CENG443
Heterogeneous Parallel Programming

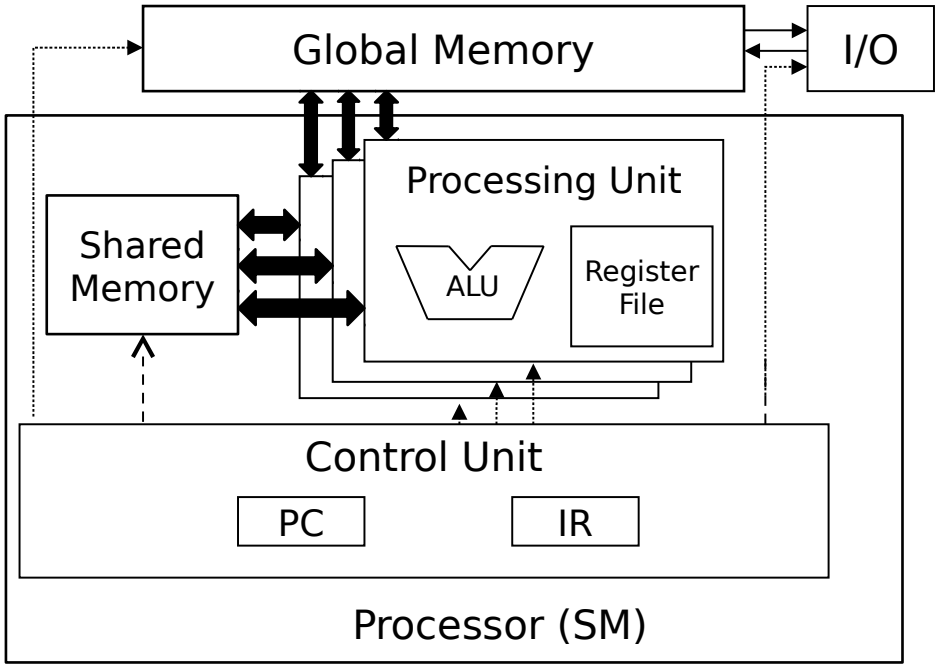
CUDA Performance

CUDA Kernel Performance

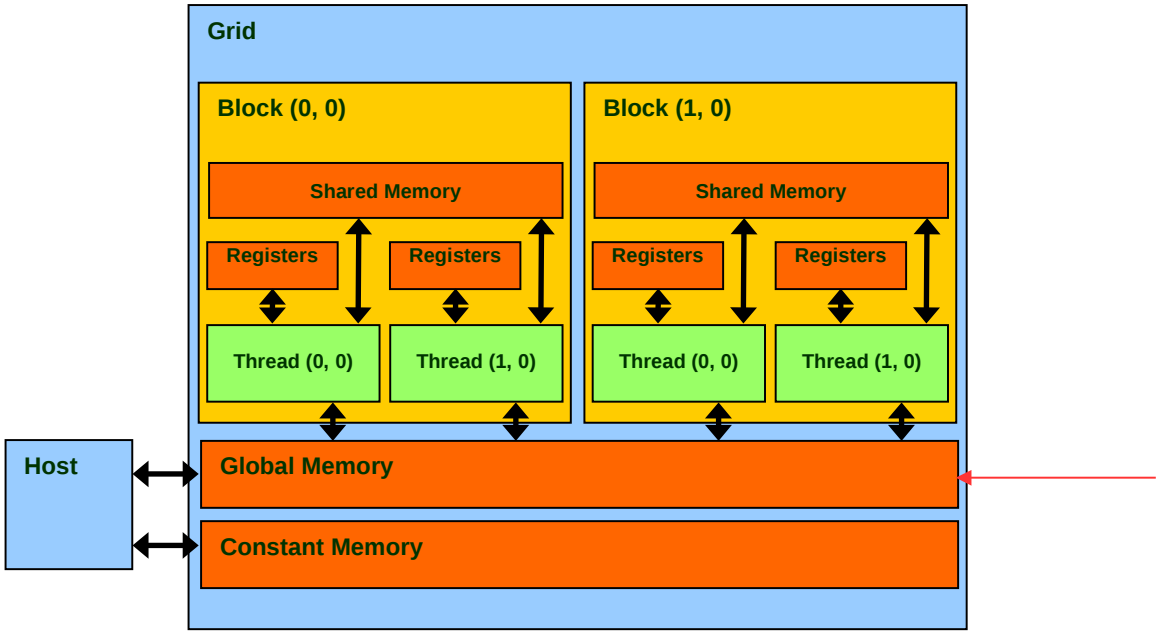
Global memory bandwidth

Warp scheduling

Hardware View of CUDA Memories



Global Memory



- Typically implemented in DRAM
- High access latency:
  400-800 cycles
- Finite access bandwidth
- Potential of traffic congestion
- Throughput up to 177GB/s

Global Memory

GPU device with the global memory bandwidth 1,000 GB/s, or 1 TB/s

4 bytes in each single-precision floating-point value

1000/4=250 giga single-precision operands per second to be loaded

- no more than 250 giga floating-point operations per second (GFLOPS)

Peak single-precision performance of 12 TFLOPS

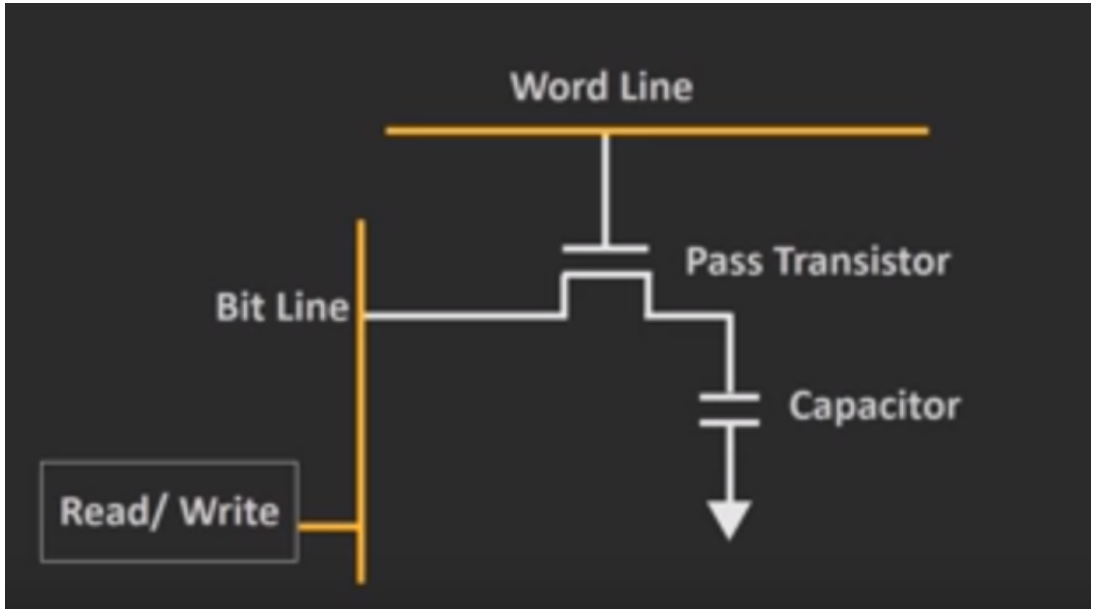
Tiny fraction = 250/12000 = 2%

Need to find ways of reducing global memory accesses!

