Parallel Computing 2020

中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

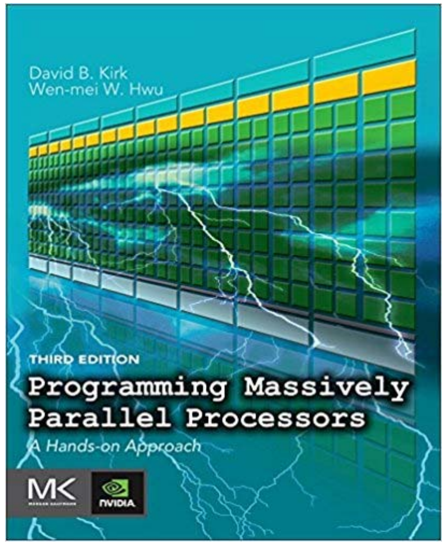
(5) Performance Considerations

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- Programming Massively Parallel Processors,
  - A Hands-on Approach
  - Third Edition

  - Chapter 5



Resources and Performance

- Performance of a parallel program can vary greatly depending on the resource constraints of the computing hardware.
- In different applications, different constraints may dominate and become *bottlenecks*.

- It is important to find:
  - the major types of resource constraints in a CUDA device;
  - how they can affect the kernel execution performance.

Content

- Global Memory Bandwidth
- More on Memory Parallelism
- Warps and SIMD Hardware
- Dynamic partitioning of resources

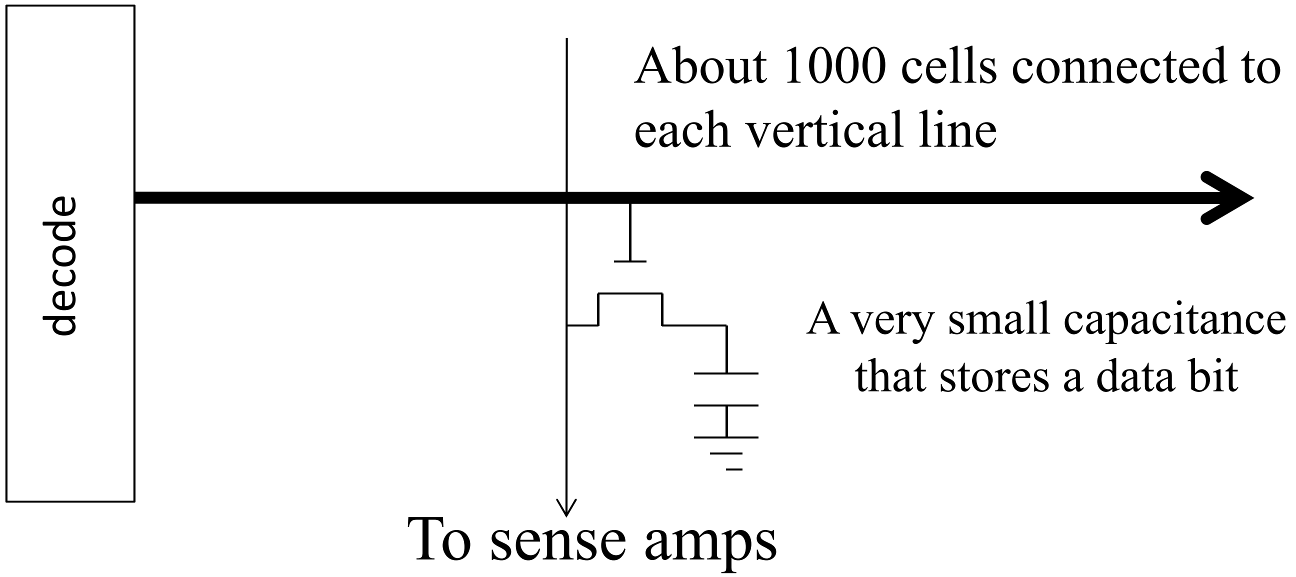
Global Memory Bandwidth

- Ideal

- Reality

DRAM core arrays are slow

- Reading from a cell in the core array is a very slow process


decode
About 1000 cells connected to each vertical line
A very small capacitance that stores a data bit
To sense amps

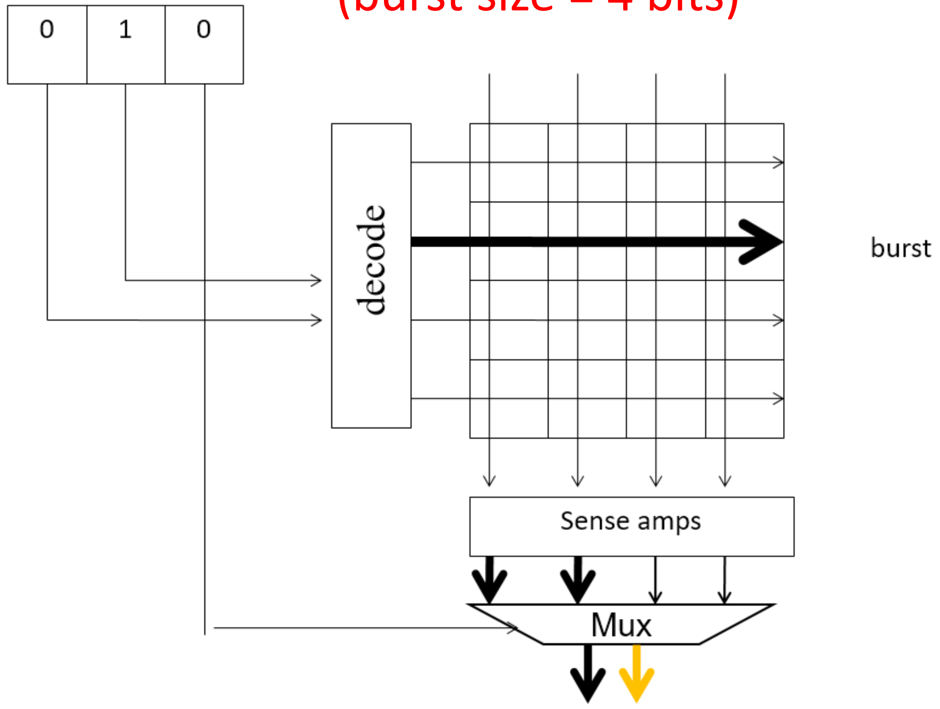
- 10s of nanoseconds v.s. sub-nanosecond clock cycle
- Like determine the flavor of cup of coffee by smelling far away

DRAM Burst

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

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

(burst size = 4 bits)

