www.bsc.es

BSC
Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

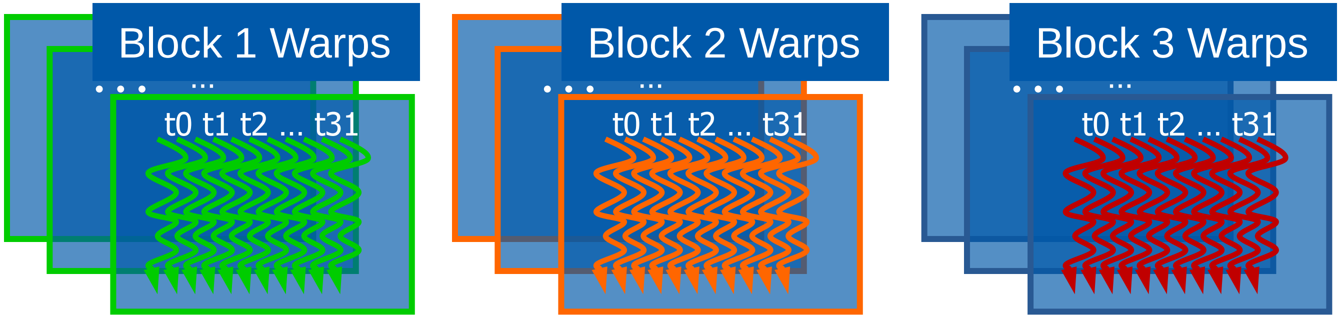
Efficiency and Performance Considerations

Marc Jordà, Antonio J. Peña

Based on material from NVIDIA’s GPU Teaching Kit

Montevideo, 21-25 October 2019

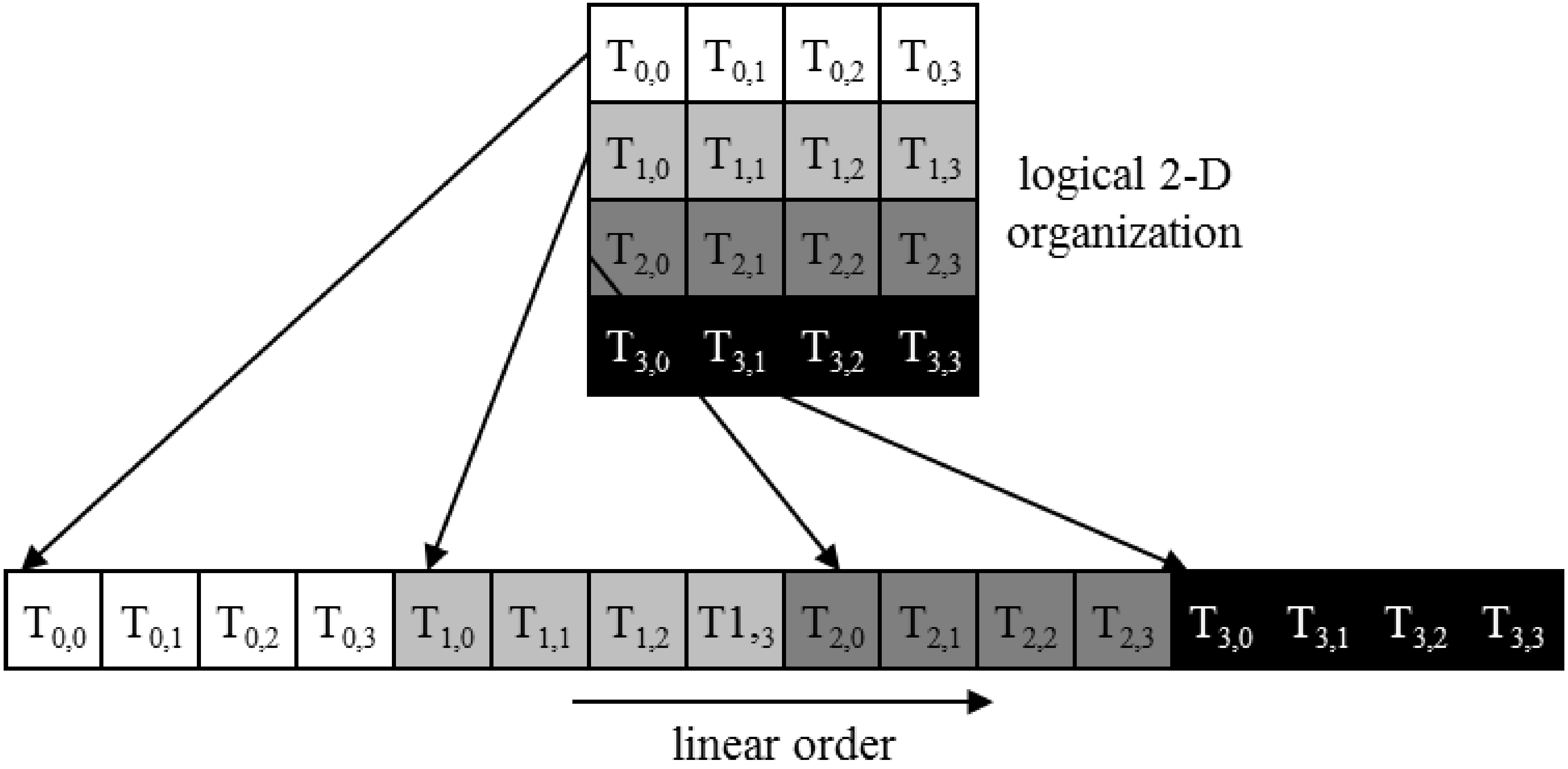
Warps as Scheduling Units



- Each block is divided into 32-thread warps
  - An implementation technique, not part of the CUDA programming model
  - 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
  - In x-dimension first, y-dimension next, and z-dimension last