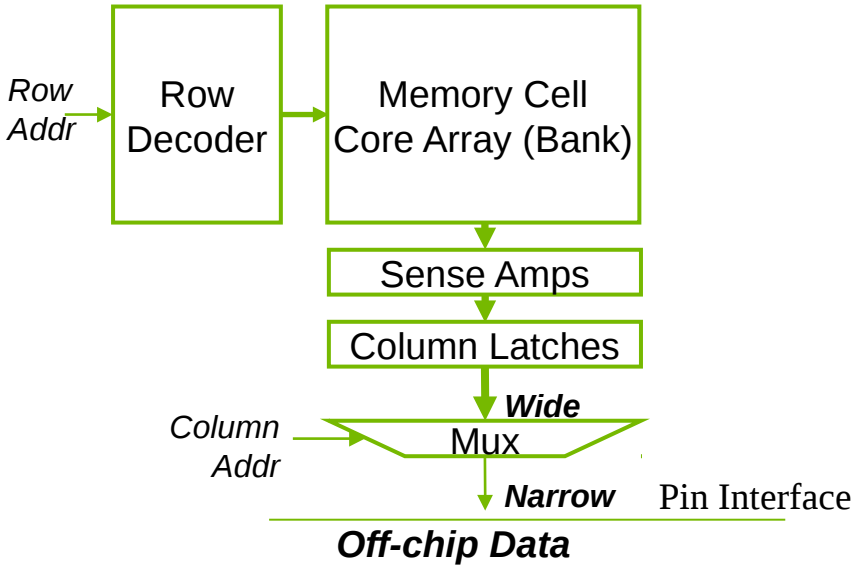
DRAM (Dynamic Random Access)

Single bit is stored as a charge in a capacitor, one transistor is used for access

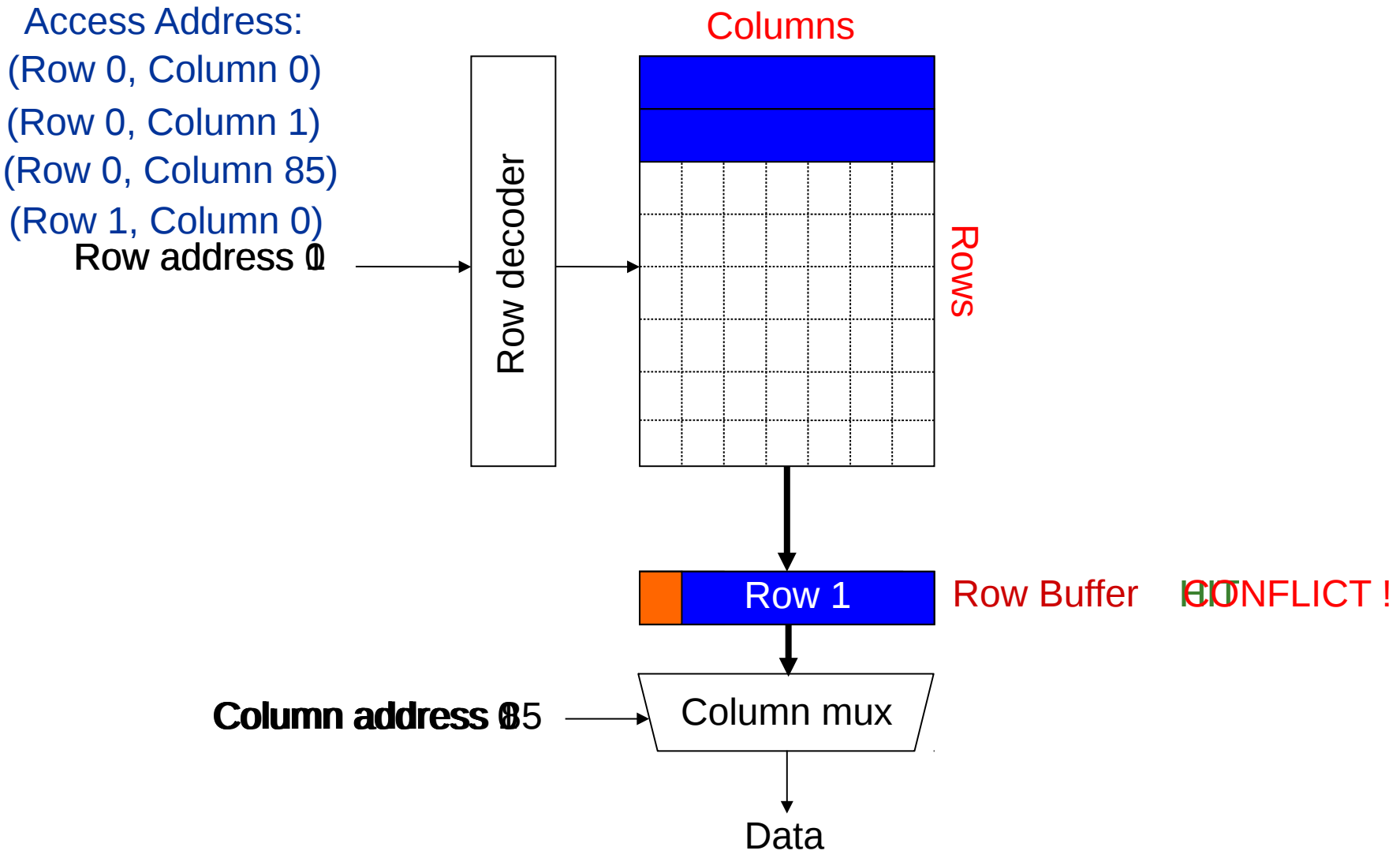
Whether the capacitor is charged or discharged indicates storage of 1 or 0



DRAM Bank Organization



DRAM Bank Operation



8

DRAM Burst

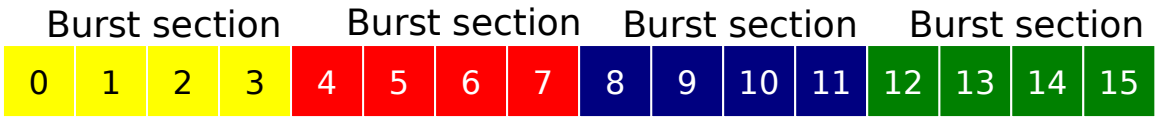
Each time a DRAM location is accessed, a range of consecutive locations that includes the requested location are actually accessed

The data from all these consecutive locations can be transferred at very high-speed to the processor

These consecutive locations accessed and delivered are referred to as DRAM *bursts*

If an application makes focused use of data from these bursts, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed

DRAM Burst-A System View



Each address space is partitioned into burst sections

- Whenever a location is accessed, all other locations in the same section are also delivered to the processor

Basic example: a 16-byte address space, 4-byte burst sections

- In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Memory Coalescing

Recognizing the burst organization of modern DRAMs, current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by organizing memory accesses of threads into favorable patterns

Takes advantage of the fact that threads in a warp execute the same instruction at any given point in time

Memory Coalescing

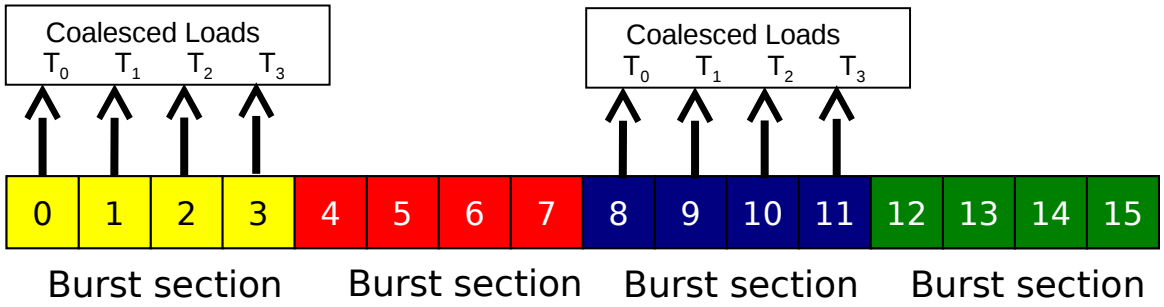
When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations

The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations

The hardware combines, or *coalesces*, all these accesses into a consolidated access to consecutive DRAM locations

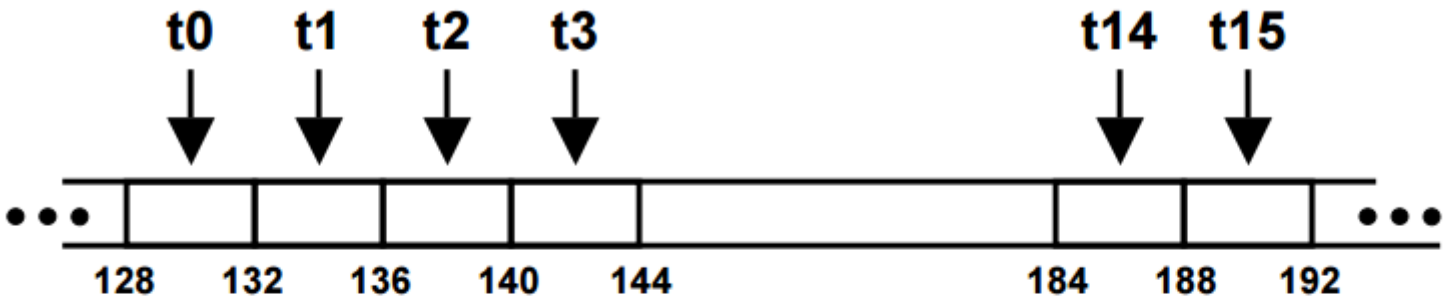
Such coalesced access allows the DRAMs to deliver data as a burst