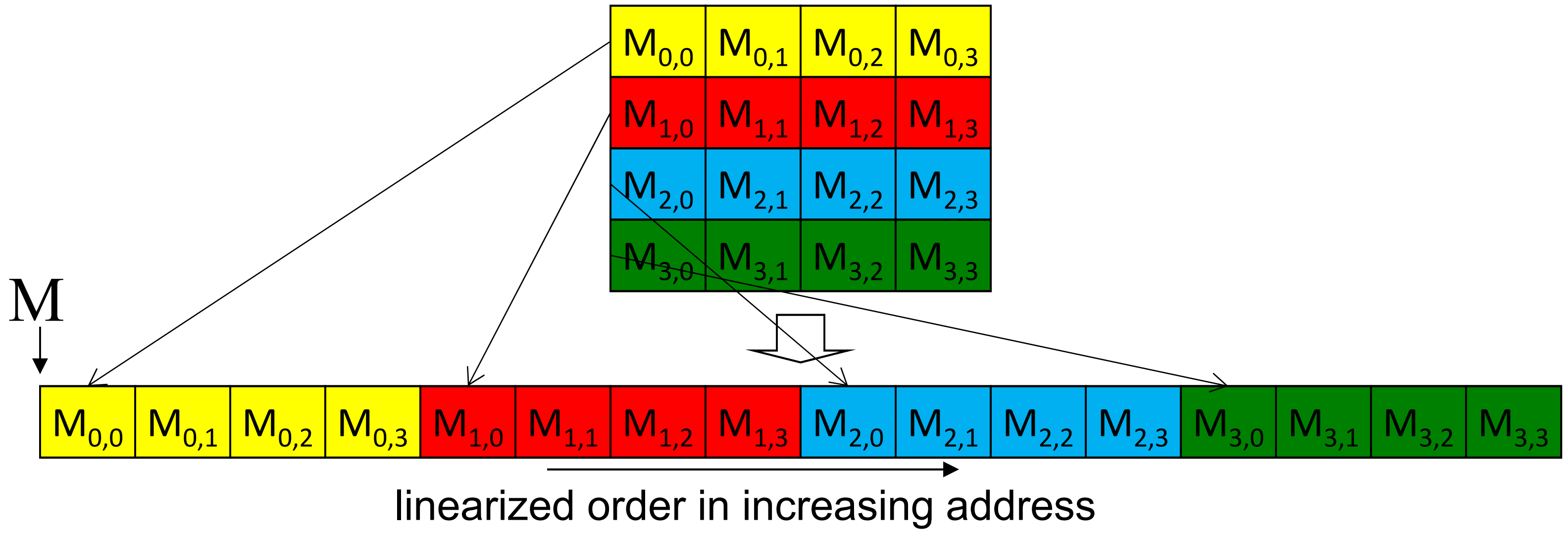


Coalesce

- Threads in a **warp** execute the same instruct at any given time.
- Modern DRAM systems are designed to be always accessed in **burst** mode.
- The favorable access pattern is achieved When all threads in a warp access consecutive global memory locations.
- In this case, the hardware combines, or **coalesces**, all these accesses into a consolidated access to consecutive DRAM locations (burst).

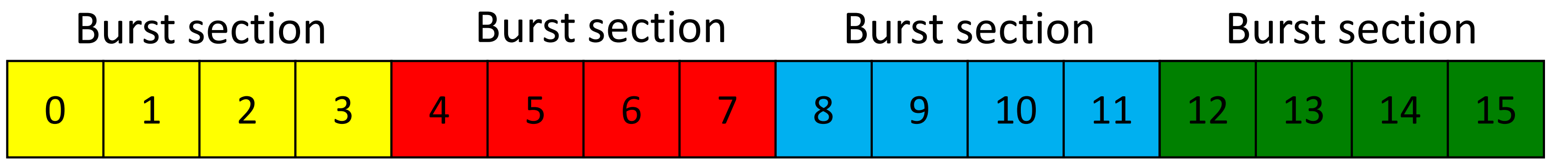
A 2D C Array in Linear Memory Space

- How the memory addresses are formed in accessing C multidimensional array elements?



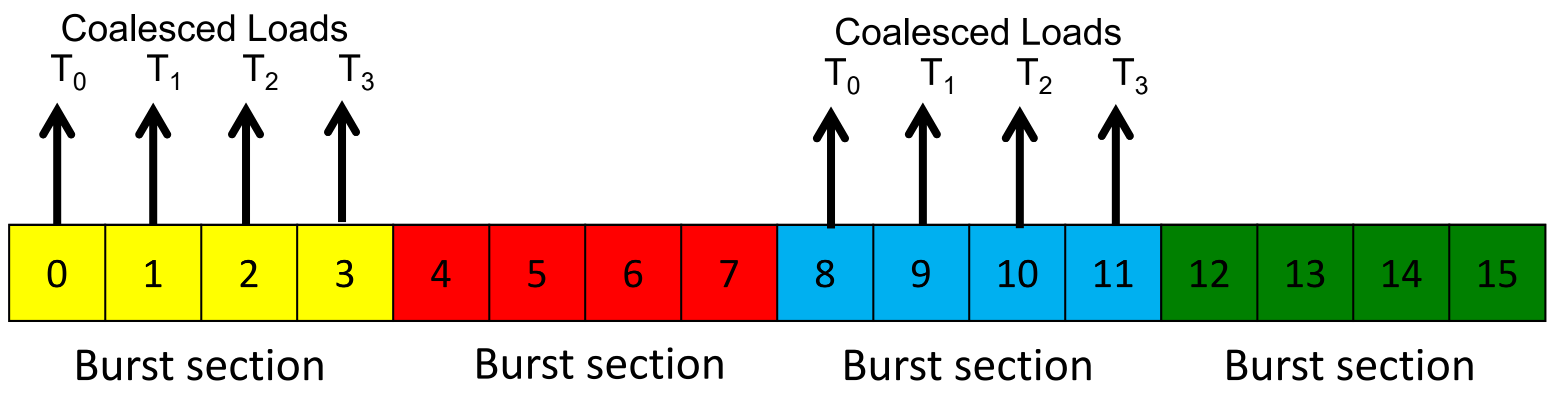
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
linearized order in increasing address

DRAM Burst – A System View



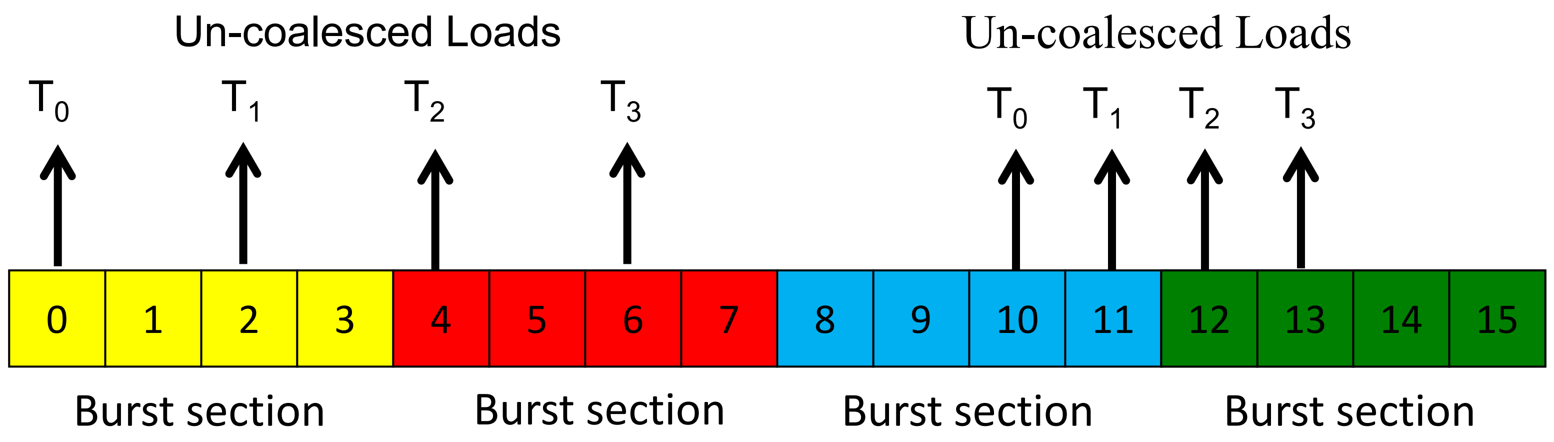
- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

Memory Coalescing



– 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.

