CUDA Memories

Memory Access Efficiency

Memory Access Efficiency

- Maximum performance impact after parallelizing the code?
  - Simple CUDA kernels will likely achieve only a small fraction of the potential speed of the underlying hardware
  - Methods to circumvent such congestions
    - Global memory (DRAM) has long access latencies
    - Traffic congestion in the global memory access paths
    - SMs are idle

CGMA ratio

CGMA ratio

- ***Compute to Global Memory Access Ratio***:
  - Defined as the number of floating point calculations performed for each access to the global memory within a critical region of a CUDA program.

CGMA ratio

- ***Compute to Global Memory Access Ratio***:
  - Defined as the number of floating point calculations performed for each access to the global memory within a critical region of a CUDA program.
- Example: In the matrix multiplication program,
  - Every iteration of for loop
    - 2 global memory accesses
    - 1 FP multiplication
    - 1 FP addition
  - Ratio of floating-point calculation to global memory access operation is 1 to 1.

CGMA ratio

CGMA ratio

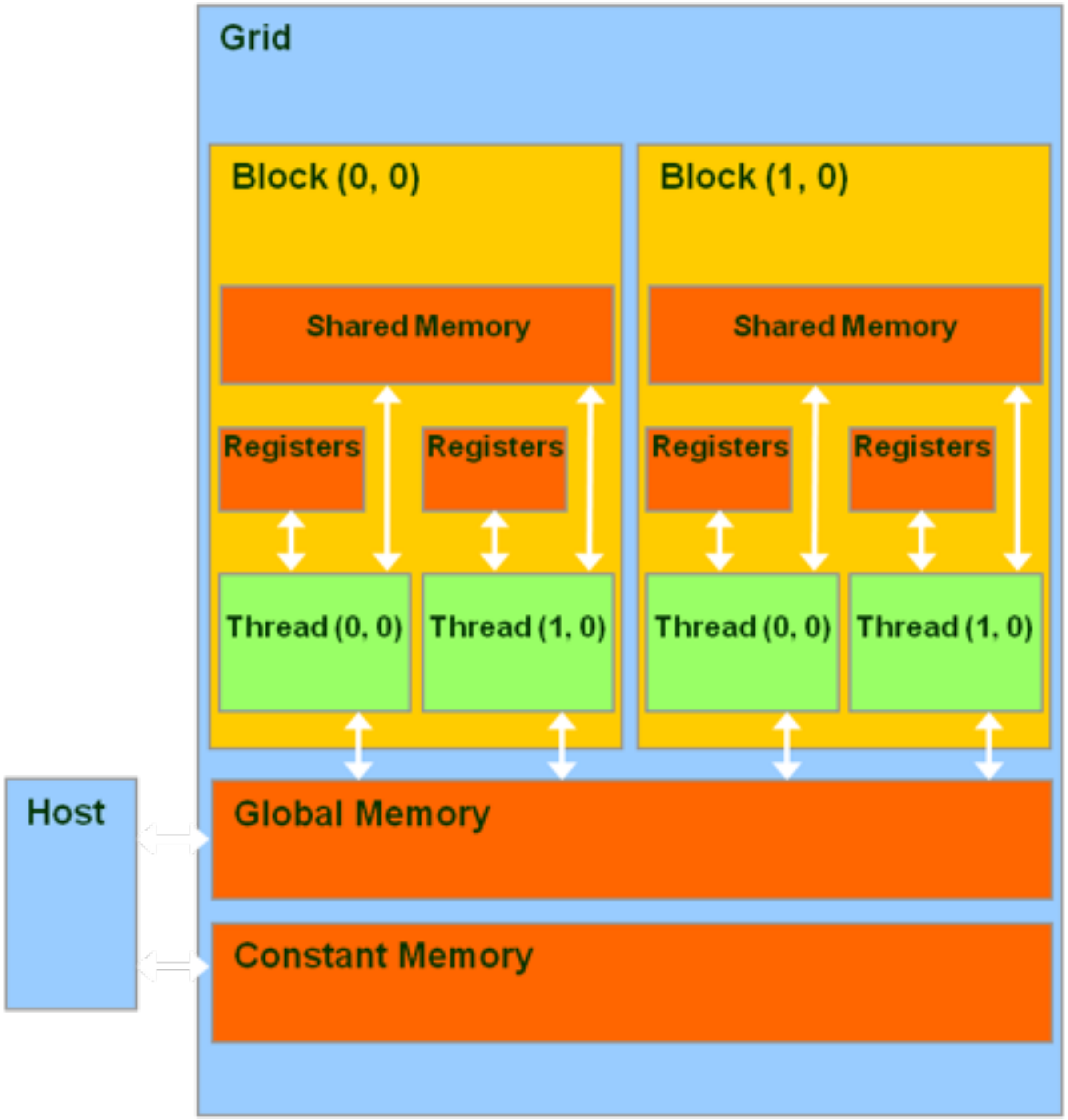
- The CGMA ratio has major implications on the performance of a CUDA kernel.

CGMA ratio

- The CGMA ratio has major implications on the performance of a CUDA kernel.
- Example : G80 card
  - Peak Compute Capacity:367 GFLOPs
  - 86.4 (GB/s) of global memory access bandwidth
  - For a Single Precision (4byte) datum, load (86.4/4)=21.6 giga single precision data
  - For CGMA=1, 21.6 billion FP operations /sec (gigaflops) – Max Limit
  - Compare Peak compute capacity and the Max limit

Calls for Better CGMA ratio!

CUDA Device Memory Types



- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory

CUDA Variable Types

CUDA Variable Types

- **Scope**:
  - Identifies the range of threads that can access the variable: by a single thread only, by all threads of a block, or by all threads of all grids
  - If scope is a single thread, a private version of the variable is created for every thread

CUDA Variable Types

CUDA Variable Types

- **Lifetime :**
  - Specifies the portion of the program's execution duration when the variable is available for use : either within a kernel's invocation or throughout the entire application
  - Lifetime within Kernel invocation
    - Declared inside function body and can be used only by kernel's code
    - Contents of the variable are not maintained across the kernel invocations
  - Lifetime throughout the application
    - Declared outside of any function body
    - Contents of variable are maintained throughout the application execution

CUDA variable types

Variable Declaration	Memory	Scope	Lifetime
int var;	Register	Thread	Kernel
int array_var[10];	Local	Thread	Kernel
__shared__ int shared_var;	Shared	Block	Kernel
__device__ int global_var;	Global	Grid	Application
__constant__ int constant_var;	Constant	Grid	Application

CUDA variable types

- **Automatic scalar variables**
  - Placed into registers
  - Private copy generated for every thread that executes the kernel function
  - When a thread terminates all the variables cease to exist
  - Accessing is fast and parallel
  - Care must be taken not to exceed the capacity of register storage

CUDA variable types

- **Automatic array variables**
  - Stored in global memory
  - May incur long access delays and congestions
  - Private version of each array is created for and used by every thread
- **Constant variable (__constant__)**
  - All threads in all grids see its same version
  - Often used for variables that provide input values to kernel functions
  - Stored in global memory but cached for efficient access

CUDA variable types

- **Shared variable (__shared__)**
  - Efficient means for collaboration of threads within a block
  - Declaration resides with kernel or device function
  - All threads of a block see same version
  - Private version created for and used by each thread block during kernel execution
  - When kernel terminates, contents of its shared variables cease to exist
  - Access is fast and parallel
  - Used to hold parts of global memory data that are used frequently used in execution

CUDA variable types

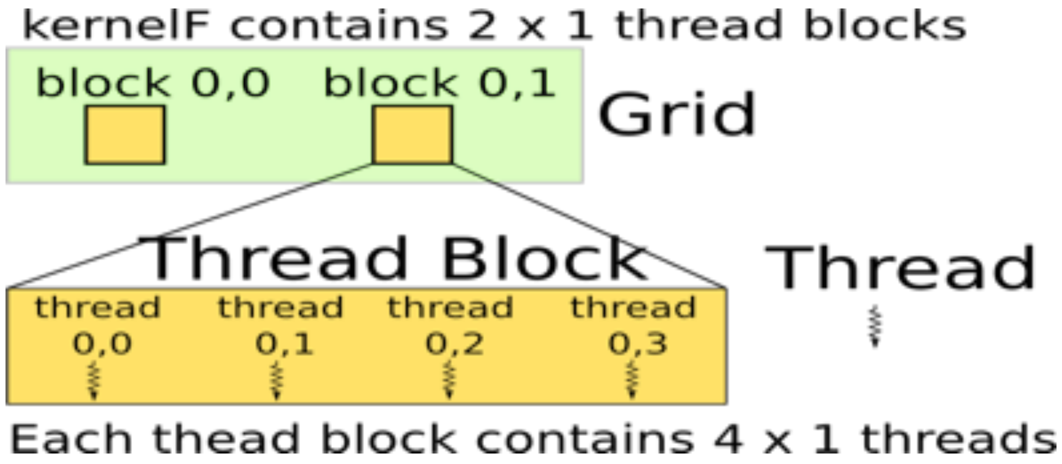
- **Device variable (__device__)**
  - Global variable placed in global memory
  - Access is slow
  - Advantage: visible to all threads of all kernels
  - Contents persist through the entire execution
  - Used as a means of collaboration among threads across blocks
  - Used to pass information from one kernel invocation to another