Memory Coalescing

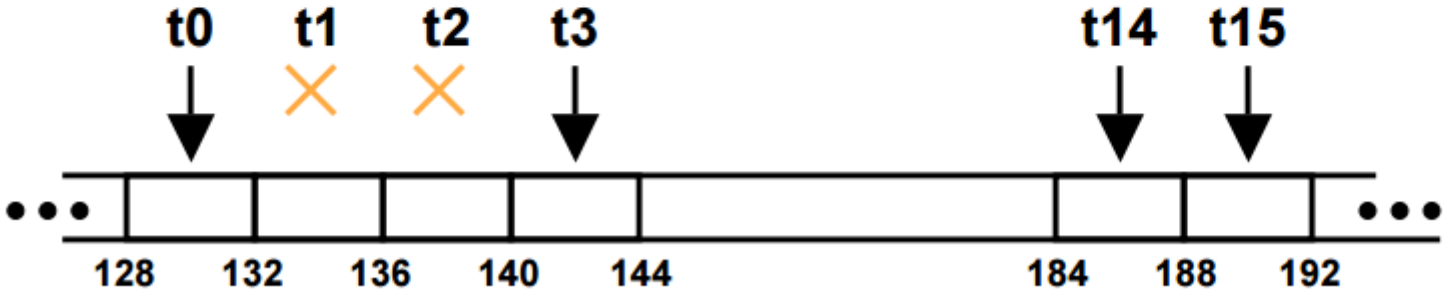


When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced

Coalesced Accesses

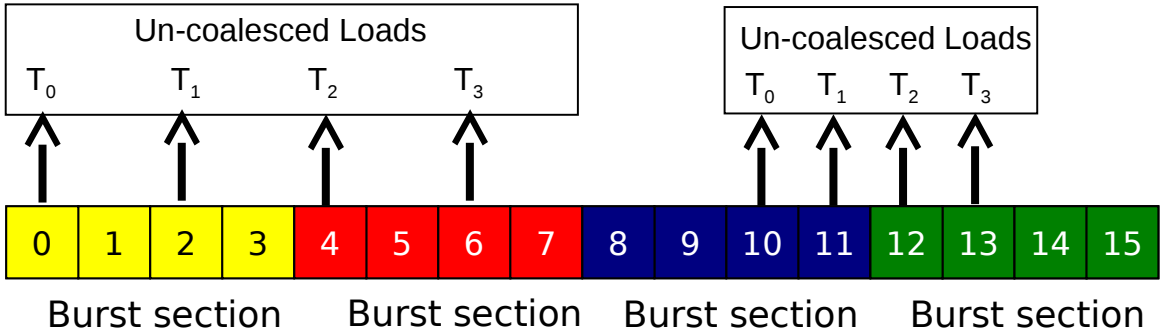


All threads participate



Some Threads Do Not Participate

Uncoalesced Accesses

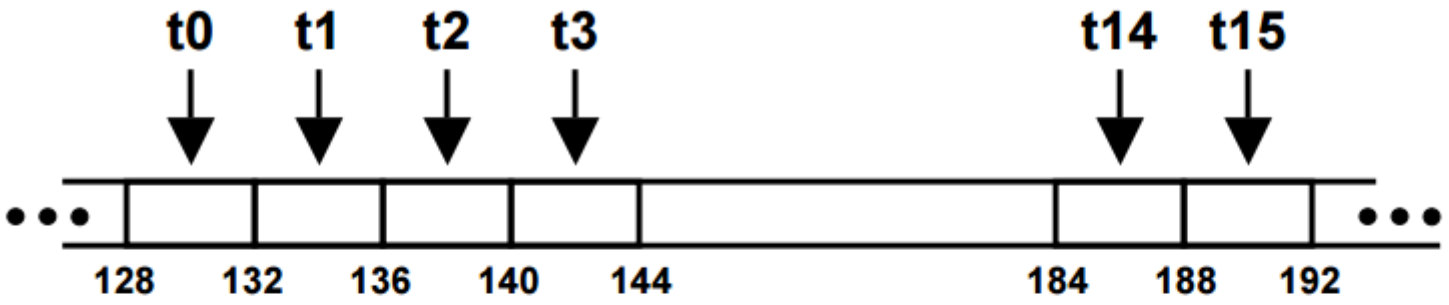


When the accessed locations spread across burst section boundaries:

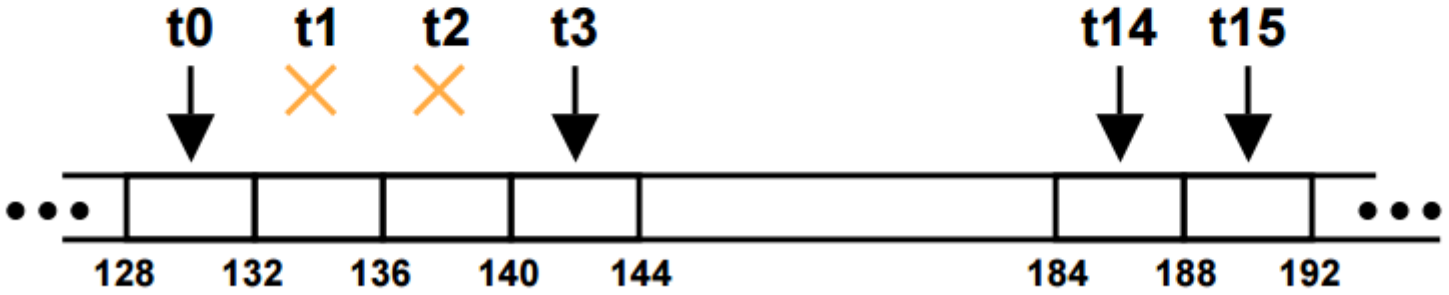
- Coalescing fails
- Multiple DRAM requests are made
- The access is not fully coalesced