Un-coalesced Accesses



- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made

A Simple Matrix Multiplication Kernel (review)

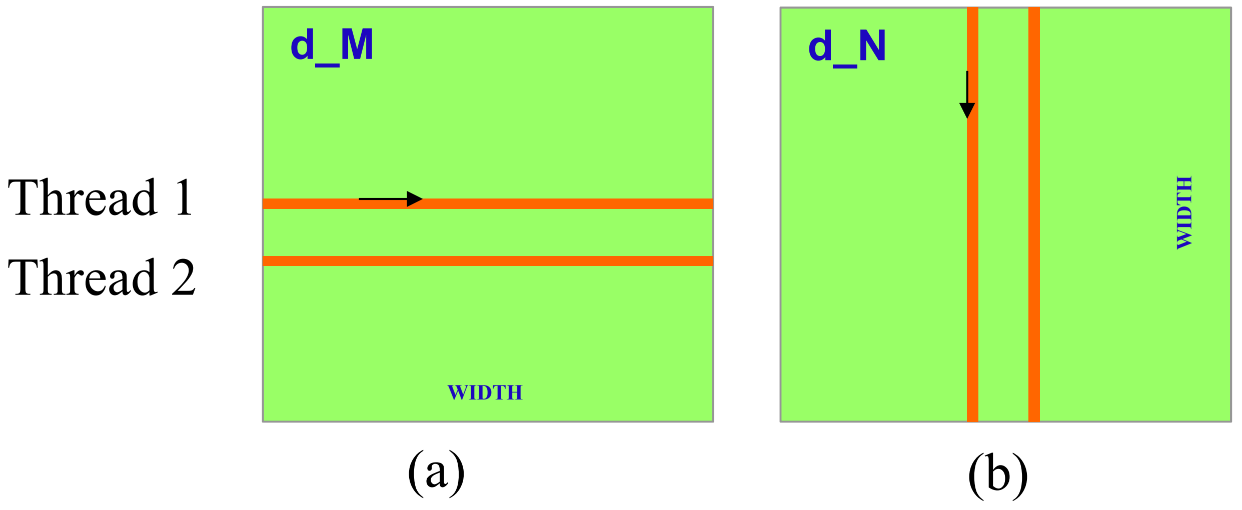
```
__global__
void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 // Calculate the row index of the P element and M
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 // Calculate the column index of P and N
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 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

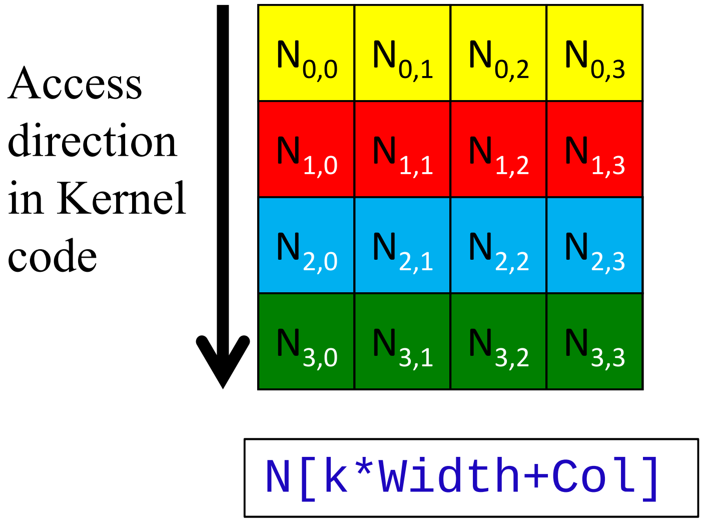
For M, threads in a warp read adjacent rows;
For N, threads in a warp read adjacent columns;



