Performance Optimizations in CUDA

IOANNIS E. VENETIS

Assistant Professor
Department of Informatics
University of Piraeus, Greece

Achieving high performance on GPUs

Modern GPUs have the potential of achieving high performance

- ≈33.5 TFLOPS for Double Precision arithmetic (H100 and H200 GPUs)
- ≈90.5 TFLOPS for Single Precision arithmetic (L40 GPU)
- ≈989.4 TFLOPS for Half Precision arithmetic (H100 GPU)

Programming in a GPU hardware-agnostic manner will not unleash the full potential of GPUs

Programming for high performance on a GPU is not an easy task

- Good knowledge of the GPU hardware is required
- Good knowledge of possible performance bottlenecks is required
- Good knowledge of how to map programming constructs to the GPU hardware to avoid bottlenecks is required

Purpose of this presentation

Present the common bottlenecks in achieving high performance on GPUs

Present strategies to overcome these bottlenecks

Provide a starting point on where to look at, in case direct programming in CUDA and achieving high performance on the GPU is required

- Achieving high performance on the GPU does not always require direct programming in CUDA
- Highly optimized libraries are available that provide commonly required operations in a range of scientific domains
  - cuBLAS, cuSPARSE, CUDA Graphs, cuSOLVER, cuFFT, cuRAND, AmgX, …

Simple Matrix Multiplication

Simple Matrix Multiplication

We will use this algorithm to exploit usage of shared memory

- One of the strategies to improve performance in programs that execute on the GPU

Preliminaries

A common strategy when programming for the GPU is to assign calculations for each element of the result to a single CUDA thread

Matrix multiplication

- Each thread performs calculations for one element of the output matrix

How to assign elements of the output matrix to threads?

Correlating threads to matrix elements

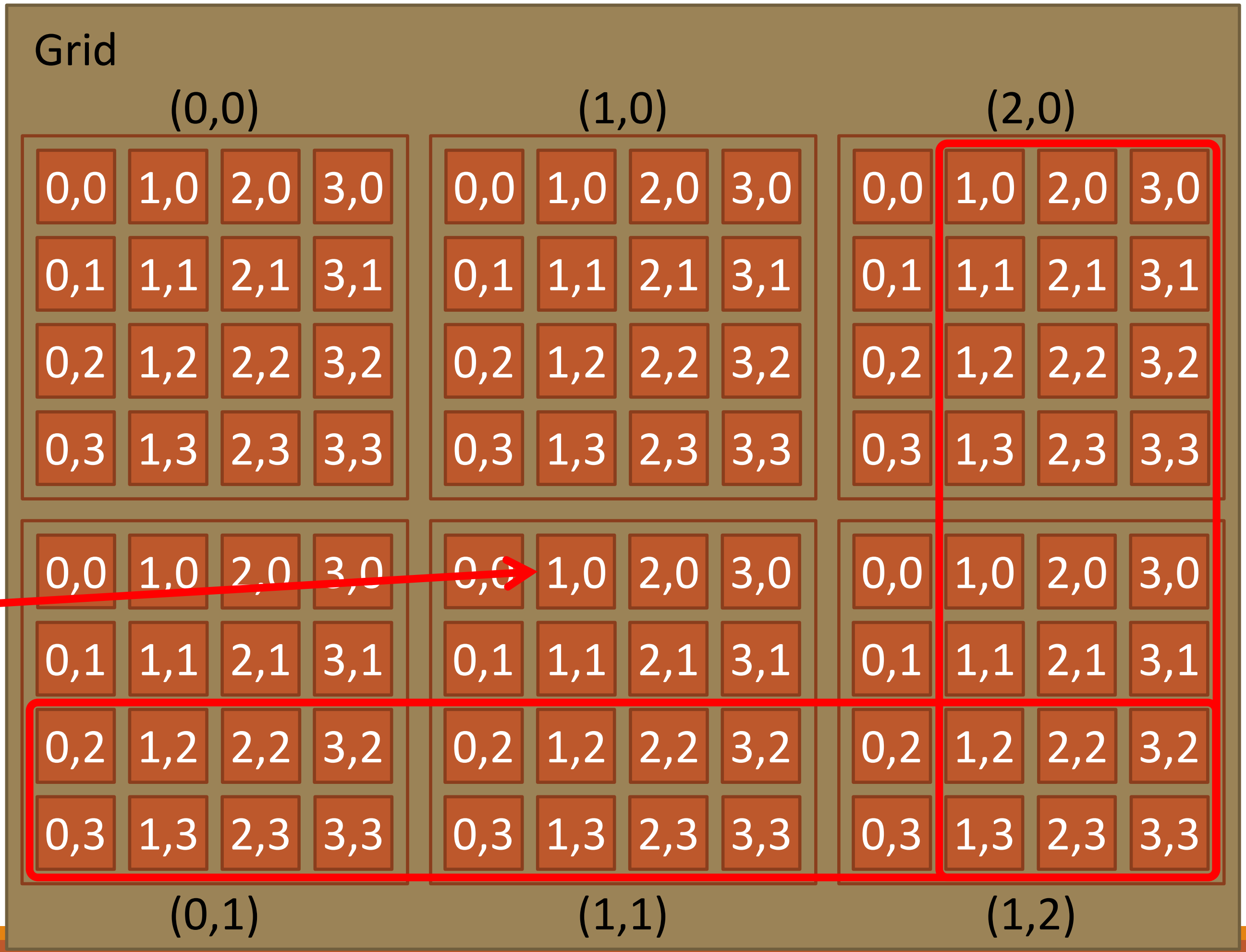
We use blocks of size 4×4

Matrix is 6×9

Matrix

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8

```
x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y
```



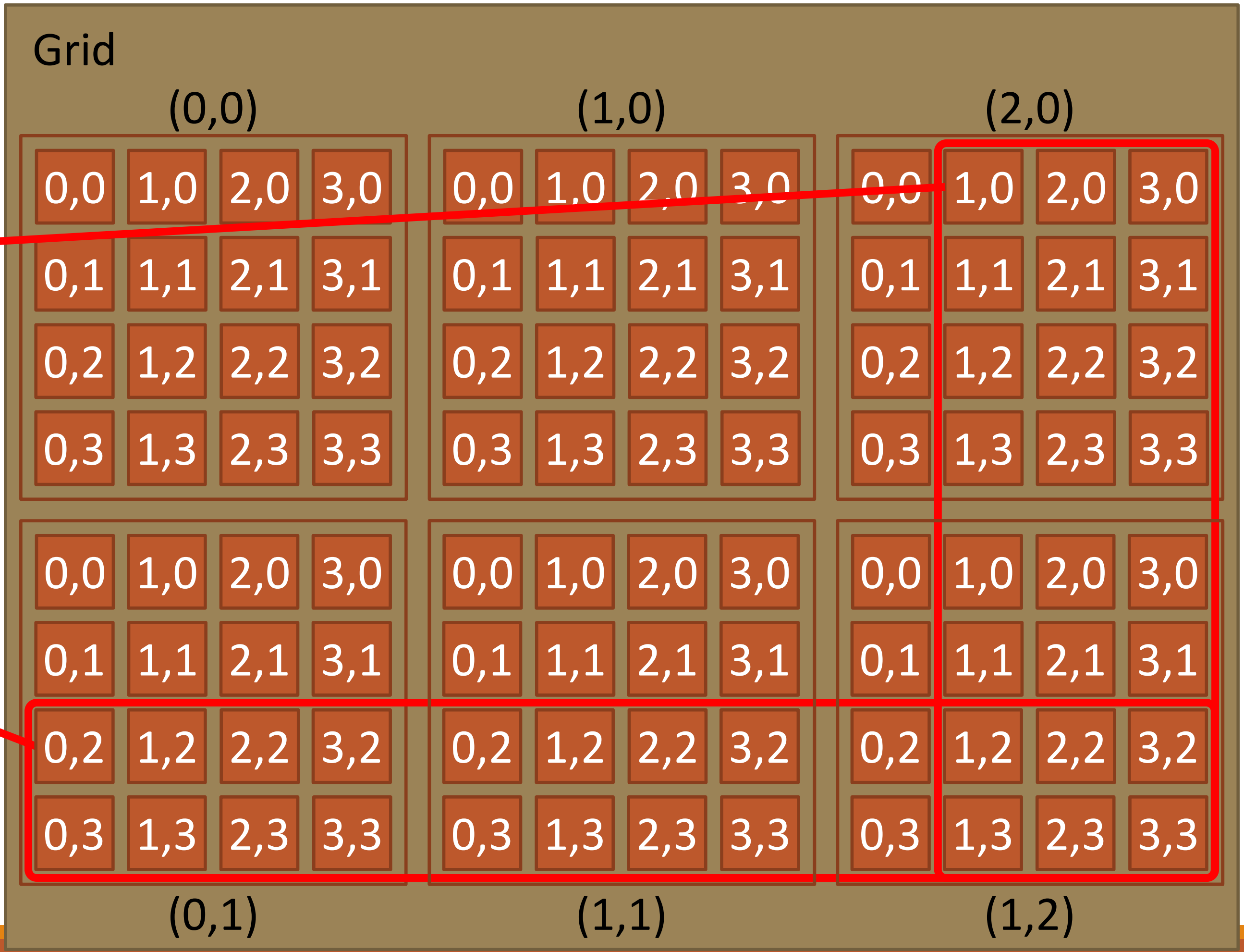
Threads exceeding matrix limits

We use blocks of size 4×4

Matrix is 6×9

```
x =blockIdx.x * blockDim.x +
    threadIdx.x = 2 * 4 + 1 = 9
```