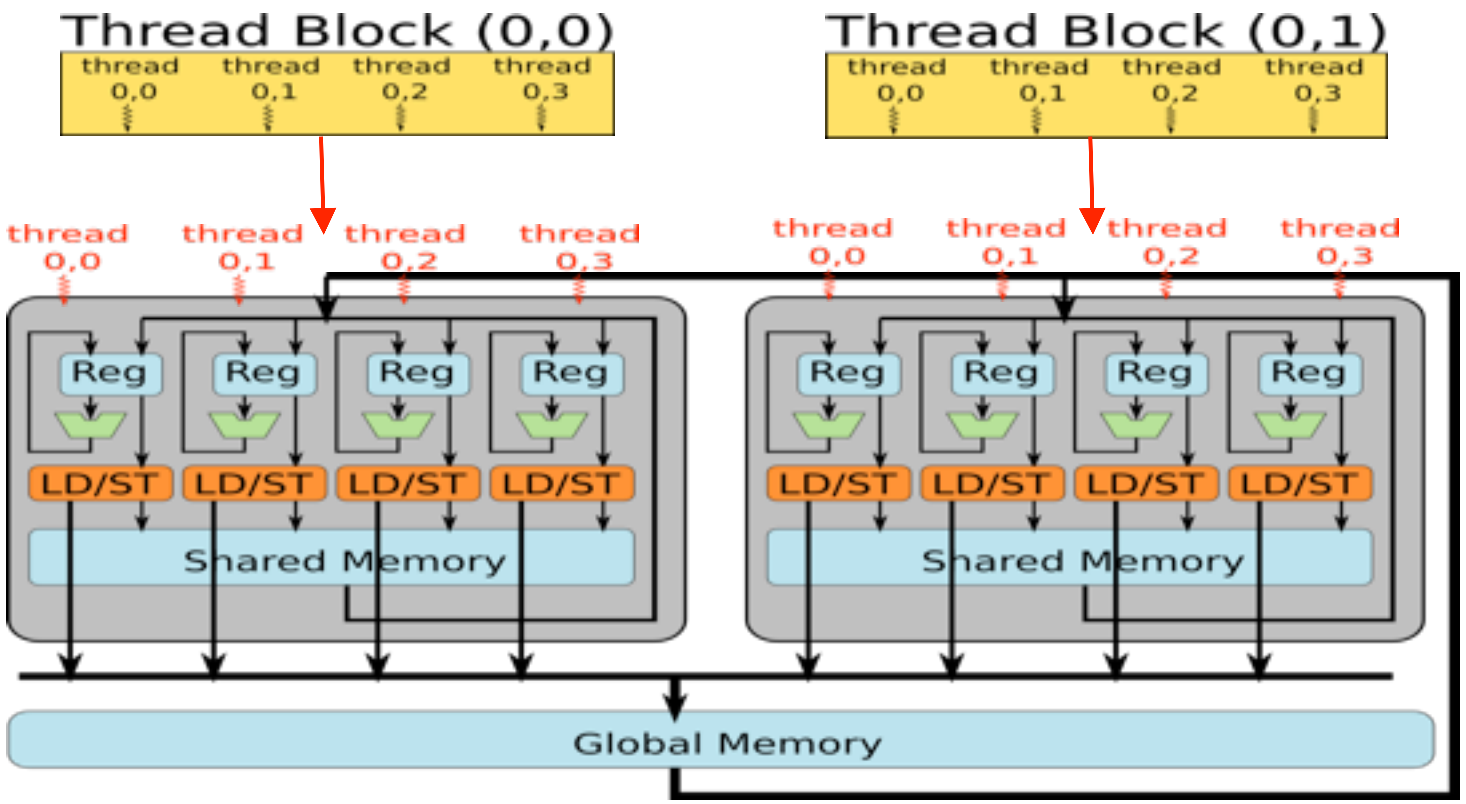
Shared memory

Shared memory

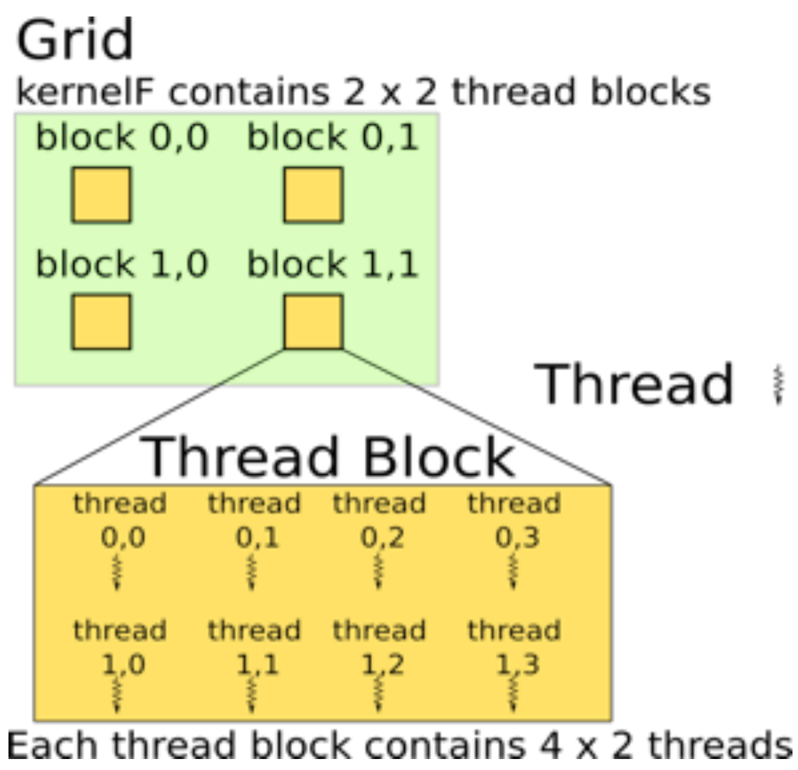
- A strategy for reducing global memory traffic
  - Global memory – Slow
  - Shared memory – Fast
  - Use Shared memory
  - Divide data sets into *tiles* such that each tile fits into the shared memory
  - Do the computation on these tiles independently

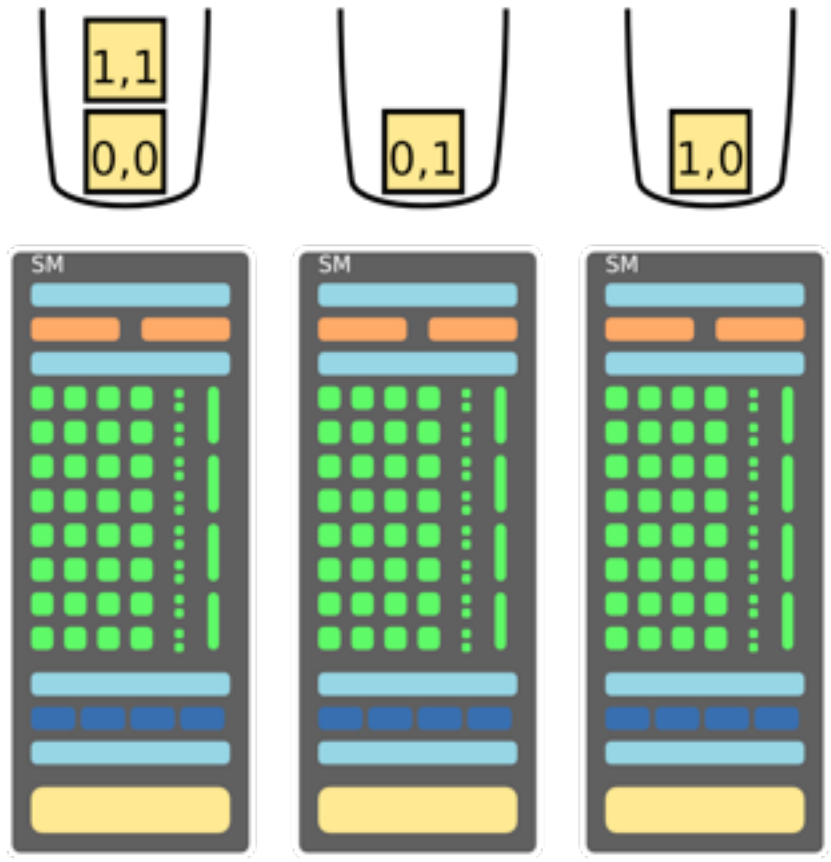
How Are Threads Scheduled?





