



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

Memory and Data Locality

Marc Jordà, Antonio J. Peña

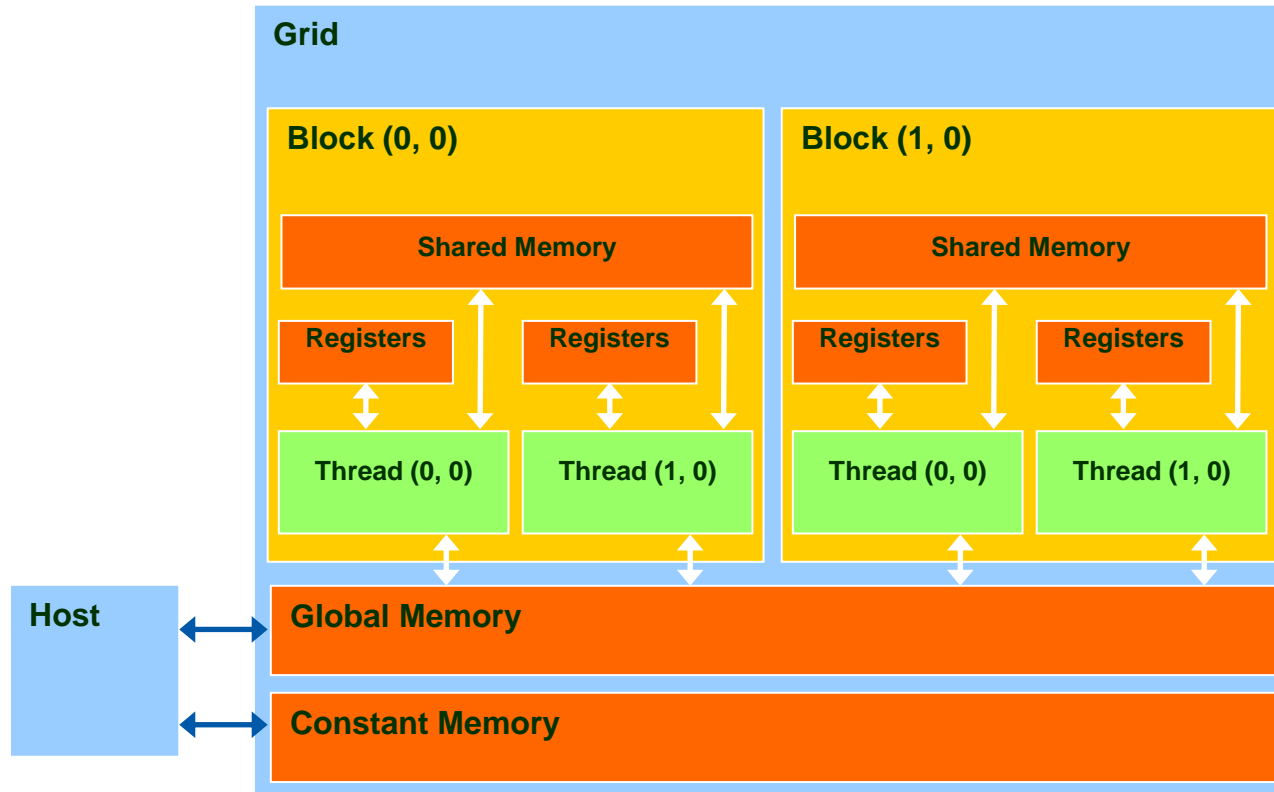
Based on material from NVIDIA's GPU Teaching Kit

Montevideo, 21-25 October 2019

How about performance on a GPU

- All threads access global memory for their input matrix elements
 - One memory access (4 bytes) per floating-point addition
 - 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
 - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
 - $4 \times 1,500 = 6,000$ GB/s required to achieve peak FLOPS rating
 - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS
- This limits the execution rate to 3.3% ($50/1500$) of the peak floating-point execution rate of the device!
- **Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS**

Programmer View of CUDA Memories



Declaring CUDA Variables

Variable declaration	Memory	Scope	Lifetime
<code>int LocalVar;</code>	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

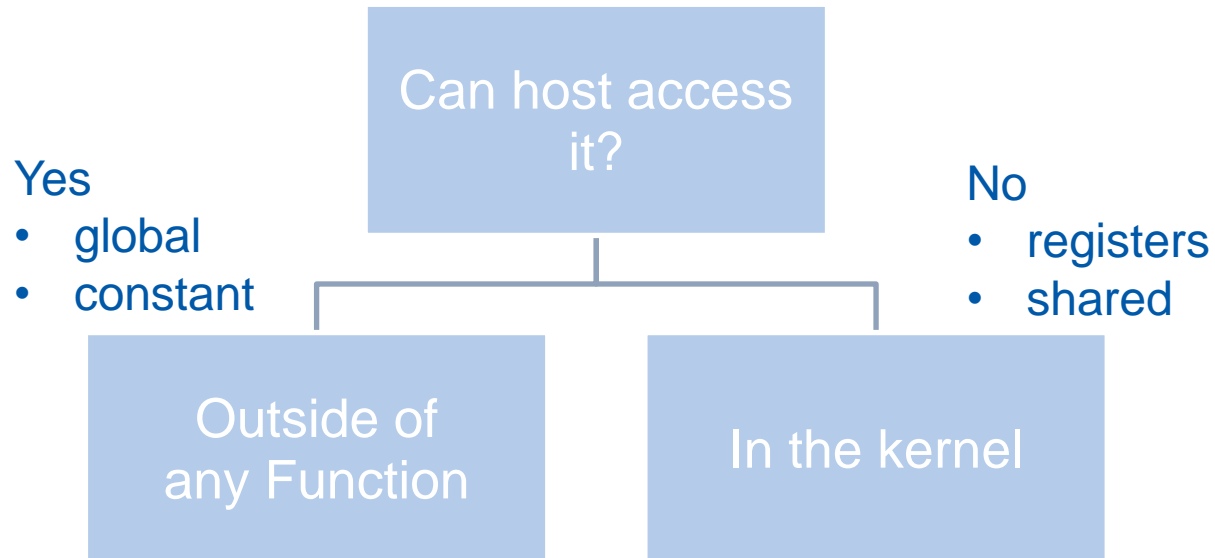
- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
 - Except per-thread arrays that usually reside in global memory
e.g. `int array[10];`

Example: Shared Memory Variable Declaration

```
__global__ void some_kernel(char* in, ...)  
{  
    __shared__ float sh_in[TILE_WIDTH][TILE_WIDTH];  
    ...  
}
```