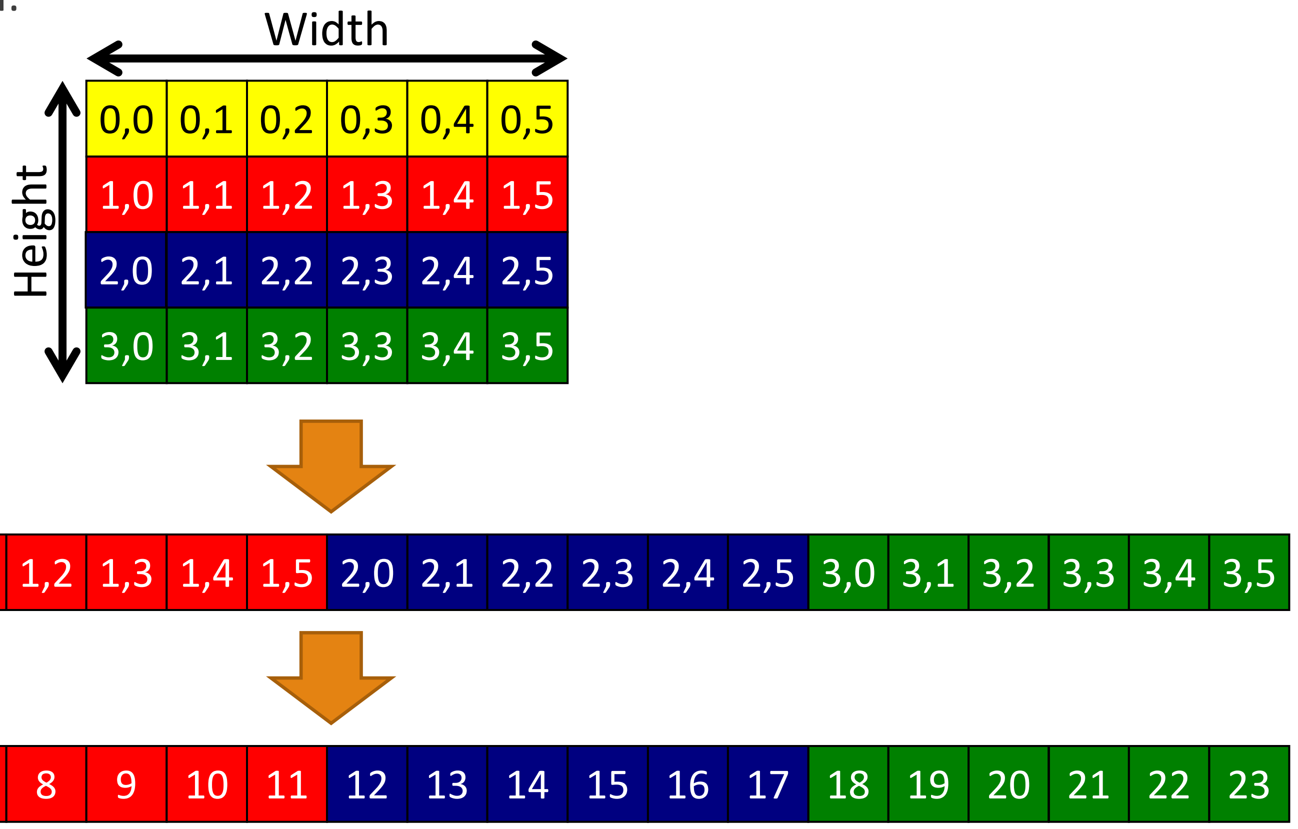
```
y =blockIdx.y * blockDim.y +
    threadIdx.y = 1 * 4 + 2 = 6
```



Row-major representation of matrices in C/C++

2-D matrices are stored in memory row-after-row

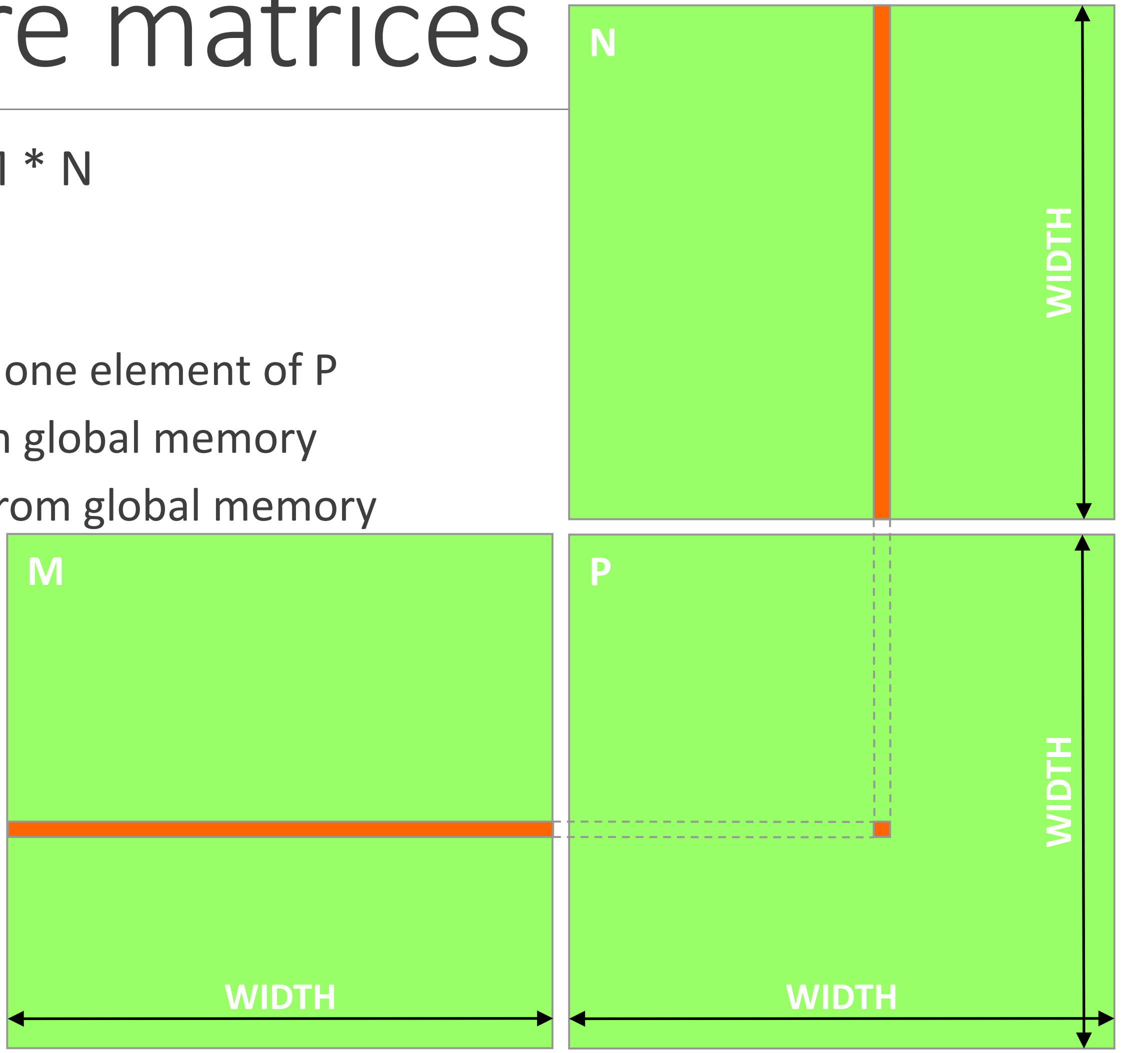
- Position in this 1-D representation:
  - Row * Width + Column