$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2-D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ T1,3 $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

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

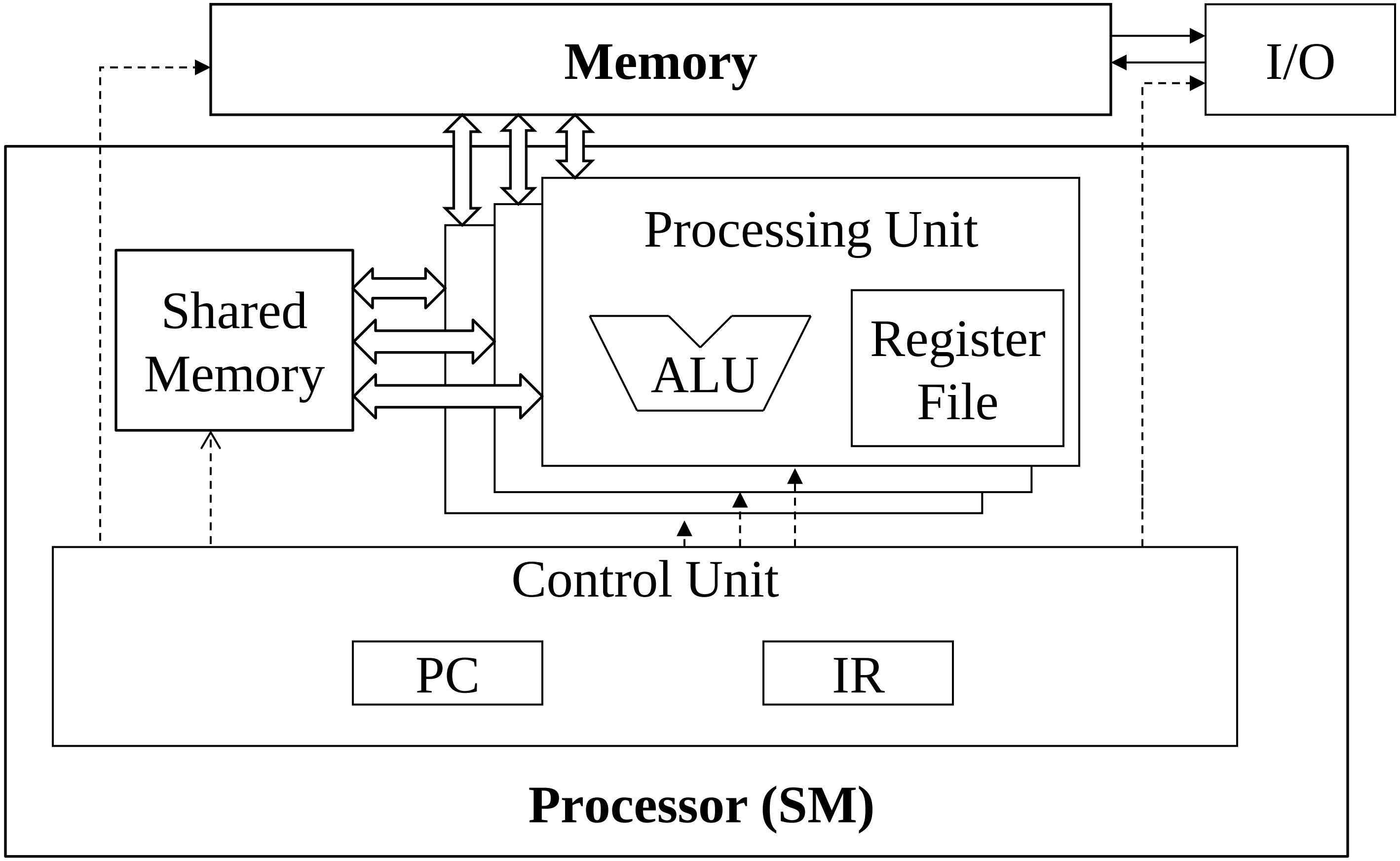
- Thus you can use this knowledge in control flow
- However, the exact size of warps may change from generation to generation

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 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 within a warp taking **different paths is 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
  - 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 have 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 performance degradation
  - For example, see the tile loading code below:

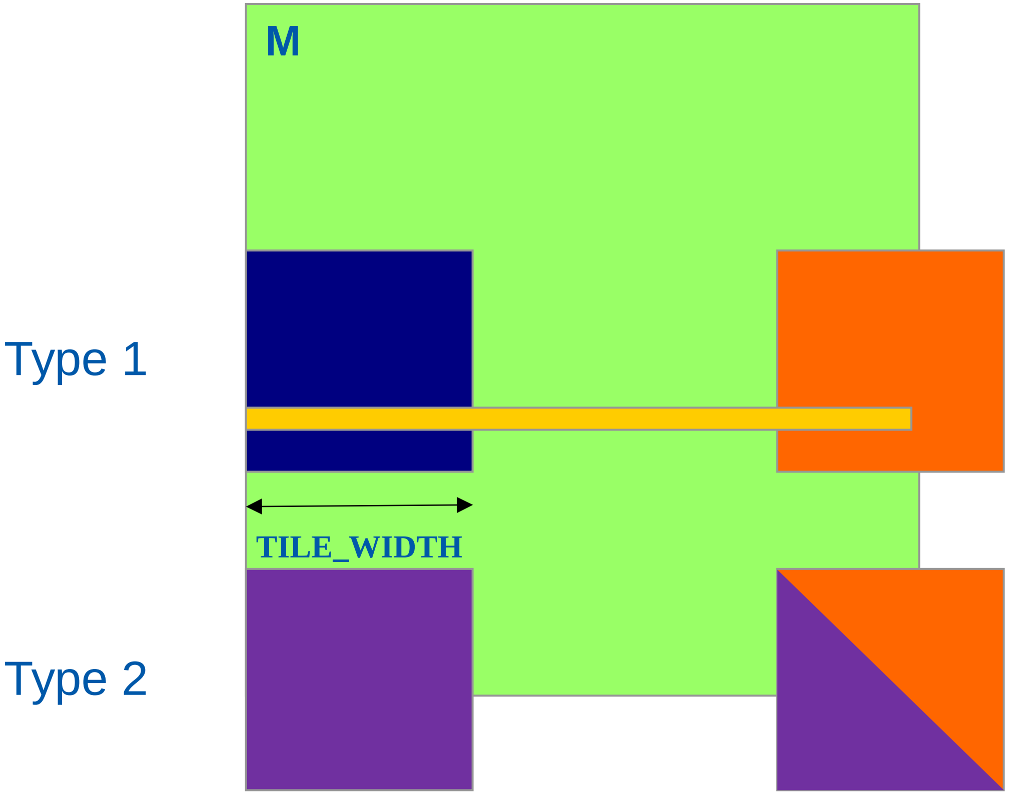
```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
   ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
} else {
   ds_M[ty][tx] = 0.0;
}

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

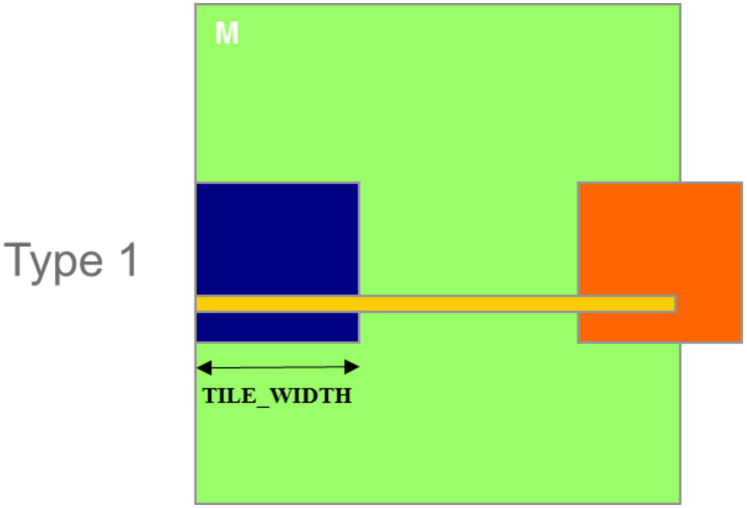
- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

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

Control Divergence in Loading M Tiles

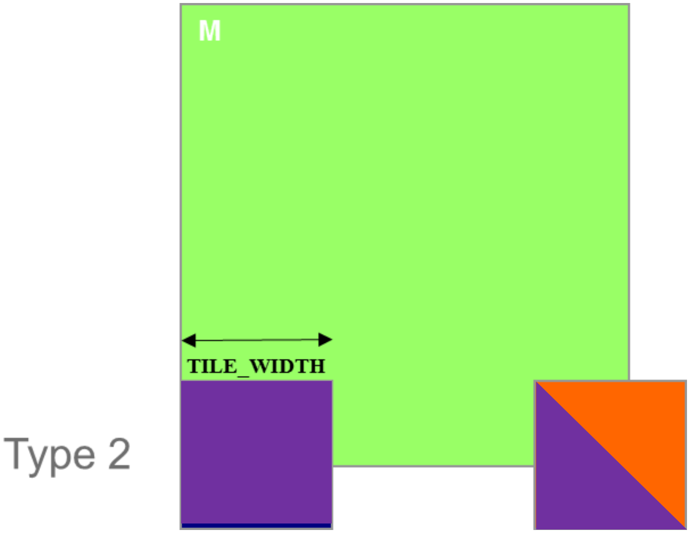
- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)

- 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



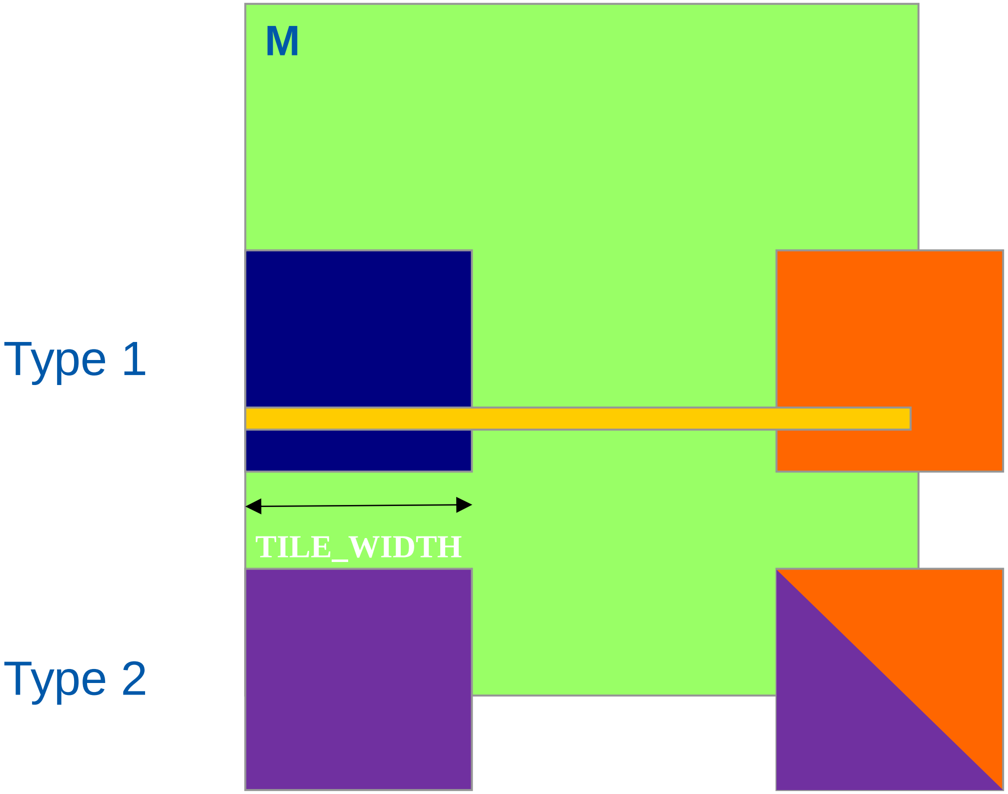
Control Divergence in Loading M Tiles (Type 2)

- Type 2: the 7 blocks 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 13% ((336+14)/(2352+392))



Additional Comments

- The calculation of impact of control divergence in loading N tiles is somewhat different and is left as an exercise

- 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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

QUIZ – PART I

Question 1

- We are to process a 600x800 (800 pixels in the *x* or horizontal direction, 600 pixels in the *y* or vertical direction) picture with the *PictureKernel()*. That is, *m*=600 and *n*=800.

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
  // Calculate the row # of the d_Pin and d_Pout element to process
  int Row = blockIdx.y*blockDim.y + threadIdx.y;
  // Calculate the column # of the d_Pin and d_Pout element to process
  int Col = blockIdx.x*blockDim.x + threadIdx.x;
  if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
     d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

Assume that we decided to use a grid of 16x16 blocks. How many warps will be generated during the kernel execution?

a) 37 * 16
b) 38 * 50
c) 38 * 8 * 50
d) 38 * 50 * 2

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Question 1 - Answer

- We are to process a 600x800 (800 pixels in the *x* or horizontal direction, 600 pixels in the *y* or vertical direction) picture with the *PictureKernel()*. That is, *m*=600 and *n*=800.

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
       d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

Assume that we decided to use a grid of 16x16 blocks. How many warps will be generated during the kernel execution?

a) 37 * 16
b) 38 * 50
c) 38 * 8 * 50
d) 38 * 50 * 2

Explanation: There are ceil(800/16.0) = 50 blocks in the *x* direction and ceil(600/16.0) = 38 blocks in the *y* direction. Each block contributes (16*16)/32=8 warps. So there are 38*50*8 warps.

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Question 2

- In the same problem (*m*=600; *n*=800), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) 37 + 50 * 8

b) 38 * 16

c) 50

d) 0