Some of the bytes accessed and transferred are not used by the threads

Coalesced Access

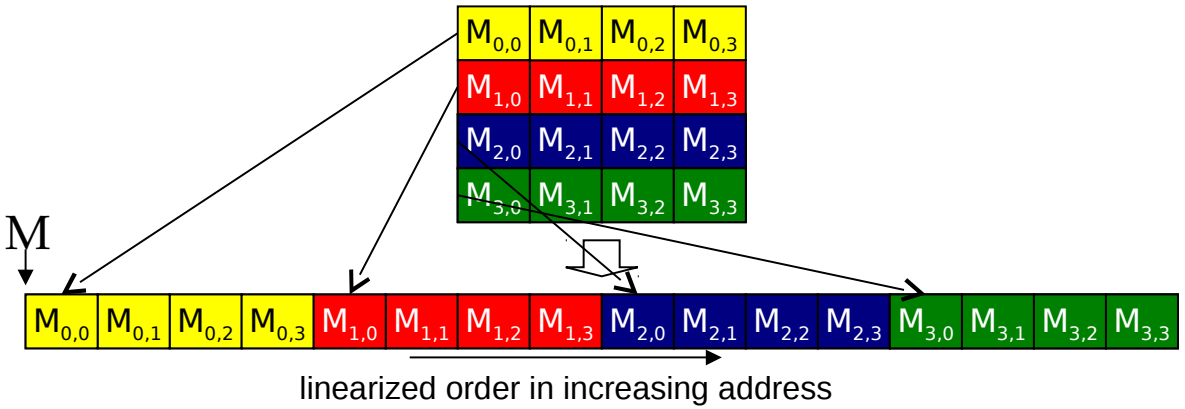


All threads participate

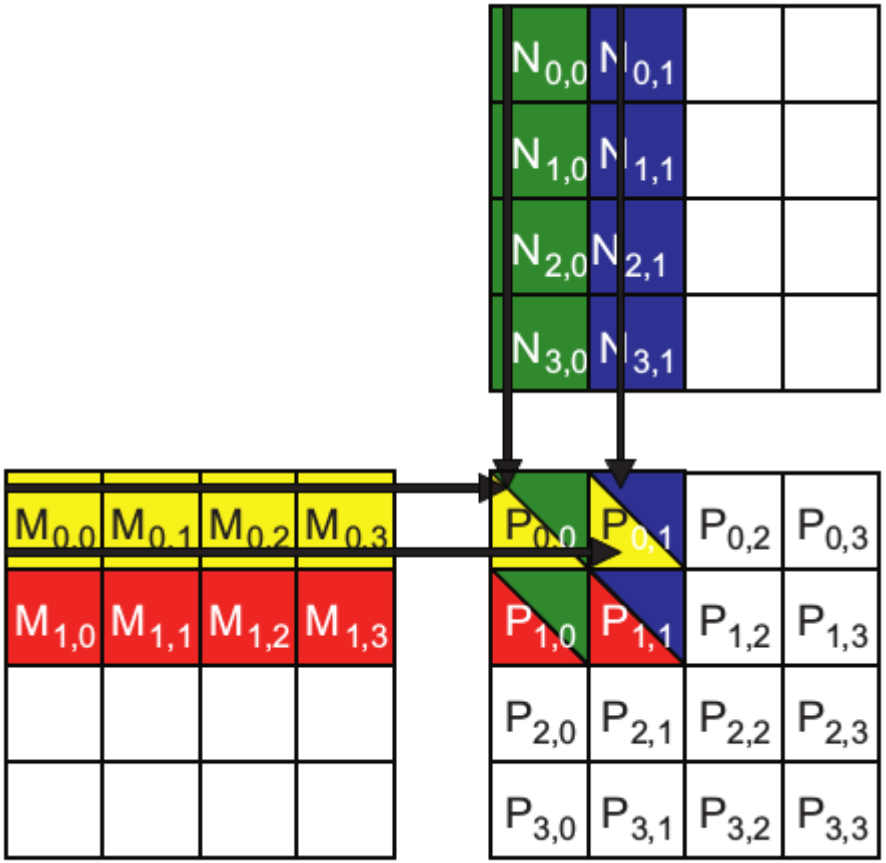


Some Threads Do Not Participate

A 2D C Array in Linear Memory Space



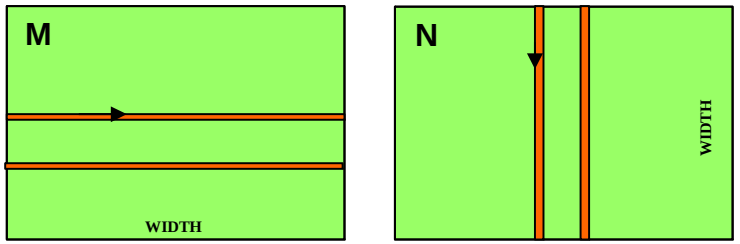
Matrix Multiplication



A Basic Matrix Multiplication Kernel

```
__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 i = 0; i < Width; ++k) {
      Pvalue += M[Row*Width+i]*N[i*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

Matrix Multiplication in C 2D arrays

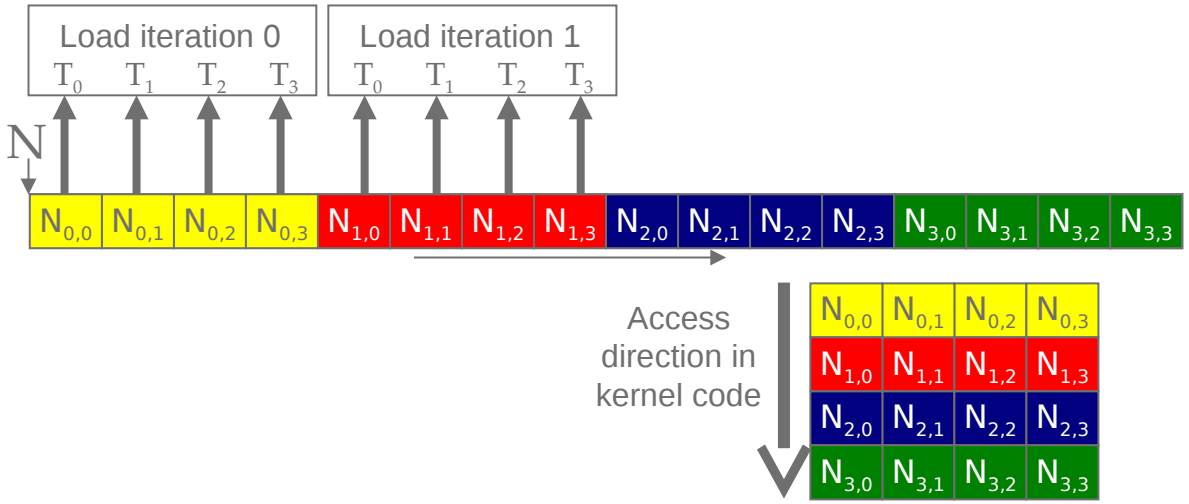


`M[Row*width+i]` `N[i*Width+Col]`

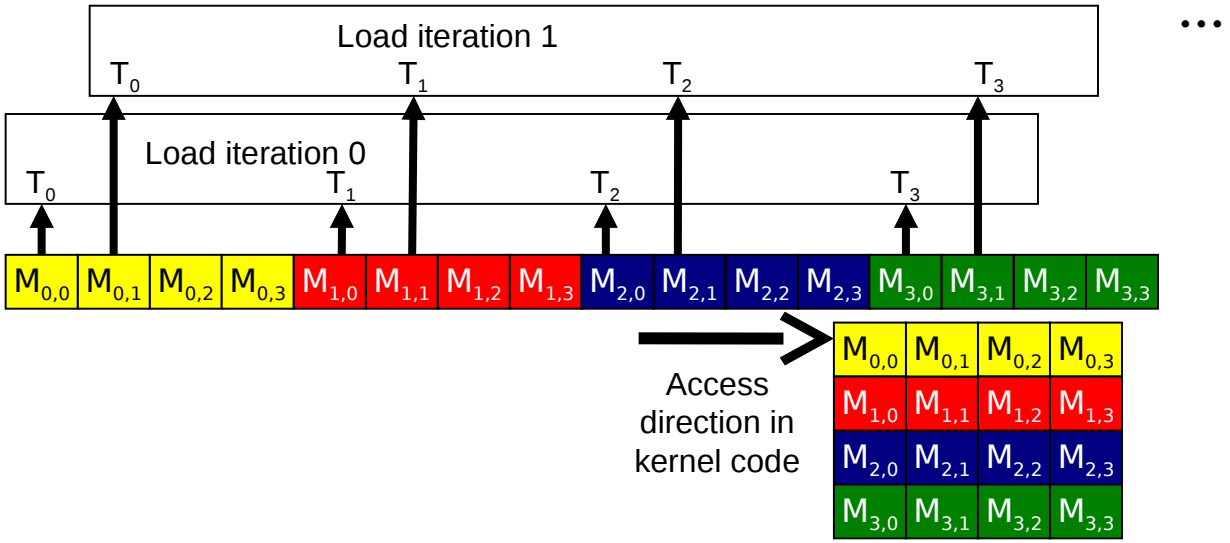
i is the loop counter in the inner product loop of the kernel code

Col = blockIdx.x*blockDim.x + threadIdx.x
Row=blockIdx.y*blockDim.y+threadIdx.y

N Accesses are Coalesced



M Accesses are not Coalesced

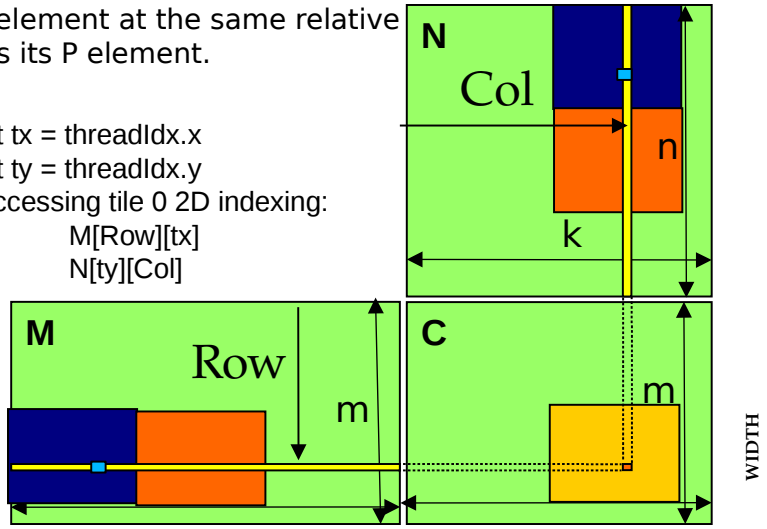


Using Shared Memory for Coalescing

If an algorithm requires a kernel code to iterate through data along the row direction, one can use the shared memory to enable memory coalescing-tiles are loaded in a coalesced pattern

Have each thread load an M element and an N element at the same relative position as its P element.

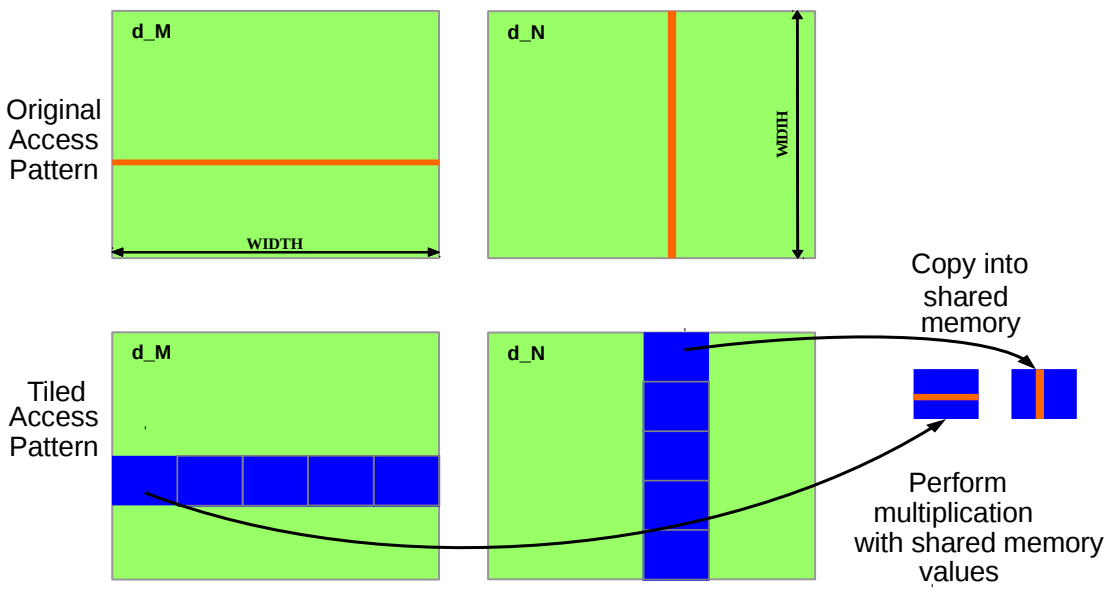
int tx = threadIdx.x
int ty = threadIdx.y
Accessing tile 0 2D indexing:
M[Row][tx]
N[ty][Col]



Tiled Matrix Multiplication Kernel

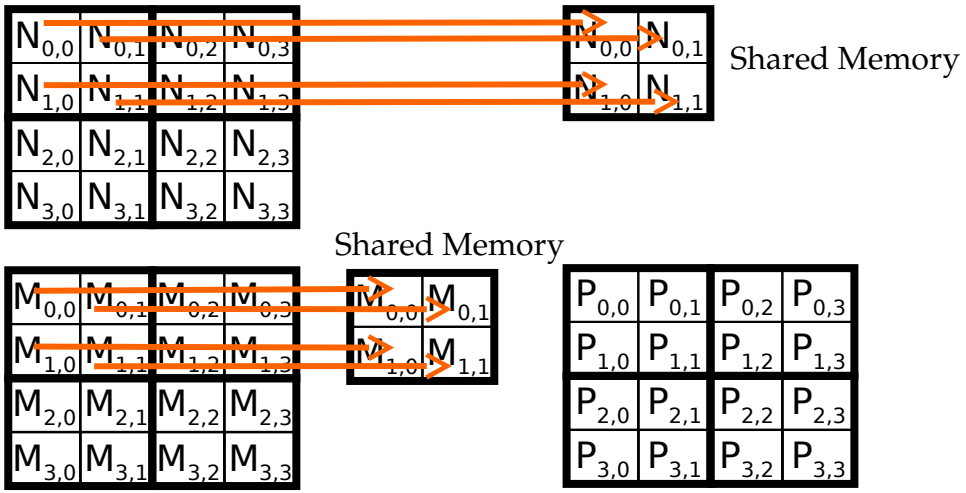
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
  __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
  __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

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

  int Row = by * blockDim.y + ty;
  int Col = bx * blockDim.x + tx;
  float Pvalue = 0;

 // Loop over the M and N tiles required to compute the P element
 for (int p = 0; p < Width/TILE_WIDTH; ++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;
}
```