Multiplication of square matrices

Assume the multiplication of square matrices: P = M * N

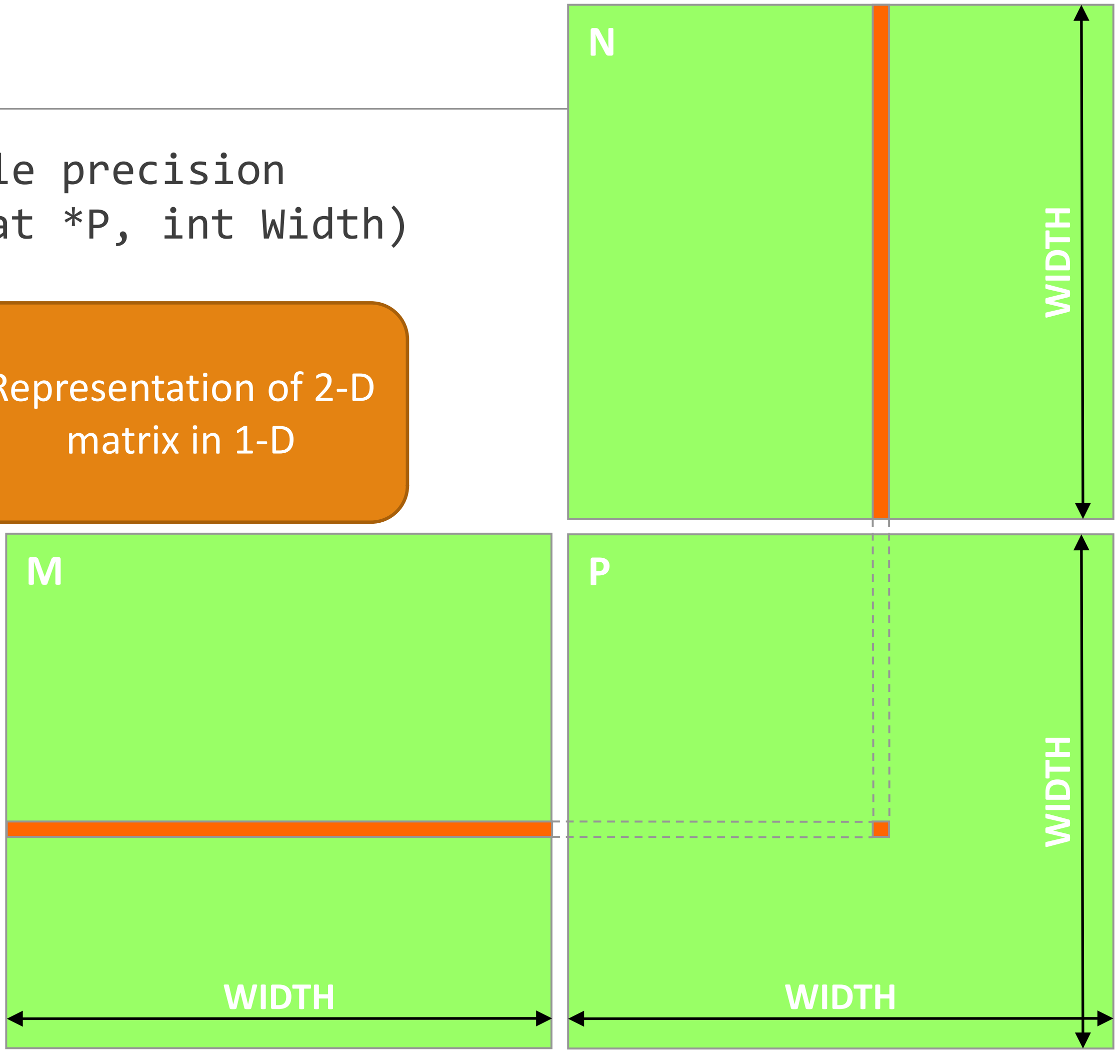
- Matrices assumed square for simplicity in the example
- Size of matrices is WIDTH x WIDTH
- Our strategy will be for each CUDA thread to calculate one element of P
- Notice that each row of M is loaded WIDTH times from global memory
- Notice that each column of N is loaded WIDTH times from global memory



2-D matrix multiplication CPU code

```
// Matrix multiplication on the host in single precision
void MatrixMulOnHost(float *M, float *N, float *P, int Width)
{
  for (int i = 0; i < Width; i++) {
    for (int j = 0; j < Width; j++) {
      double Pvalue = 0.0;
      for (int k = 0; k < Width; k++) {
        double a = M[i * Width + k];
        double b = N[k * Width + j];
        Pvalue += a * b;
      }
      P[i * Width + j] = Pvalue;
    }
  }
}
```

Representation of 2-D matrix in 1-D

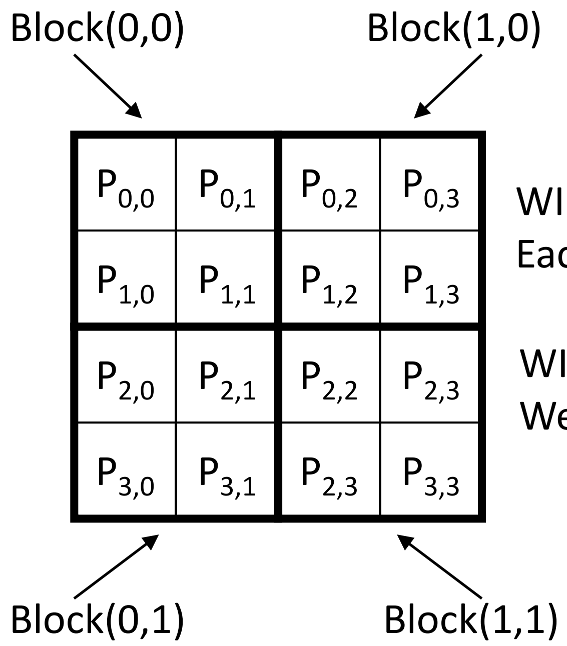


How to organize calculations

We will create blocks of threads of size TILE_WIDTH × TILE_WIDTH

Each block will calculate a tile of size TILE_WIDTH × TILE_WIDTH of the result matrix P

The 2-D grid must have size (WIDTH/TILE_WIDTH) × (WIDTH/TILE_WIDTH)


Block(0,0)
Block(1,0)
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{2,3}$ $P_{3,3}$
Block(0,1)
Block(1,1)

WIDTH = 4;   TILE_WIDTH = 2;
Each block has 2 * 2 = 4 threads

WIDTH / TILE_WIDTH = 2;
We will use 2 * 2 = 4 blocks

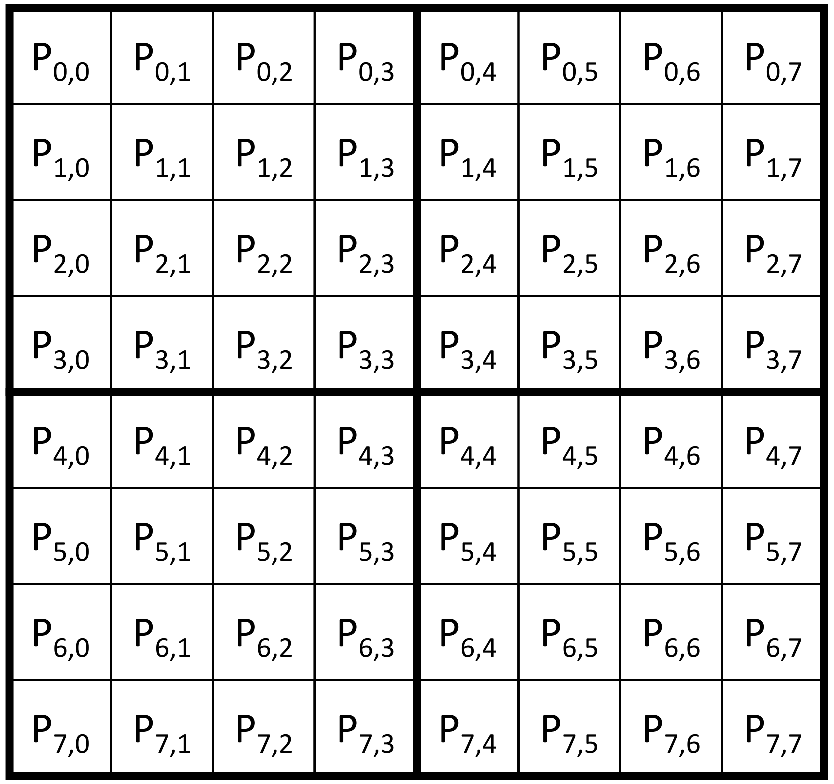
A larger example

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8;   TILE_WIDTH = 2;
Each block has 2 * 2 = 4 threads

WIDTH / TILE_WIDTH = 4;
We will use 4 * 4 = 16 blocks

Using a different block size

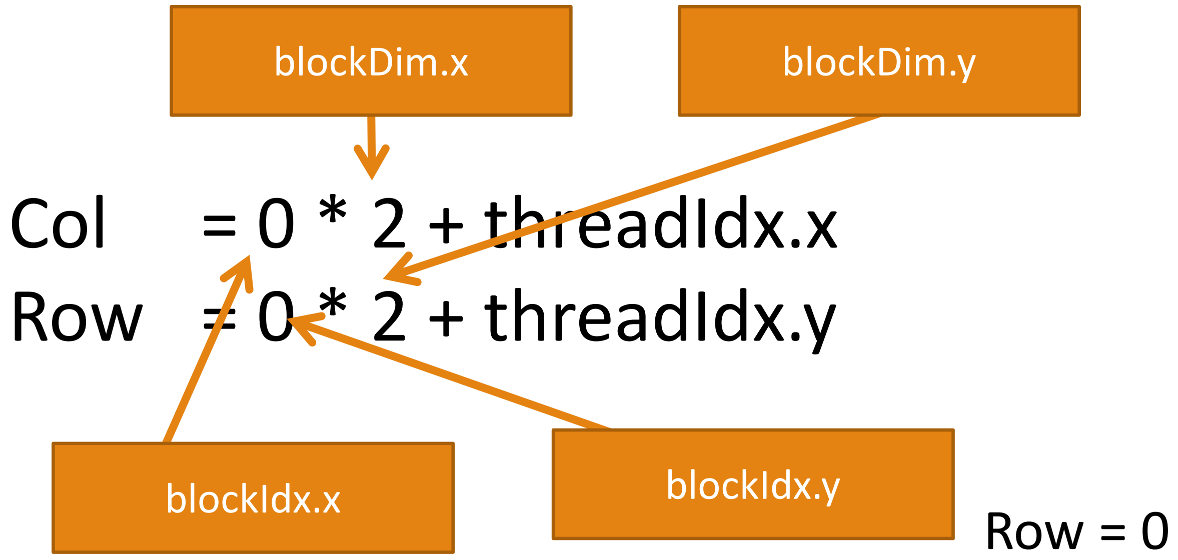


$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8;   TILE_WIDTH = 4;
Each block has 4 * 4 = 16 threads