Blocks Are Dynamically Scheduled



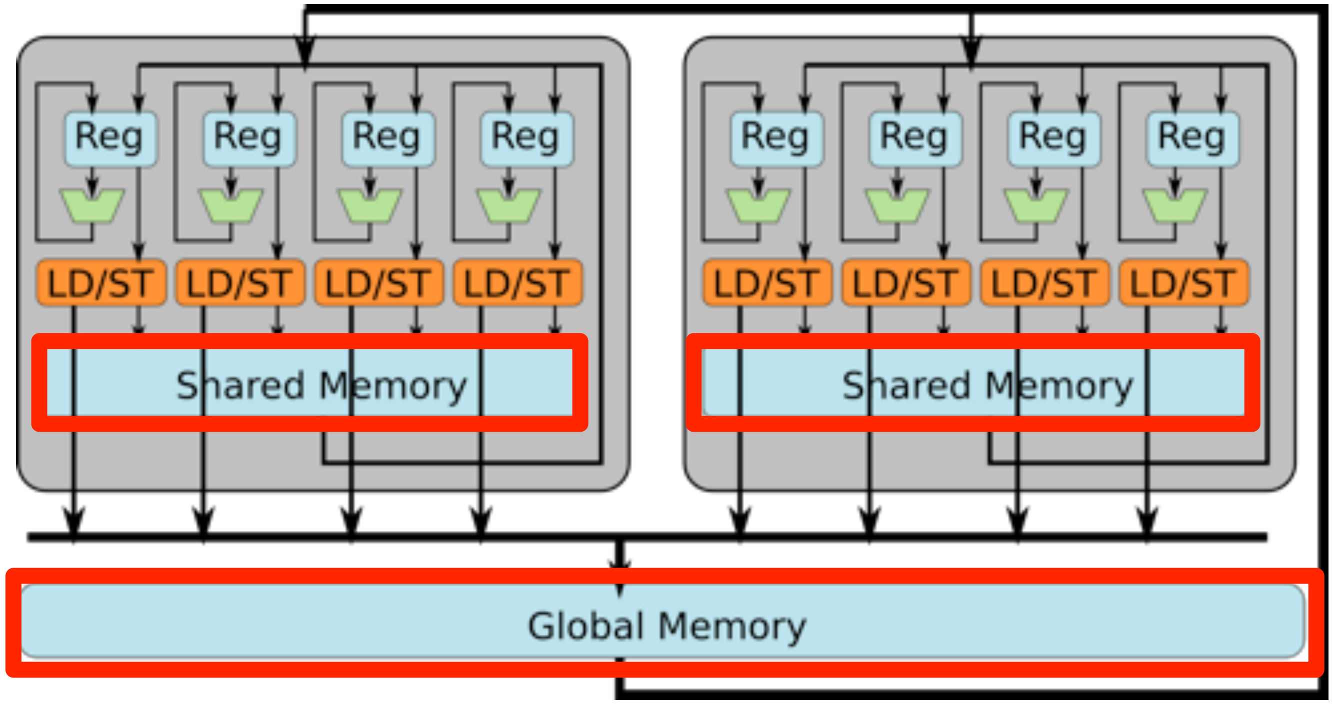


Utilizing Memory Hierarchy

Memory
access
latency

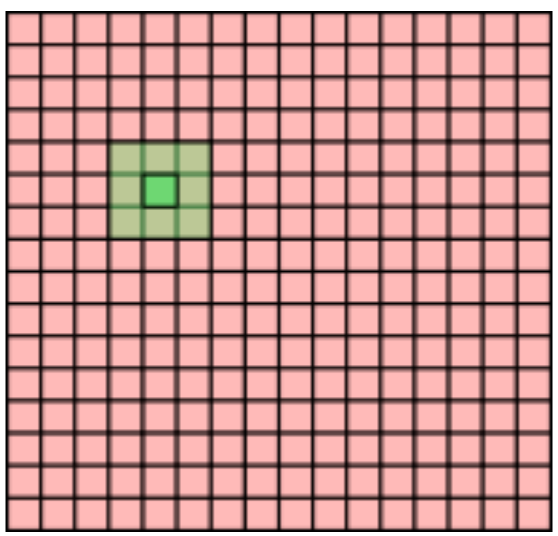
several cycles

100+ cycles



Example: Average Filters

Average over a 3x3 window for a 16x16 array

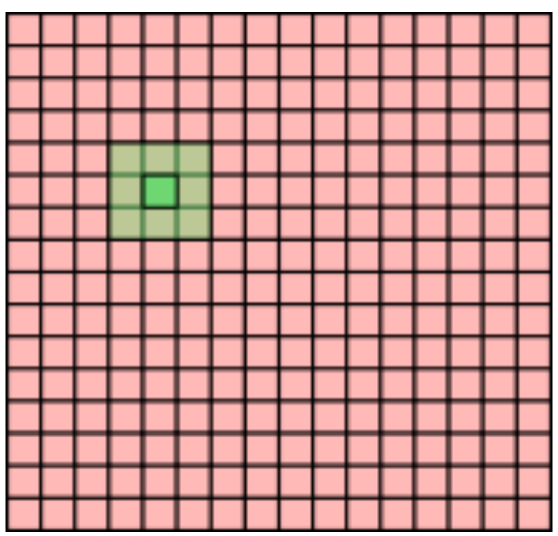


```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__   kernelF(A){
   i = threadIdx.y;
   j = threadIdx.x;
   tmp = (A[i-1][j-1] + A[i-1][j] +
            ... + A[i+1][i+1] ) / 9;
       A[i][j] = tmp;
}
```

Each thread loads 9 elements from global memory.
It takes hundreds of cycles.

Example: Average Filters

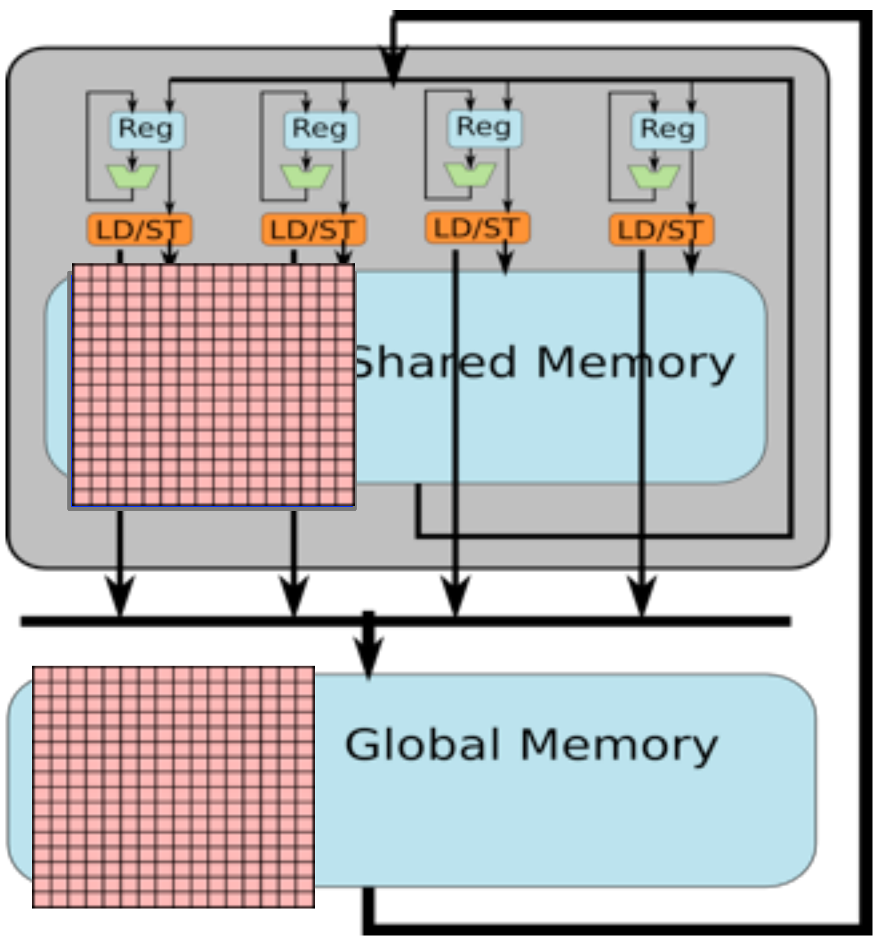
Average over a 3x3 window for a 16x16 array



```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                    ... + smem[i+1][i+1] ) / 9;
}
```

Example: Average Filters

Average over a 3x3 window
for a 16x16 array




```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__  smem[16][16];   allocate shared mem
    i = threadIdx.y;
    j = threadIdx.x;               Each thread loads one element from
                                   global memory.
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                   ... + smem[i+1][i+1] ) / 9;
}
```

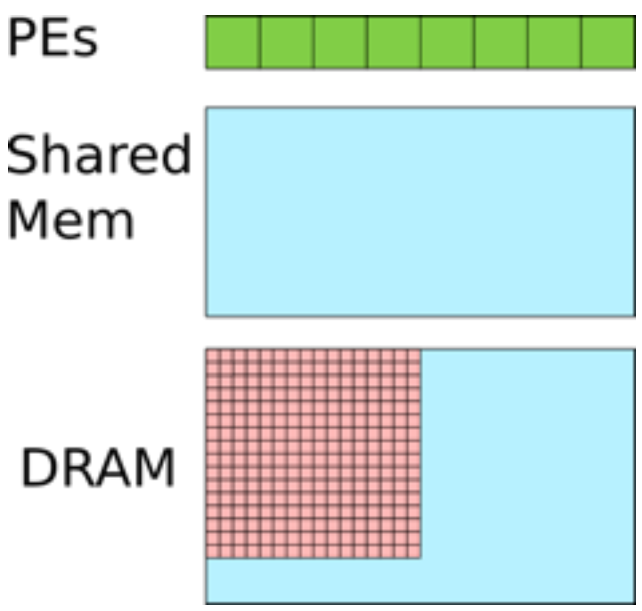
However, the Program Is Incorrect

Hazards!

```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                      ... + smem[i+1][i+1] ) / 9;
}
```

Let's See What's Wrong

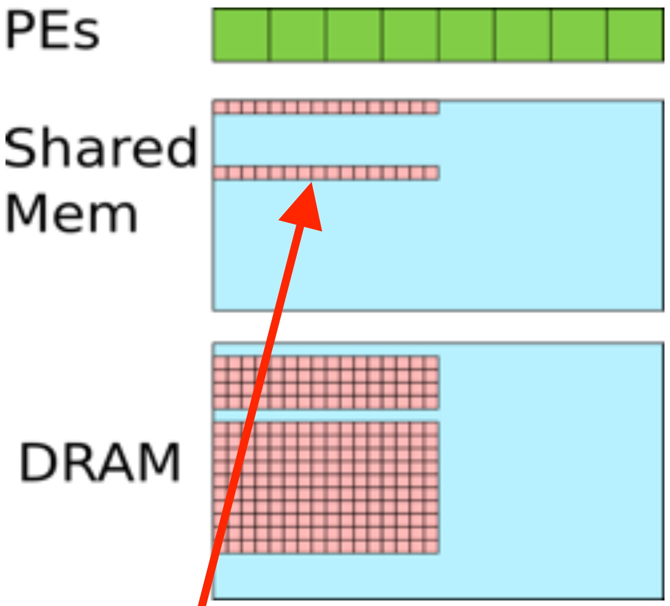
Assume 256 threads are scheduled on 8 PEs.




```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;            Before load instruction
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                ... + smem[i+1][i+1] ) / 9;
}
```