Corner Turning



Phase 0 Load for Block (0,0)

Each thread loads one M element and one N element in tiled code



Turned a vertical access pattern into a horizontal access pattern

Tiled Algorithm

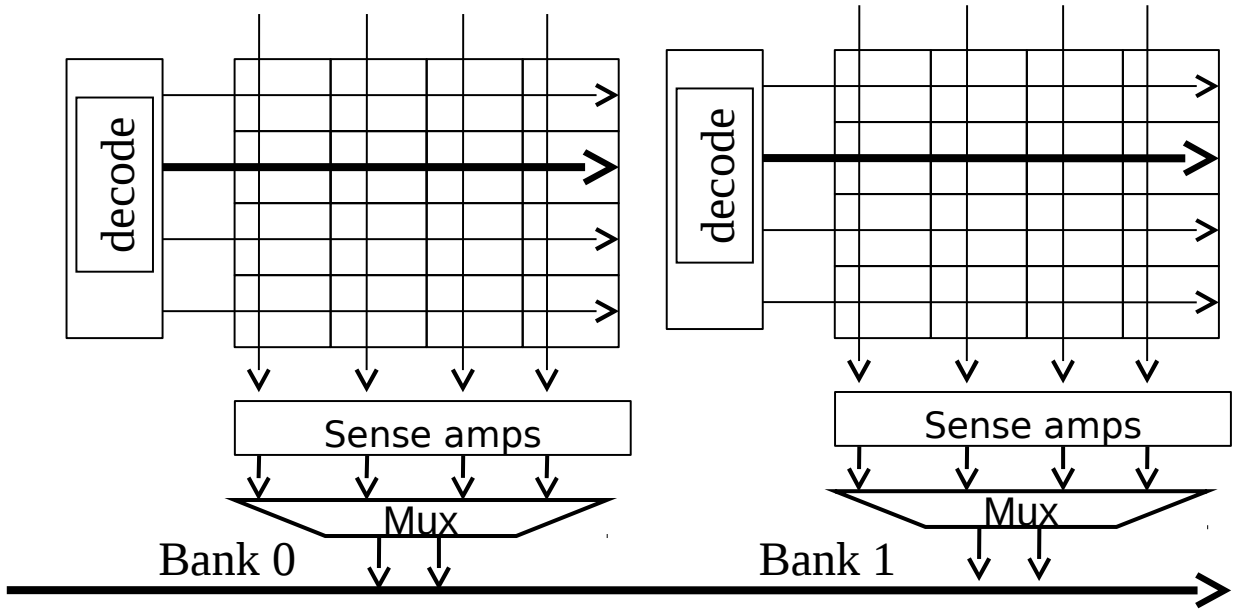
Loads to both the M and N elements are coalesced

1) number of memory loads is reduced due to the reuse of data in the shared memory