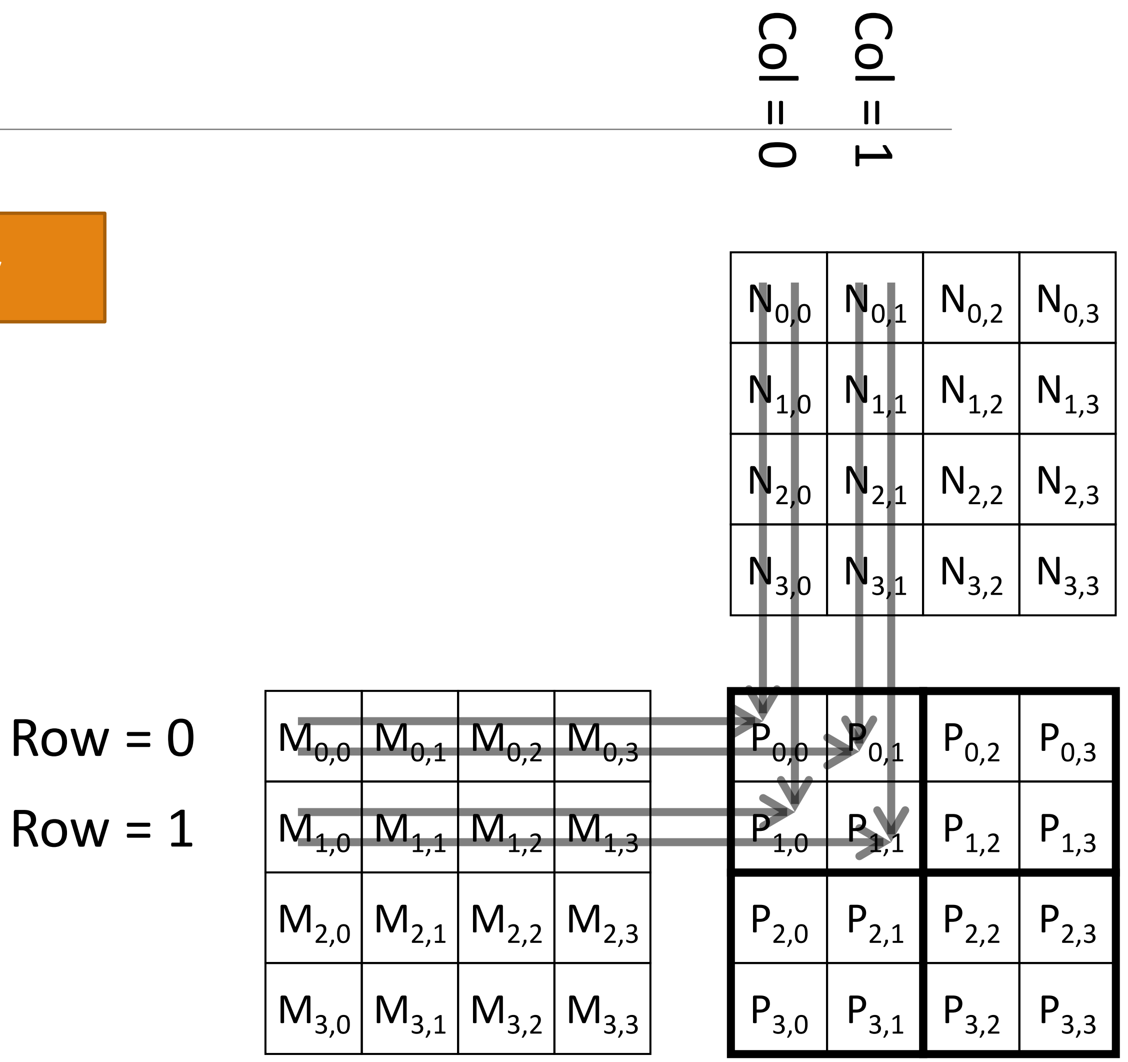
WIDTH / TILE_WIDTH = 2;
We will use 2 * 2 = 4 blocks

Calculations on block (0, 0) for TILE_WIDTH = 2

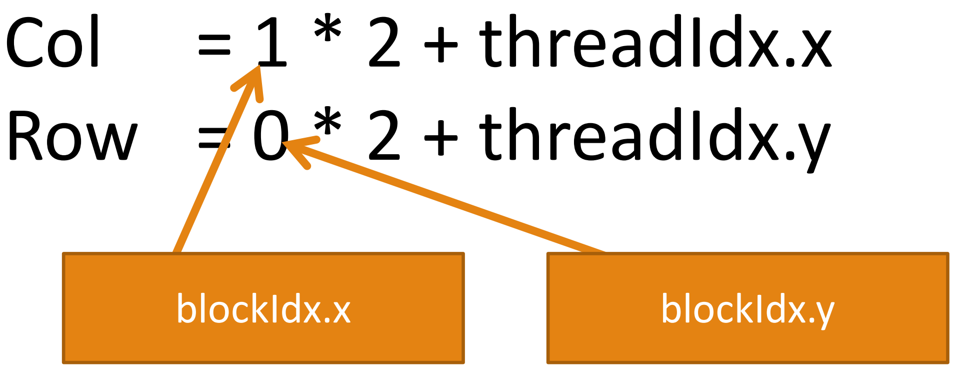


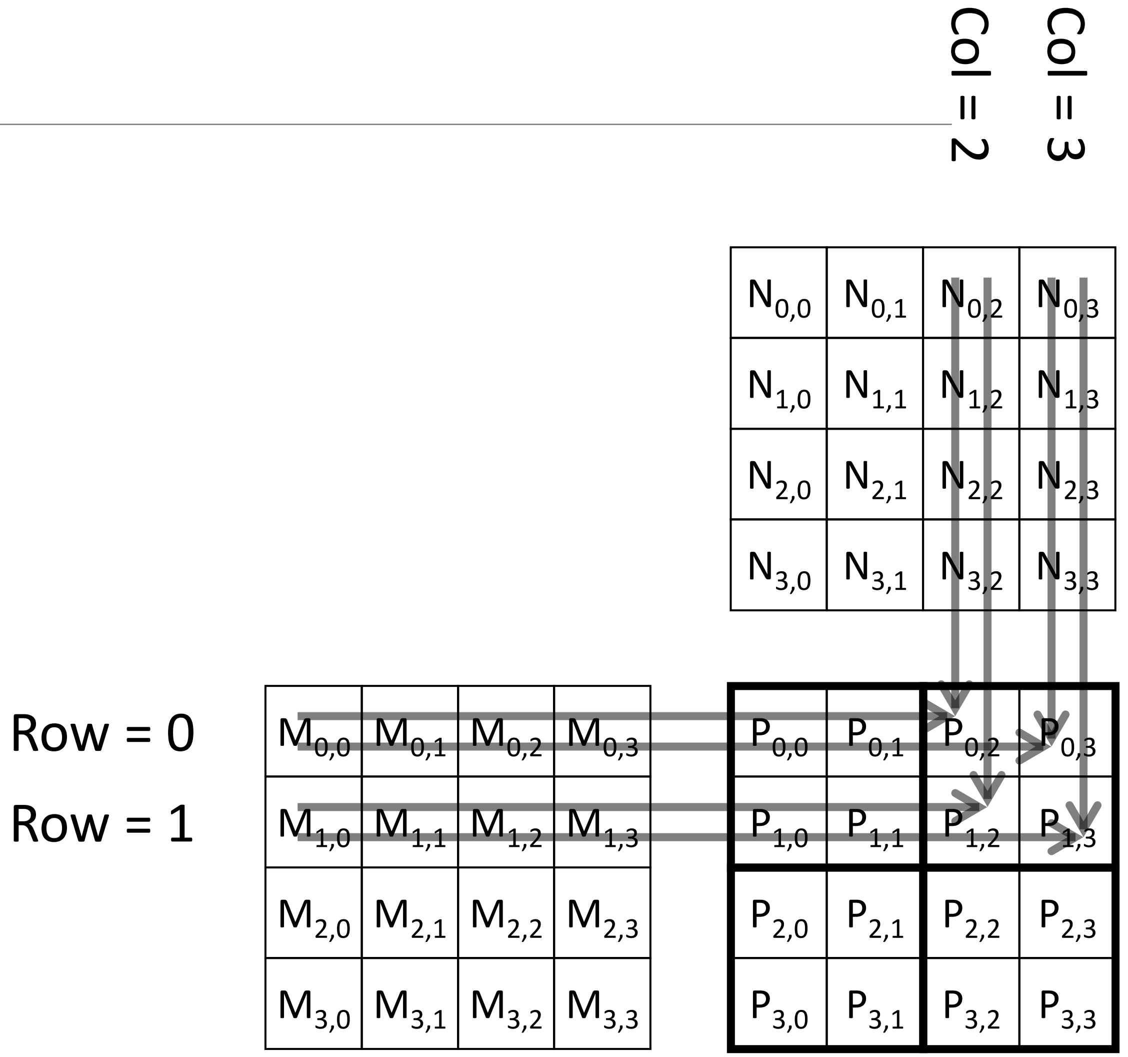
Col = 0 * 2 + threadIdx.x

Row = 0 * 2 + threadIdx.y


Col = 0
Col = 1
Row = 0
Row = 1
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Calculations on block (0, 1) for TILE_WIDTH = 2