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

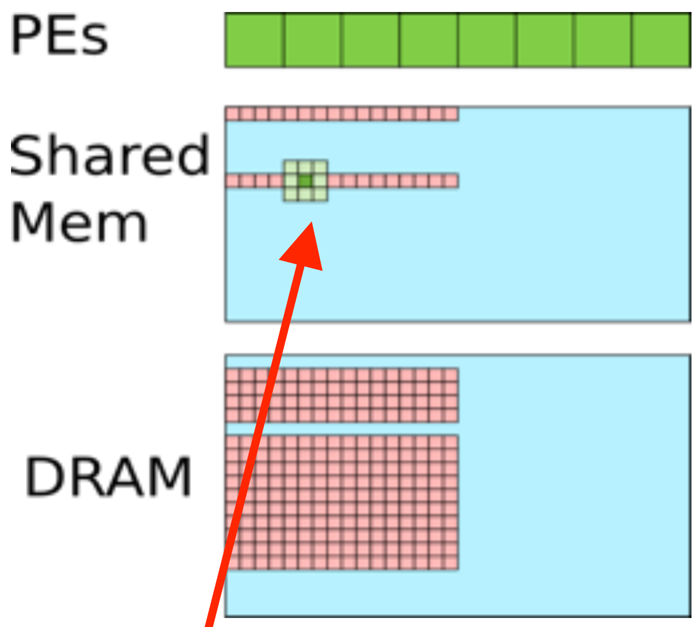



```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                   ... + smem[i+1][i+1] ) / 9;
}
```

Some threads finish the load earlier than others.

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

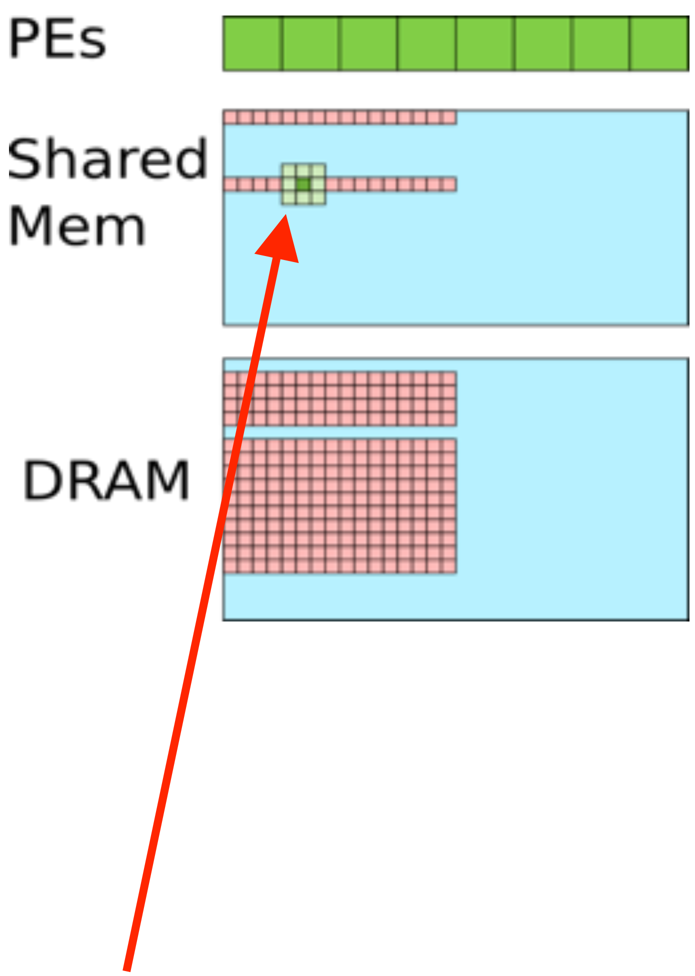



```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                ... + smem[i+1][i+1] ) / 9;
}
```

Some elements in the window are not yet loaded by other threads. Error!

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

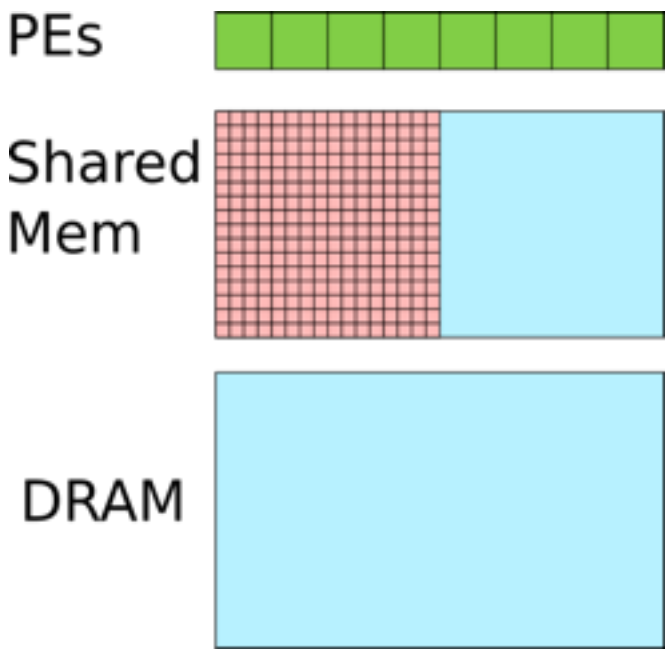



```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                ... + smem[i+1][i+1] ) / 9;
}
```

Some elements in the window are not yet loaded by other threads. Error!

How to Solve It?

Assume 256 threads are scheduled on 8 PEs.

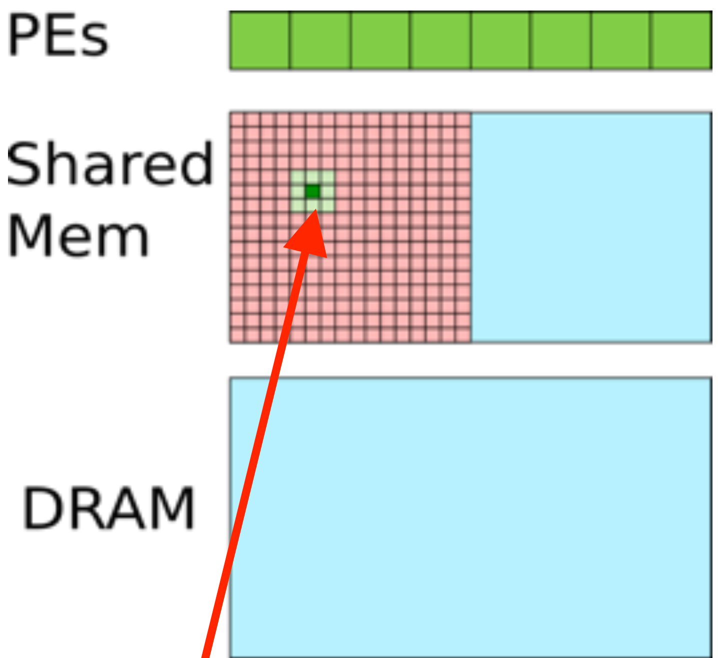



```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__  kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __ syncthreads();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                 ... + smem[i+1][i+1] ) / 9;
}
```

Wait until all threads hit barrier

Use Synchronization barrier

Assume 256 threads are scheduled on 8 PEs.