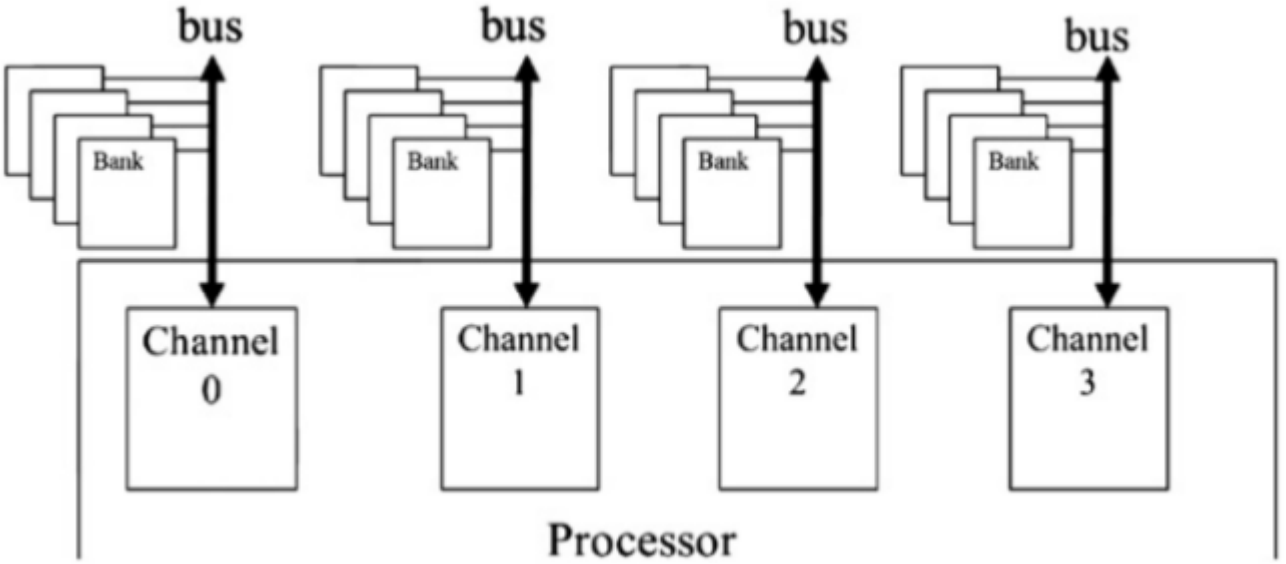
2) the remaining memory loads are coalesced so the DRAM bandwidth utilization is further improved

Tiled kernel can run 30x faster than the simple kernel

Multiple DRAM Banks



Channels and Banks in DRAM



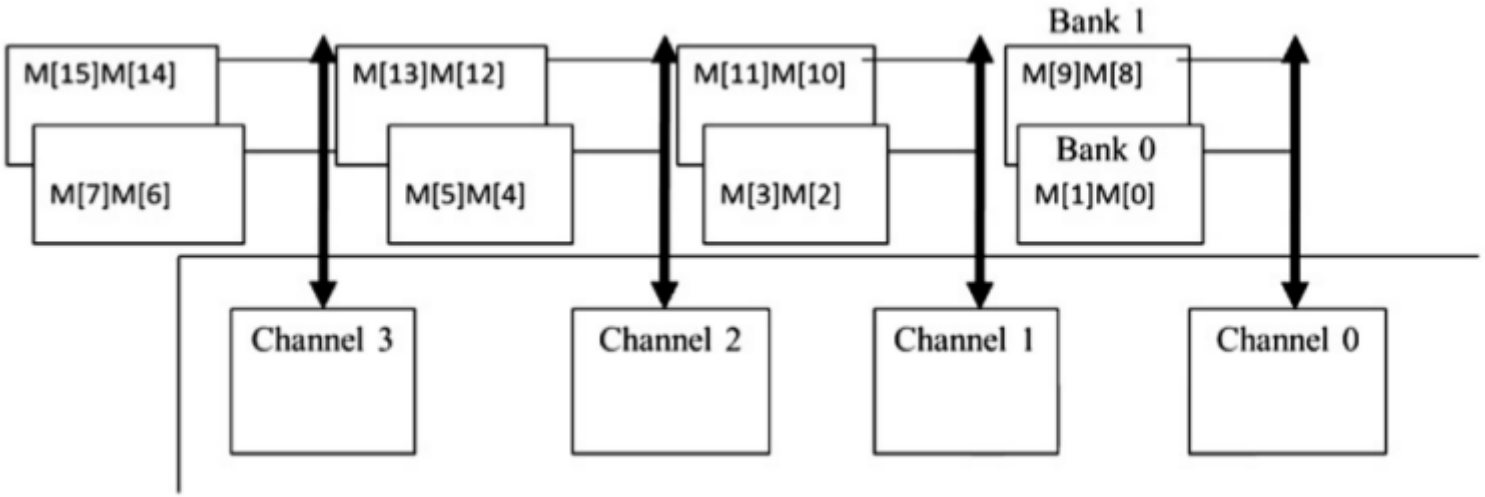
DRAM Bursting with Banking

Single-Bank burst timing, dead time on interface

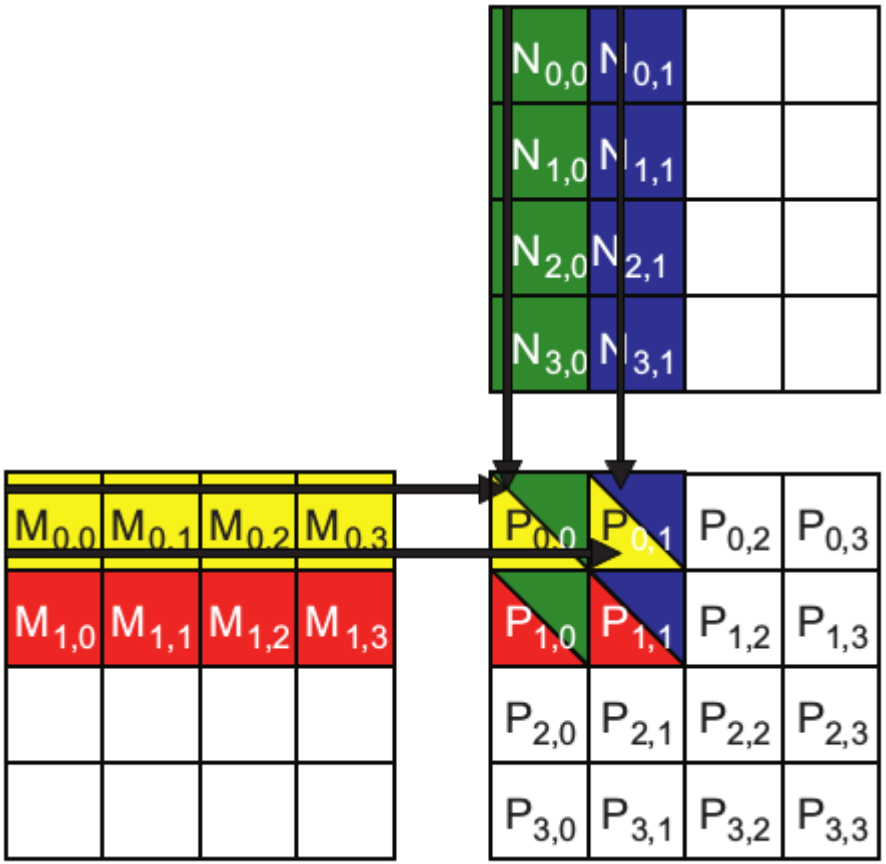
Multi-Bank burst timing, reduced dead time

