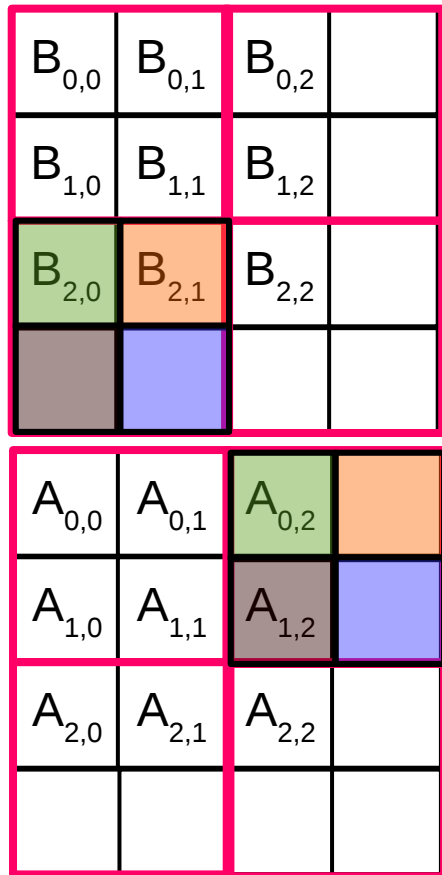
Loads for Block (0,0) – Phase 1



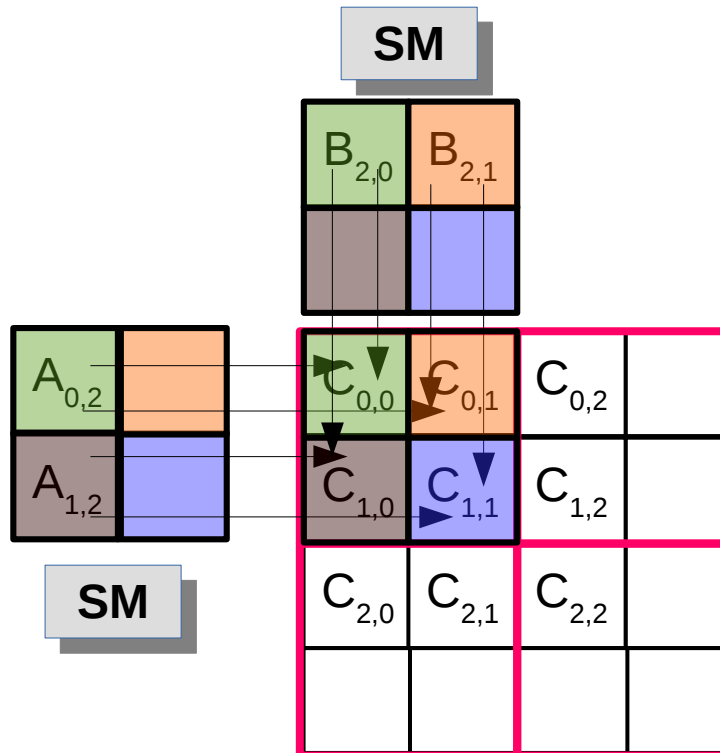
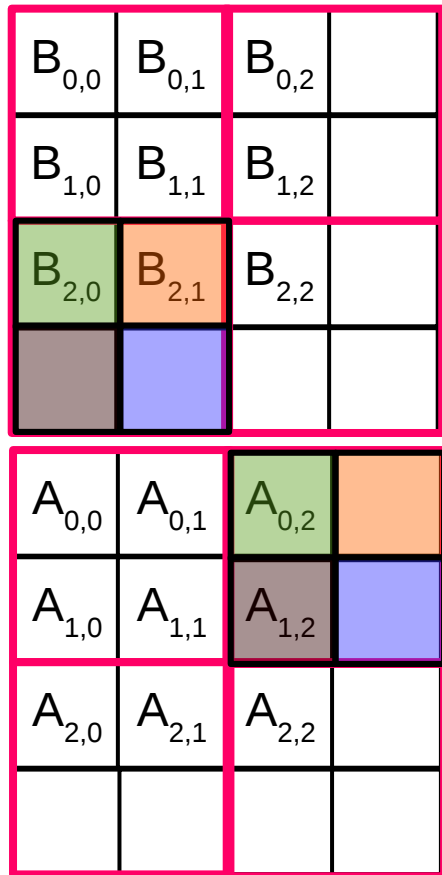
Computation after Phase 1 Loads