Shared memory array dimension(s) must be known at compile time

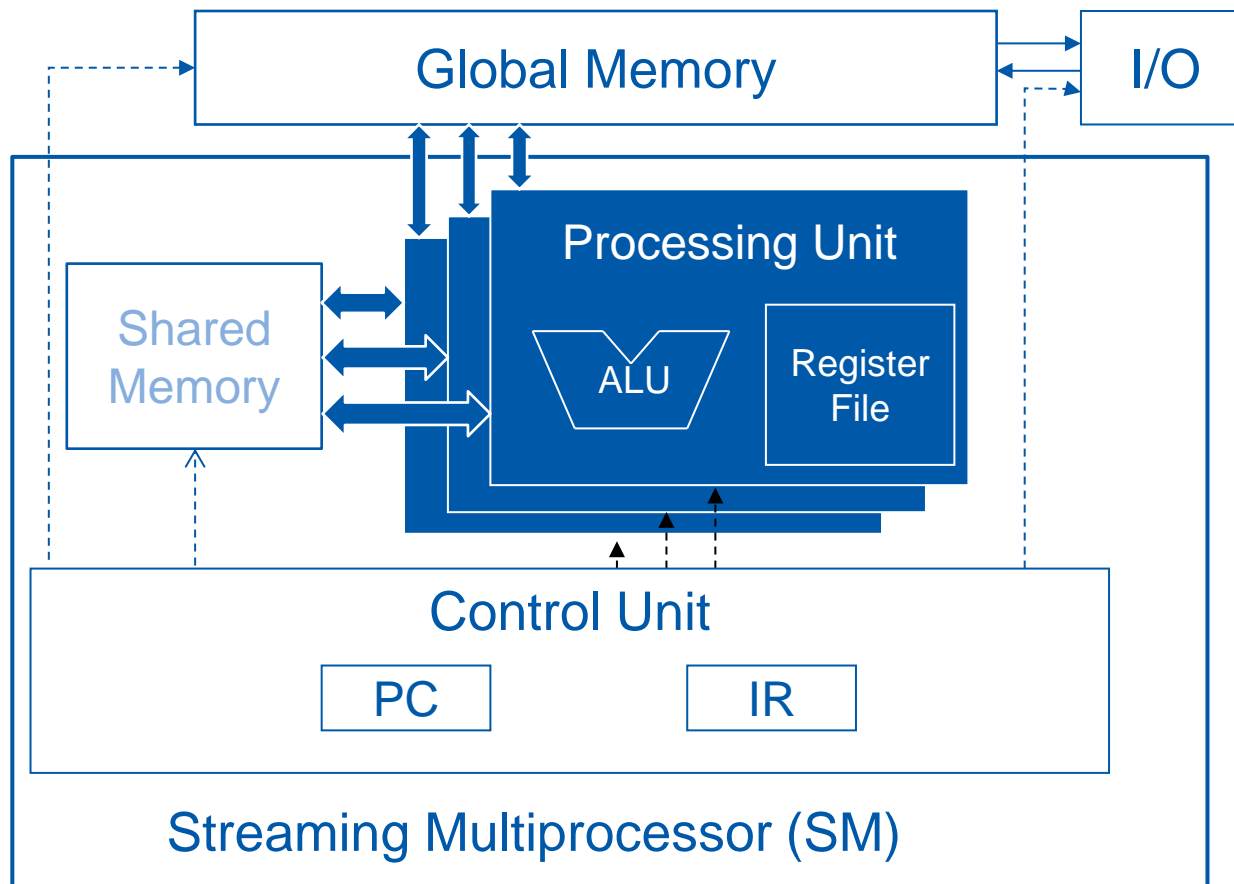
Where to Declare Variables?



Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
 - One in each SM
 - Accessed at much higher speed (in both latency and throughput) than global memory
 - Scope of access and sharing - thread blocks
 - Lifetime – thread block
 - contents will disappear after the corresponding thread block finishes/terminates execution
 - Accessed by memory load/store instructions
 - A form of scratchpad memory in computer architecture

Hardware View of CUDA Memories



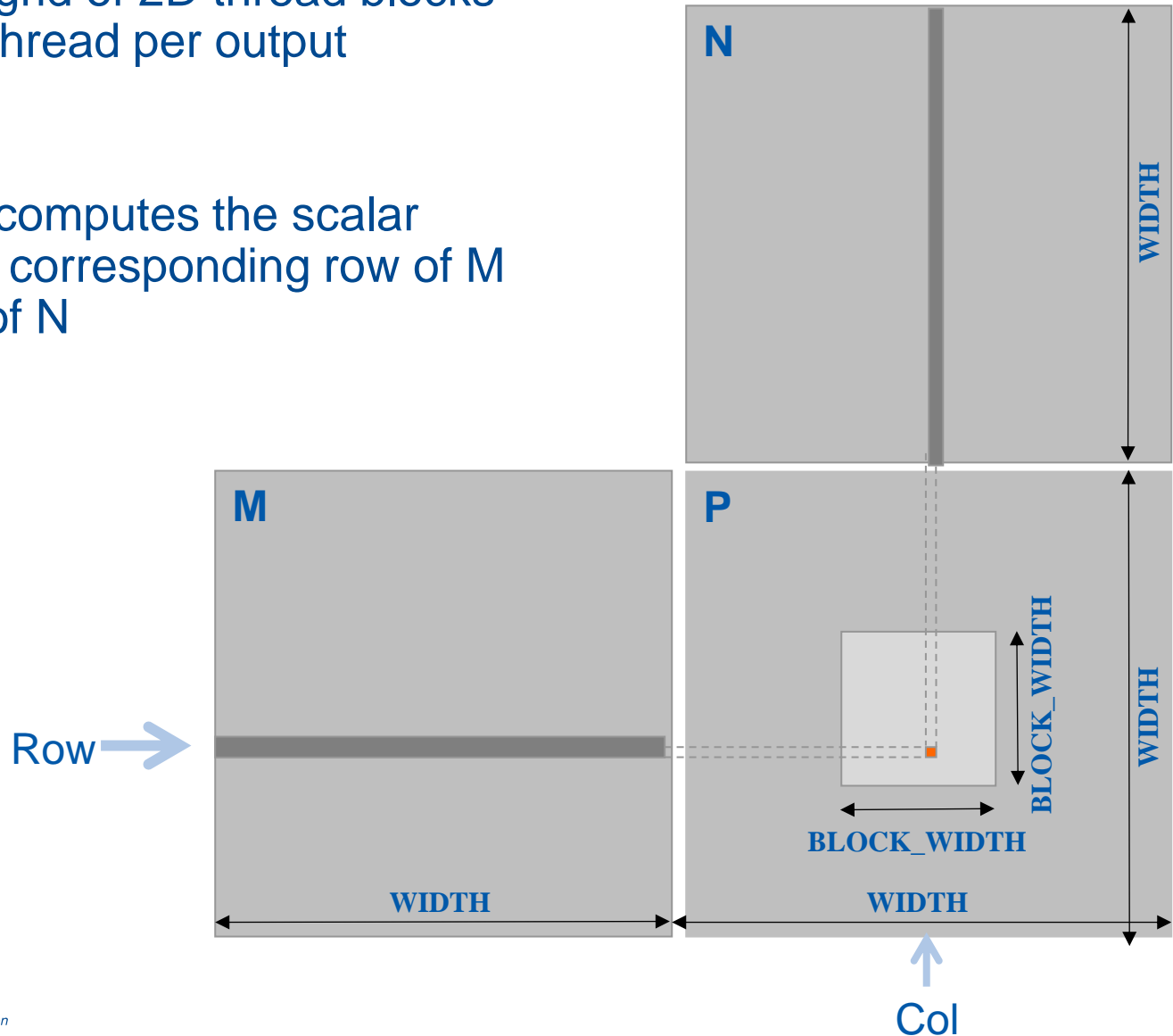


**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación

TILED PARALLEL ALGORITHMS

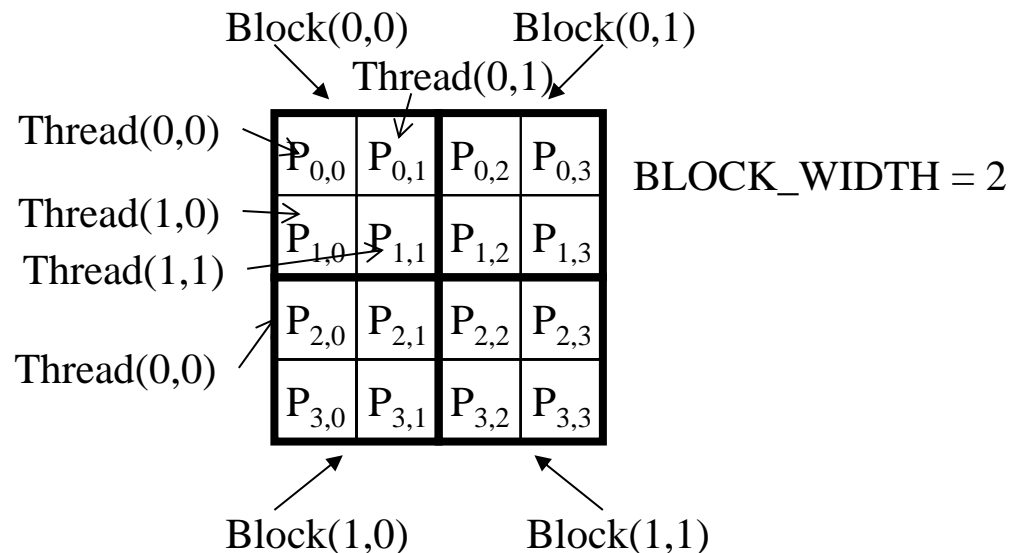
Use case – Matrix Multiplication

- Create a 2D grid of 2D thread blocks to have one thread per output element
- Each thread computes the scalar product of its corresponding row of M and column of N