Question 2 - Answer

- In the same problem (*m*=600; *n*=800), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
   // Calculate the row # of the d_Pin and d_Pout element to process
   int Row = blockIdx.y*blockDim.y + threadIdx.y;
   // Calculate the column # of the d_Pin and d_Pout element to process
   int Col = blockIdx.x*blockDim.x + threadIdx.x;
   if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
      d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) 37 + 50 * 8

b) 38 * 16

c) 50

d) 0

Explanation: The size of the picture in the *x* dimension is a multiple of 16 so there is no block in the *x* direction that has any threads in the invalid range. The size of the picture in the *y* dimension is 37.5 times of 16. This means that the threads in the last block are divided into halves: 128 in the valid range and 128 in the invalid range. Since 128 is a multiple of 32, all warps will fall into either one or the other range. There is no control divergence.

Question 3

- In the same code, if we process an 800x600 picture (600 pixels in the *x* direction; 800 pixels in *y*), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in 
range
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) 37 + 50 * 8
b) 38 * 16
c) 50 * 8
d) 0

Question 3 - Answer

- In the same code, if we process an 800x600 picture (600 pixels in the *x* direction; 800 pixels in *y*), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) 37 + 50 * 8
b) 38 * 16
c) 50 * 8
d) 0

Explanation: The size of the picture in the *x* dimension is 600, which is 37.5 times of 16. This means that every warp processing the right edge of the picture will have control divergence. There are 50*8 such warps (50 blocks, 8 warps in each block). Since the size of the picture in the *y* dimension is a multiple of 16, there is no more divergence in the warps that process the lower edge of the picture.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 4

- In the same code, if we are to process a 799x600 picture (600 pixels in the *x* direction; 799 pixels in *y*), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) 37 + 50 * 8

b) (37 + 50) * 8

c) 50 * 8

d) 0

Question 4 - Answer

- In the same code, if we are to process a 799x600 picture (600 pixels in the *x* direction; 799 pixels in *y*), how many warps will have control divergence?

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {
    // Calculate the row # of the d_Pin and d_Pout element to process
    int Row = blockIdx.y*blockDim.y + threadIdx.y;
    // Calculate the column # of the d_Pin and d_Pout element to process
    int Col = blockIdx.x*blockDim.x + threadIdx.x;
    if ((Row < m) && (Col < n))  // Each thread computes one element of d_Pout if in range
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
}
```