Computation after Phase 1 Loads



Computation after Phase 1 Loads



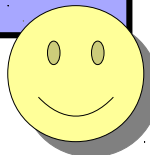
Iteration 0

$C_{0,0} += A_{0,2} * B_{2,0}$

$C_{0,1} += A_{0,2} * B_{2,1}$

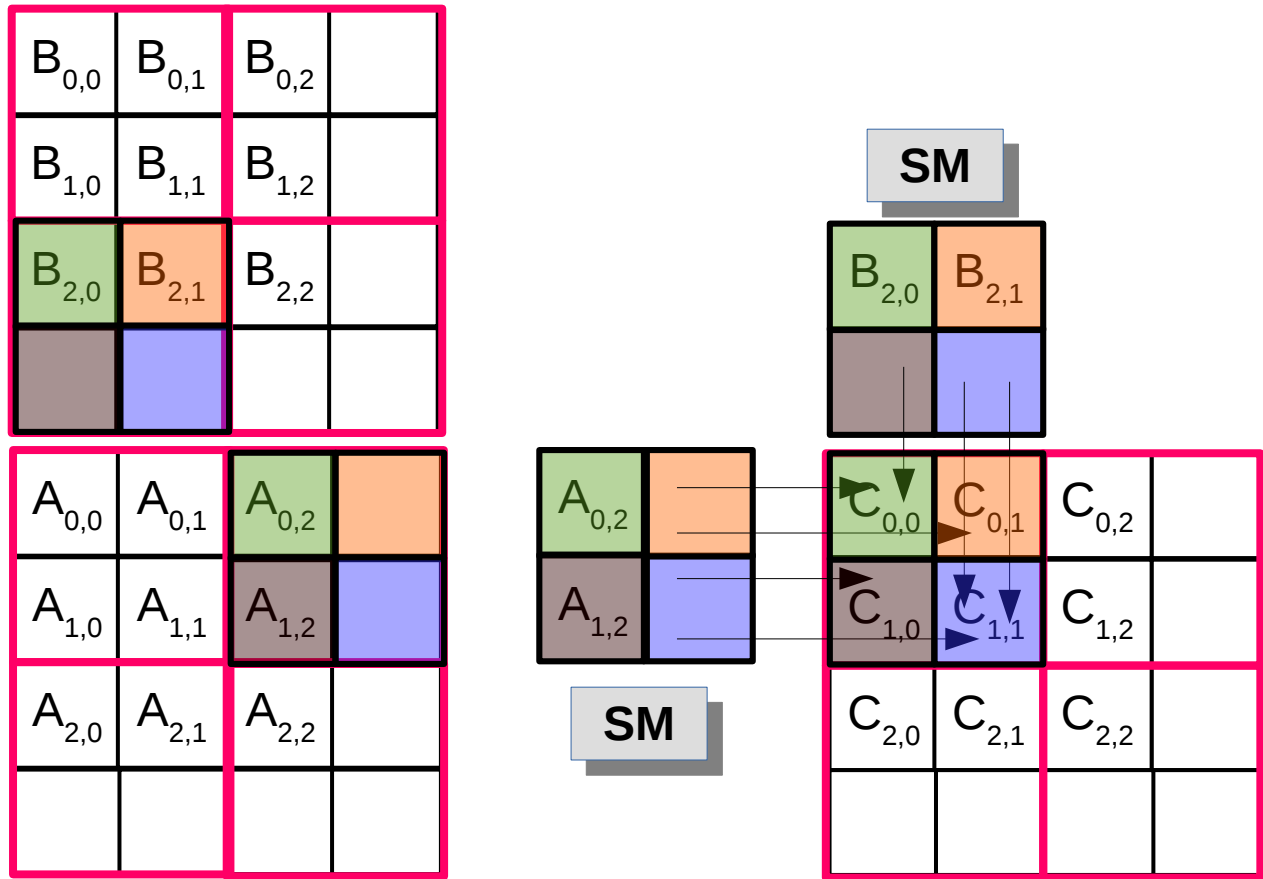
$C_{1,0} += A_{1,2} * B_{2,0}$

$C_{1,1} += A_{1,2} * B_{2,1}$

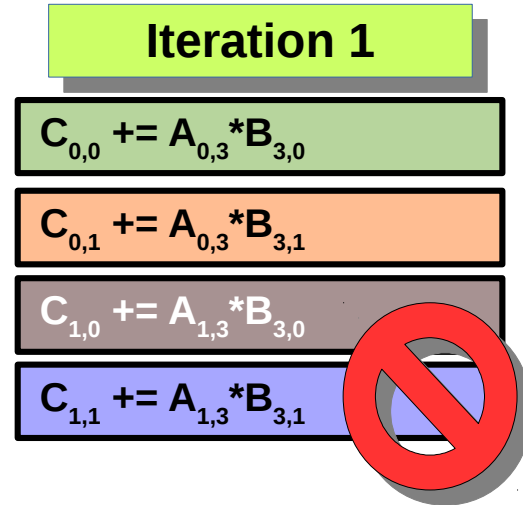
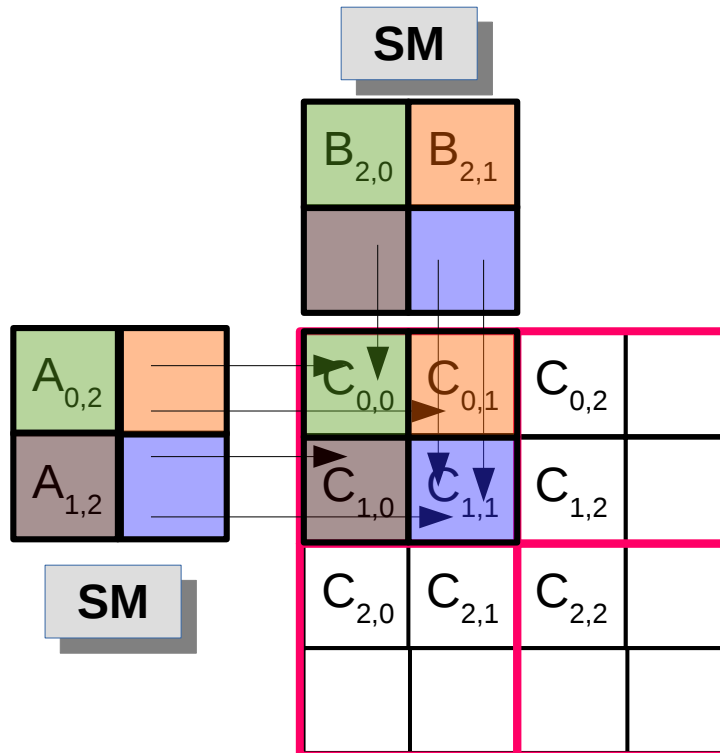
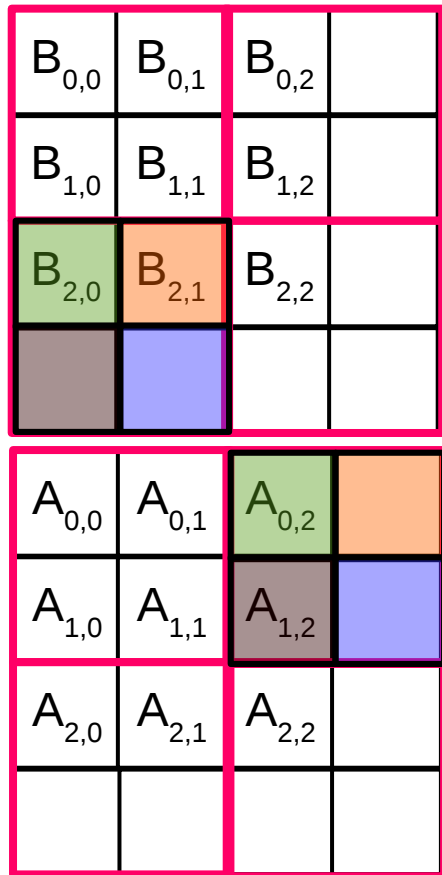