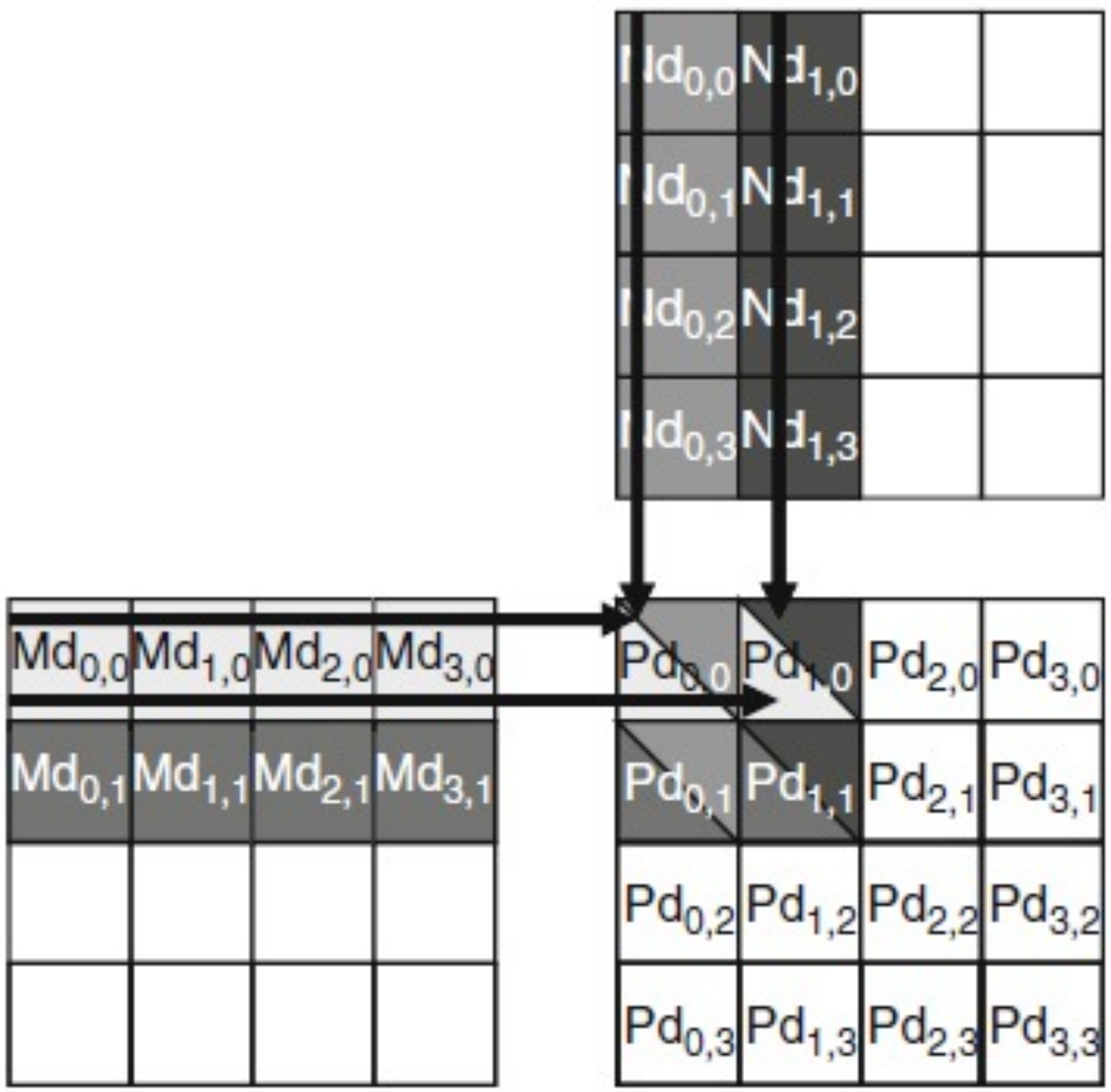


All elements in the window are loaded when each thread starts averaging.

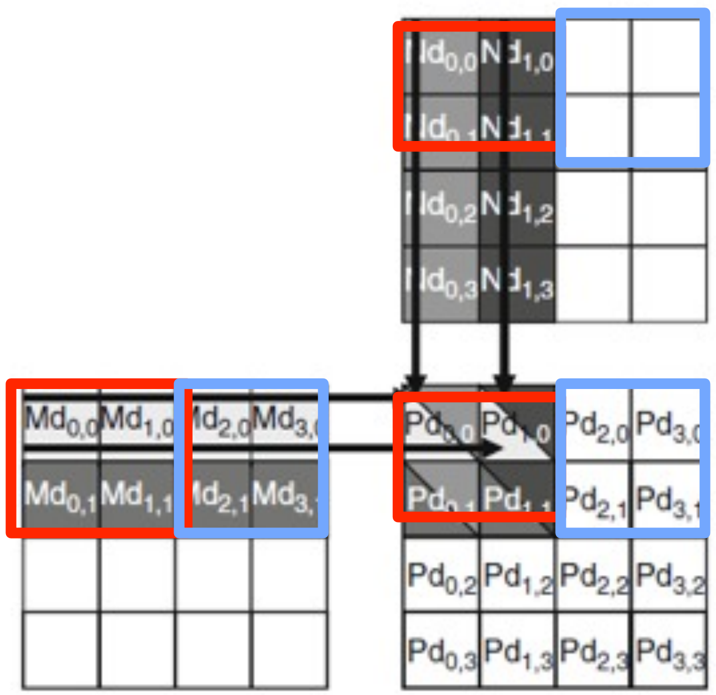
```
dim3 blocks(1,1), threads(16,16);
kernelF<<<blocks,threads>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __ syncthreads();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                ... + smem[i+1][i+1] ) / 9;
}
```

Matrix multiplication –Parallel code

Using Multiple blocks (using shared memory)



Global memory accesses performed by threads in $\text{block}_{0,0}$

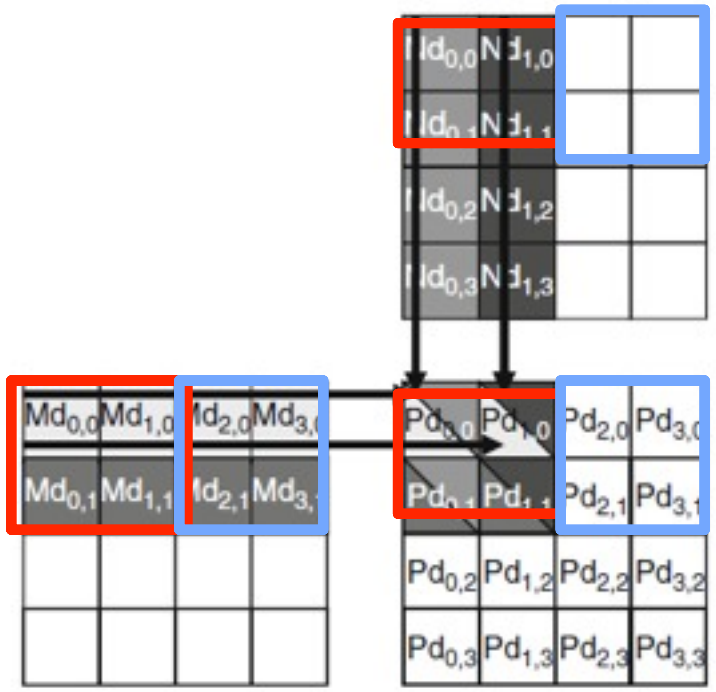


Access order

$Pd_{0,0}$ Thread(0,0)	$Pd_{1,0}$ Thread(1,0)	$Pd_{0,1}$ Thread(0,1)	$Pd_{1,1}$ Thread(1,1)
$Md_{0,0} * Nd_{0,0}$	$Md_{0,0} * Nd_{1,0}$	$Md_{0,1} * Nd_{0,0}$	$Md_{0,1} * Nd_{1,0}$
$Md_{1,0} * Nd_{0,1}$	$Md_{1,0} * Nd_{1,1}$	$Md_{1,1} * Nd_{0,1}$	$Md_{1,1} * Nd_{1,1}$
$Md_{2,0} * Nd_{0,2}$	$Md_{2,0} * Nd_{1,2}$	$Md_{2,1} * Nd_{0,2}$	$Md_{2,1} * Nd_{1,2}$
$Md_{3,0} * Nd_{0,3}$	$Md_{3,0} * Nd_{1,3}$	$Md_{3,1} * Nd_{0,3}$	$Md_{3,1} * Nd_{1,3}$

Global memory accesses performed by threads in $block_{0,0}$

Every M and N element is accessed exactly twice! (for 2 x 2 block)



Access order

$Pd_{0,0}$ Thread(0,0)	$Pd_{1,0}$ Thread(1,0)	$Pd_{0,1}$ Thread(0,1)	$Pd_{1,1}$ Thread(1,1)
$Md_{0,0} * Nd_{0,0}$	$Md_{0,0} * Nd_{1,0}$	$Md_{0,1} * Nd_{0,0}$	$Md_{0,1} * Nd_{1,0}$
$Md_{1,0} * Nd_{0,1}$	$Md_{1,0} * Nd_{1,1}$	$Md_{1,1} * Nd_{0,1}$	$Md_{1,1} * Nd_{1,1}$
$Md_{2,0} * Nd_{0,2}$	$Md_{2,0} * Nd_{1,2}$	$Md_{2,1} * Nd_{0,2}$	$Md_{2,1} * Nd_{1,2}$
$Md_{3,0} * Nd_{0,3}$	$Md_{3,0} * Nd_{1,3}$	$Md_{3,1} * Nd_{0,3}$	$Md_{3,1} * Nd_{1,3}$

Global memory accesses performed by threads in $\text{block}_{0,0}$

Global memory accesses performed by threads in $\text{block}_{0,0}$

Collaborate 4 threads in their accesses to global memory ?

Global memory accesses performed by threads in $\text{block}_{0,0}$

Collaborate 4 threads in their accesses to global memory ?

Reduce global memory access by half!

Global memory accesses performed by threads in $\text{block}_{0,0}$

Collaborate 4 threads in their accesses to global memory ?

Reduce global memory access by half!

Reduction is proportional to the block size

Global memory accesses performed by threads in block

Global memory accesses performed by threads in block

- *The potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used.*

Global memory accesses performed by threads in block

- *The potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used.*
- With N×N blocks, the potential reduction of global memory traffic would be N.

Global memory accesses performed by threads in block

- *The potential reduction in global memory traffic in the matrix multiplication example is proportional to the dimension of the blocks used.*
- With N×N blocks, the potential reduction of global memory traffic would be N.
- That is, if we use 16×16 blocks, we could potentially reduce the global memory traffic to 1/16 through collaboration between threads.

Alternate Algorithm

Alternate Algorithm

- Collaborate threads to reduce global memory traffic

Alternate Algorithm

- Collaborate threads to reduce global memory traffic
- Collaboratively load Md and Nd elements into shared memory before using the elements for dot product

Alternate Algorithm

- Collaborate threads to reduce global memory traffic
- Collaboratively load Md and Nd elements into shared memory before using the elements for dot product
- Do not exceed the capacity of shared memory!