a) **37 + 50 * 8**

b) (37 + 50) * 8

c) 50 * 8

d) 0

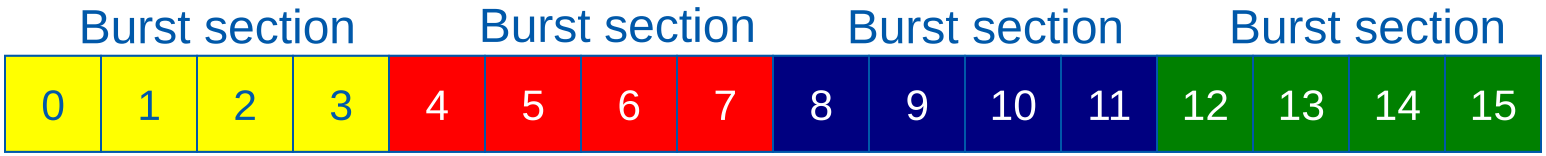
Explanation: The number of warps processing the right edge remains 50*8, all of which will have control divergence. However, the warps processing the lower edge of the picture will also have control divergence. There are 38 of them. One of them is already counted for processing the right edge. So we have 50*8+38-1 = 50*8+37.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

MEMORY ACCESS PERFORMANCE

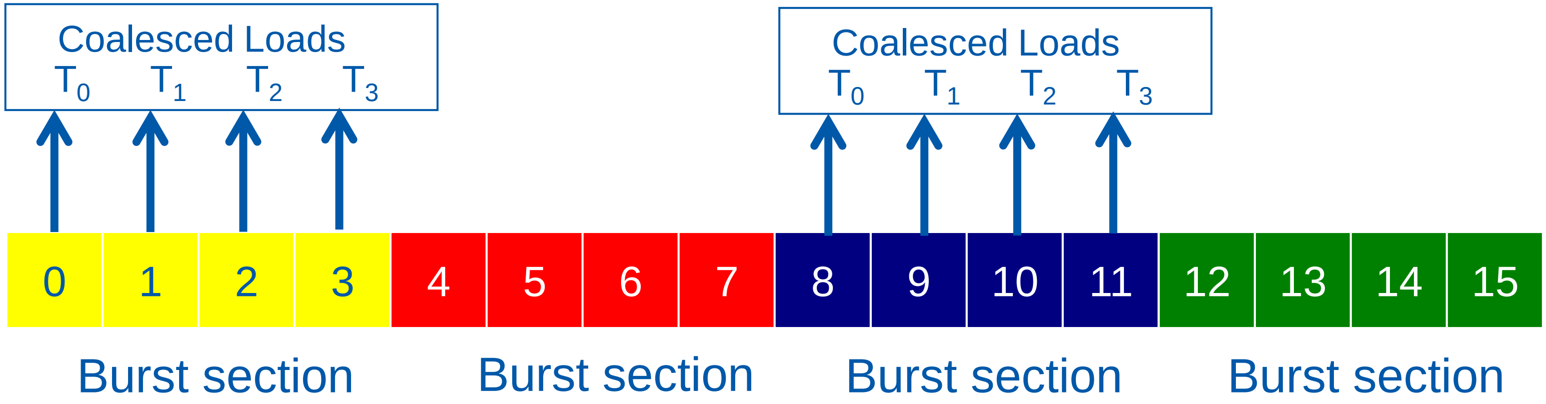
DRAM Burst – A System View



Burst section				Burst section				Burst section				Burst section			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- The 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
- Toy 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

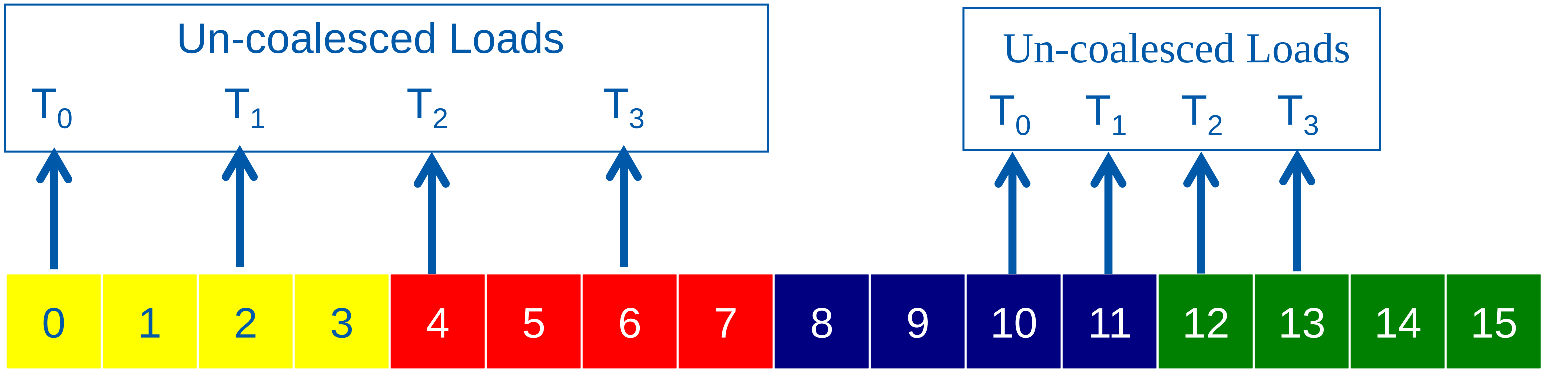
Memory Coalescing



- 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.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Un-coalesced Accesses