Computation after Phase 1 Loads

Iteration 1



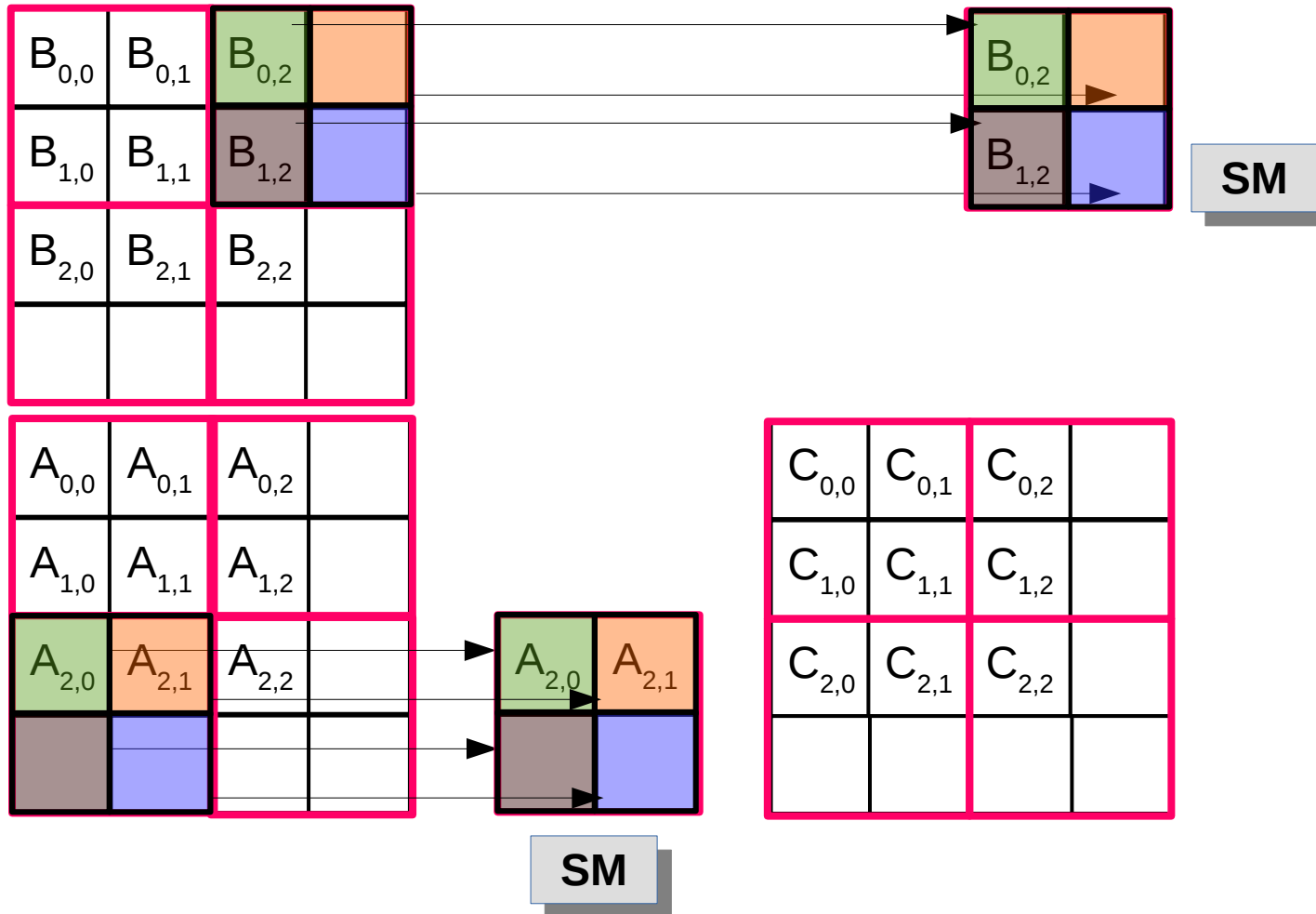
Computation after Phase 1 Loads



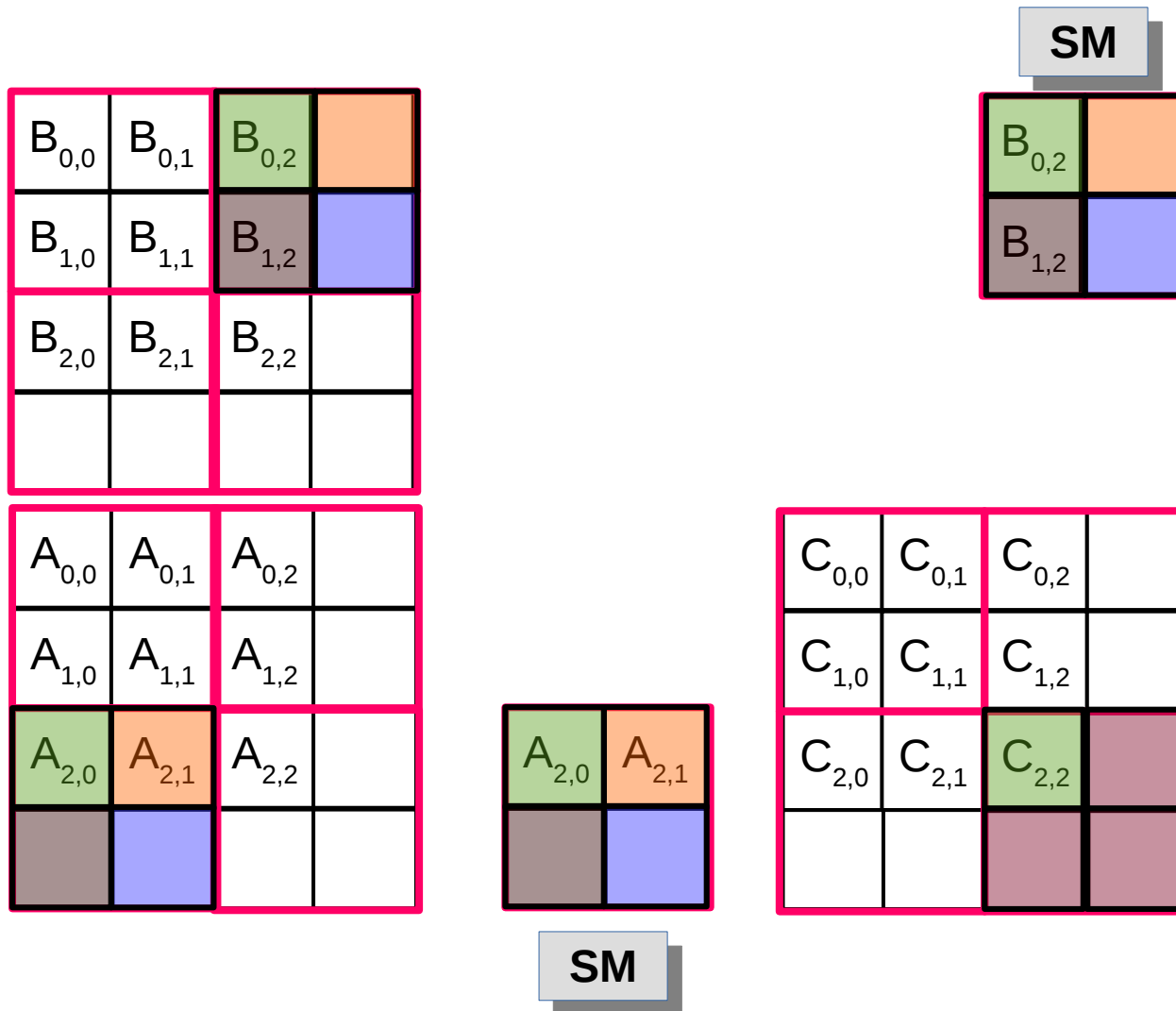
Loads for Block (1,1) Phase 0

Thread 0,1 attempts to load $B_{0,3}$
Thread 1,1 attempts to load $B_{1,3}$

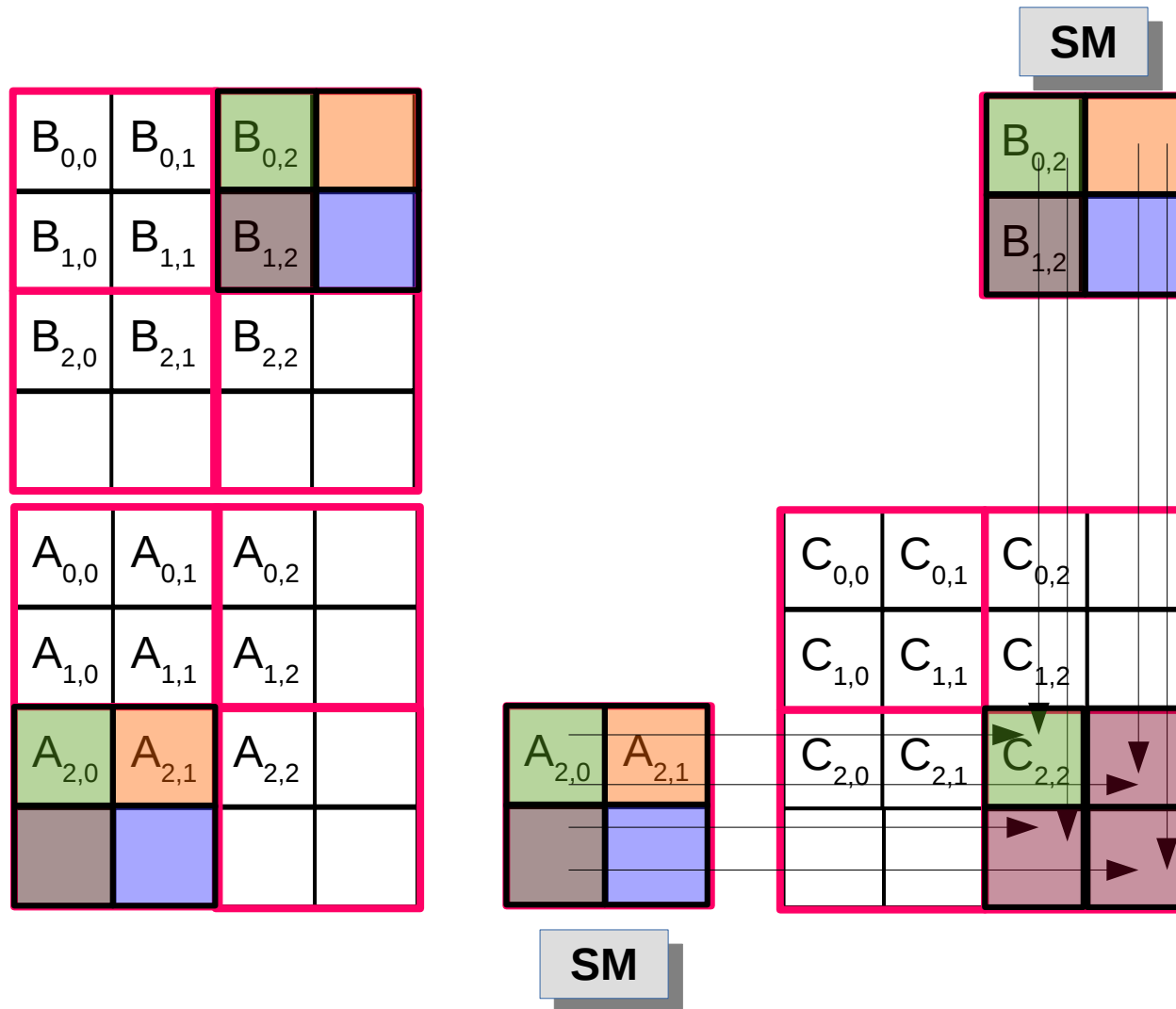
Thread 1,0 attempts to load $A_{3,0}$
Thread 1,1 attempts to load $A_{3,1}$