```
M[Row*Width+k]          N[k*Width+Col]
```

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

N accesses are coalesced.


Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
N[k*Width+Col]

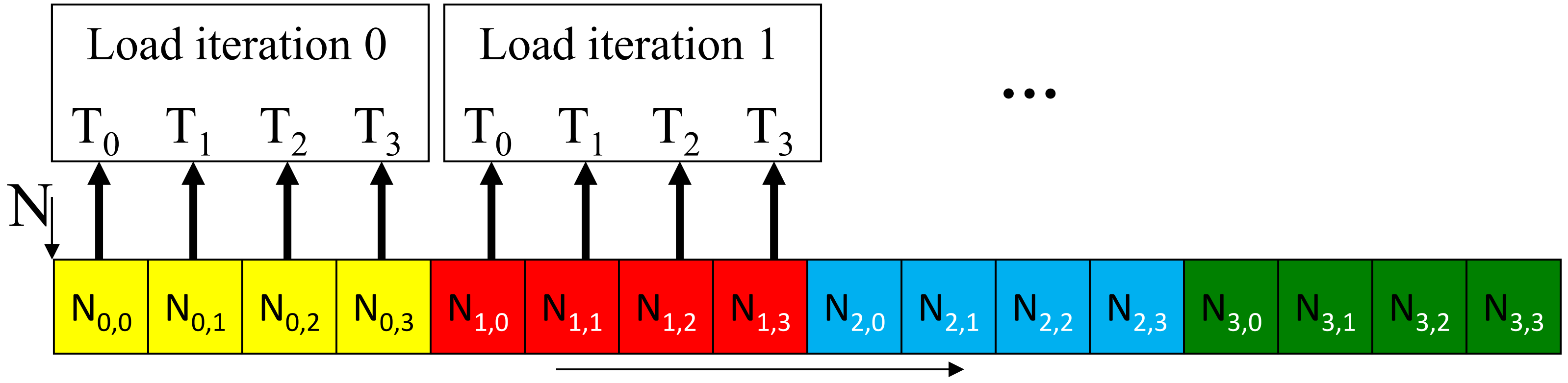
`k*Width:`

same across all threads within a given iteration of the k loop.

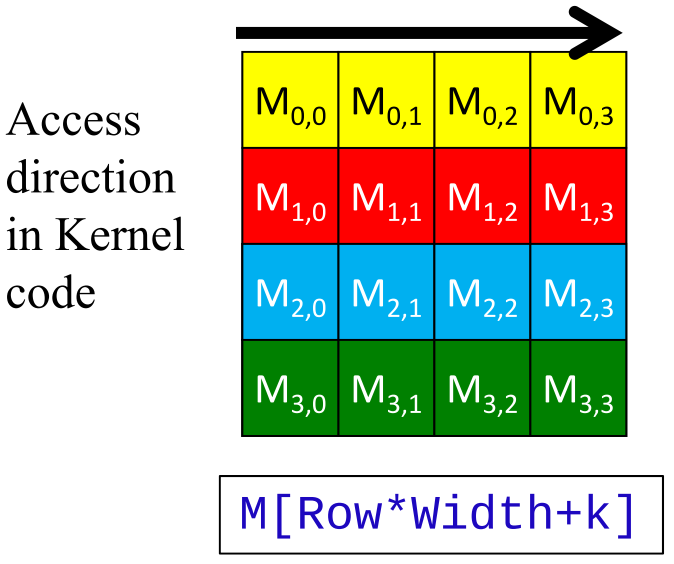
`Col:`

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

blockIndx.x and blockDim.x are same for all threads in the same block.


Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
…
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

M accesses are not coalesced.



Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
M[Row*Width+k]

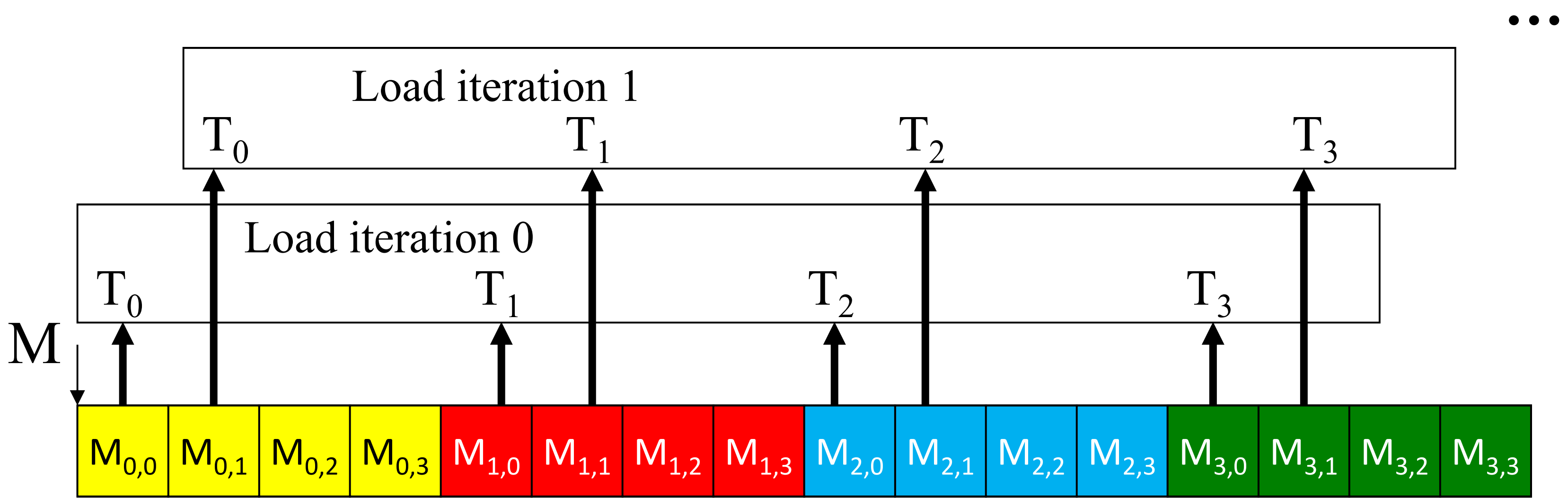
k & Width:

same across all threads within a given iteration of the k loop.

Row:

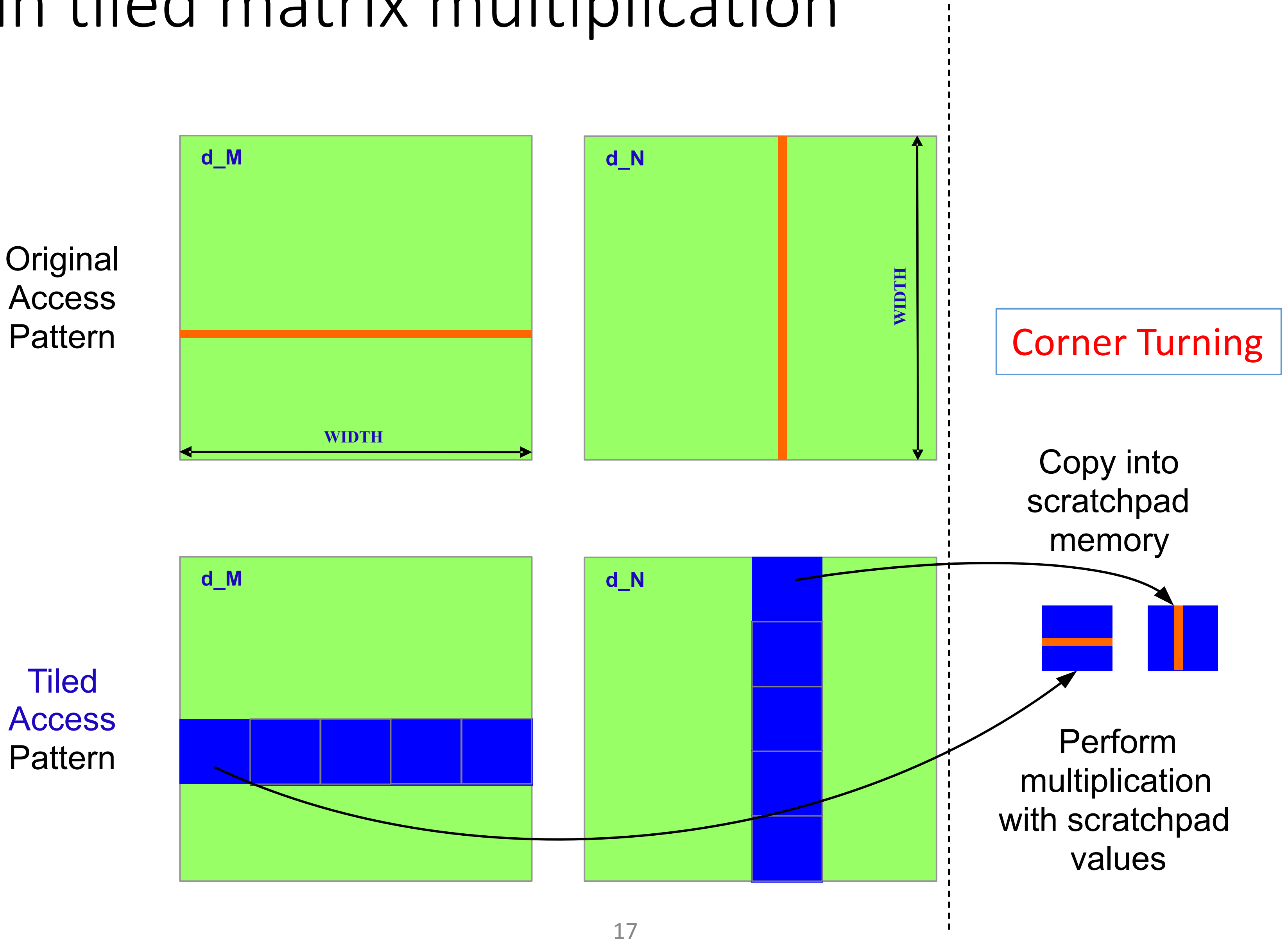
Row=blockIdx.y*blockDim.y+threadIdx.y

blockIndx.x and blockDim.x are same for all threads in the same block.



…
Load iteration 1
T_0 T_1 T_2 T_3
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

Use shared memory to enable coalescing in tiled matrix multiplication



Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width){
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    // Loop over the M and N tiles required to compute the P element
    for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
       // Collaborative loading of M and N tiles into shared memory
        Mds[? ][? ] = M[          ?                ];
        Nds[? ][? ] = N[          ?                ];
        __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __synchthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Load M elements

- The linearized index calculation is equivalent to the two-dimensional array access expression:
- M[Row][ph*TILE_WIDTH+tx].
    - column index used by the threads only differs in terms of threadIdx.
    - row index is determined by blockIdx.y and threadIdx.y
    - threads in the same thread block have **identical** blockIdx.y/threadIdx.y and **adjacent** threadIdx.x

- Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx]

Load N elements

- The linearized index calculation is equivalent to the two-dimensional array access expression:
- N[ph*TILE_WIDTH+ty][bx*TILE_WIDTH+tx].
    - the row index ph*TILE_WIDTH+ty has the same value for all threads with the **same** threadIdx.y value.
    - the column index calculation for each thread, Col=bx*TILE_WIDTH+tx, where bx*TILE_WIDTH is the same for all threads in the same block

- Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width){
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;
    // Identify the row and column of the P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;
    // Loop over the M and N tiles required to compute the P element
    for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {
       // Collaborative loading of M and N tiles into shared memory
        Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
        Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
        __syncthreads();
       for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[ty][k] * Nds[k][tx];
        __synchthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

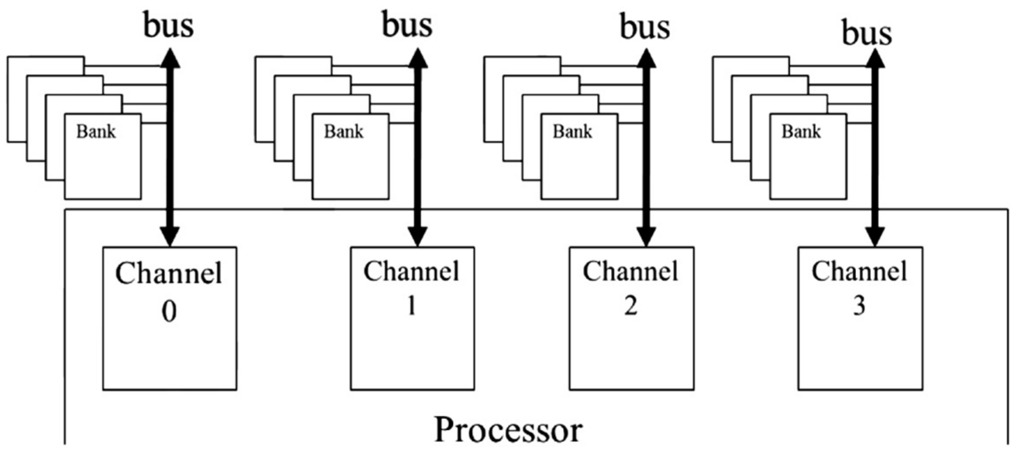
~30x

Content

- Global Memory Bandwidth
- More on Memory Parallelism
- Warps and SIMD Hardware
- Dynamic partitioning of resources

Bursting is not sufficient

- DRAM systems typically employ two more forms of parallel organization:
  - banks and channels.
    - At the highest level, a processor contains one or more channels.
    - Each channel is a memory controller with a bus that connects a set of DRAM banks to the processor.



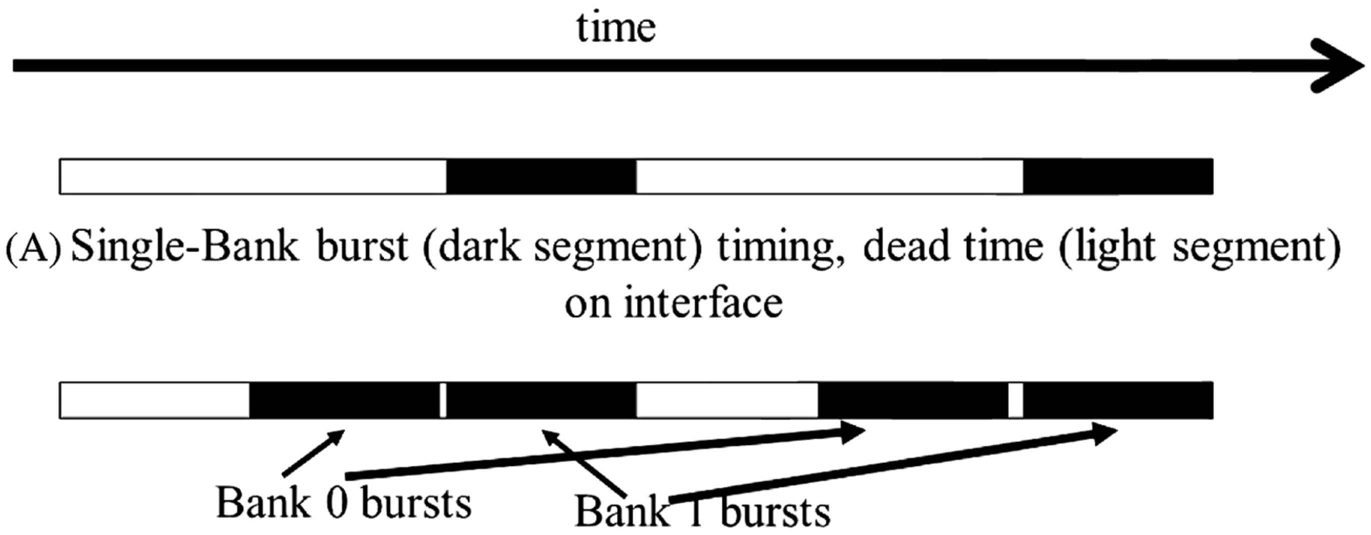
Number of Channels

- The data transfer *bandwidth* of a bus is defined by its width and clock frequency:
  - For example, a 64-bit *DDR* bus with a clock frequency of 1 GHz has a bandwidth of 8B*2*1 GHz = 16 GB/sec

- This seems to be a large number but is often too small for modern CPUs and GPUs.
  - CPU → 32GB/s → 2 channels
  - GPU→ 128GB/s → 8 channels

Number of Banks

- For each channel, the *number of banks* is determined by fully utilizing the bandwidth of the bus:

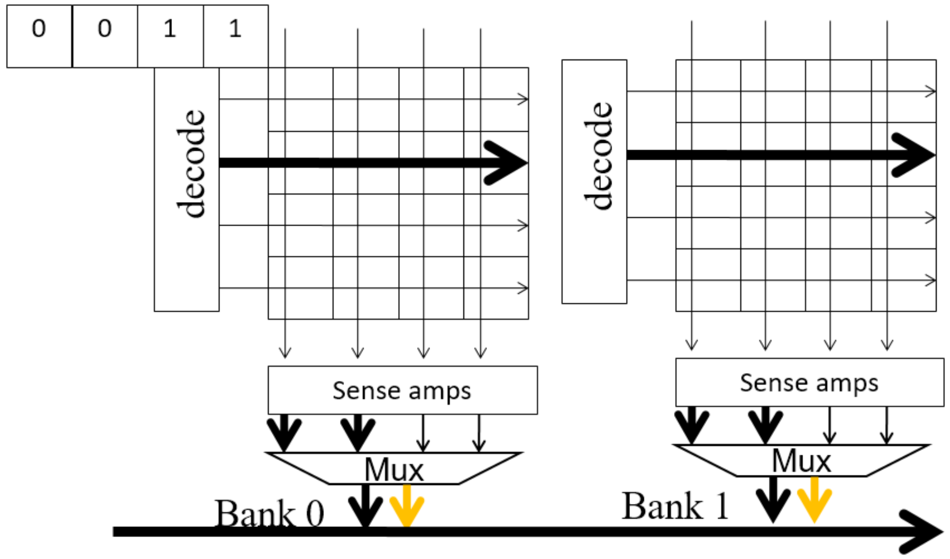


(A) Single-Bank burst (dark segment) timing, dead time (light segment) on interface

(B) Two-Bank burst timing, reduced dead time

Multiple DRAM Banks

In general, if the ratio of the cell array access latency and data transfer time is R, we need to have at least R+1 banks.



- Reasons to have more banks:
  - Avoid bank conflicts;
  - Manufacturability;

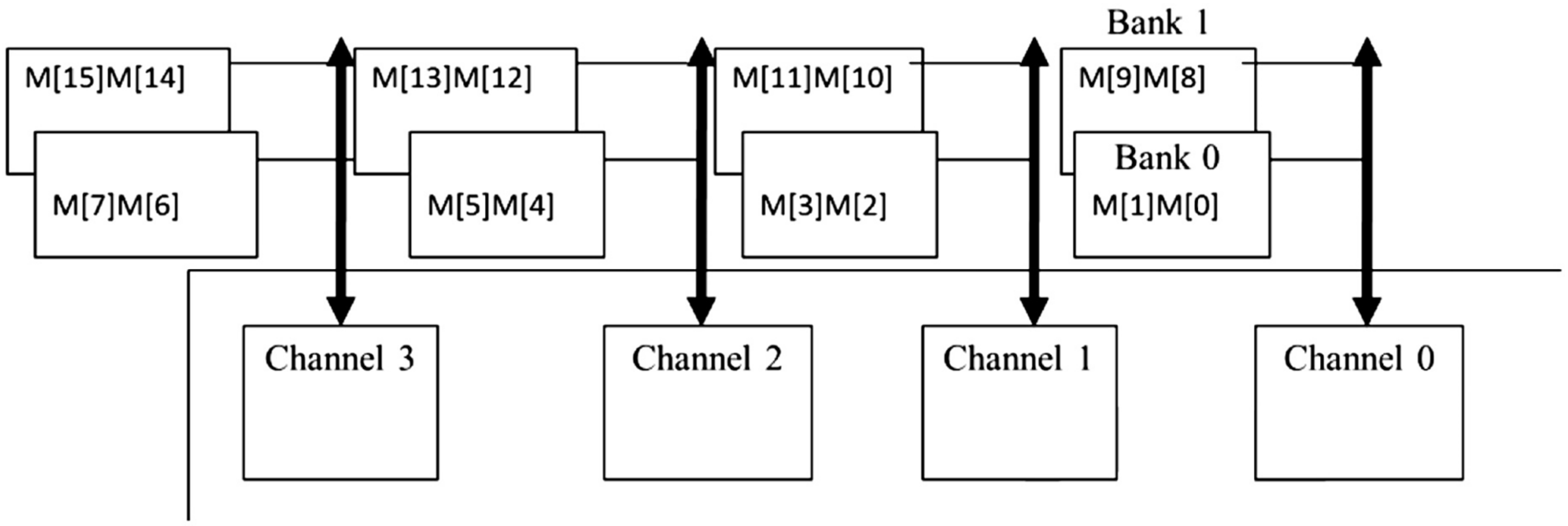
Data Distribution

- There is an important connection between the parallel execution of threads and the parallel organization of the DRAM system:

    1. sufficient number of threads making simultaneous memory accesses.
    2. these memory accesses must be evenly distributed to the channels and banks.