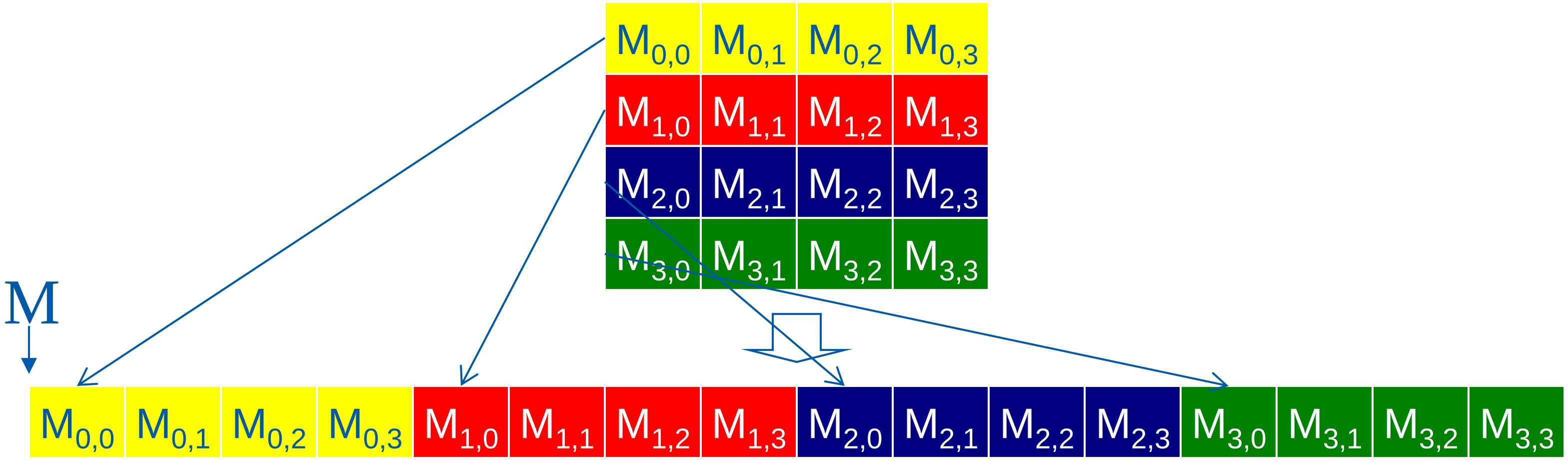


- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

How to judge if an access is coalesced?

- Accesses in a warp are to consecutive locations if the index in an array access is in the form of
  - A[(expression with terms independent of threadIdx.x) + threadIdx.x];