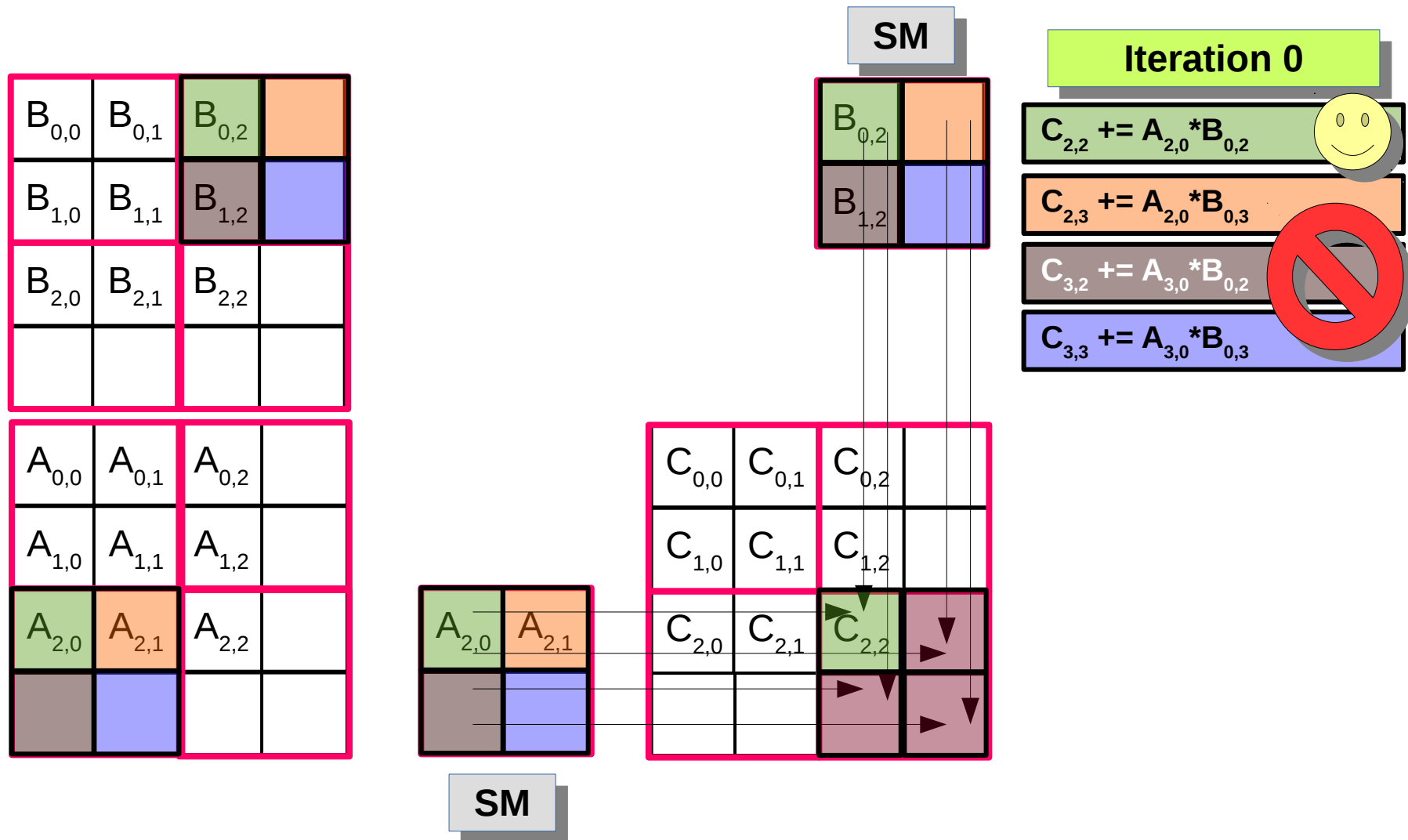
Computation after Phase 0 Loads



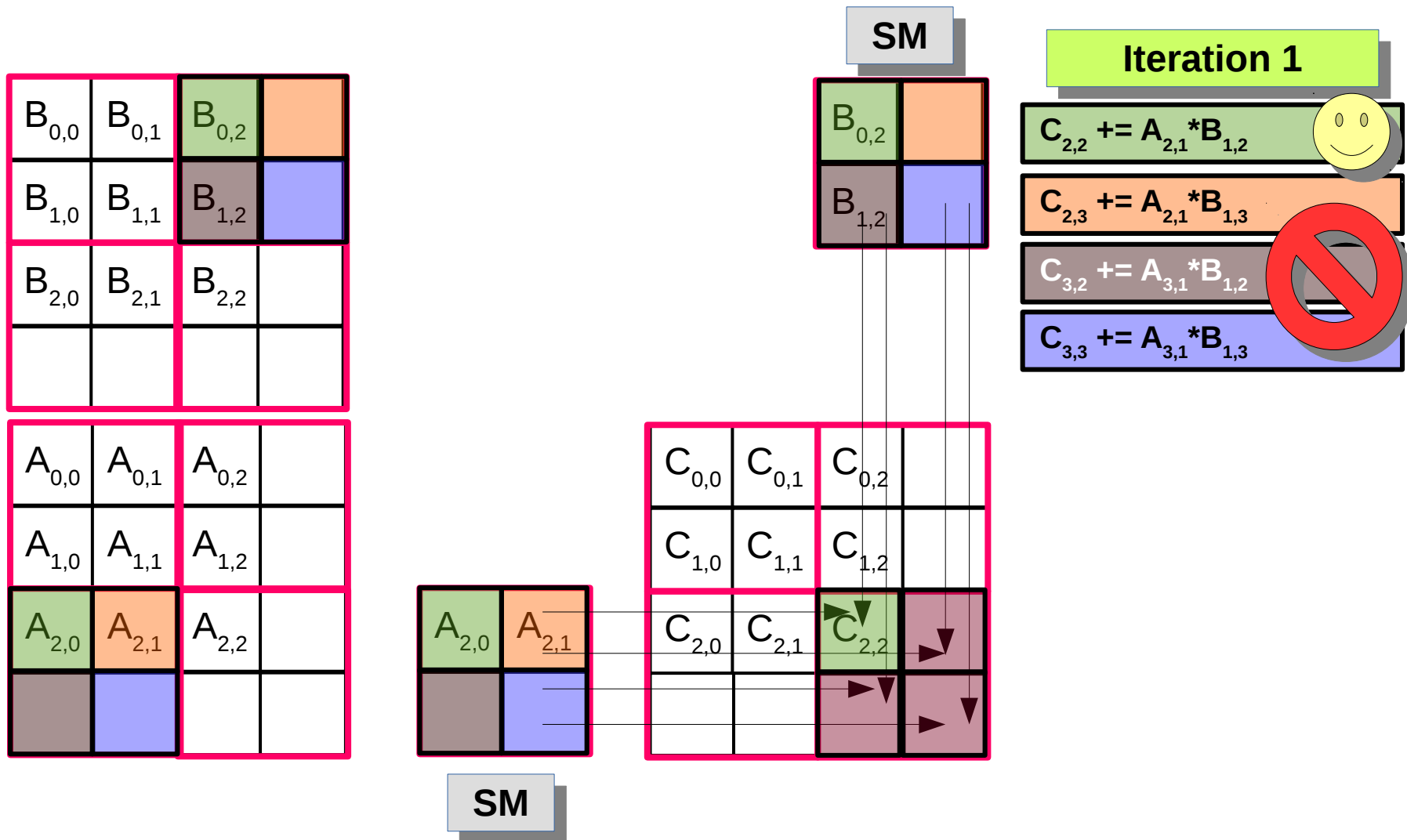
Computation after Phase 0 Loads



Computation after Phase 0 Loads



Computation after Phase 0 Loads



Major Cases from the Example

- Threads that calculate valid C elements but can step outside valid input

Major Cases from the Example

- Threads that calculate valid C elements but can step outside valid input
 - Phase 1 of Block(0,0), 2nd step, all threads

Major Cases from the Example

- Threads that calculate valid C elements but can step outside valid input
 - Phase 1 of Block(0,0), 2nd step, all threads
- Threads that do not calculate valid C elements but still need to participate in loading the input tiles

Major Cases from the Example

- Threads that calculate valid C elements but can step outside valid input
 - Phase 1 of Block(0,0), 2nd step, all threads
- Threads that do not calculate valid C elements but still need to participate in loading the input tiles
 - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent C[3,2] but need to participate in loading tile element B[1,2]

Tiled MM – Arbitrary Matrix Dimensions

- When a thread is to load any input element, test if it is in the valid index range