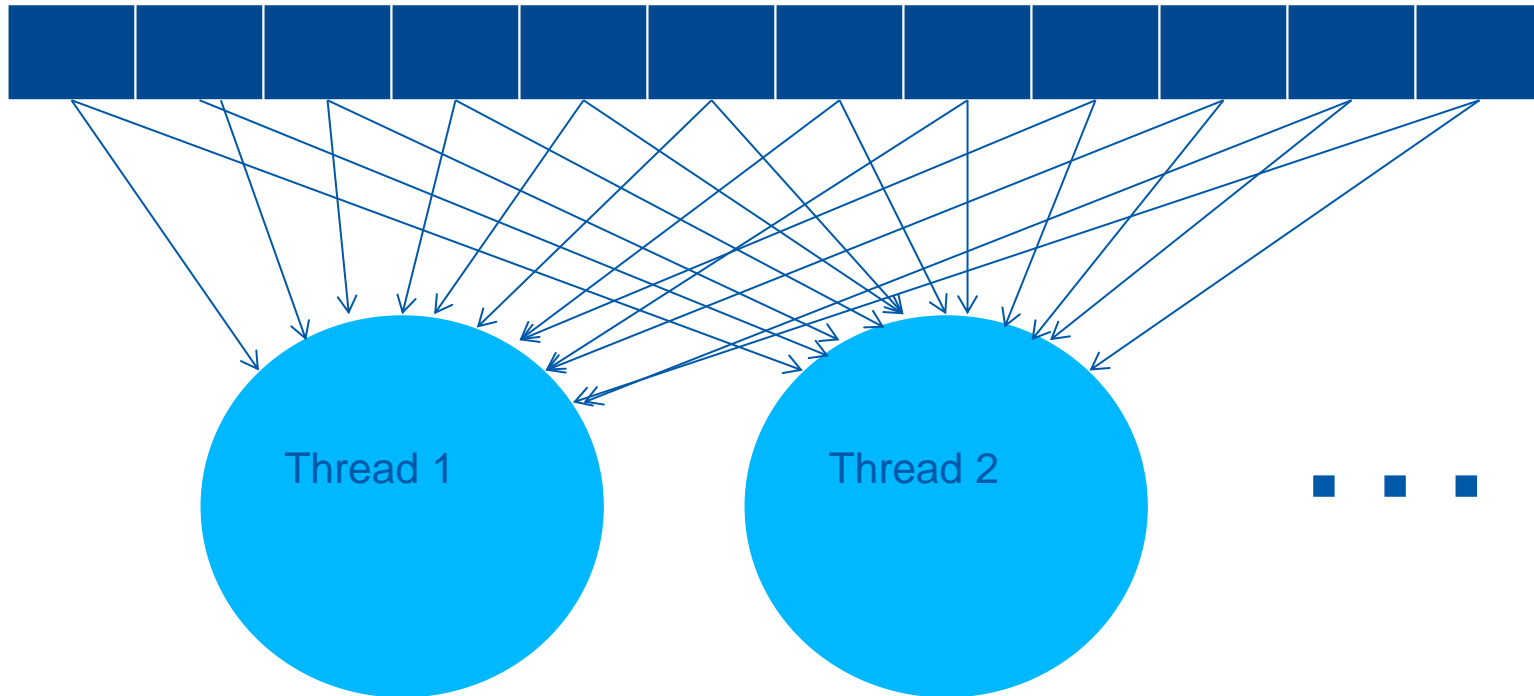
A Basic Matrix Multiplication

```
__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;  
  
    // compute element (Row, Col) of matrix P  
    ...  
}
```



Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory



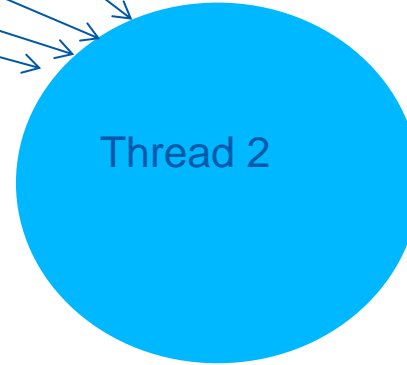
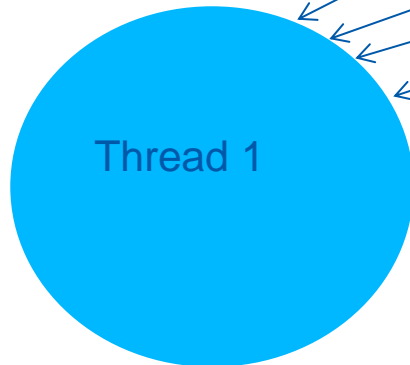
Data reuse by different threads

Tiling/Blocking - Basic Idea

Global Memory



On-chip Memory



Divide the global memory content into tiles

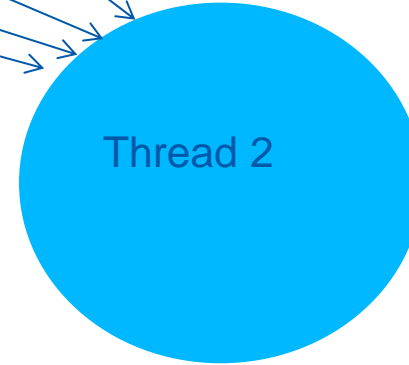
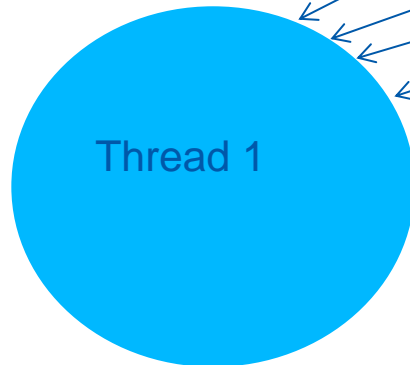
Focus the computation of threads on one tile at each point in time

Tiling/Blocking - Basic Idea

Global Memory



On-chip Memory



Divide the global memory content into tiles

Focus the computation of threads on one tile at each point in time

Basic Concept of Tiling

- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
 - Carpooling for commuters
 - Tiling for global memory accesses
 - drivers = threads accessing their memory data operands
 - cars = memory access requests