Col = 2
Col = 3
Row = 0
Row = 1
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

A first, simple computational kernel for 2-D matrix multiplication

```
__global__ void MatrixMulKernel(float *M_d, float *N_d, float *P_d, int Width)
{
  // Calculate the row index of the P_d element and M_d
  int Row = blockIdx.y * blockDim.y + threadIdx.y;

  // Calculate the column index of P_d and N_d
  int Col = blockIdx.x * blockDim.x + threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0.0;
    // Each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; k++) {
      Pvalue += M_d[Row * Width + k] * N_d[k * Width + Col];
    }
    P_d[Row * Width + Col] = Pvalue;
  }
}
```

Shared memory

Purpose

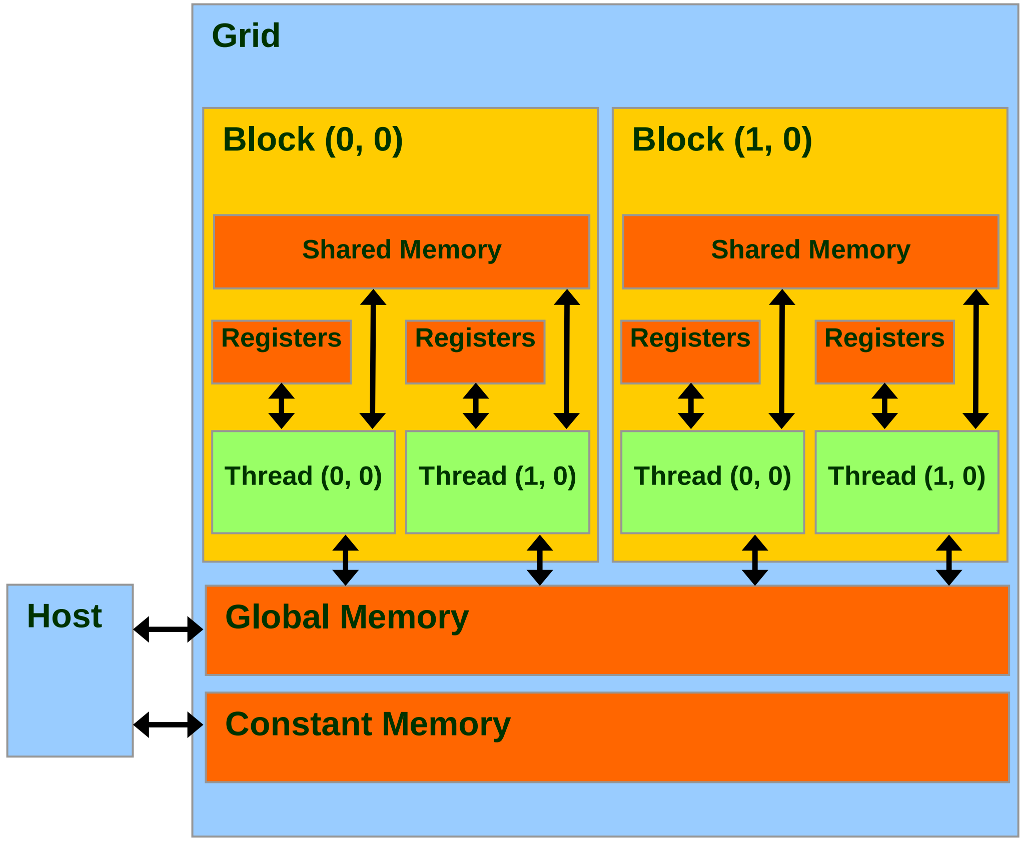
Understand how to use the memory hierarchy in CUDA

- Registers, shared memory, global memory
- Algorithms using “tiles”
- How to use barriers

Memory hierarchy from the programmers point of view

Each thread:

- Reads/Writes to registers that belong to it (per thread registers) (~1 cycle)
- Reads/Writes to shared memory that belongs to a block (per block shared memory) (~30-50 cycles)
- Reads/Writes to global memory that belongs to the grid (per grid global memory) (~80-2750 cycles)
- Read from constant memory that belongs to the grid (per grid constant memory) (~1-7 cycles if data is in cache)



Shared memory in CUDA

Special type of memory

- Its use must be explicitly specified in the source code
- Resides in each Streaming Multiprocessor (SM)
- Very fast access, compared to global memory
- Accessed in the same way that global memory is accessed

Data type specifiers in CUDA

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

__device__ is optional when __shared__ or __constant__ is used

Automatic variables reside in registers

- Except arrays, which reside in global memory

Declaring variables that reside in shared memory

```
__global__ void MatrixMulKernel(float *M_d, float *N_d, float *P_d, int Width)
{
  __shared__ float M_ds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float N_ds[TILE_WIDTH][TILE_WIDTH];
```

Programming strategy

DRAM memory modules are used to implement global memory– slow access

Better strategy: Divide input data into smaller parts (tiles) in order to exploit shared memory

- Decide how input data will be divided into tiles, so that each tile fits into shared memory
- Handle each tile with a block of threads:
  - Copy tile from global to shared memory
    - Use multiple threads to exploit parallelism at the memory level
  - Perform computations on the tile that resides in shared memory
    - Each thread can (and must!) use multiple times each copied data element
  - Copy results from the shared to the global memory

When not to use shared memory

Assume that:

- N data elements must be read from memory
- Each element will be used only 1 time
- Access time per element
  - If in global memory: t_g
  - If in shared memory: $t_s << t_g$

Total time to access elements

- From global memory
  - $T_g = N \cdot t_g$
- From shared memory
  - $T_s = N \cdot t_g + N \cdot t_s > T_g$

More time is required!

When to use shared memory

Assume now that:

- N elements must be read from memory
- Each element will be used $K > 1$ times

Total time to access elements

- From global memory
  - $T_g = N \cdot K \cdot t_g$
- From shared memory
  - $T_s = N \cdot t_g + N \cdot (K-1) \cdot t_s$

But because $t_s << t_g$ there is a huge gain!

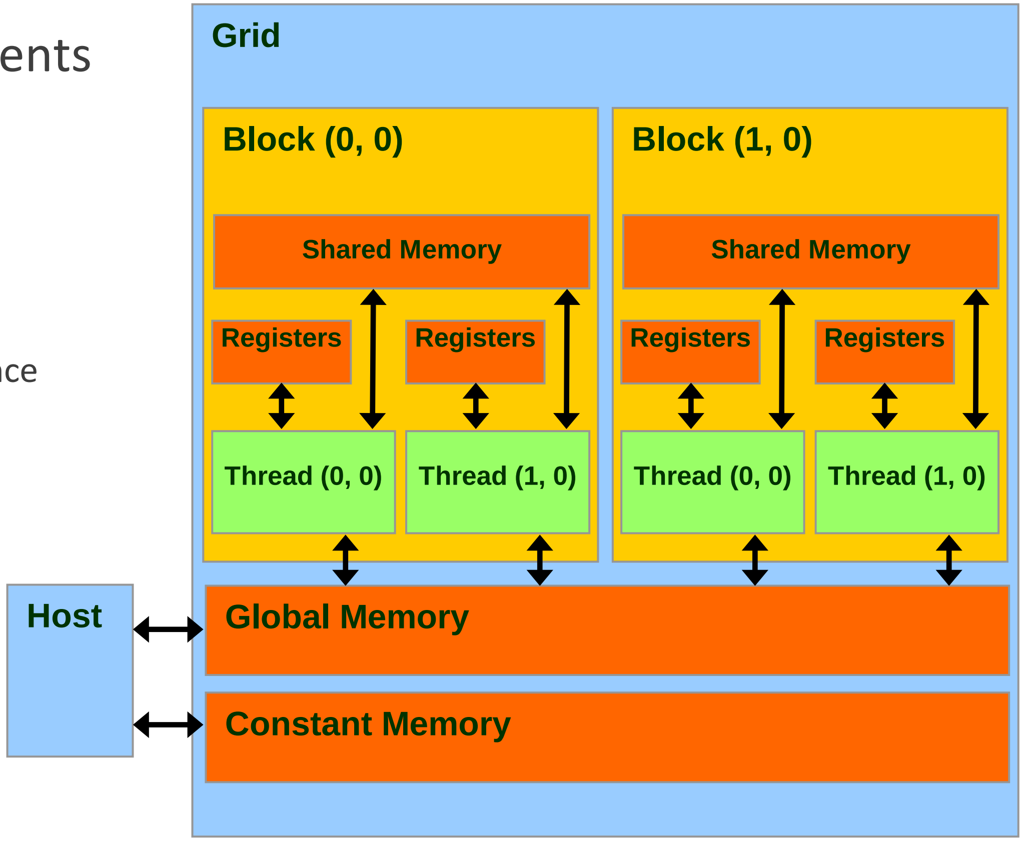
2-D matrix multiplication using shared memory

Performance issues on the Fermi architecture

All threads access global memory to read input matrix elements

- 2 accesses (8 bytes) for one multiplication and one addition of single precision numbers
- 4 Bytes of memory transfers / FLOP (Floating-Point Operation)
- Max. performance of the architecture is 1000 GFLOPS
  - 4*1000 = 4000 GB/sec memory bandwidth required to achieve max. performance
- But Fermi provides 150 GB/s
  - 150 / 4 = 37.5 GFLOPS can be achieved
  - Real code achieves only about 25 GFLOPS