Array Elements into Channels and Banks

Interleaved data distribution



Matrix Multiplication



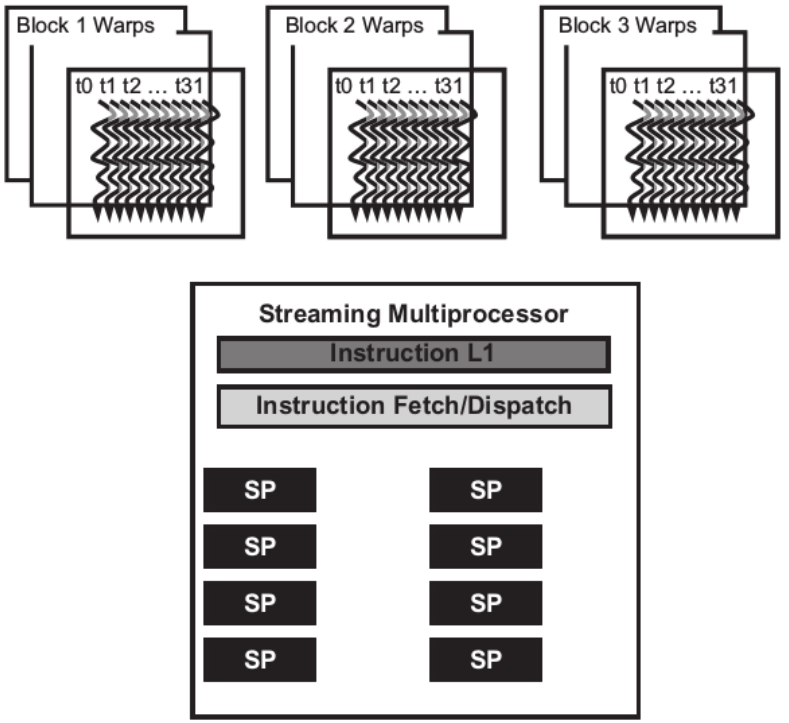
M Elements

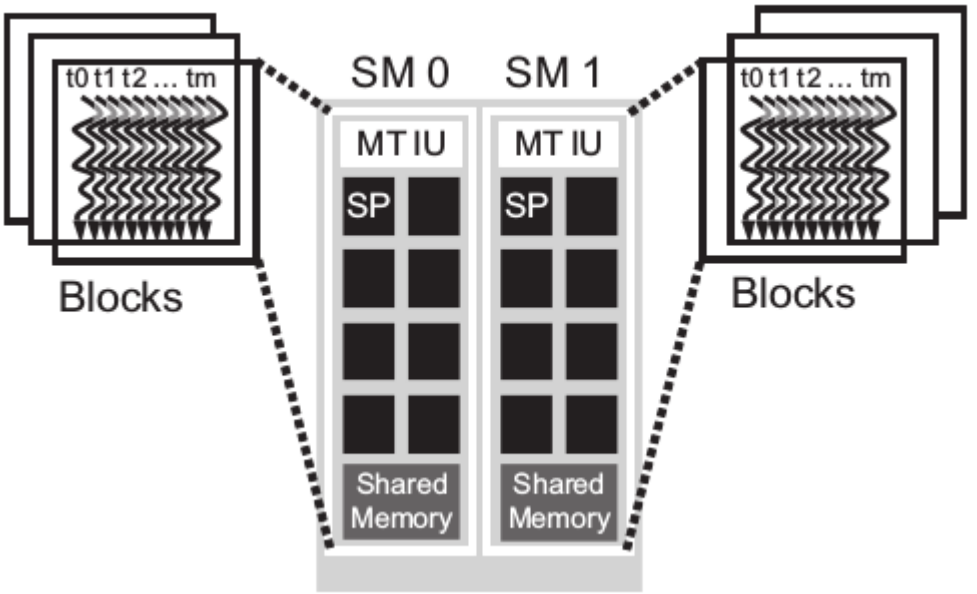
Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

Warps

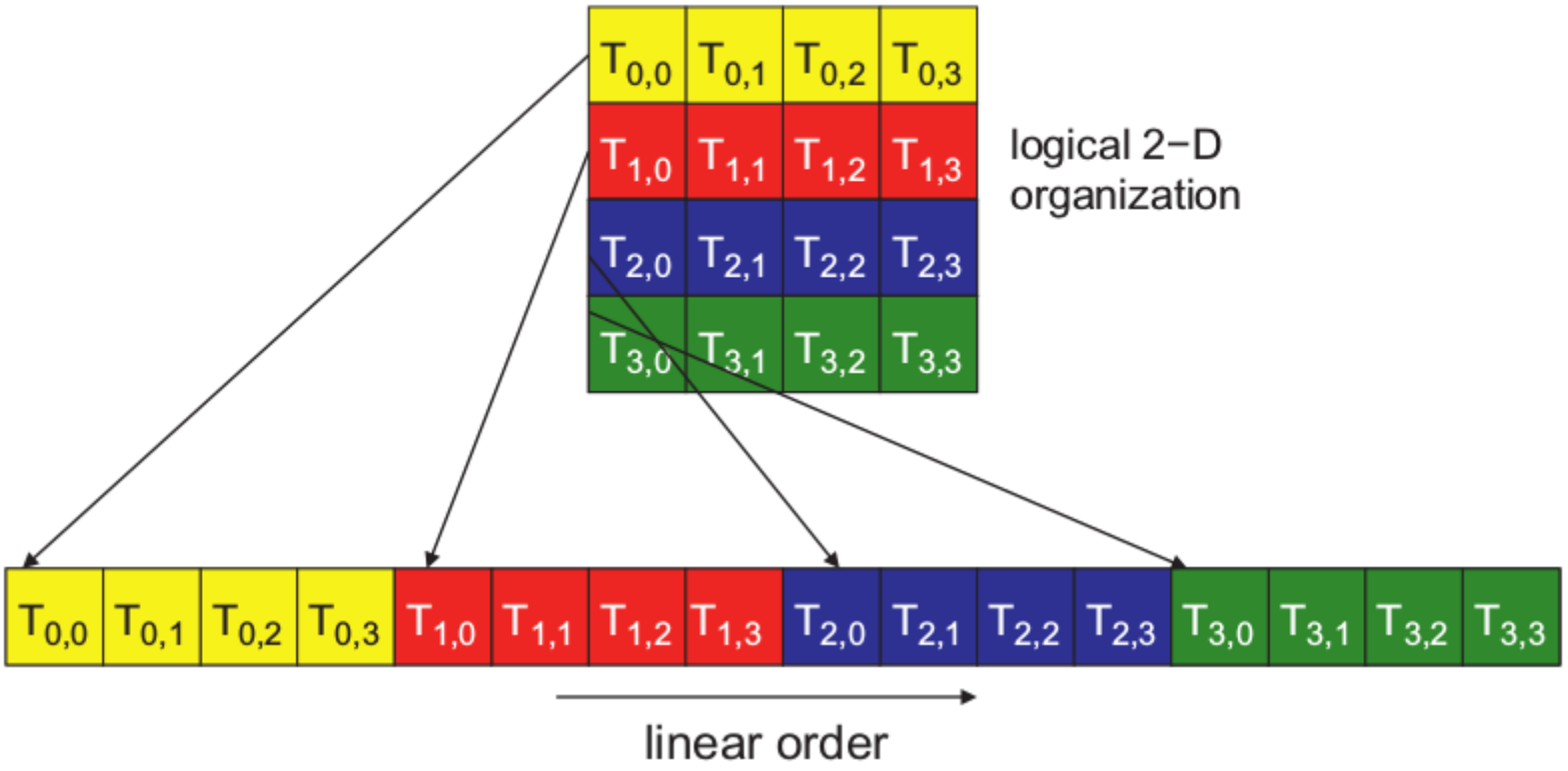
Each block is divided into 32-thread warps

Warps are scheduling units in SM



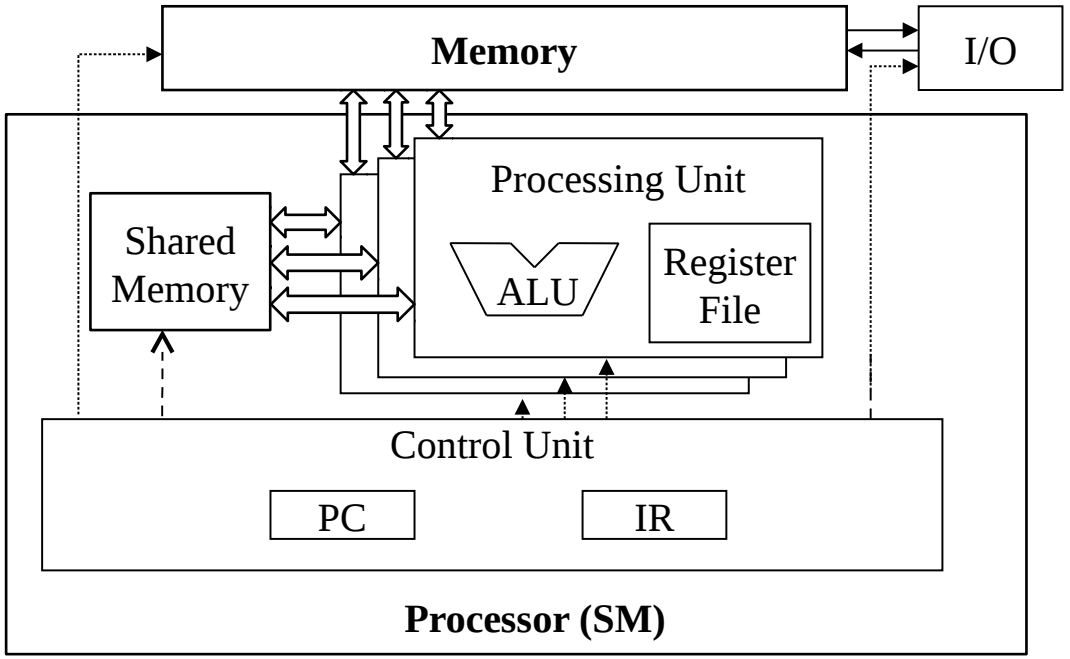


Placing 2D Threads into Linear Order



SMs are SIMD Processors

Control unit for instruction fetch, decode, and control is shared among multiple processing units