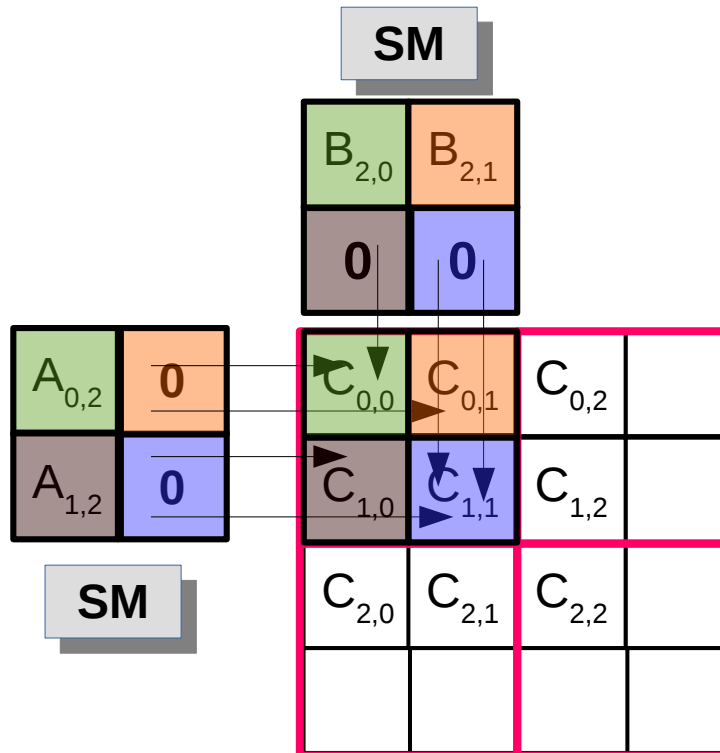
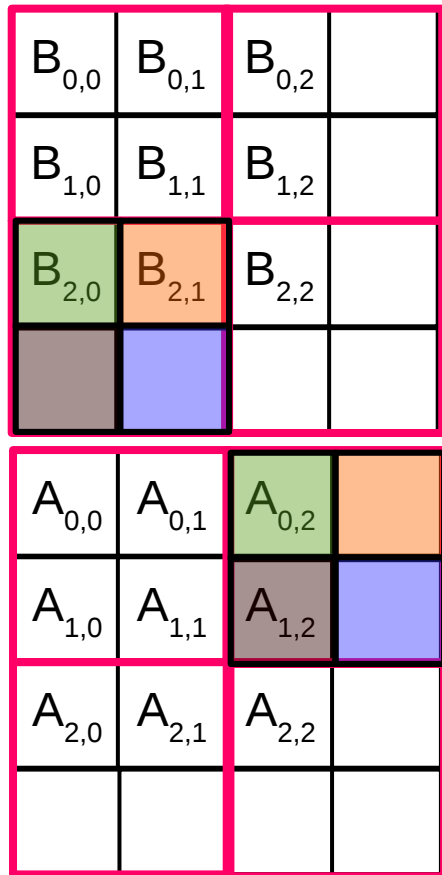
Tiled MM – Arbitrary Matrix Dimensions

- 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

Tiled MM – Arbitrary Matrix Dimensions

- 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
- 0 will not affect the multiply-add step – functional correctness

Computation after Phase 1 Loads




Iteration 1

$C_{0,0} += A_{0,3} * B_{3,0}$

$C_{0,1} += A_{0,3} * B_{3,1}$

$C_{1,0} += A_{1,3} * B_{3,0}$

$C_{1,1} += A_{1,3} * B_{3,1}$



Simple Solution contd.

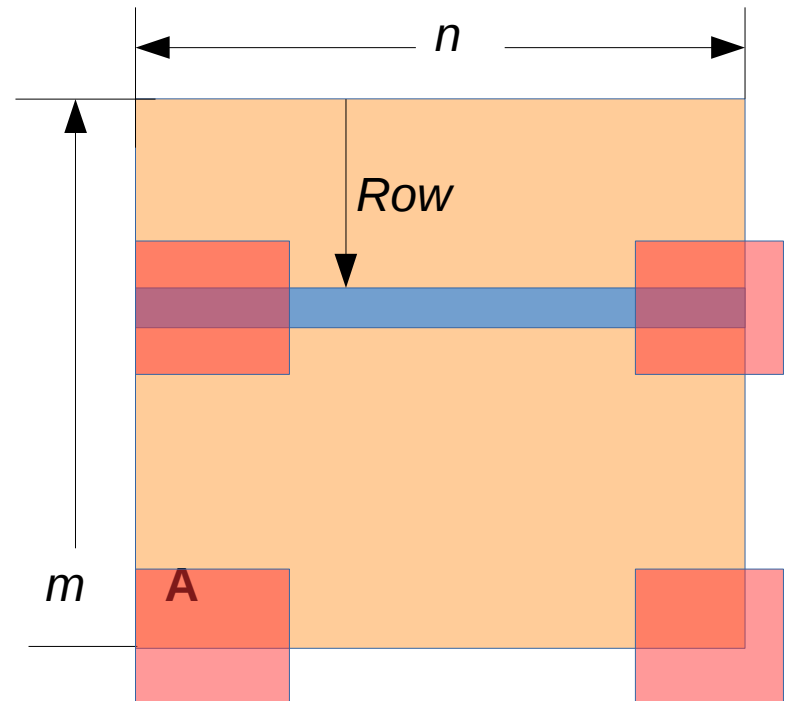
- If a thread does not calculate a valid C element
 - Can still perform multiply-add into its register
 - Shouldn't write its Cvalue to the global memory at the end of the kernel
 - Thread participates in the tile loading process

Boundary Condition for Input A Tile

Each thread loads $A[\text{Row}][t*\text{TILE_WIDTH}+tx]$

Each thread loads $A[\text{Row}*n + t*\text{TILE_WIDTH}+tx]$

```
if (Row < m) && (t*TILE_WIDTH+tx < n) then
  load A element
else
  load 0
```

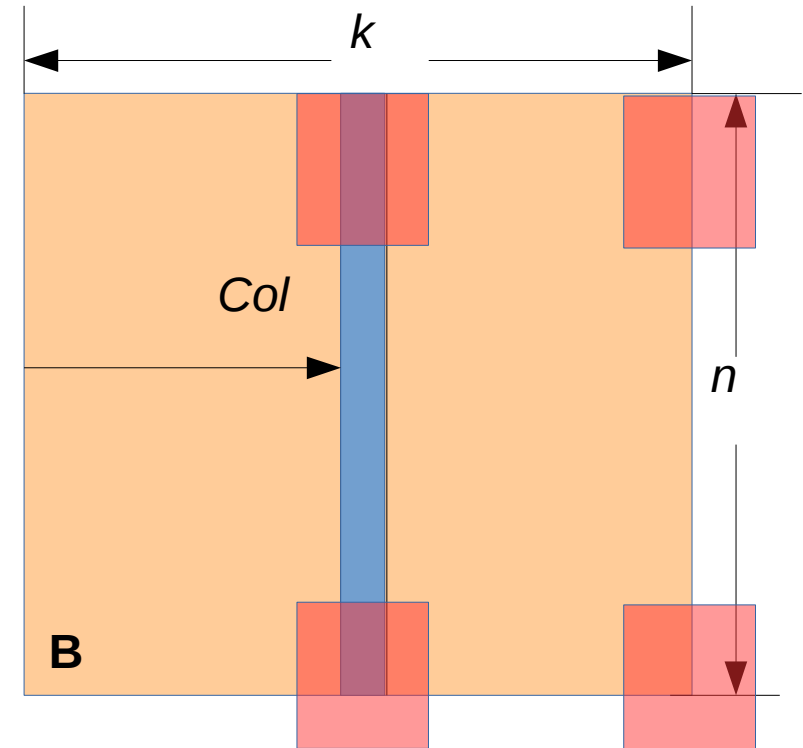


Boundary Condition for Input B Tile

Each thread loads $B[t * \text{TILE_WIDTH} + ty][\text{Col}]$

Each thread loads $B[(t * \text{TILE_WIDTH} + ty) * k + \text{Col}]$

```
if (t * TILE_WIDTH + ty < n) && (Col < k) then
  load B element
else
  load 0
```



Tiled Matrix Multiplication Kernel

```
...
for (int t = 0; t < (n-1)/TILE_WIDTH + 1; ++t) {
    ...
    ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH + tx];
    ...
    ds_B[ty][tx] = B[(t*TILE_WIDTH + ty)*k + Col];

    __syncthreads();
}
...
```


Tiled Matrix Multiplication Kernel

```
...
for (int t = 0; t < (n-1)/TILE_WIDTH + 1; ++t) {

    if(Row < m && t*TILE_WIDTH+tx < n) {
        ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH + tx];
    } else {
        ds_A[ty][tx] = 0.0;
    }

    ...
    ds_B[ty][tx] = B[(t*TILE_WIDTH + ty)*k + Col];

    __syncthreads();

    ...
}
```

Tiled Matrix Multiplication Kernel

```
...
for (int t = 0; t < (n-1)/TILE_WIDTH + 1; ++t) {

    if(Row < m && t*TILE_WIDTH+tx < n) {
        ds_A[ty][tx] = A[Row*n + t*TILE_WIDTH + tx];
    } else {
        ds_A[ty][tx] = 0.0;
    }

    if (t*TILE_WIDTH+ty < n && Col < k) {
        ds_B[ty][tx] = B[(t*TILE_WIDTH + ty)*k + Col];
    } else {
        ds_B[ty][tx] = 0.0;
    }

    __syncthreads();

...

```

Tiled Matrix Multiplication Kernel

```
...  
    for (int i = 0; i < TILE_WIDTH; ++i) {  
        Cvalue += ds_A[ty][i] * ds_B[i][tx];  
    }  
}  
__syncthreads();  
  
}  
  
if (Row < m && Col < k)  
    C[Row*k + Col] = Cvalue;  
...  