Accesses to global memory have to be reduced drastically to get close to the max. of 1000 GFLOPS



Overview of the technique

Repeat

- Find a tile in global memory that is accessed by multiple threads
- Copy the tile from global to shared memory
- Make the threads access the data they need from shared memory

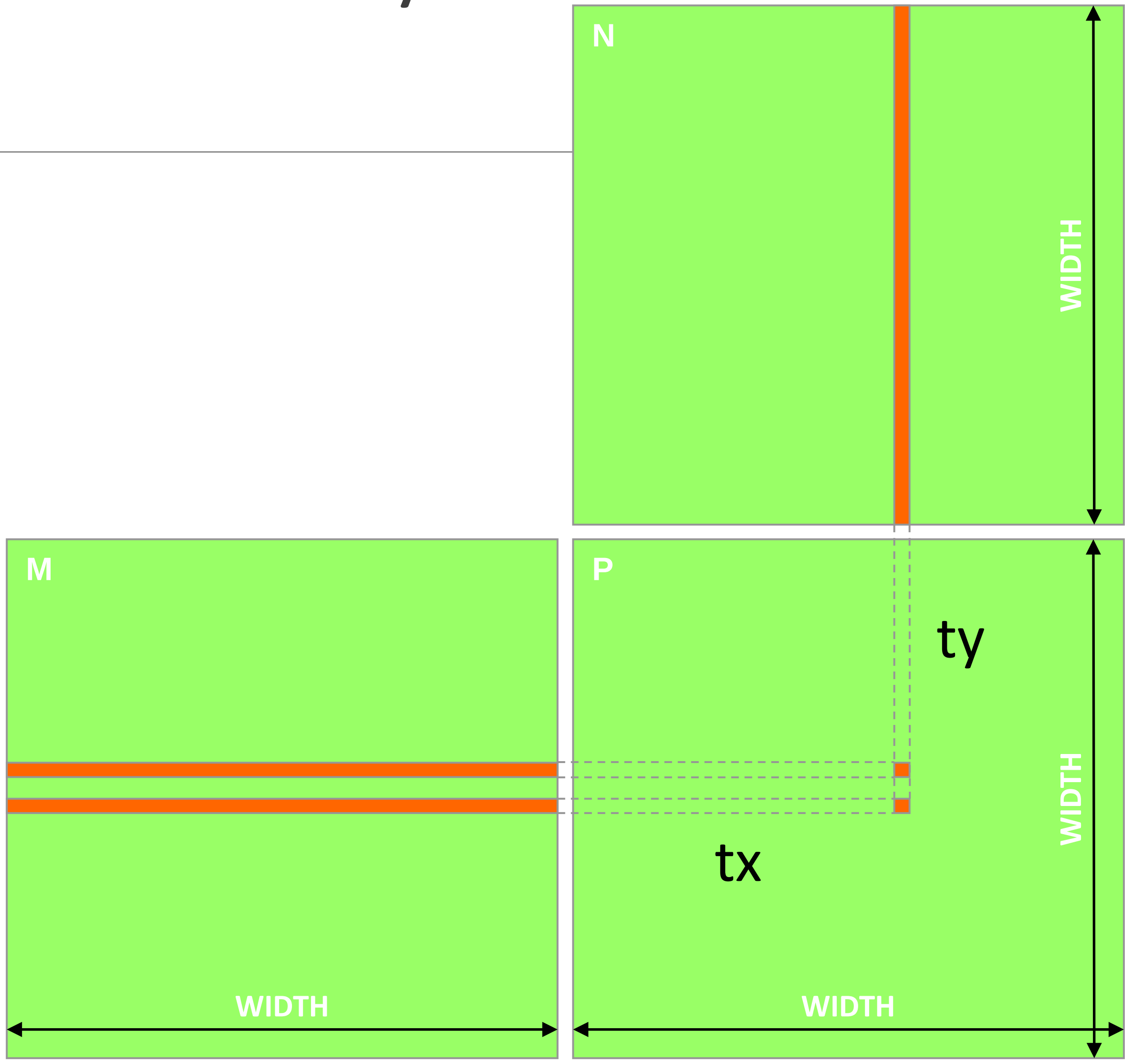
while more tiles are available

Main idea: Use shared memory to reuse data

Every input element is read from WIDTH threads

Copy each element into shared memory

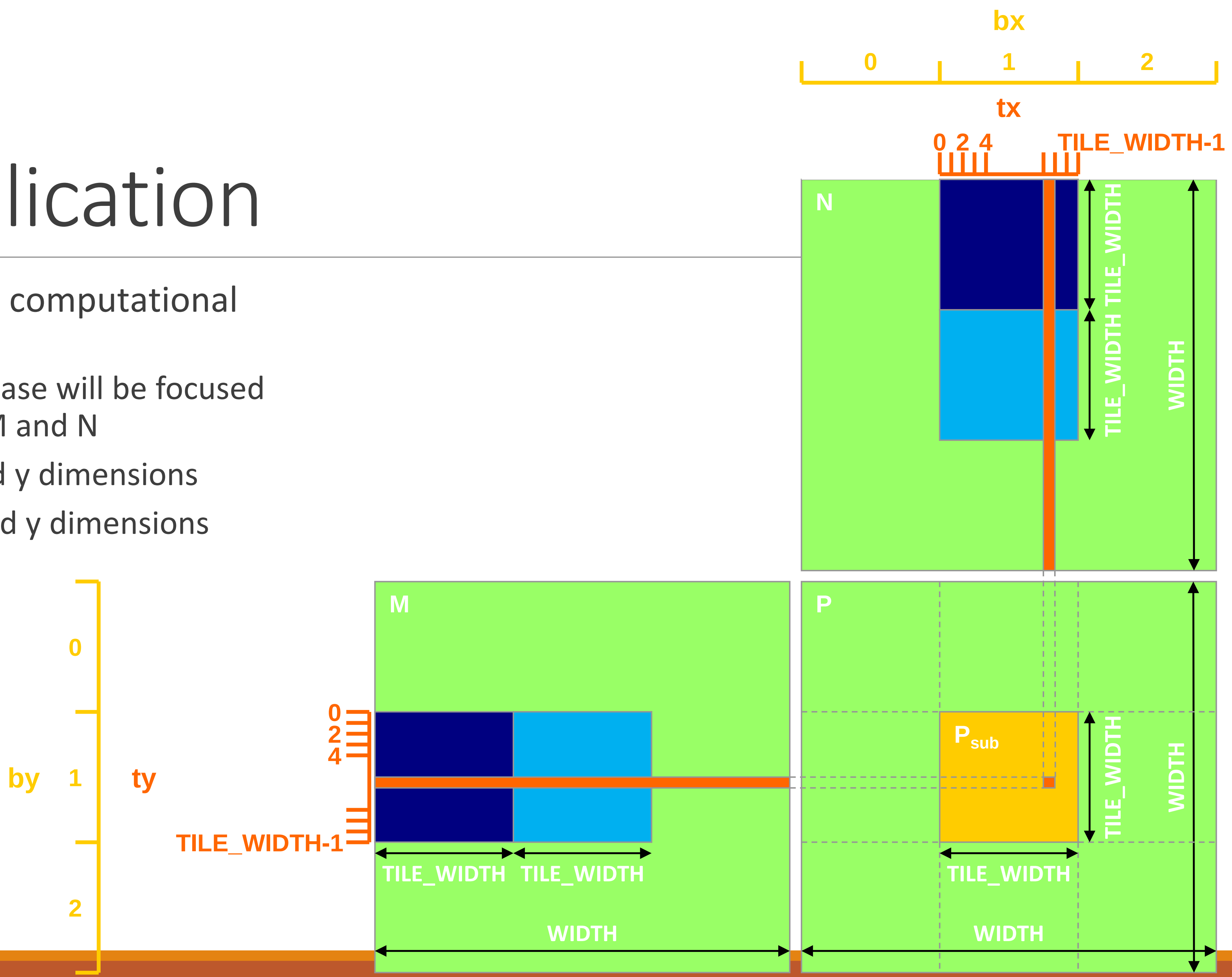
- Multiple threads will read it from there
- Reduces required bandwidth to global memory
  - Tiled algorithms



Tiled multiplication

Divide the execution of the computational kernel into phases

- Data access during each phase will be focused on a single tile of each of M and N
- bx, by: Block index on x and y dimensions
- tx, ty: Thread index on x and y dimensions



First step: Copy tile to shared memory

All threads of the block will participate

- Each thread will copy a single element from M_d and a single element from N_d

Try to make memory accesses to the global memory coalesced within a warp during the copy