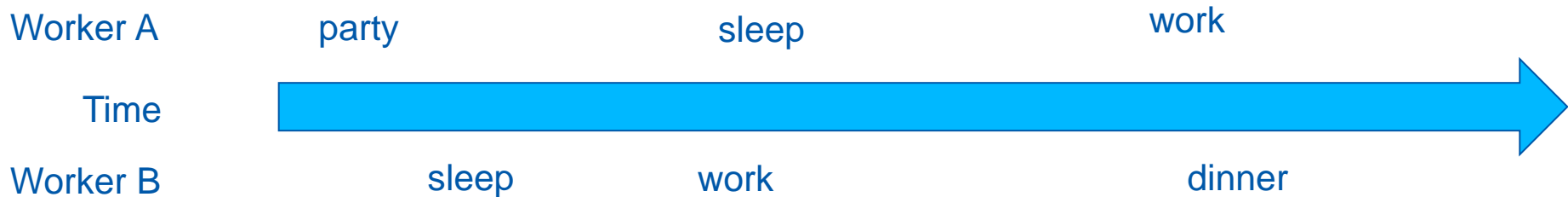
Carpools Need Synchronization

- Good: when people have similar schedule



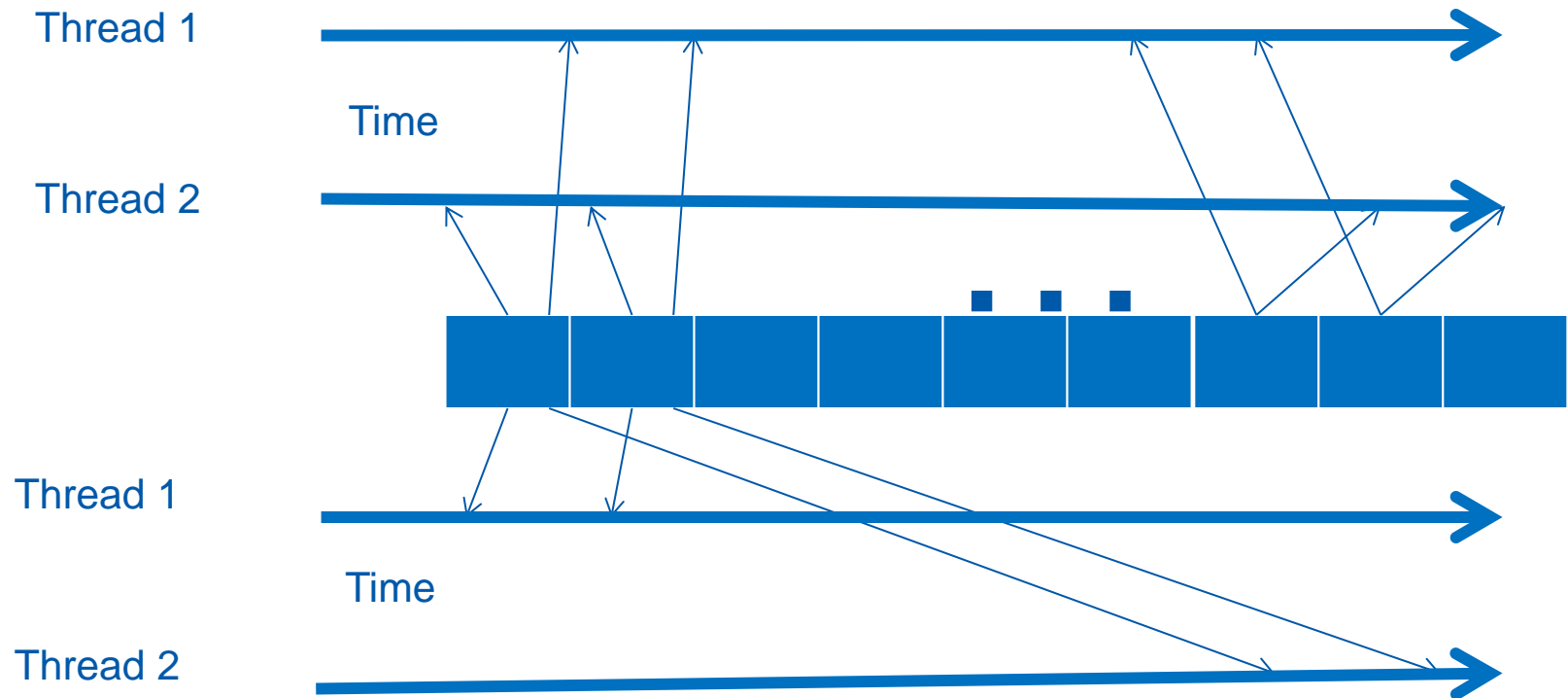
Carpools Need Synchronization

- Bad: when people have very different schedule



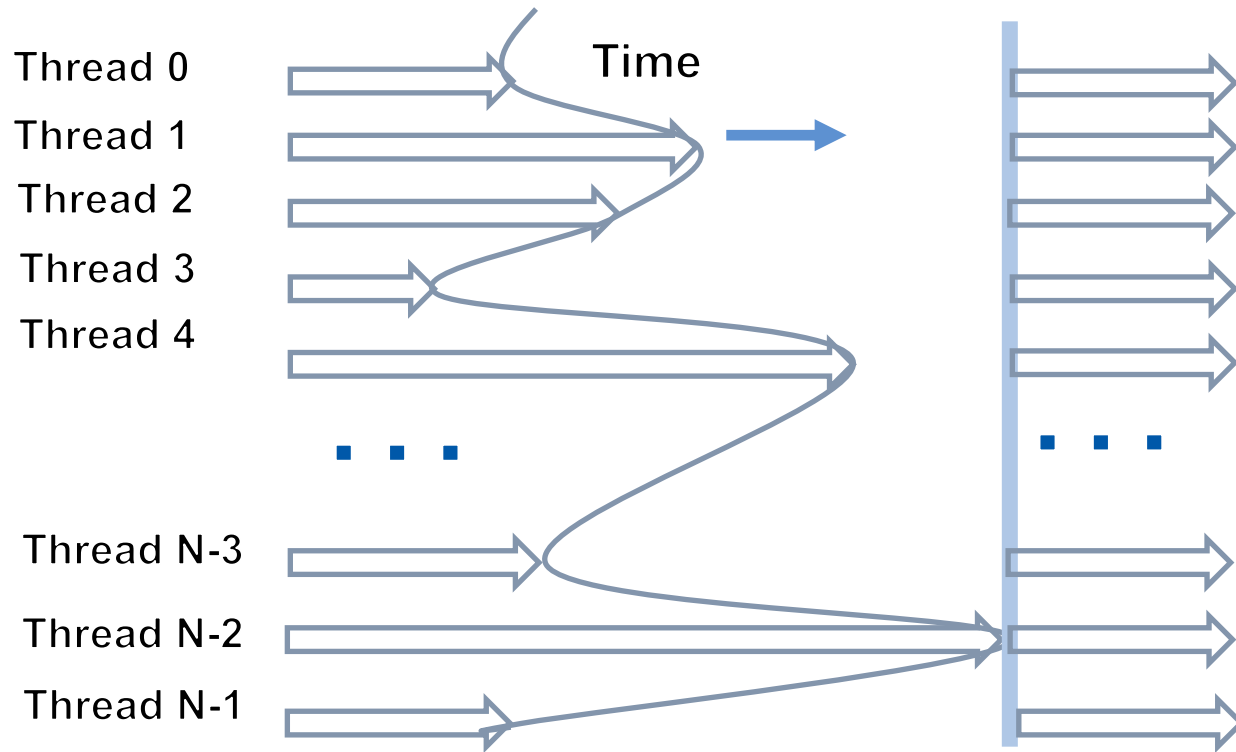
Same with Tiling

Good: when threads have similar access timing



Bad: when threads have very different timing

Barrier Synchronization for Tiling



Tiling needs synchronization to keep threads in the same phase