    3. each access to a bank must also be a coalesced access.

Data Distribution

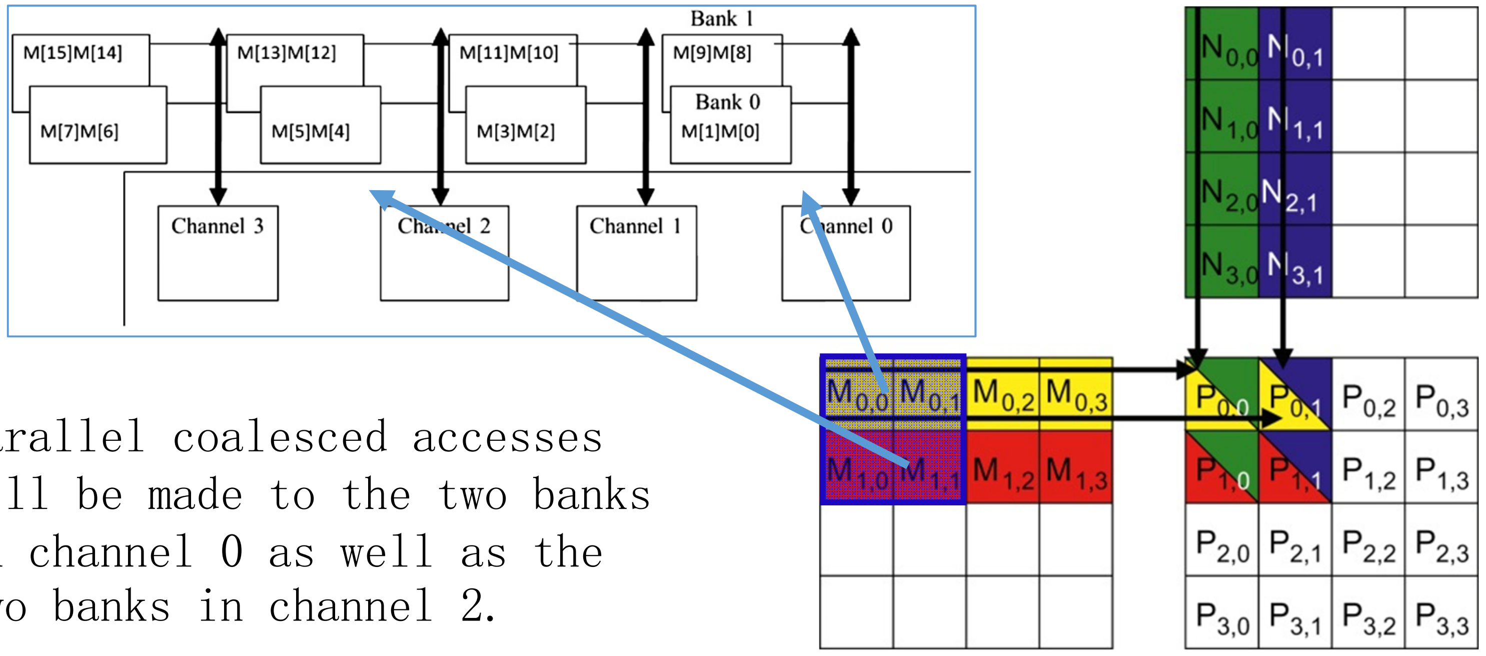
- The distribution scheme ***spreads*** the elements across the banks and channels in the system:



Interleaved data distribution

Interleaved Data Distribution

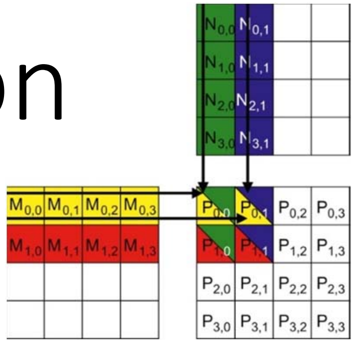
- Example:
  - multiplication will be performed with 2×2 thread blocks and 2×2 tiles



Bank 1
M[15]M[14]
M[13]M[12]
M[11]M[10]
M[9]M[8]
Bank 0
M[7]M[6]
M[5]M[4]
M[3]M[2]
M[1]M[0]
Channel 3
Channel 2
Channel 1
Channel 0
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
$N_{2,0}$ $N_{2,1}$
$N_{3,0}$ $N_{3,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Parallel coalesced accesses will be made to the two banks in channel 0 as well as the two banks in channel 2.

Interleaved Data Distribution



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

- Example:

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]

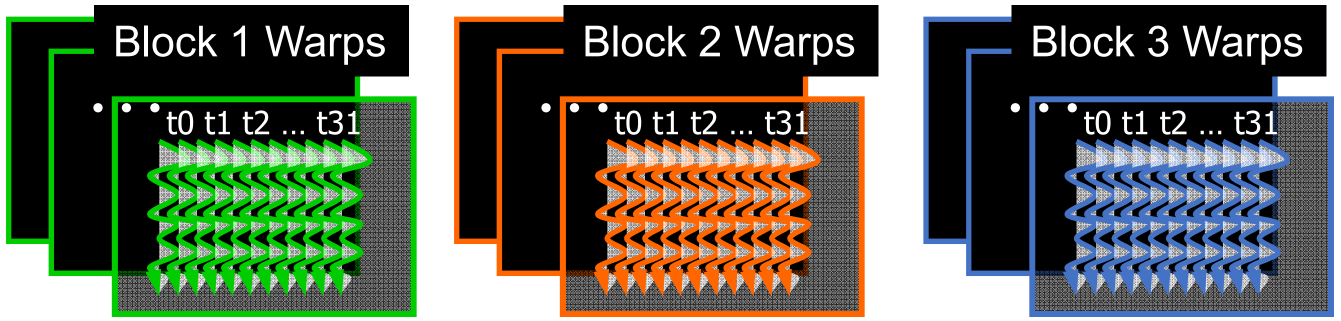
cache

Multiplying 8×8 matrices with the same 2×2 thread block configuration: What's the number of channels to make use?

Content

- Global Memory Bandwidth
- More on Memory Parallelism
- <u>Warps and SIMD Hardware</u>
- Dynamic partitioning of resources

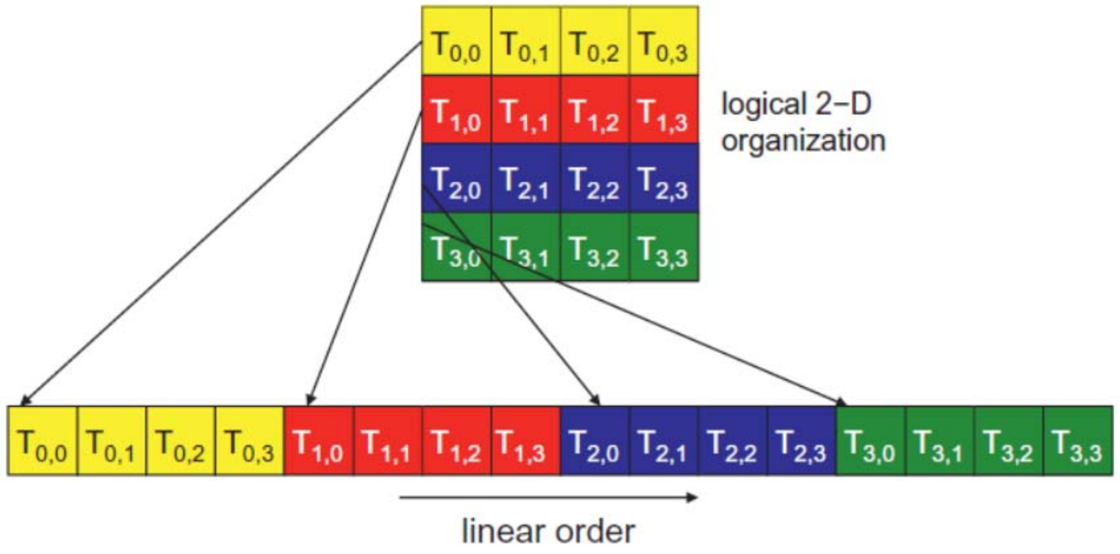
Warps as Scheduling Units



- Each block is divided into 32-thread warps
  - An implementation technique
  - Warps are scheduling units in SM
  - Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  - The number of threads in a warp may vary in future generations

Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order before partitioning into warps.
  - In x-dimension first, y-dimension next, and z-dimension last
  - For this example, all 16 threads form half a warp.
  - The warp will be padded with another 16 threads to complete a 32-thread warp.



$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2–D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$ $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

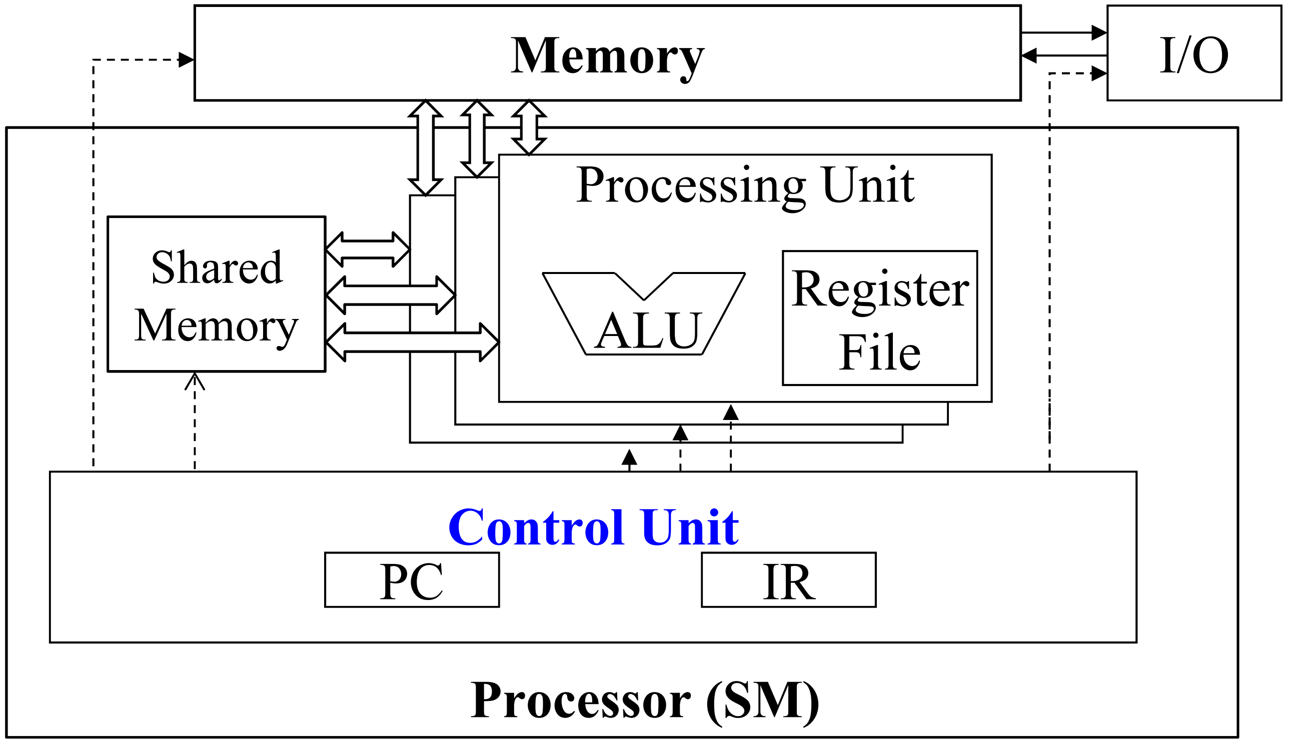
$T_{y,x}$, x being `threadIdx.x` and y being `threadIdx.y`.

Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

SMs are SIMD Processors

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



SIMD Execution 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 <u>same control flow path</u>
  - 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 <u>function of thread indices</u>:
  - 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

Device Code