A 2D C Array in Linear Memory Space

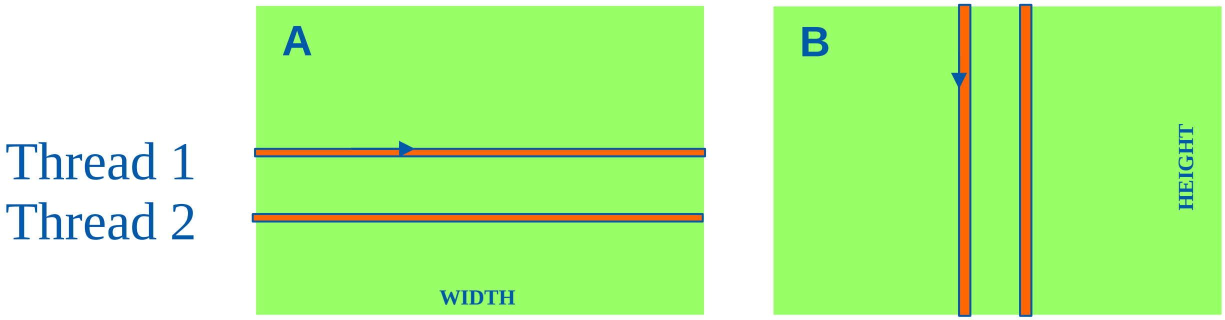


$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

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

linearized order in increasing address

Two Access Patterns of Basic Matrix Multiplication



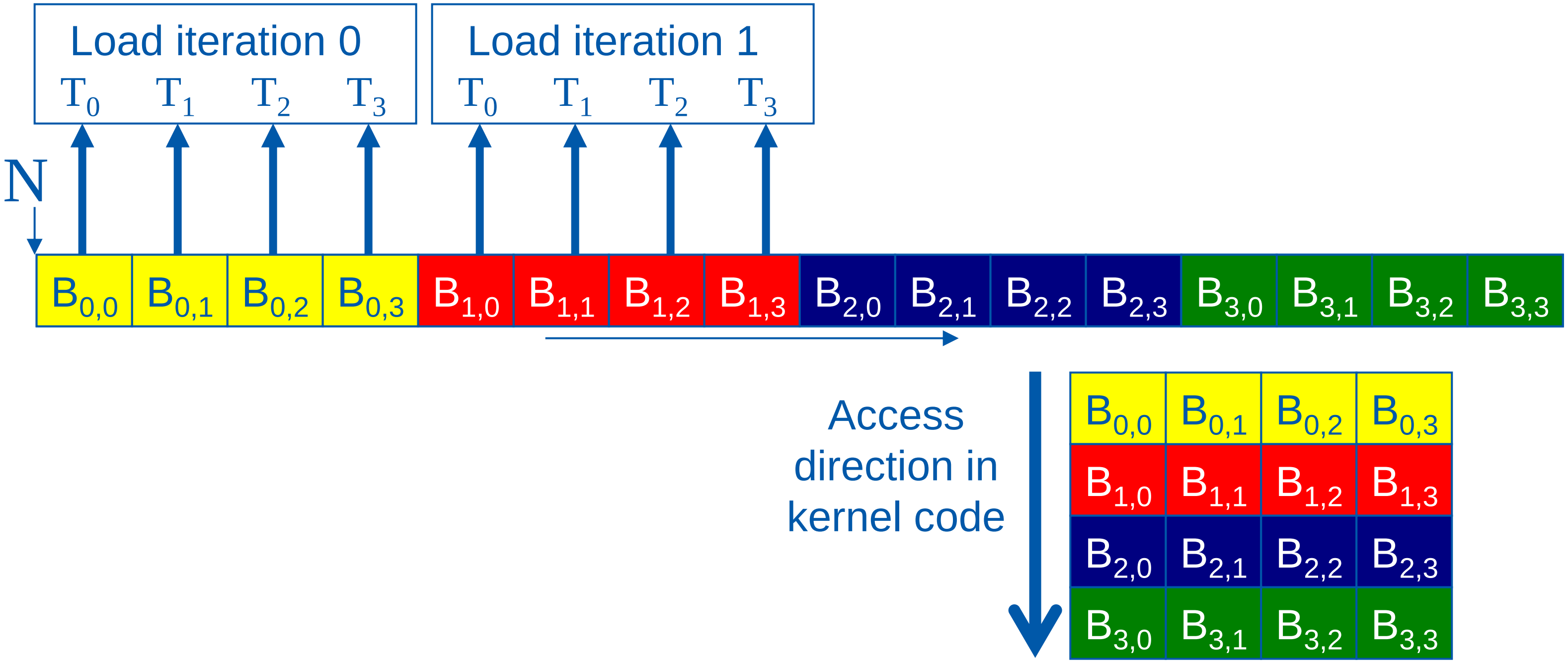
`A[Row*n+i]`   `B[i*k+Col]`

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

A is m × n, B is n × k

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

B Accesses are Coalesced



Load iteration 0

T_0 T_1 T_2 T_3

Load iteration 1

T_0 T_1 T_2 T_3

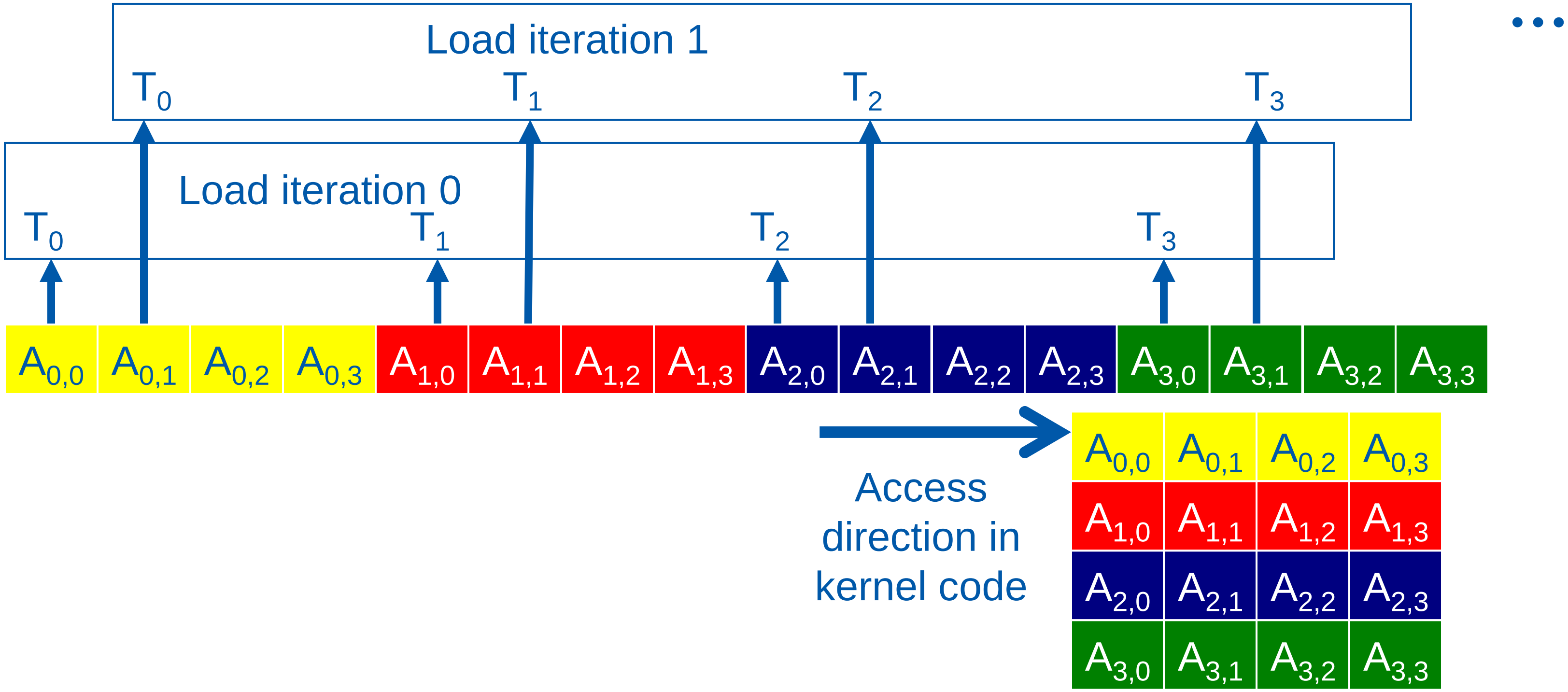
N

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

Access direction in kernel code

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

A Accesses are Not Coalesced



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

Access direction in kernel code

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

BSC

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

QUIZ – PART II

Question 1

- We want to use each thread to calculate two output elements of a vector addition. Assume that variable *i* should be initialized with the index for the first element to be processed by a thread. Which of the following should be used for such initialization to allow correct, coalesced memory accesses to these first elements in the following statement?

```
if(i<n) C[i] = A[i] + B[i];
```

a) int i = (blockIdx.x * blockDim.x) * 2 + threadIdx;

b) int i = (blockIdx.x * blockDim.x + threadIdx.x) * 2;

c) int i = (threadIdx.x * blockDim.x) * 2 + blockIdx.x;

d) int i = (threadIdx.x * blockDim.x + blockIdx.x) * 2;

Question 1 - Answer

- We want to use each thread to calculate two output elements of a vector addition. Assume that variable *i* should be initialized with the index for the first element to be processed by a thread. Which of the following should be used for such initialization to allow correct, coalesced memory accesses to these first elements in the following statement?

```
if(i<n) C[i] = A[i] + B[i];
```

a) **int i = (blockIdx.x * blockDim.x) * 2 + threadIdx;**

b) int i = (blockIdx.x

c) int i = (threadIdx.

d) int i = (threadIdx.

Explanation: Each thread block will process a section of each vector that contains twice as many elements as the number of threads in the thread block. We want to have all threads to process the first half of the section with each adjacent thread processing adjacent elements. The expression in **a)** allows such a pattern.

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Question 2

- Continuing from Question 1, what would be the correct statement for each thread to process the second element?

if(i<n) C[i] = A[i] + B[i];