Alternate Algorithm

- Collaborate threads to reduce global memory traffic
- Collaboratively load Md and Nd elements into shared memory before using the elements for dot product
- Do not exceed the capacity of shared memory!
- Divide the matrices into smaller tiles

Alternate Algorithm

- Collaborate threads to reduce global memory traffic
- Collaboratively load Md and Nd elements into shared memory before using the elements for dot product
- Do not exceed the capacity of shared memory!
- Divide the matrices into smaller tiles
- Size of tiles chosen so that they fit in shared memory
  - Tile dimension equal those of the block

Alternate Algorithm

- Collaborate threads to reduce global memory traffic
- Collaboratively load Md and Nd elements into shared memory before using the elements for dot product
- Do not exceed the capacity of shared memory!
- Divide the matrices into smaller tiles
- Size of tiles chosen so that they fit in shared memory
  - Tile dimension equal those of the block
- Dot product performed by each thread is divided into phases

The Algorithm

The Algorithm

- Divide Md and Nd matrix into blocks of size 2×2

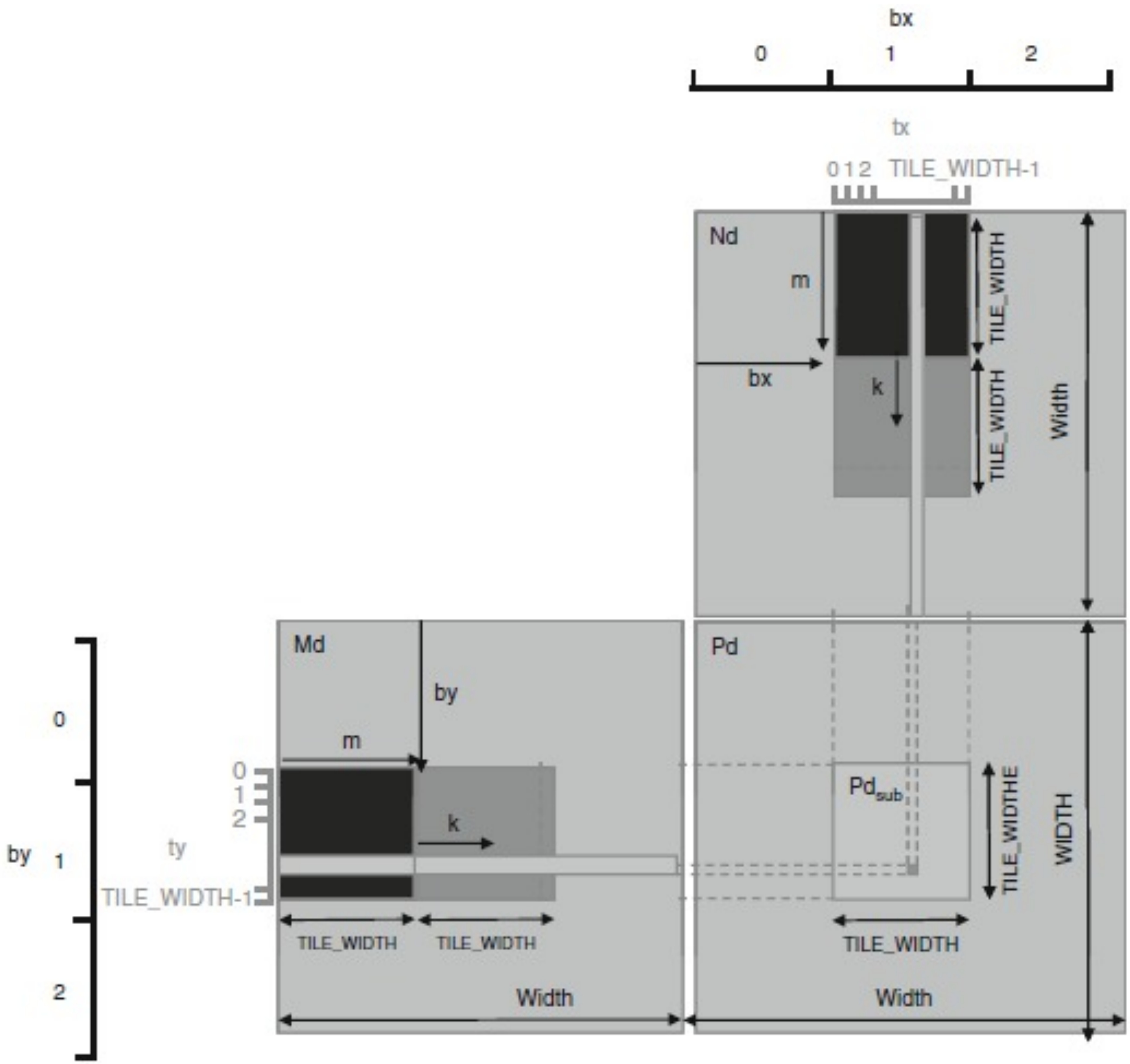
The Algorithm

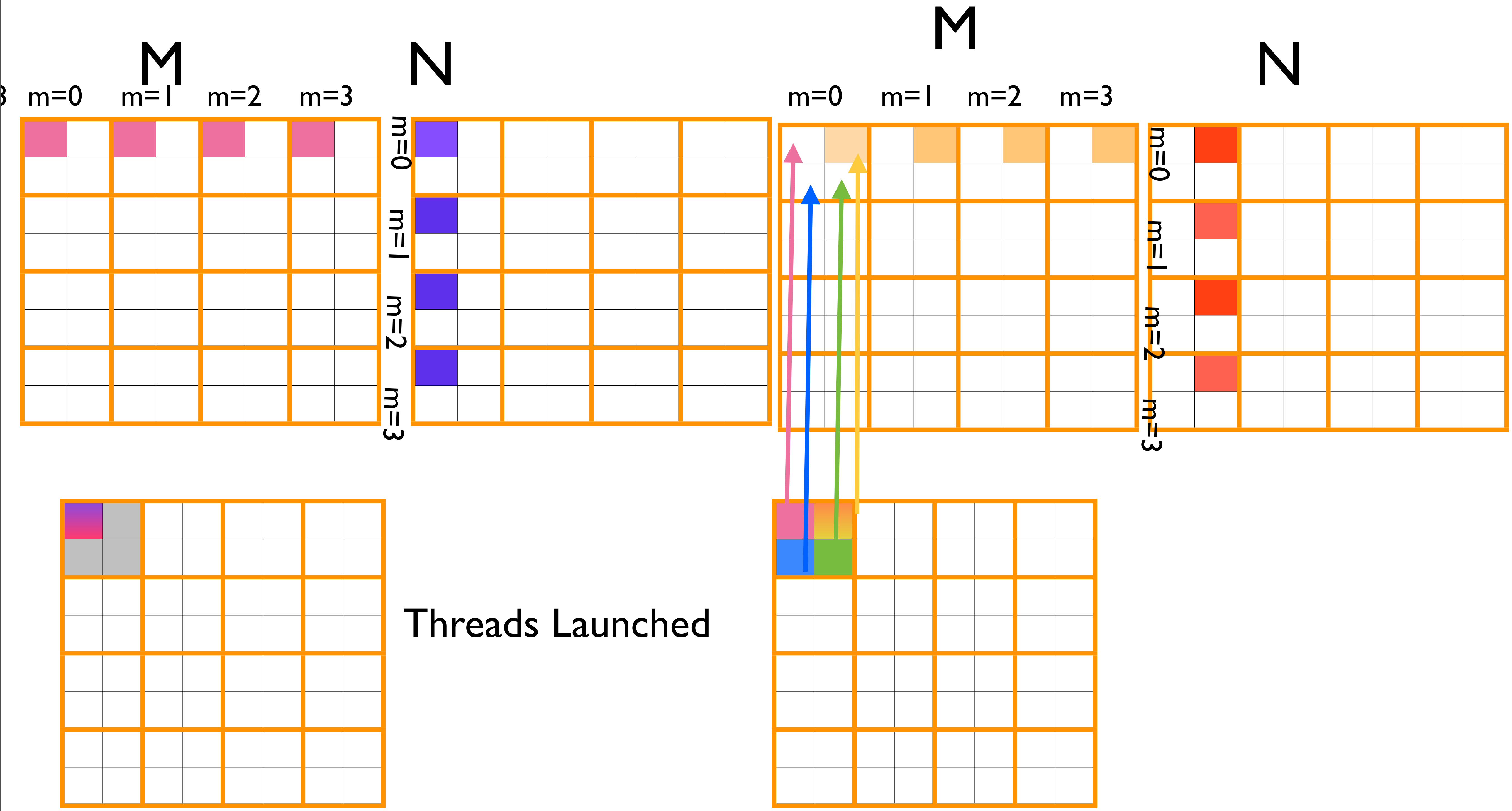
- Divide Md and Nd matrix into blocks of size 2×2
- Dot product in 2 phases
  - Threads in the block load one tile of Md and one tile of Nd into shared memory
  - Each thread in the block loads one element of Md and one element of Nd
  - Dot product is done after the tiles of Md and Nd are loaded into memory