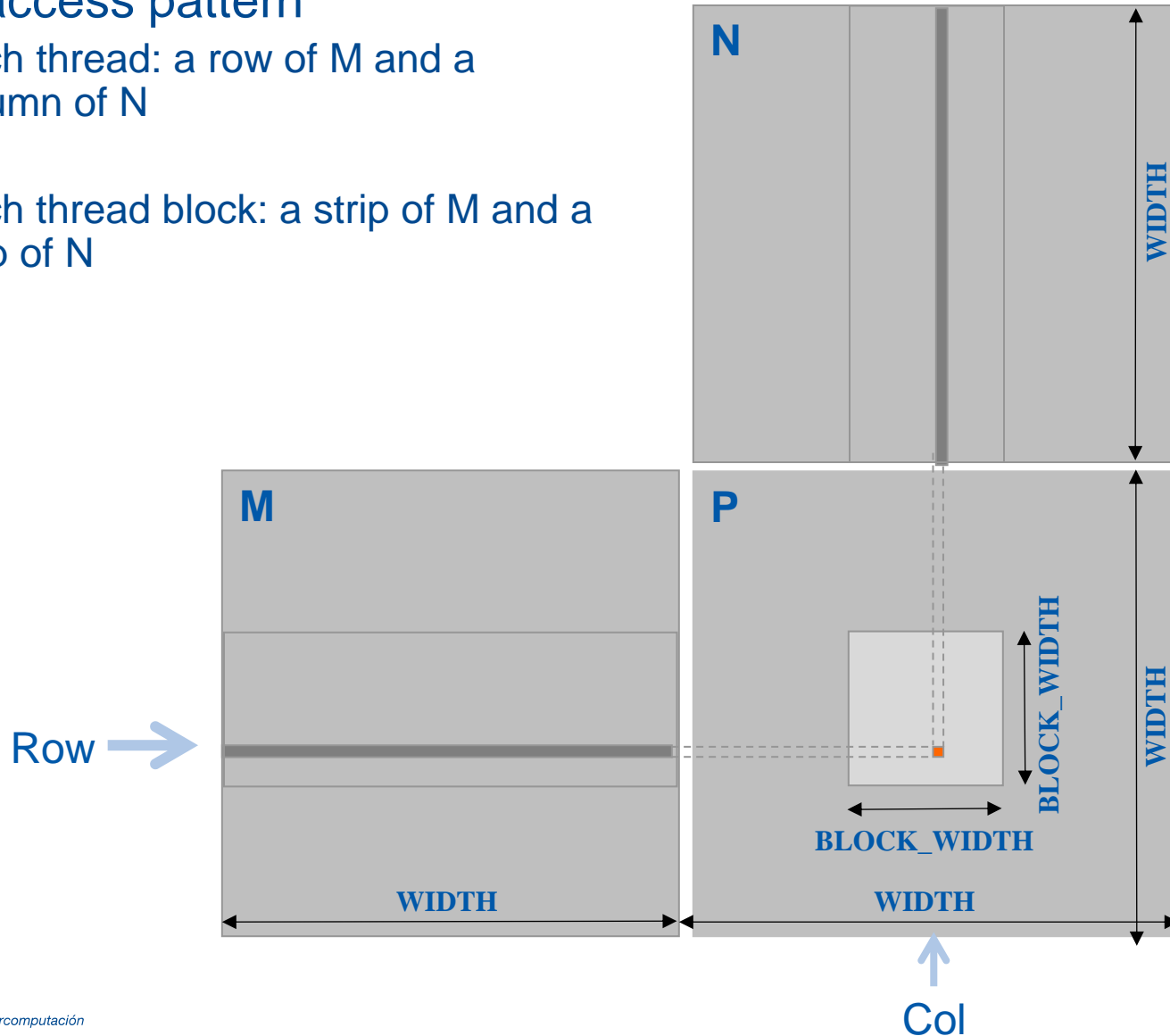
- CUDA provides barriers to synchronize the threads in a thread block

Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

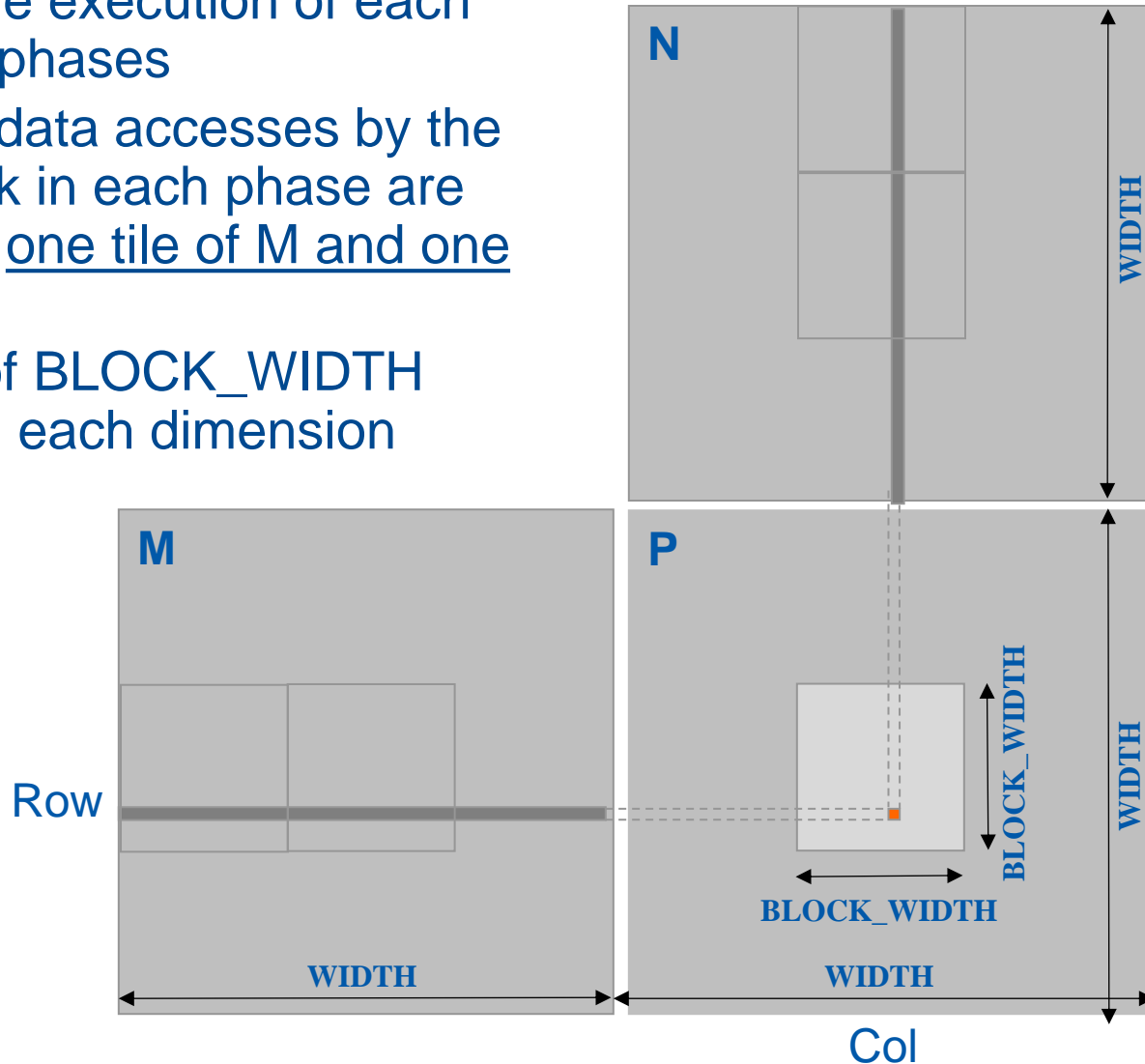
Tiled Matrix Multiplication

- Data access pattern
 - Each thread: a row of M and a column of N
 - Each thread block: a strip of M and a strip of N



Tiled Matrix Multiplication

- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of `BLOCK_WIDTH` elements in each dimension



Phase 0 Load for Block (0,0)