- Better bandwidth
- More in a while

Second step: Perform partial multiplication

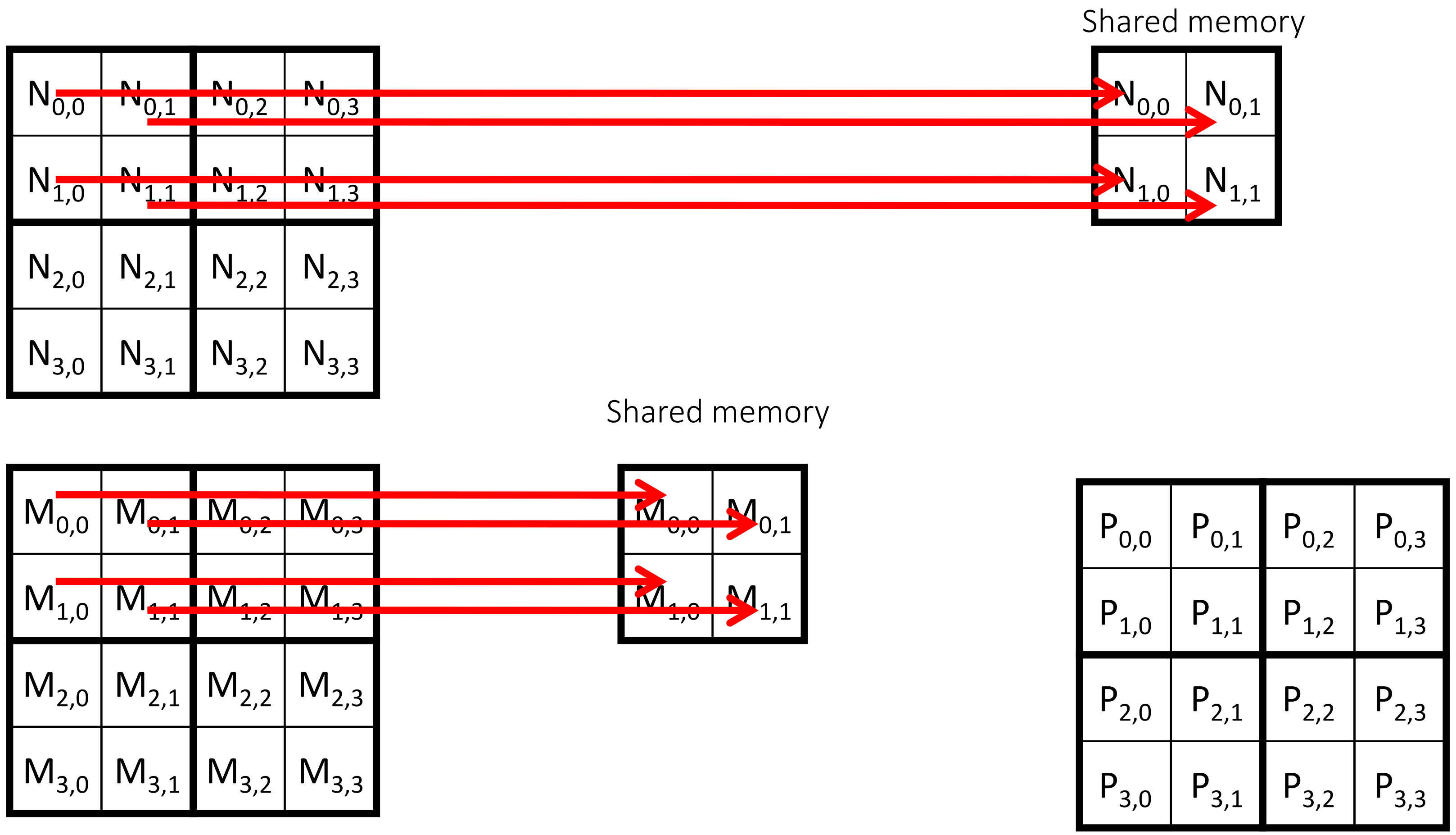
Using the elements copied into shared memory perform as many operations as possible towards the final result

- In the case of matrix multiplication, to calculate partially each element within the current block of threads

Third step: Write back final result

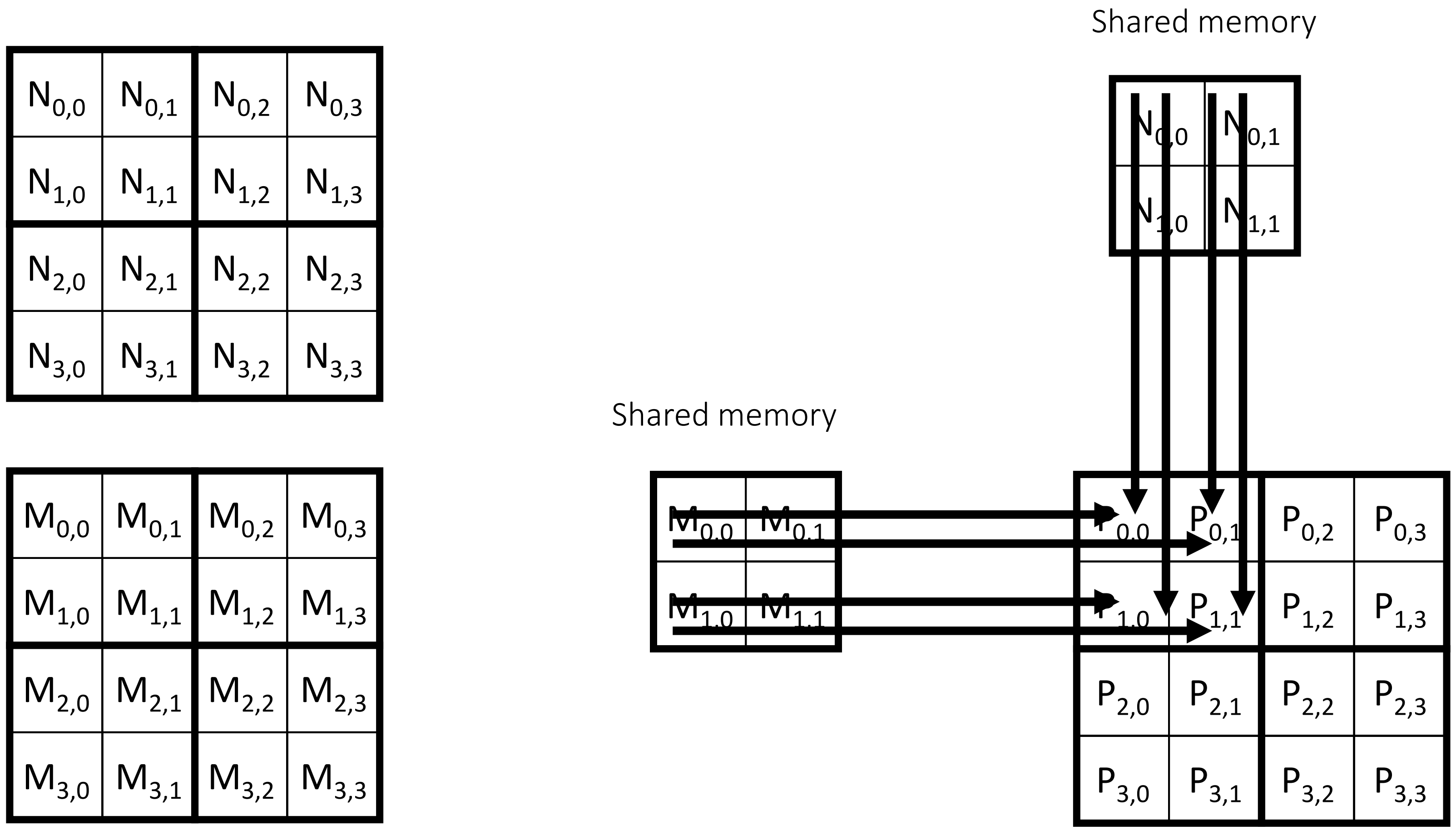
After a block of threads passes over all tiles, the final result is copied to global memory

Processing block (0,0)