```
// 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 vector size of 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%

Performance Impact of Control Divergence

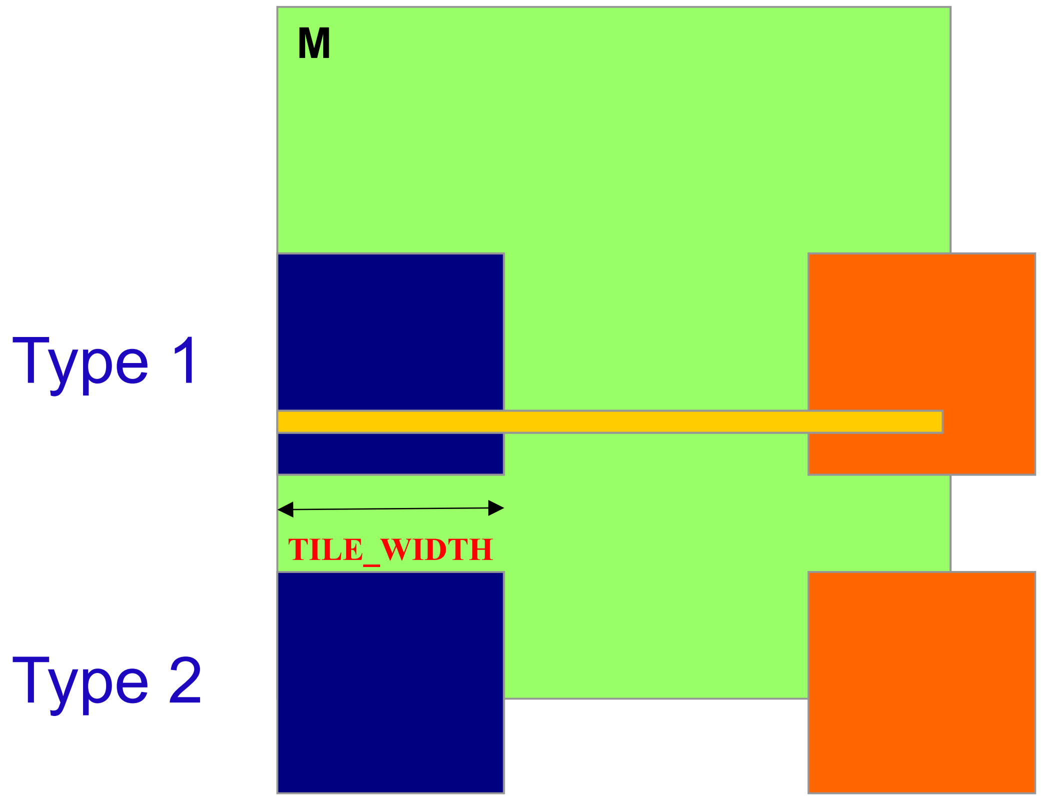
- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The **tiled matrix multiplication** kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation
  - For example, see the tile loading code below:

```
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;
}
```

Two types of blocks in loading M Tiles

- 1. Blocks whose tiles are all within valid range until the last phase.
- 2. Blocks whose tiles are partially outside the valid range all the way