Global Memory

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

$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Shared Memory

Global Memory

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

Shared Memory

$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

Global Memory

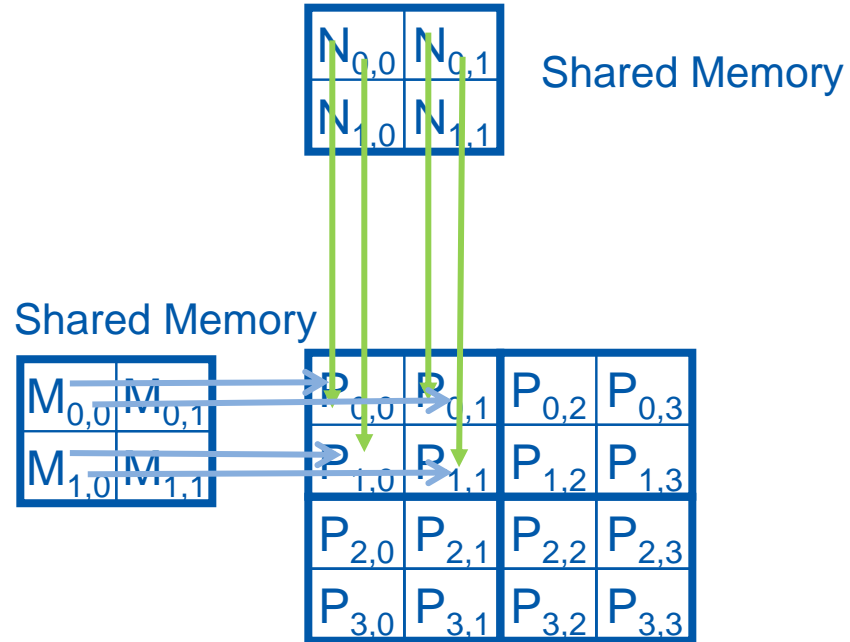
$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

2D Thread grid with 2D thread blocks, one thread per element of P

Phase 0 Use for Block (0,0) (iteration 0)

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

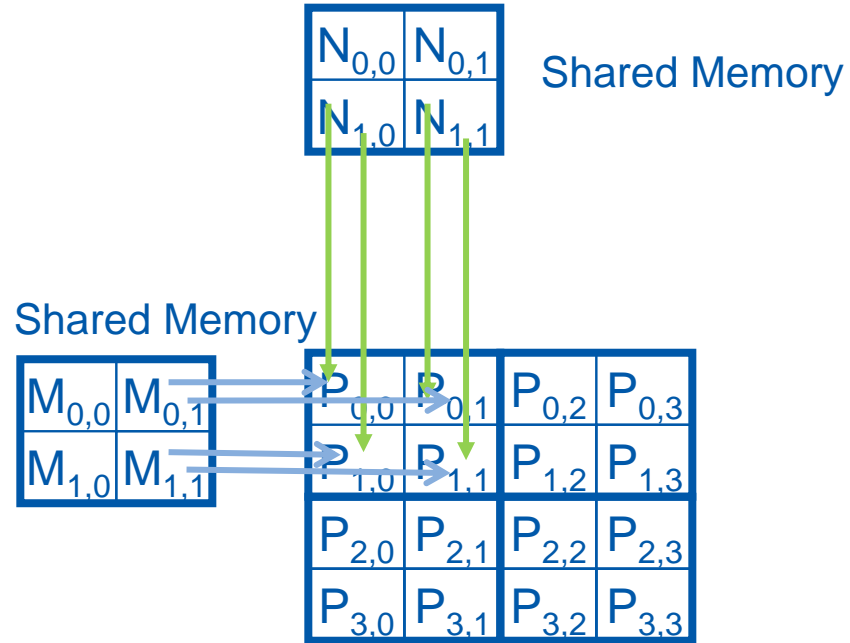
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



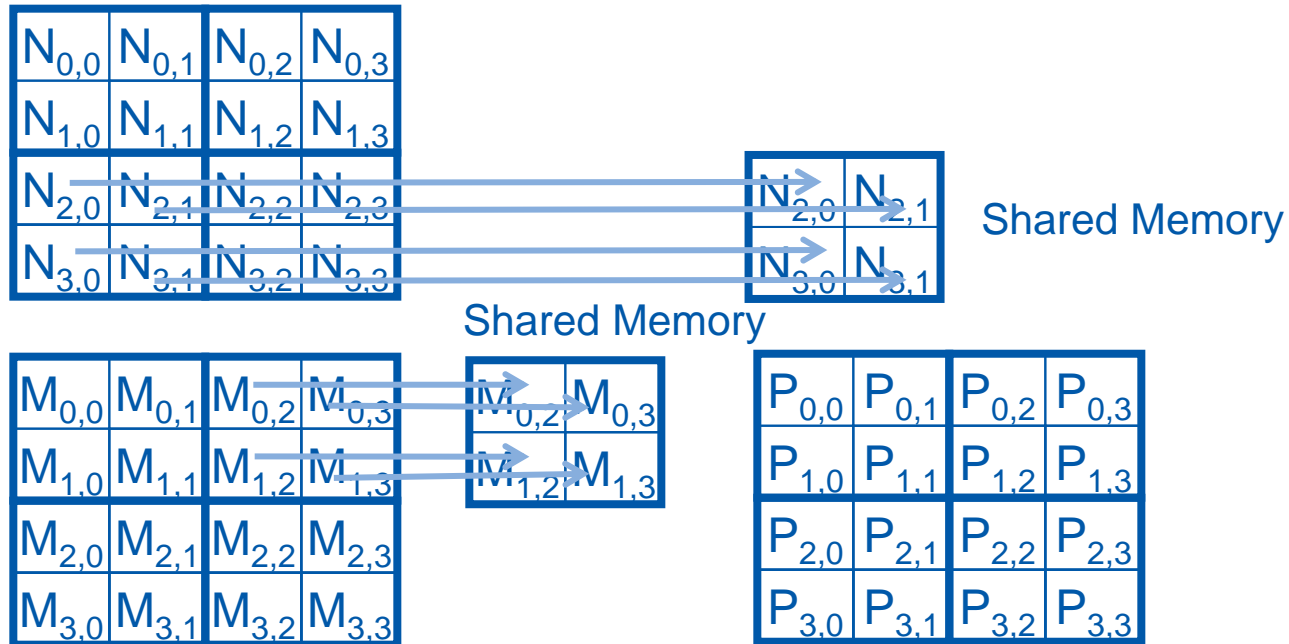
Phase 0 Use for Block (0,0) (iteration 1)

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

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



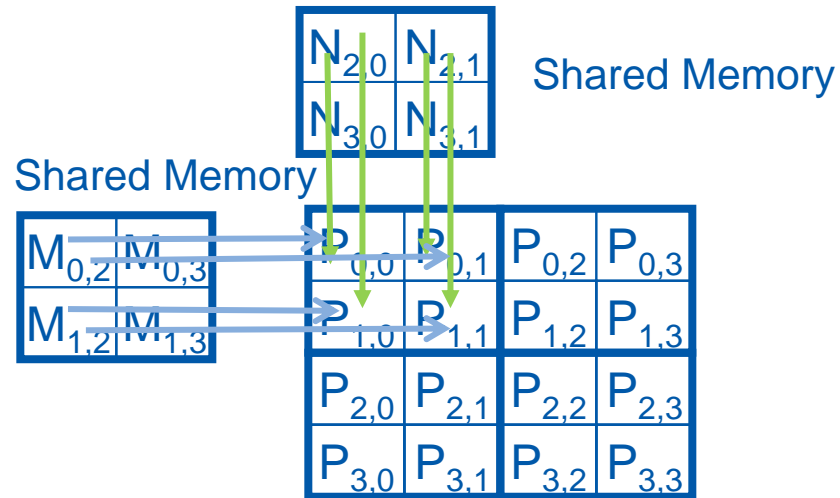
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)