The Algorithm

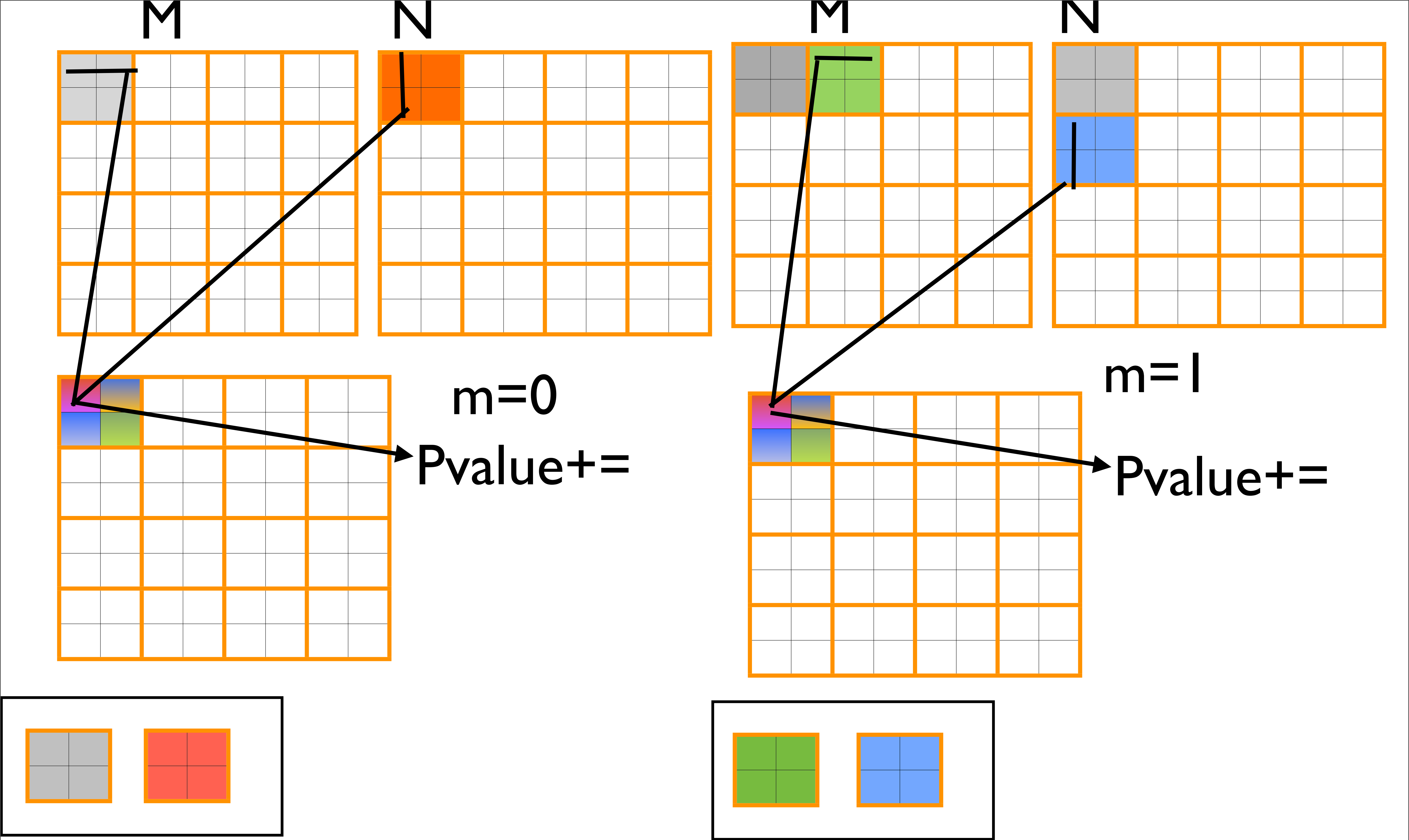
- Divide Md and Nd matrix into blocks of size 2×2
- Dot product in 2 phases
  - Threads in the block load one tile of Md and one tile of Nd into shared memory
  - Each thread in the block loads one element of Md and one element of Nd
  - Dot product is done after the tiles of Md and Nd are loaded into memory
- In general, Number of phases=Width/ Tile_Width

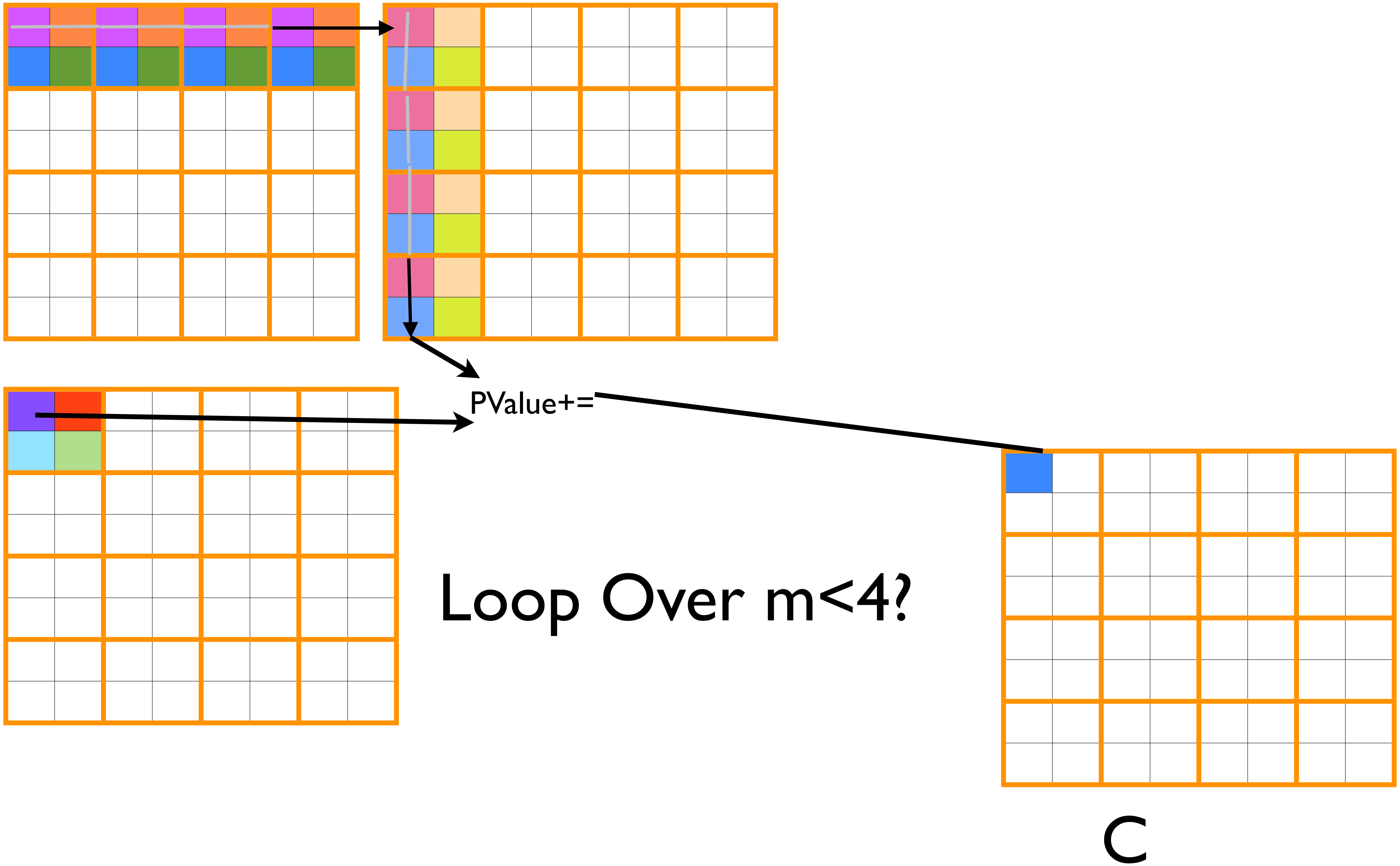
Calculating the row and column indices



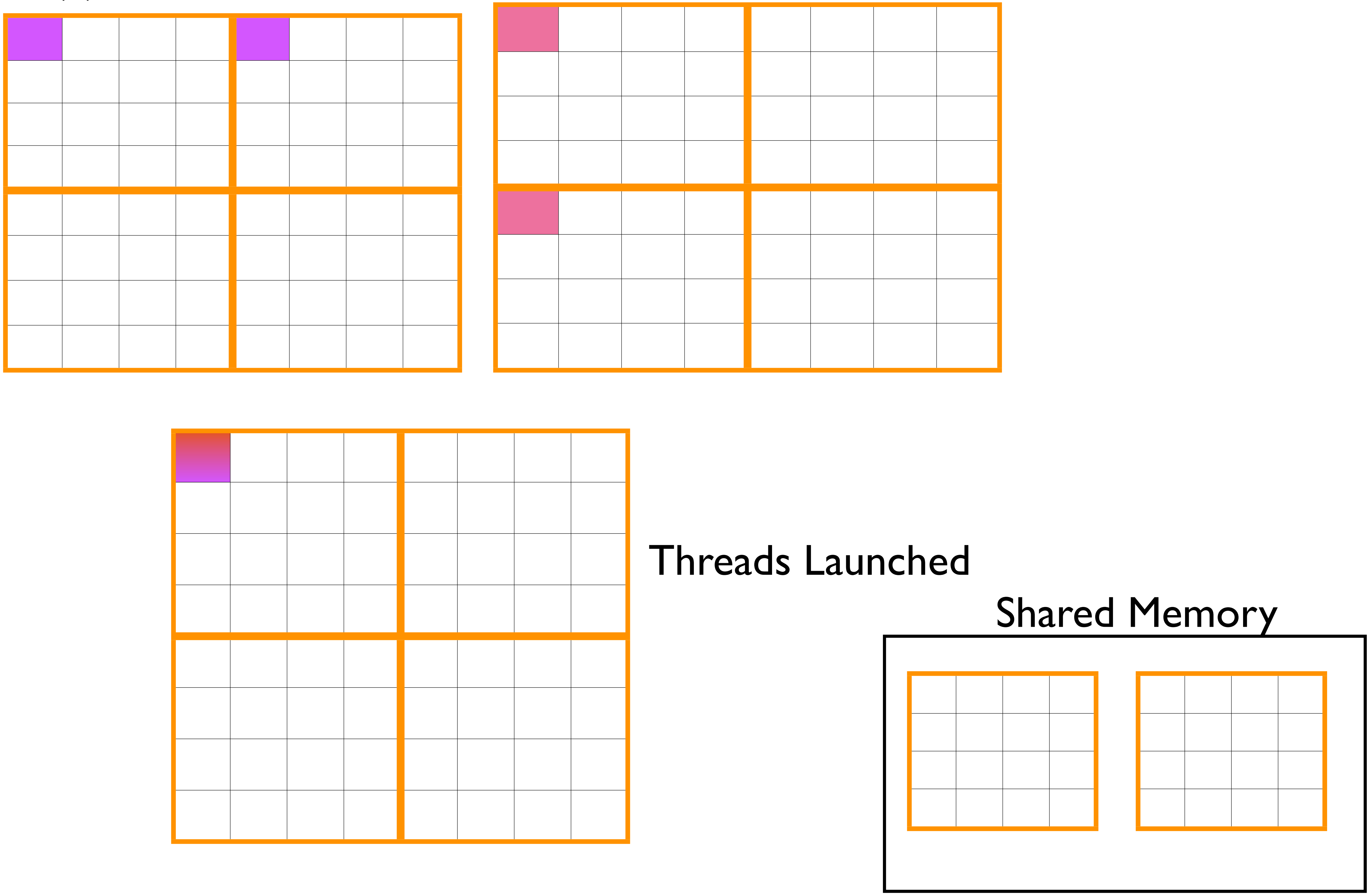


M
N
m=0
m=1
m=2
m=3
m=0
m=1
m=2
m=3
M
N
m=0
m=1
m=2
m=3
m=0
m=1
m=2
m=3
Threads Launched