```

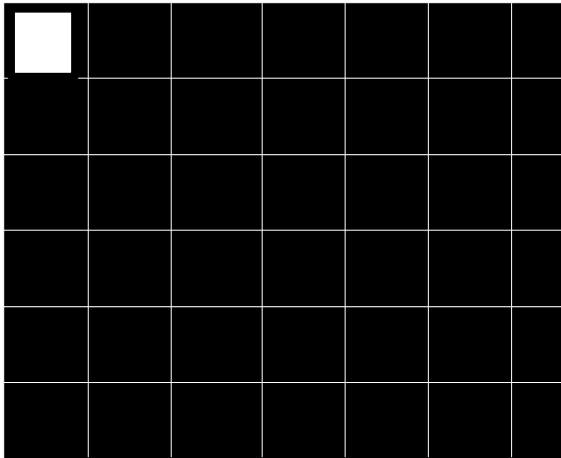
Important Points

- For each thread the conditions are different for
 - Loading A element
 - Loading B element
 - Storing output elements
- The effect of control divergence should be small for large matrices

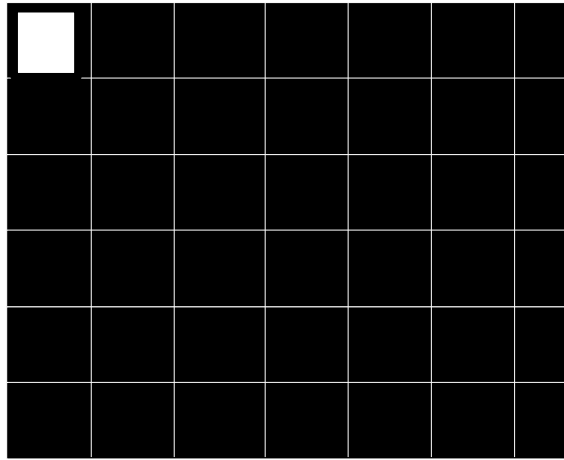
MM using Tiling

Matrix Multiplication

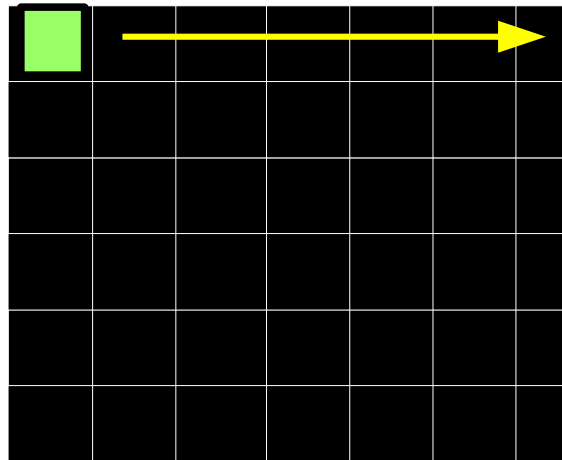
```
double X[N][N], Y[N][N], Z[N][N];  
for (i=0;i<N;i++)  
  for (j=0;j<N;j++)  
    for (k=0;k<N;k++)  
      X[i][j] += Y[i][k] * Z[k][j];
```