SIMD Among Threads in a Warp

All threads in a warp must execute the same instruction at any point in time

This works efficiently if all threads follow the same control flow path

- All if-then-else statements make the same decision
- All loops iterate the same number of times

Control Divergence

Control divergence occurs when threads in a warp take different control flow paths by making different control decisions

- Some take the then-path and others take the else-path of an if-statement
- Some threads take different number of loop iterations than others

The execution of threads taking different paths are serialized in current GPUs

- The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- During the execution of each path, all threads taking that path will be executed in parallel
- The number of different paths can be large when considering nested control flow statements

Control Divergence Examples

Divergence can arise when branch or loop condition is a function of thread indices

Example kernel statement with divergence:

if (threadIdx.x > 2) { }

This creates two different control paths for threads in a block

Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp

Example without divergence:

If (blockIdx.x > 2) { }

Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path

Example: Vector Addition Kernel

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
   if(i<n) C[i] = A[i] + B[i];
}
```

Analysis for 1,000 Elements

Assume that block size is 256 threads

- 8 warps in each block

All threads in Blocks 0, 1, and 2 are within valid range

- i values from 0 to 767
- There are 24 warps in these three blocks, none will have control divergence

Most warps in Block 3 will not control divergence

- Threads in the warps 0-6 are all within valid range, thus no control divergence

One warp in Block 3 will have control divergence

- Threads with i values 992-999 will all be within valid range
- Threads with i values of 1000-1023 will be outside valid range

Effect of serialization on control divergence will be small

- 1 out of 32 warps has control divergence
- The impact on performance will likely be less than 3%

Example: Matrix Multiplication

Boundary condition checks are vital for complete functionality and robustness of parallel code

The tiled matrix multiplication kernel has many boundary condition checks

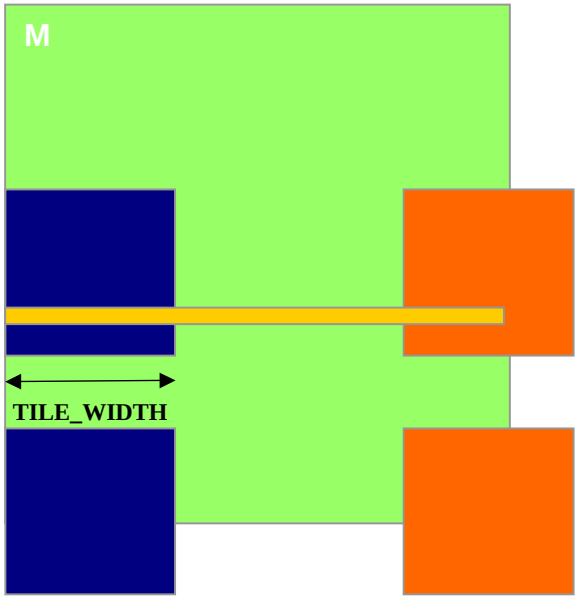
The concern is that these checks may cause significant performance degradation

```
if(Row < Width && t * 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;
}
```

Example: Matrix Multiplication

Type 1. Blocks whose tiles are all within valid range until the last phase

Type 2. Blocks whose tiles are partially outside the valid range all the way



Assume 16x16 tiles and thread blocks

Each thread block has 8 warps (256/32)

Assume square matrices of 100x100

Each thread will go through 7 phases (ceiling of 100/16)

There are 49 thread blocks (7 in each dimension)

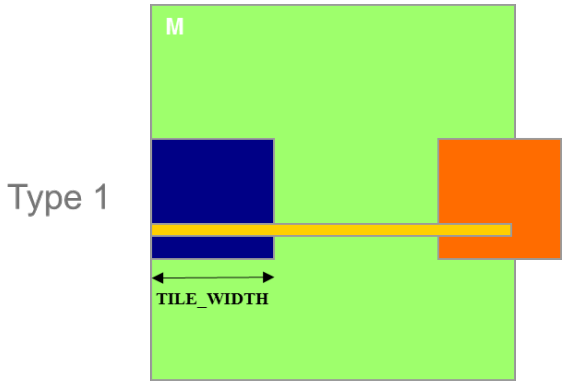
Type 1

There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps

They all have 7 phases, so there are 2,352 (336*7) warp-phases

The warps have control divergence only in their last phase

336 warp-phases have control divergence



Type 2

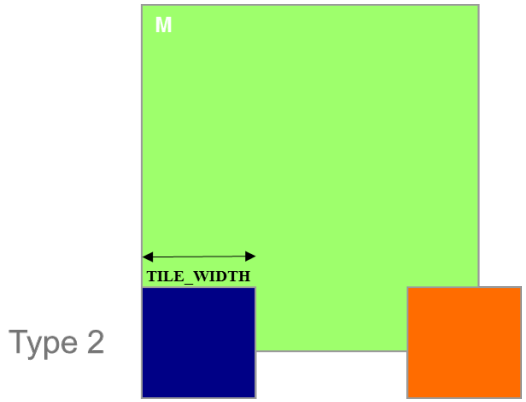
There are 7 Type 2 block assigned to load the bottom tiles, with a total of 56 (8*7) warps

They all have 7 phases, so there are 392 (56*7) warp-phases

The first 2 warps in each Type 2 block will stay within the valid range until the last phase

The 6 remaining warps stay outside the valid range

So, only 14 (2*7) warp-phases have control divergence



Type 2

Performance Impact of Control Divergence

Type 1 Blocks:

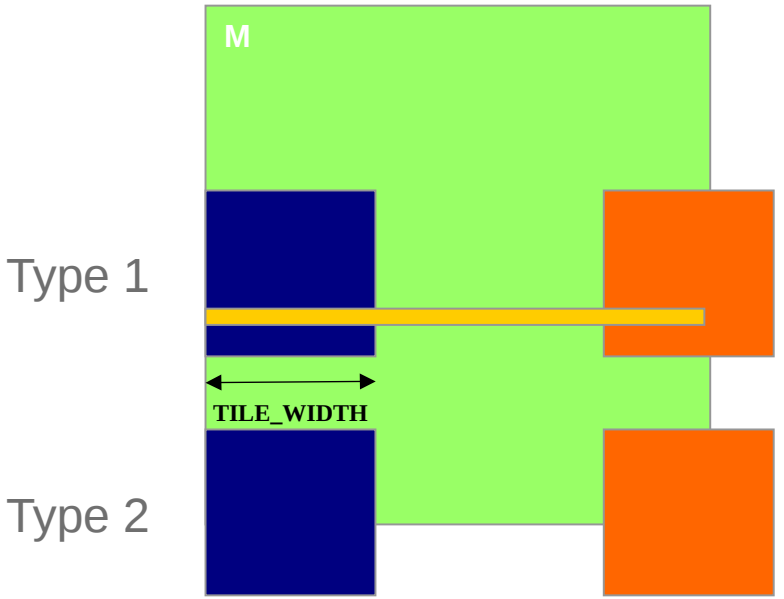
336 out of 2,352 warp-phases have control divergence

Type 2 Blocks:

14 out of 392 warp-phases have control divergence

The performance impact is expected to be less than 12%

(350/2,944)



Additional Comments

The estimated performance impact is data dependent.

- For larger matrices, the impact will be significantly smaller

In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant

- One should not hesitate to use boundary checks to ensure full functionality

The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence

More control divergence for some algorithm patterns (such as parallel reduction)