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

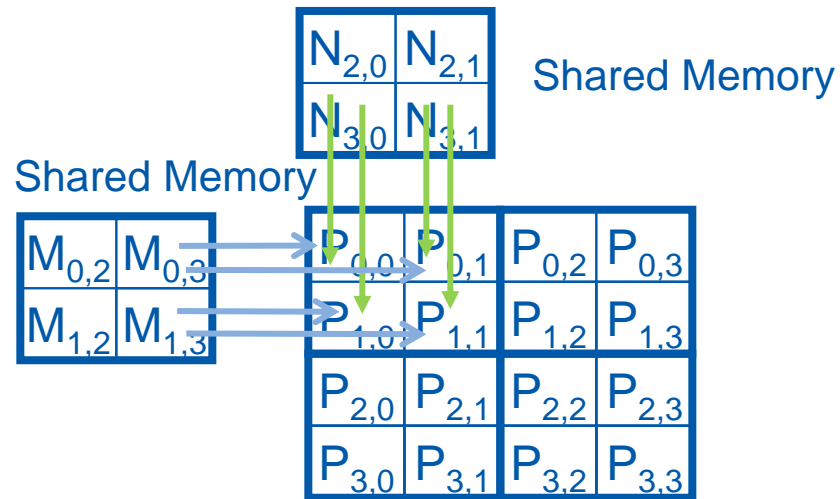
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Phase 1 Use for Block (0,0) (iteration 1)

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

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



Barrier Synchronization

- Synchronize all threads in a thread block
 __syncthreads()
- All threads in the same block must reach the __syncthreads() before any of the them can move on
 - Be careful with barriers inside if conditions
- Used to coordinate the phased execution of tiled algorithms
 - To ensure that all elements of a tile are loaded at the beginning of a phase
 - To ensure that all elements of a tile are consumed at the end of a phase