DGEMM



X



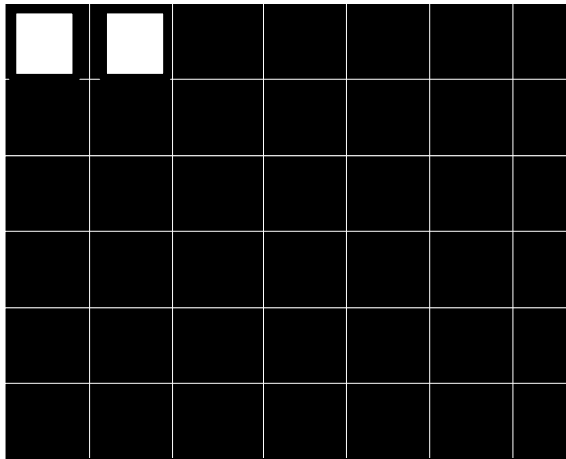
Y



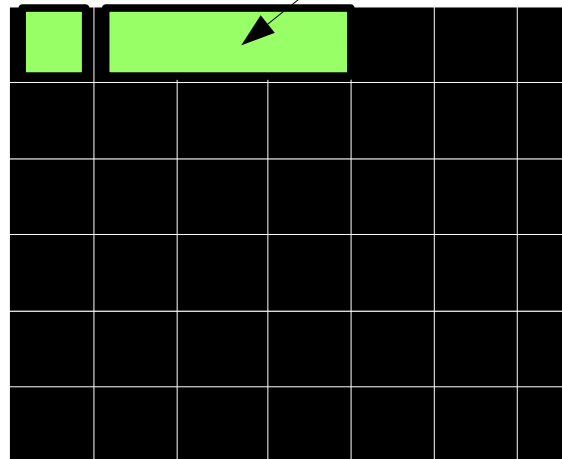
Z

Blocking / Tiling

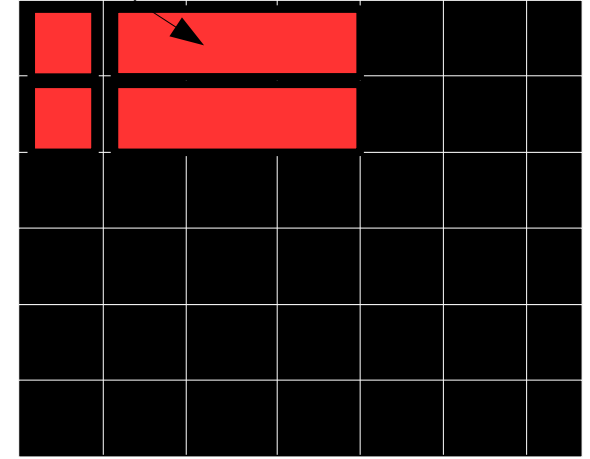
These elements are in the Cache



X



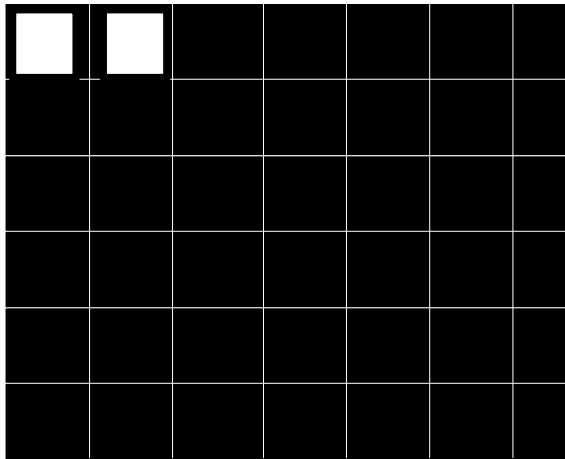
Y



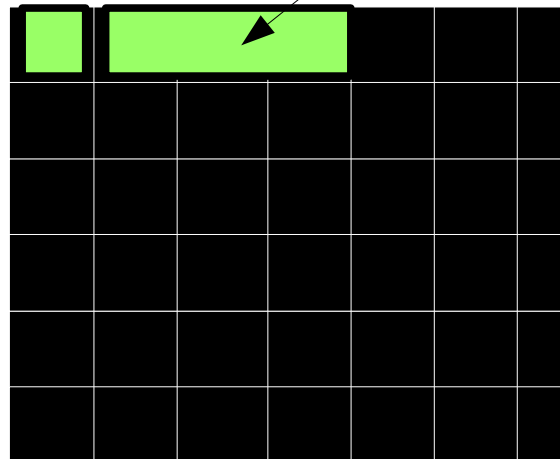
Z

Blocking / Tiling

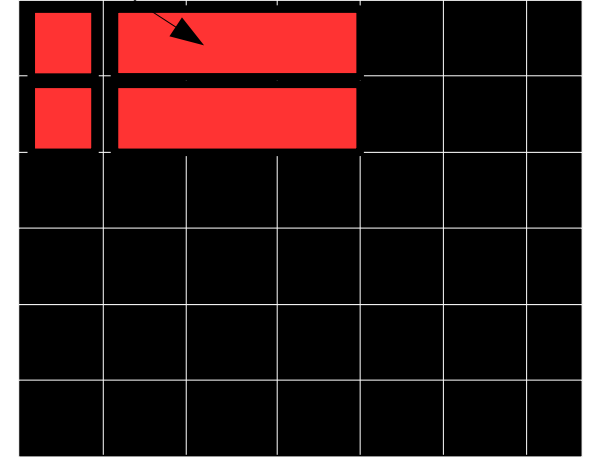
These elements are in the Cache



X



Y

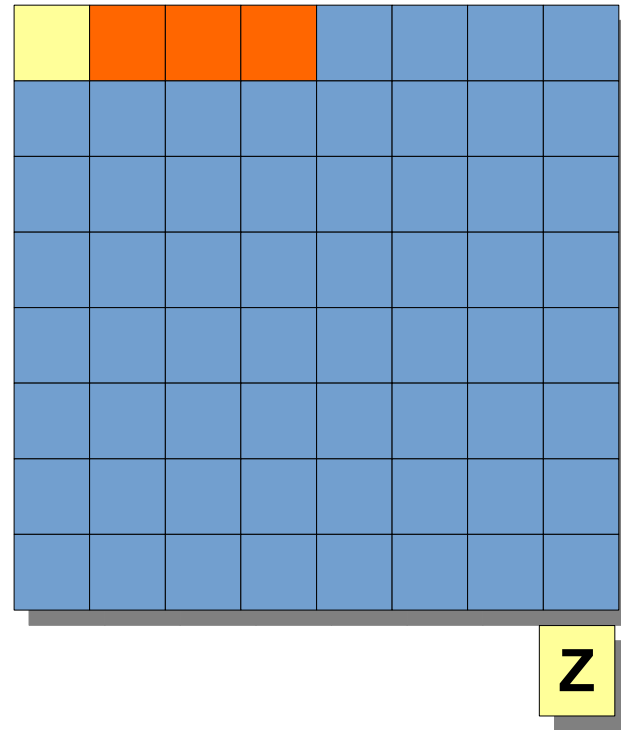
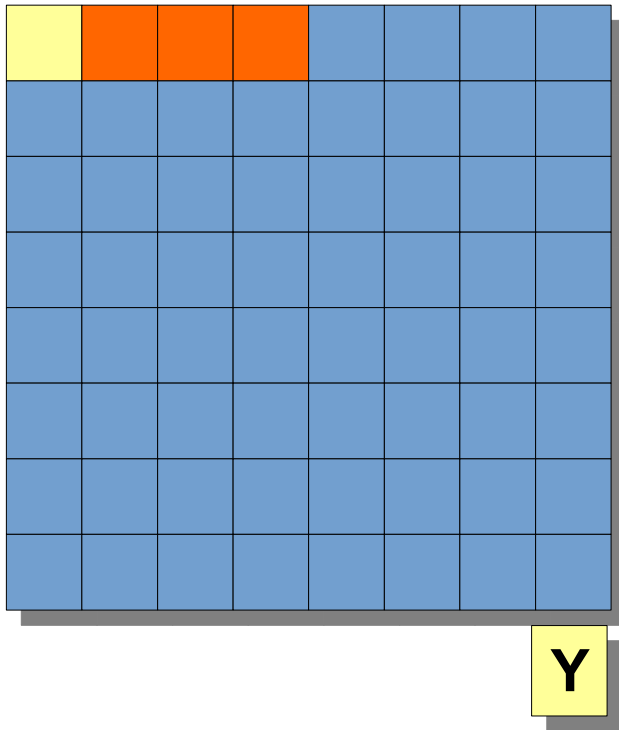


Z

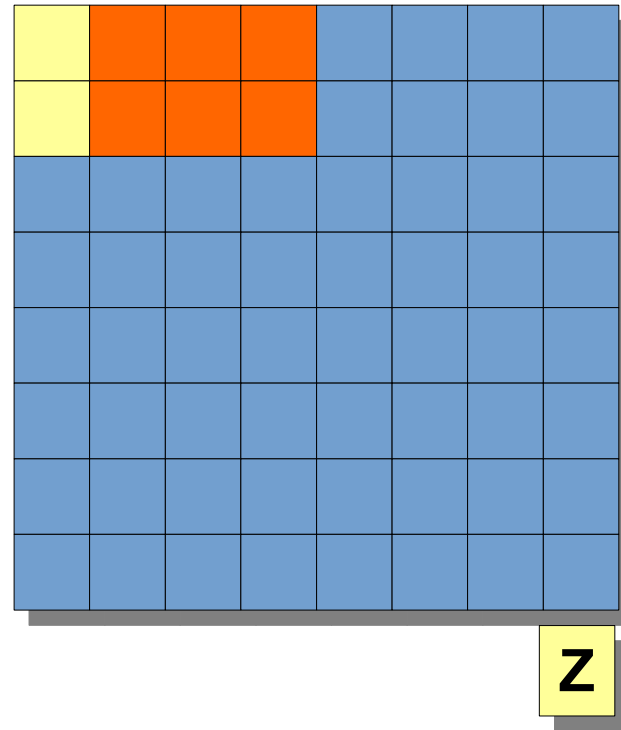
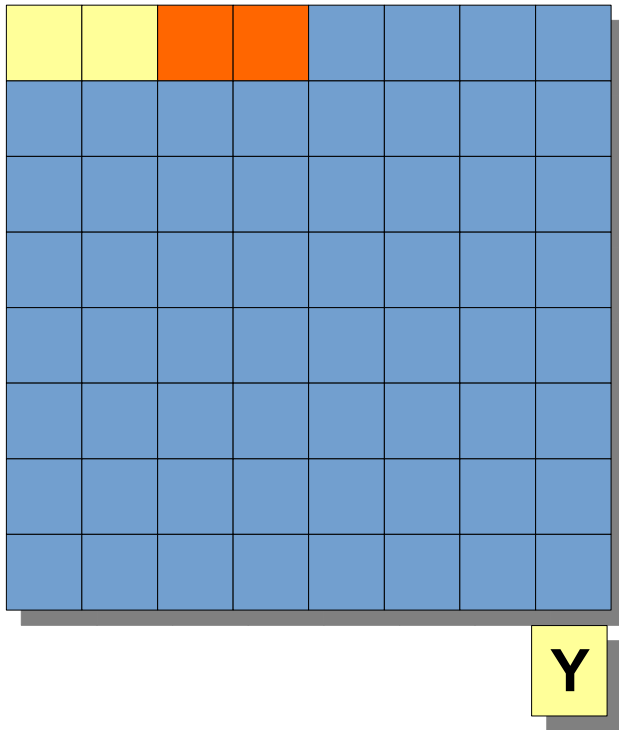
Idea of Blocking

Make full use of elements of when they are brought into the cache

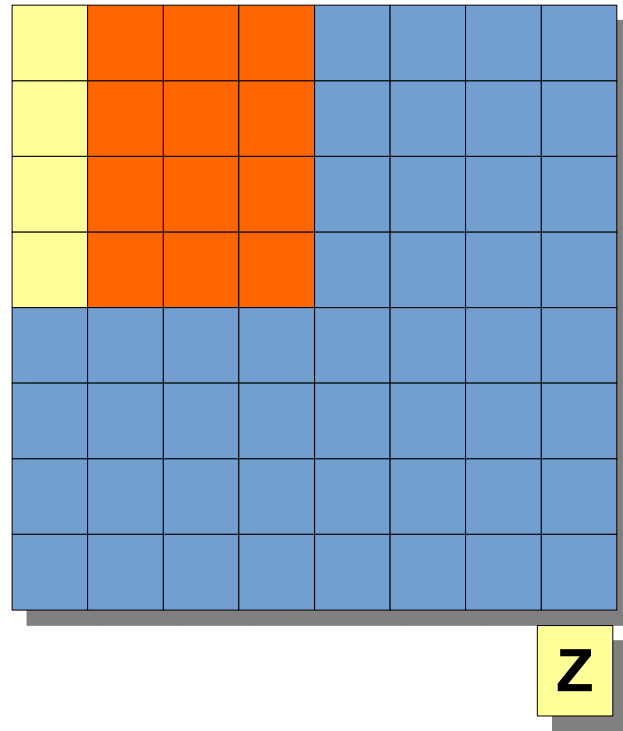
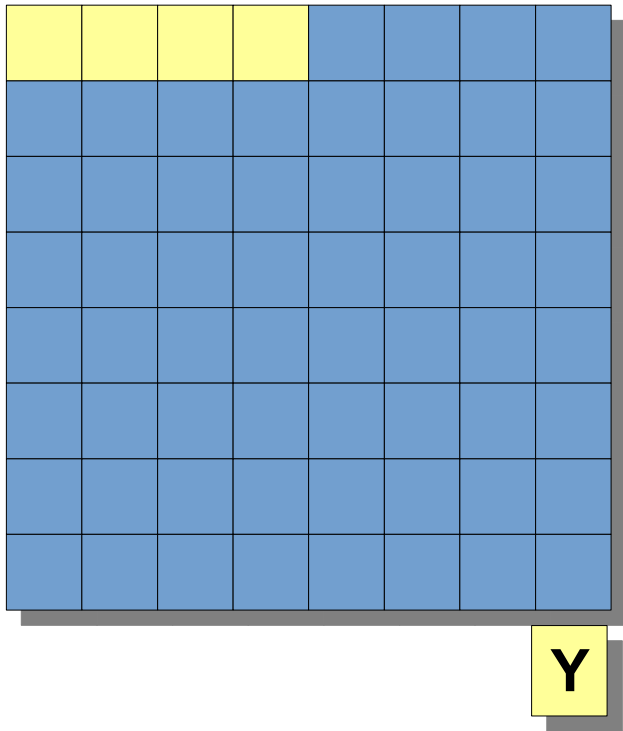
Blocking / Tiling