Analysis of Control Divergence Impact

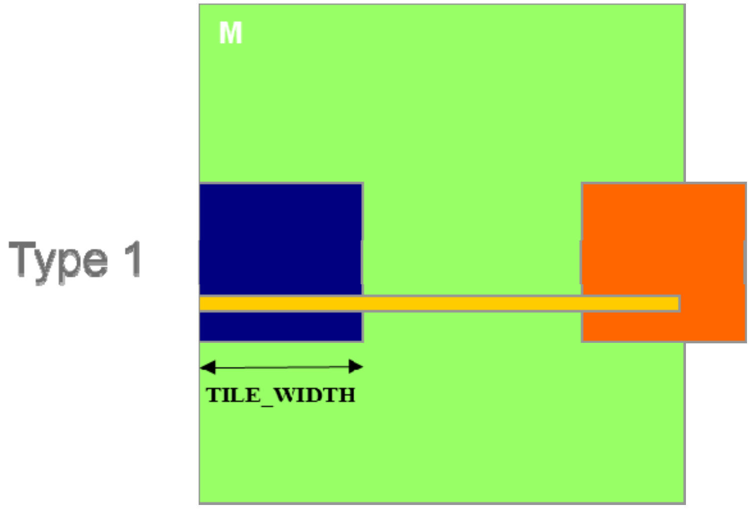
- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of 100/16)

- There are 49 thread blocks (7 in each dimension)

Control Divergence in Loading M Tiles

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

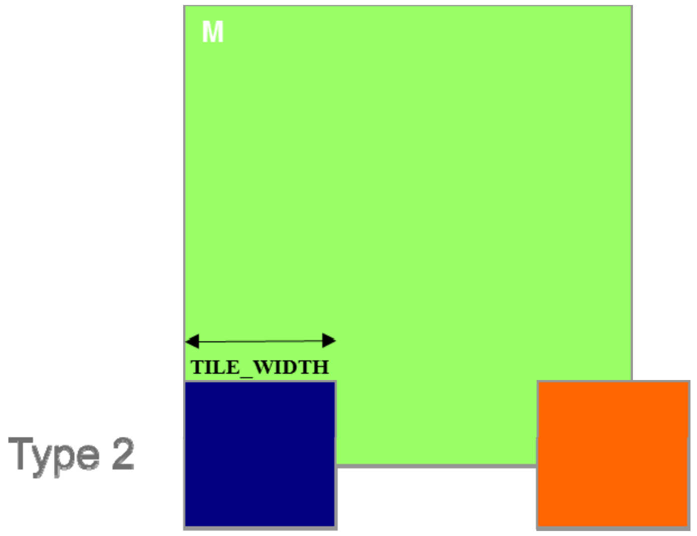
- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence



Control Divergence in Loading M Tiles (Type 2)

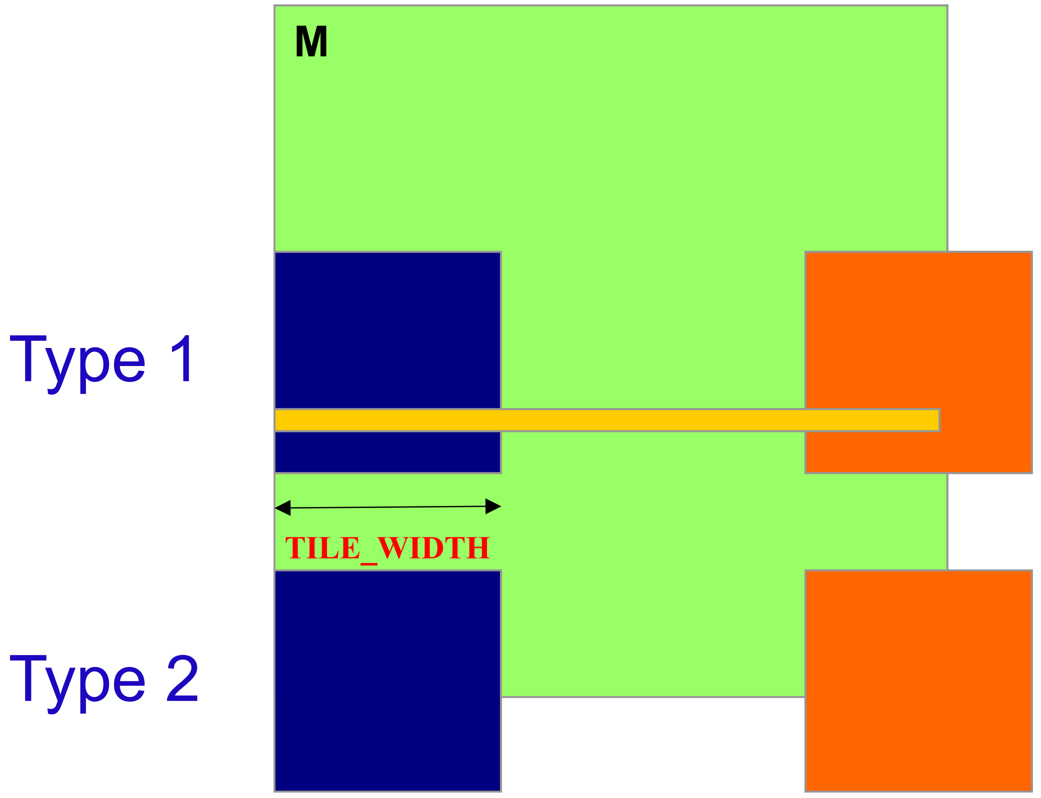
Type 2:

- the 7 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



Overall 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 or (336+14)/(2352+14))



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

Content

Execution resources

- The execution resources in an SM include:
  - registers,
  - shared memory,
  - thread block slots,
  - and thread slots.
- These resources are dynamically partitioned and assigned to threads to support their execution.

Execution resources

- **Thread slots:**
  - These thread slots are partitioned and assigned to thread blocks during runtime.
- For example: Fermi devices have 1536 thread slots.
  - If each thread block consists of 512 threads, the 1536 thread slots are partitioned and assigned to three blocks.
  - In this case, each SM can accommodate up to three thread blocks due to limitations on thread slots.
  - If each thread block contains 256 threads, the 1536 thread slots are partitioned and assigned to 6 thread blocks.
- The ability to dynamically partition the thread slots among thread blocks makes SMs versatile.

Execution resources

- **Block slots:**
  - subtle interactions between block slots and thread slots.
- For example: Fermi devices have 1536 thread slots.
  - If each block has 128 threads, the 1536 thread slots can be partitioned and assigned to 12 blocks.
  - However, since there are only 8 block slots in each SM, only 8 blocks will be allowed.
  - This means that in the end, only 1024 of the thread slots will be utilized.
- Therefore, to fully utilize both the block slots and thread slots, one needs at least 256 threads in each block.

Execution resources

- **Registers:**
  - the automatic variables declared in a CUDA kernel are placed into registers.
- For example: assume each SM has 16,384 registers.
  - If the kernel uses 10 registers per thread for 16×16 thread blocks, how many threads can run on each SM? 10*16*16 = 2560
  - The number of registers required by six blocks is 15,360. OK
  - Adding another block would require 17,920 registers? NOT
  - How about additional two automatic variables?
  - 12*16*16 = 3072; registers required by six blocks is now 18,432. NOT
- The CUDA runtime will reduce the number of blocks assigned to each SM by one.
  - This, however, reduces the number of threads running on an SM from 1536 to 1280.
  - “performance cliff”

Execution resources

- **Shared memory:**
  - another resource that is dynamically partitioned at run-time.
- For example: Tiled algorithms
  - require a large amount of shared memory to be effective.
  - large shared memory usage can reduce the number of thread blocks running on an SM.
- Reduced thread parallelism can negatively affect the utilization of the memory access bandwidth;
  - reduced memory access throughput, in turn, can further reduce the thread execution throughput.

Execution resources

- Constraints of all the dynamically partitioned resources interact with each other in a complex manner.
- CUDA Occupancy Calculator
  - a downloadable Excel sheet
  - calculates the actual number of threads running on each SM for a particular device given the usage of resources by a kernel.

summary

- The memory bandwidth is a first-order performance factor in a massively parallel processor
  - DRAM bursts, banks, and channels
  - All concepts are also applicable to modern multicore processors

- The memory coalescing is important for effectively utilizing memory bandwidth in CUDA
  - Its origin in DRAM burst
  - Checking if a CUDA memory access is coalesced
  - Techniques for improving memory coalescing in CUDA code