a) if (i<n) C[i+1] = A[i+1] + B[i+1];

b) if (i+1<n) C[i+1] = A[i+1] + B[i+1];

c) if (i+threadIdx.x < n)
 C[i+threadIdx.x] = A[i+threadIdx.x] + B[i+threadIdx.x];

a) if(i+blockDim.x < n)
 C[i+blockDim.x] = A[i+blockDim.x] + B[i+blockDim.x];

Question 2 - Answer

- Continuing from Question 1, what would be the correct statement for each thread to process the second element?

if(i<n) C[i] = A[i] + B[i];

a) If (i<n) C[i+1] = A[i+1] + B[i+1];

b) If (i+1<n) C[i+1] = A[i+1] + B[i+1];

c) If (i+threadIdx.x < n)
C[i+threadIdx.x] = A[i+threadIdx.x] + B[i+threadIdx.x];

**a) if(i+blockDim.x < n)
C[i+blockDim.x] = A[i+blockDim.x] + B[i+blockDim.x];**

Explanation: all threads should be shifting to the next half of the section. Thus everyone should be moving to the element that is of BlockDim.x elements away from the first element it processed. The expression is actually (blockIdx.x*blockDim.x)*2 + blockDim.x + threadIdx.x.

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Question 3

- Assuming the following simple matrix multiplication kernel, which of the following is true?

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
      float Pvalue = 0;
      for (int k = 0; k < Width; ++k) {
          Pvalue += M[Row*Width+k] * N[k*Width+Col];
      }
      P[Row*Width+Col] = Pvalue;
    }
}
```

a) M[Row*Width+k] and N[k*Width+Col] are coalesced but P[Row*Width+Col] is not
b) M[Row*Width+k], N[k*Width+Col] and P[Row*Width+Col] are all coalesced
c) M[Row*Width+k] is not coalesced but N[k*Width+Col] and P[Row*Width+Col] both are
d) M[Row*Width+k] is coalesced but N[k*Width+Col] andt P[Row*Width+Col] are not

Question 3 - Answer

- Assuming the following simple matrix multiplication kernel, which of the following is true?

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
      float Pvalue = 0;
      for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k] * N[k*Width+Col];
      }
      P[Row*Width+Col] = Pvalue;
    }
}
```

a) M[Row*Width+k] and N[k*Width+Col] are coalesced but P[Row*Width+Col] is not

b) M[Row*Width+k], N[k*Width+Col] and P[Row*Width+Col] are all coalesced

c) M[Row*Width+k] is not coalesced but N[k*Width+Col] and P[Row*Width+Col] both are

d) M[Row*Width+k] is coalesced but N[k*Width+Col] andt P[Row*Width+Col] are not

Explanation: M is accessed with Row*Width+k, which is actually (blockIdx.y*blockDin.y+threadIdx.y)*Width + k where threadIdx.y has Width coefficient. This violates the criterion. On the other hand, N and P are accessed with k*Width+Col, which is actually (k*Width + blockIdx.x*blockDim.x)+threadIdx.x. This meets the criterion

www.bsc.es

Barcelona
Supercomputing
Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact
marc.jorda@bsc.es, antonio.pena@bsc.es