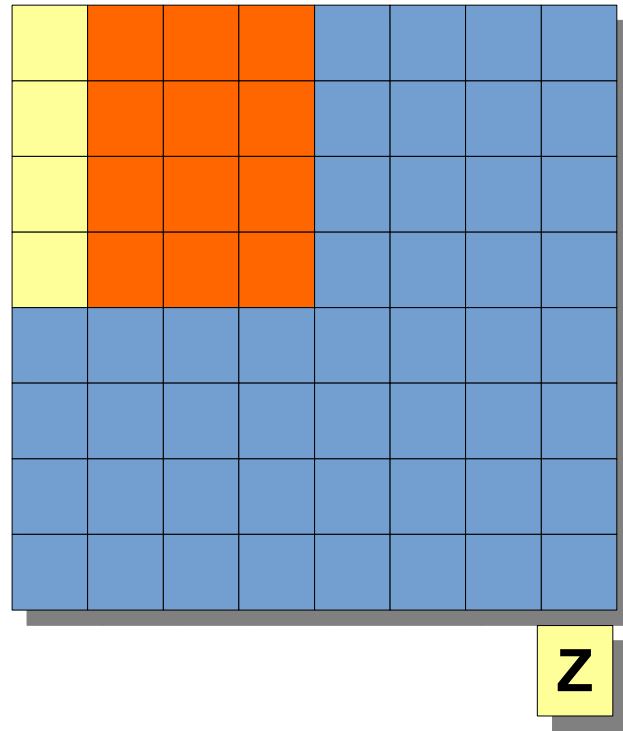
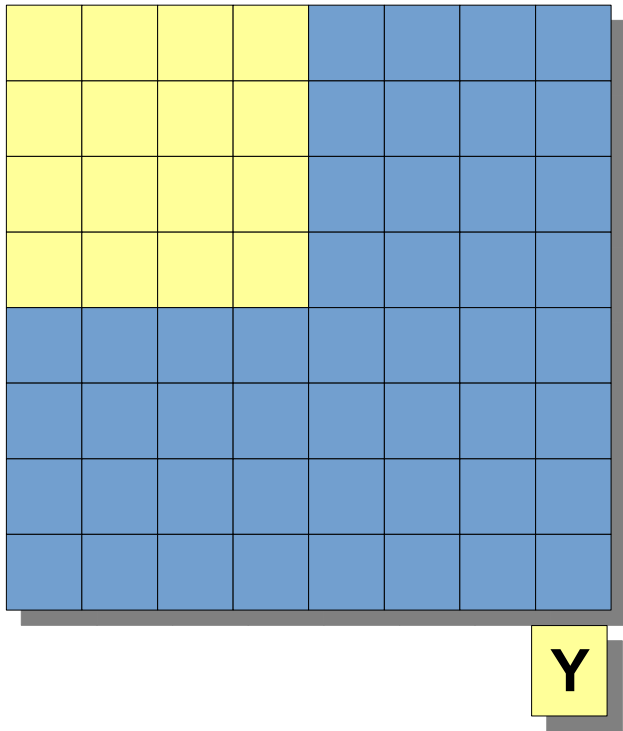
Blocking / Tiling



Blocking / Tiling



Blocking / Tiling



Blocking

- Make full use of the elements of Z when they are brought into the cache

(0,0)	(0,1)
(1,0)	(1,1)

Y

(0,0)	(0,1)
(1,0)	(1,1)

Z

X	Y x Z
0,0	$0,0 \times 0,0 + 0,1 \times 1,0$
1,0	$1,0 \times 0,0 + 1,1 \times 1,0$

Blocking

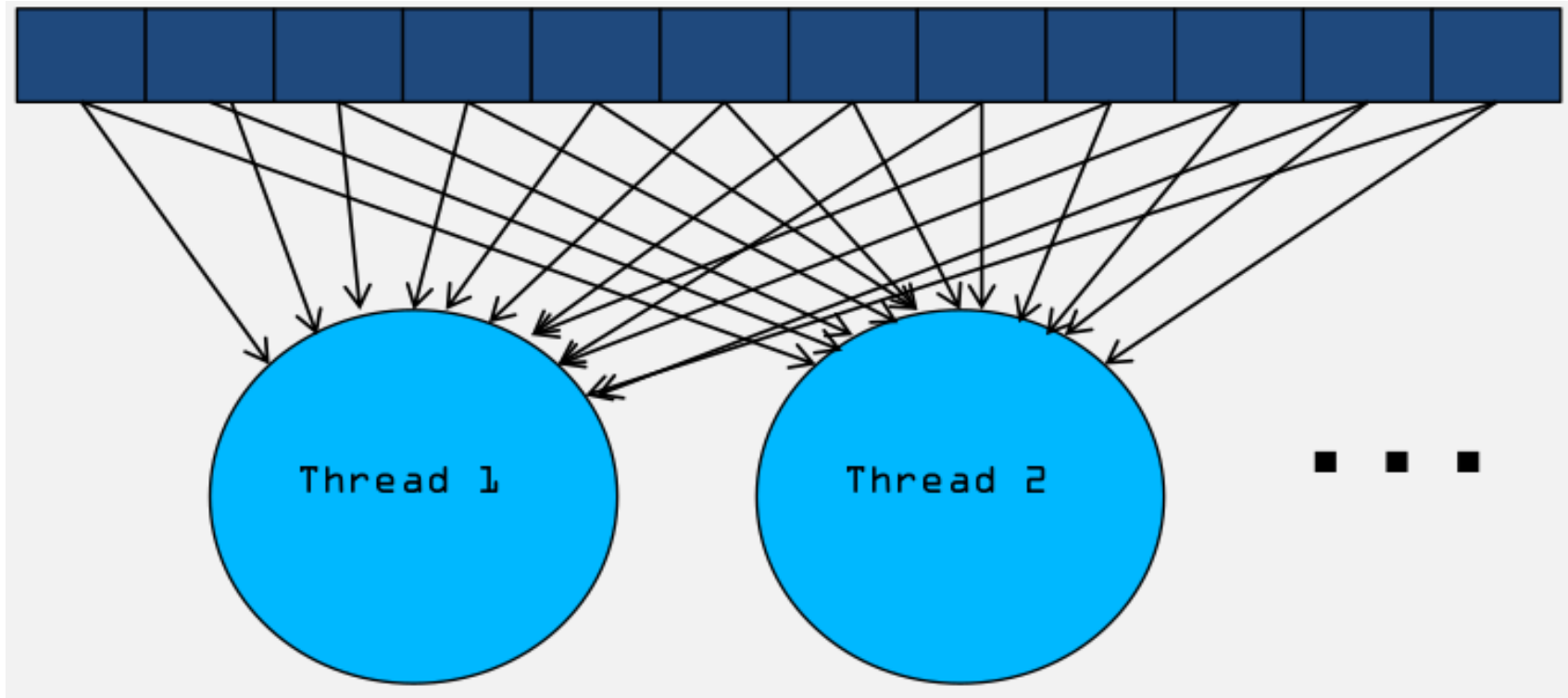
```
double X[N][N], Y[N][N], Z[N][N];

for (J=0; J<N; J+=B)
for (K=0; K<N; K+=B)

for (i=0; i<N; i++)
    for (j=J; j<min(J+B,N); j++)
        for (k=K,r=0; k<min(K+B,N); k++)
            r += Y[i][k] * Z[k][j];
        X[i][j] += r;
```

Extra Slides

Global Memory Access Pattern of the Basic MM Kernel



Computation after Phase 1 Loads