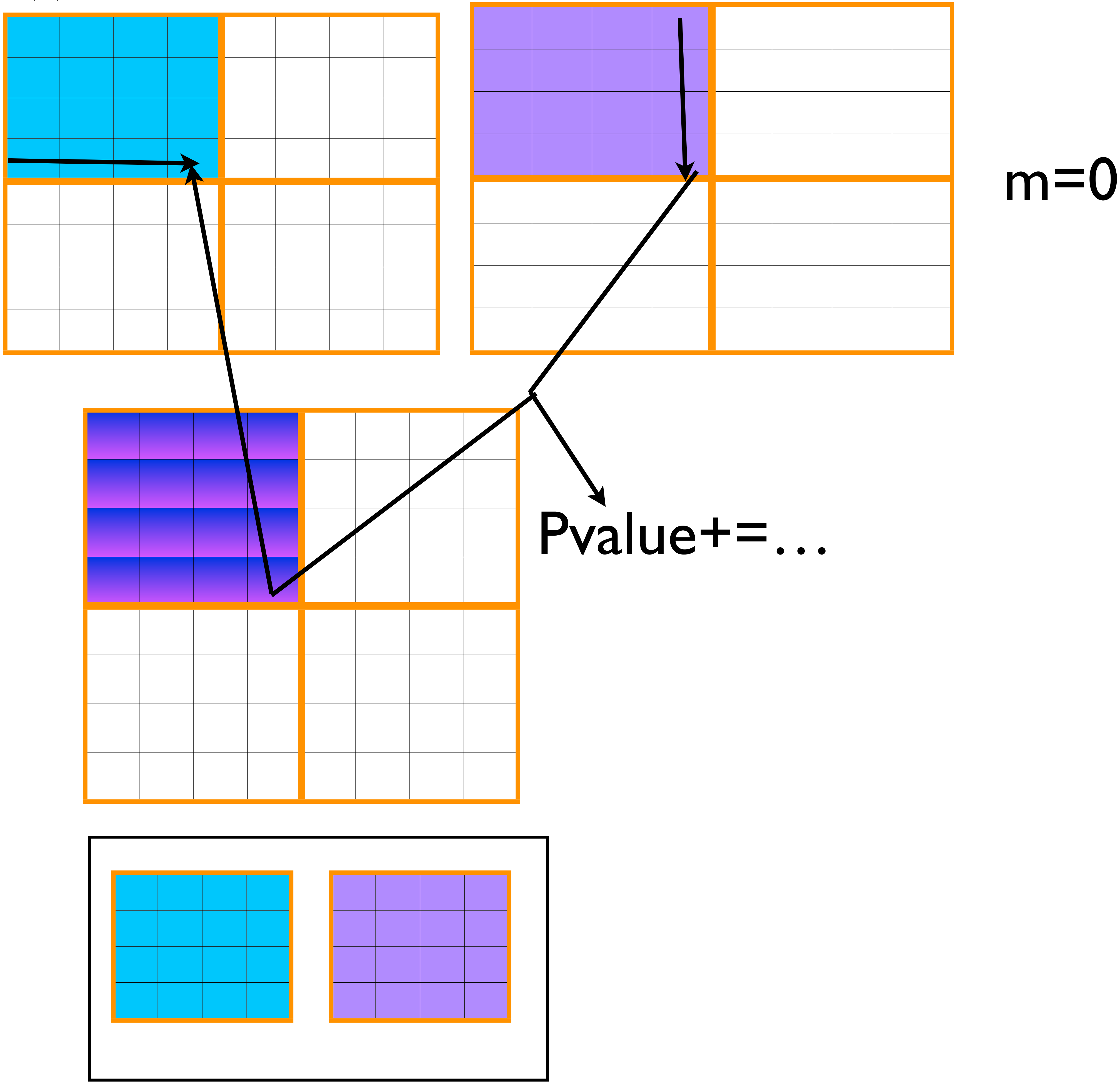

M
N
M
N
m=0
Pvalue+=
m=1
Pvalue+=


PValue+=
Loop Over m<4?
C

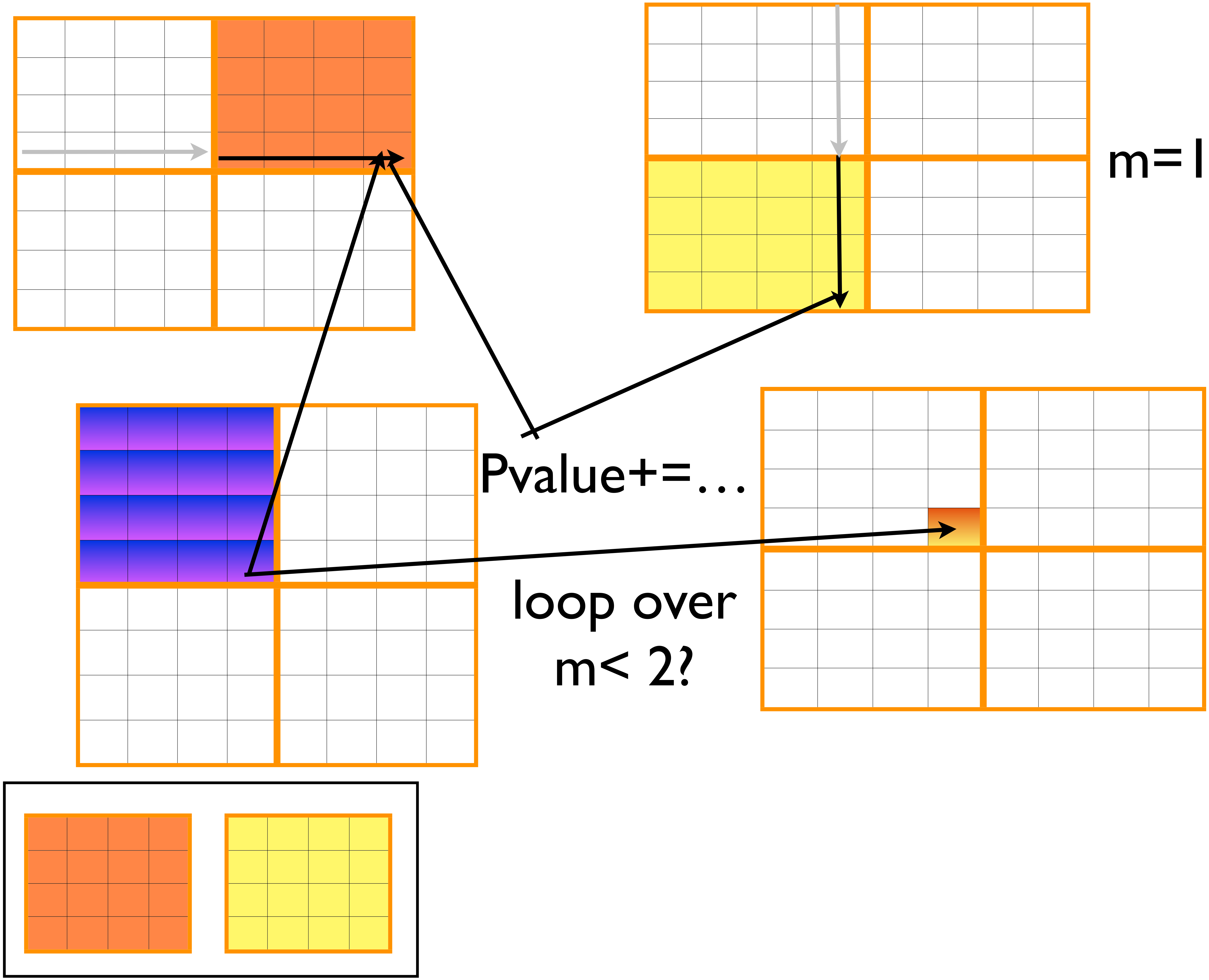
What if block dimension is 4?



What if block dimension is 4?...



What if block dimension is 4?...



Global memory access

- G80 card revisited
  - Theoretical global memory bandwidth =86.4GB/s
  - With tiled multiplication
    - Global memory accesses reduced by a factor of TILE_WIDTH
    - For 16×16 tiles, can reduce global memory accesses by a factor of 16.
    - CGMA increases from 1 to 16
    - Computation rate is (86.4/4)*16=345.6GFLOPS/s which is close to the theoretical computation rate of 367GFLOPS!
  - Removes global memory bandwidth as the limiting factor of matrix multiplication