Tiled Matrix Multiplication Kernel

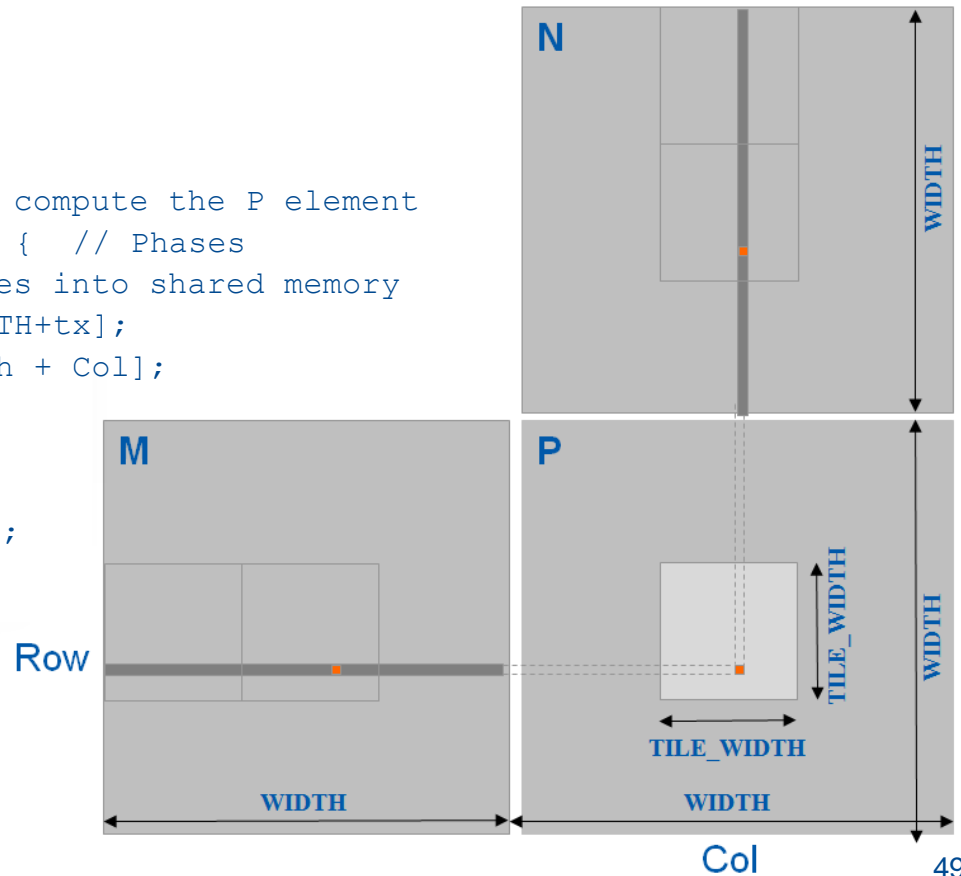
```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    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) { // Phases
        // 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[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i)
            Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```



Shared Memory and Threading

Shared memory size is variable across GPU models!

- For `TILE_WIDTH = 16`, each thread block uses $2*256*4B = 2KB$ of shared memory.
- For 16KB shared memory, one can potentially have up to 8 thread blocks executing
- `TILE_WIDTH 32` would lead to $2*32*32*4B = 8KB$ of shared memory usage per thread block, allowing 2 thread blocks active at the same time
 - However, the thread count limitation of 1536 threads per SM in current generation GPUs will reduce the number of blocks per SM to one!

There are hardware constraints on the size a thread block, too

- Maximum of 1024 threads per thread block
- Will see GPU limitations in the deviceQuery lab



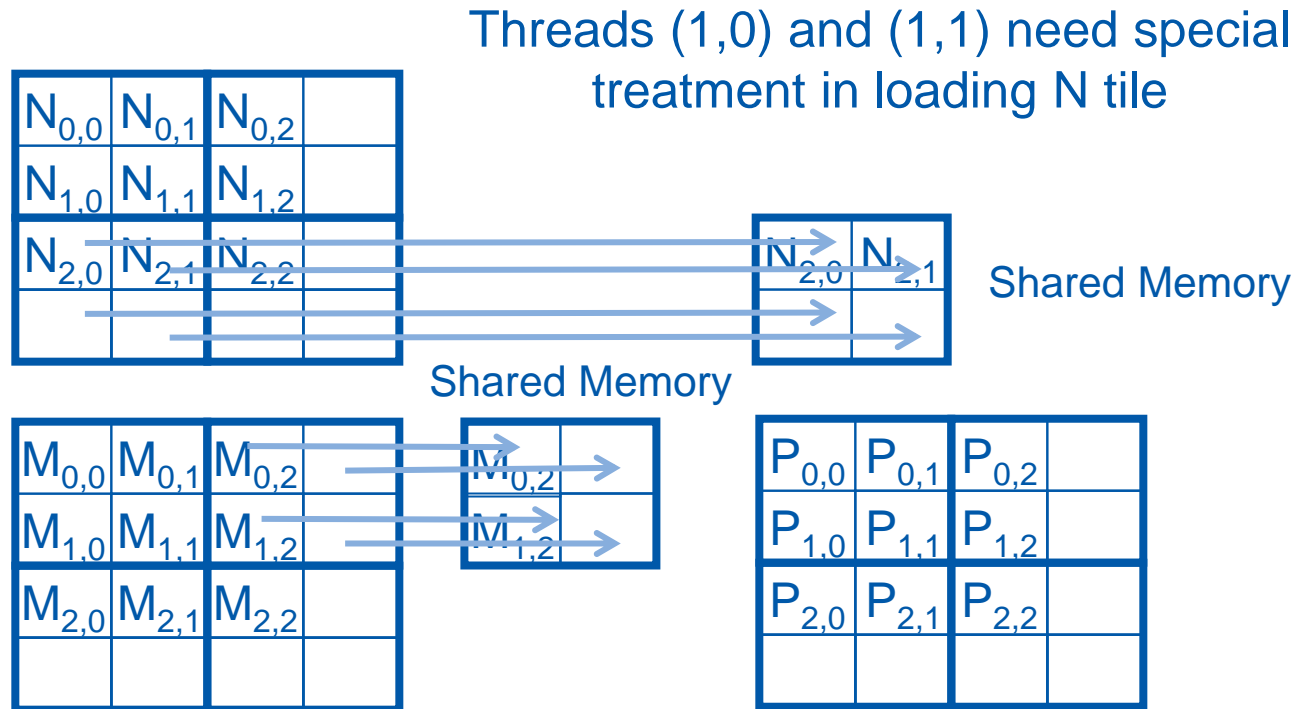
**Barcelona
Supercomputing
Center**
Centro Nacional de Supercomputación

HANDLING ARBITRARY MATRIX SIZES IN TILED ALGORITHMS

Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
 - However, real applications need to handle arbitrary sized matrices.
 - One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.

Phase 1 Loads for Block (0,0) for a 3x3 Example

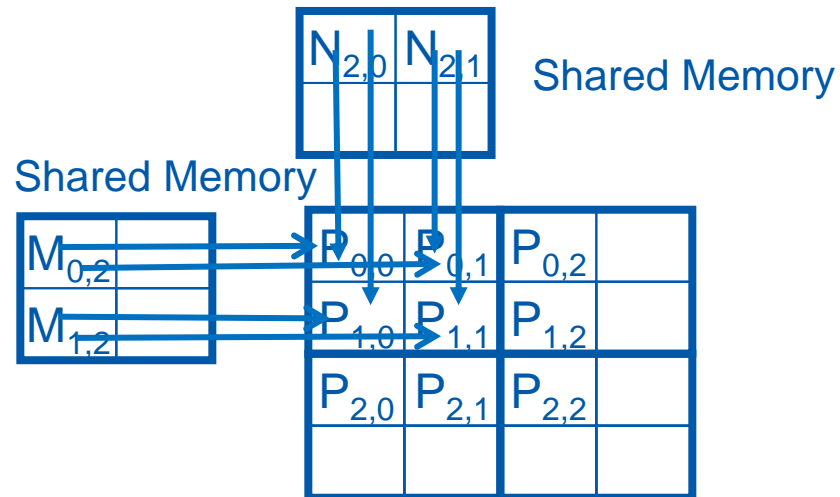


Threads (0,1) and (1,1) need special treatment in loading M tile

Phase 1 Use for Block (0,0) (iteration 0)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

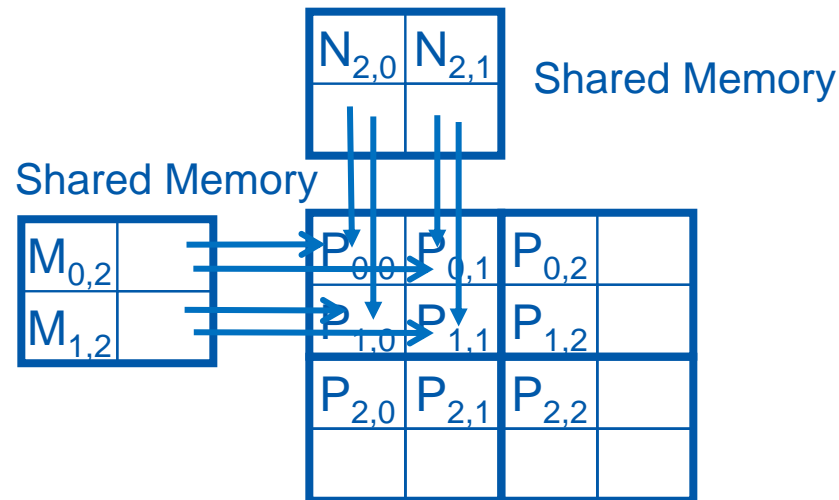
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



Phase 1 Use for Block (0,0) (iteration 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

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



All Threads need special treatment. None of them should introduce invalidate contributions to their P elements.

Major Cases in Toy Example

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
 - Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent $P[3,2]$ but need to participate in loading tile element $N[1,2]$
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
 - Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid $P[1,0]$ but attempts to load non-existing $N[3,0]$

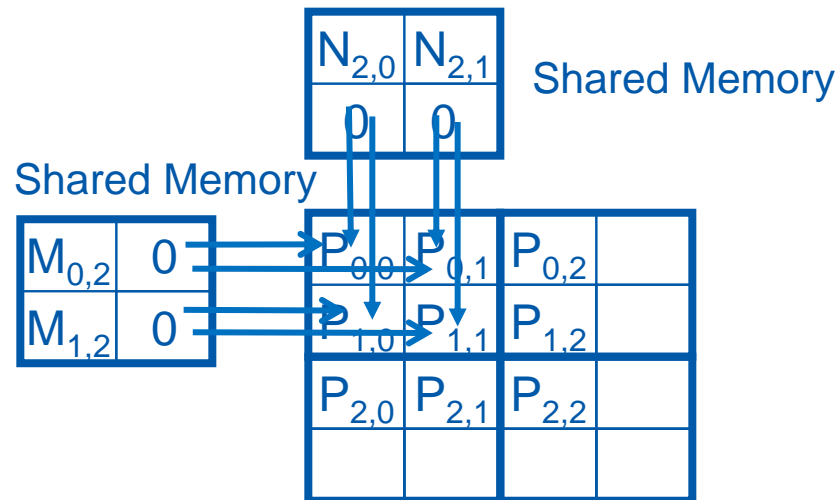
A “Simple” Solution

- When a thread is to load any input element, test if it is in the valid index range
 - If valid, proceed to load
 - Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for loading input elements is different from the test for calculating output P element
 - A thread that does not calculate valid P element can still participate in loading input tile elements
- For each thread the conditions are different for
 - Loading M element
 - Loading N element
 - Calculating and storing output elements

Phase 1 Use for Block (0,0) (iteration 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

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



Handling General Rectangular Matrices

- In general, the matrix multiplication is defined in terms of rectangular matrices
 - A $j \times k$ M matrix multiplied with a $k \times l$ N matrix results in a $j \times l$ P matrix
- We have presented square matrix multiplication, a special case
- The kernel function needs to be generalized to handle general rectangular matrices
 - The Width argument is replaced by three arguments: j , k , l
 - When Width is used to refer to the height of M or height of P , replace it with j
 - When Width is used to refer to the width of M or height of N , replace it with k
 - When Width is used to refer to the width of N or width of P , replace it with l



**Barcelona
Supercomputing
Center**

Centro Nacional de Supercomputación

QUIZ

Question 1

Assume that a kernel is launched with 1,000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?

- a) 1
- b) 1,000
- c) 512
- d) 512,000

Question 1 - Answer

Assume that a kernel is launched with 1000 thread blocks each of which has 512 threads. If a variable is declared as a shared memory variable, how many versions of the variable will be created through the lifetime of the execution of the kernel?

- a) 1
- b) 1,000**
- c) 512
- d) 512,000

Explanation: Shared memory variables are allocated to thread blocks. So, the number of versions is the number of thread blocks, 1,000.

Question 2

- ⌘ For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices A and B?
- a) 1/8 of the original usage
 - b) 1/16 of the original usage
 - c) 1/32 of the original usage
 - d) 1/64 of the original usage

Question 2 - Answer

- ⌘ For our tiled matrix-matrix multiplication kernel, if we use a 32x32 tile, what is the reduction of memory bandwidth usage for input matrices A and B?
- a) 1/8 of the original usage
 - b) 1/16 of the original usage
 - c) 1/32 of the original usage**
 - d) 1/64 of the original usage

Explanation: Each element in the tile is used 32 times

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