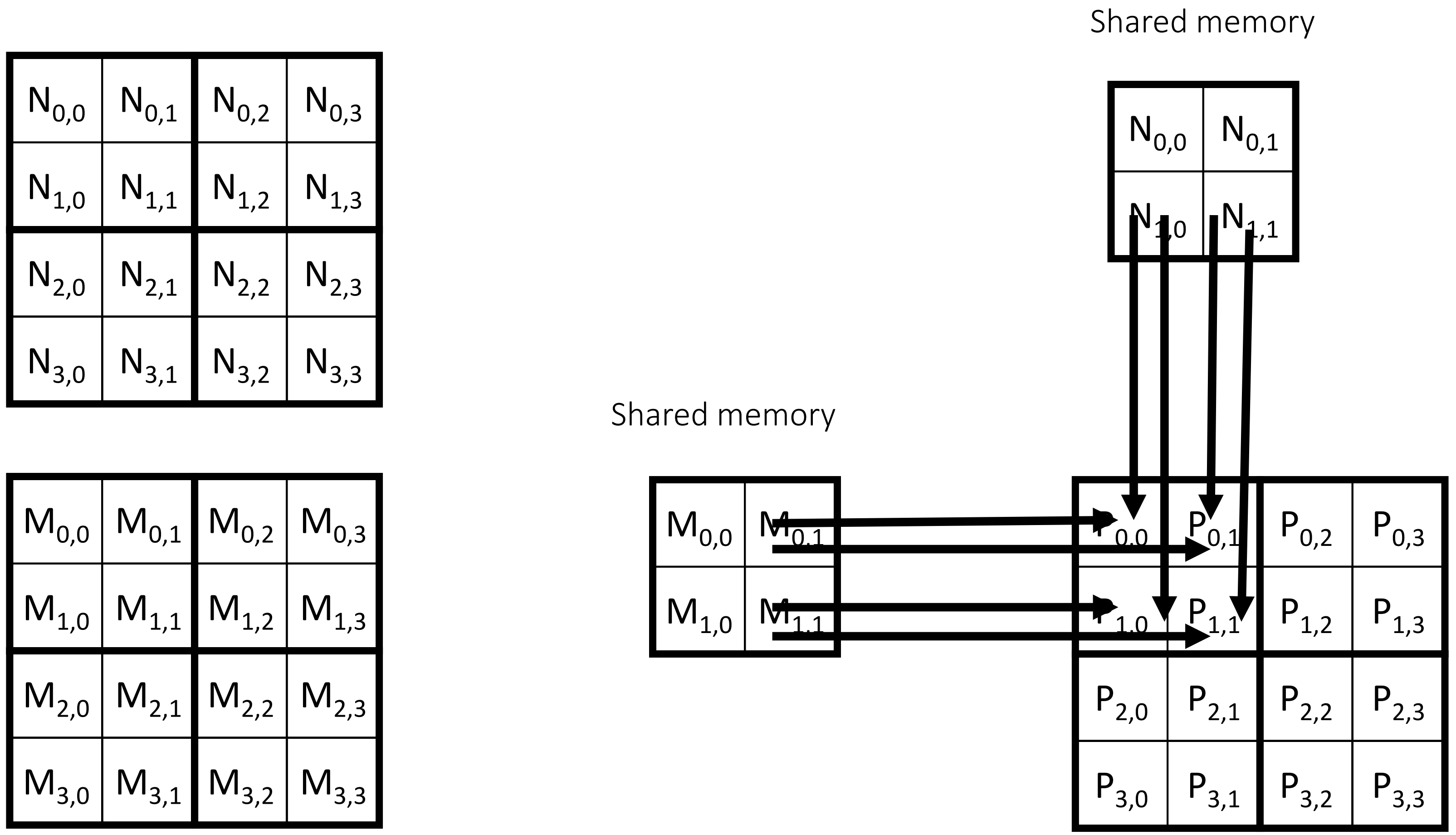


Shared memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared memory
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$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

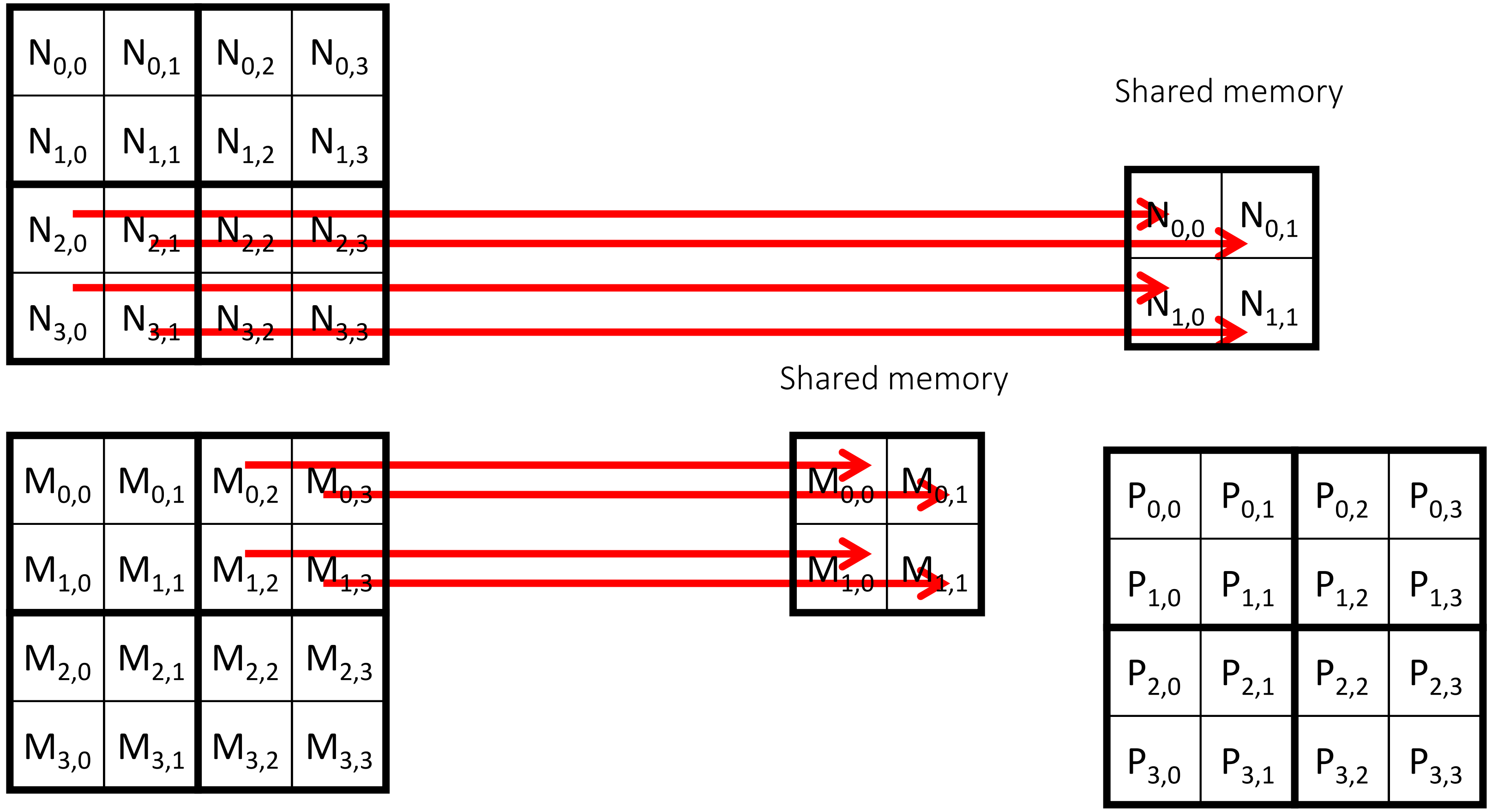
Processing block (0,0)


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

Processing block (0,0)

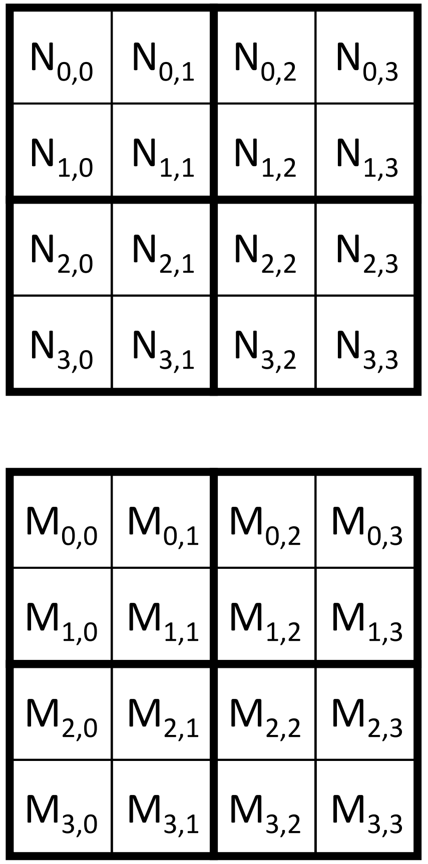

Shared memory
Shared memory
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$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

Processing block (0,0)

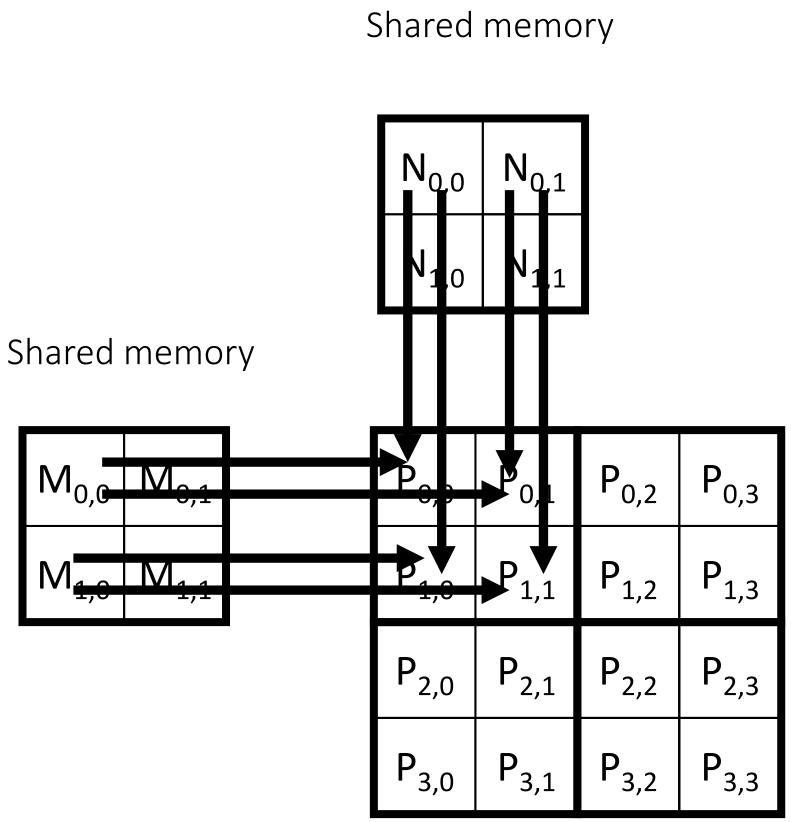


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Shared memory
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$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

Processing block (0,0)



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



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

Required synchronization construct: Barrier

CUDA API function call

- __syncthreads()

All threads of a block must call __syncthreads() before they can continue execution

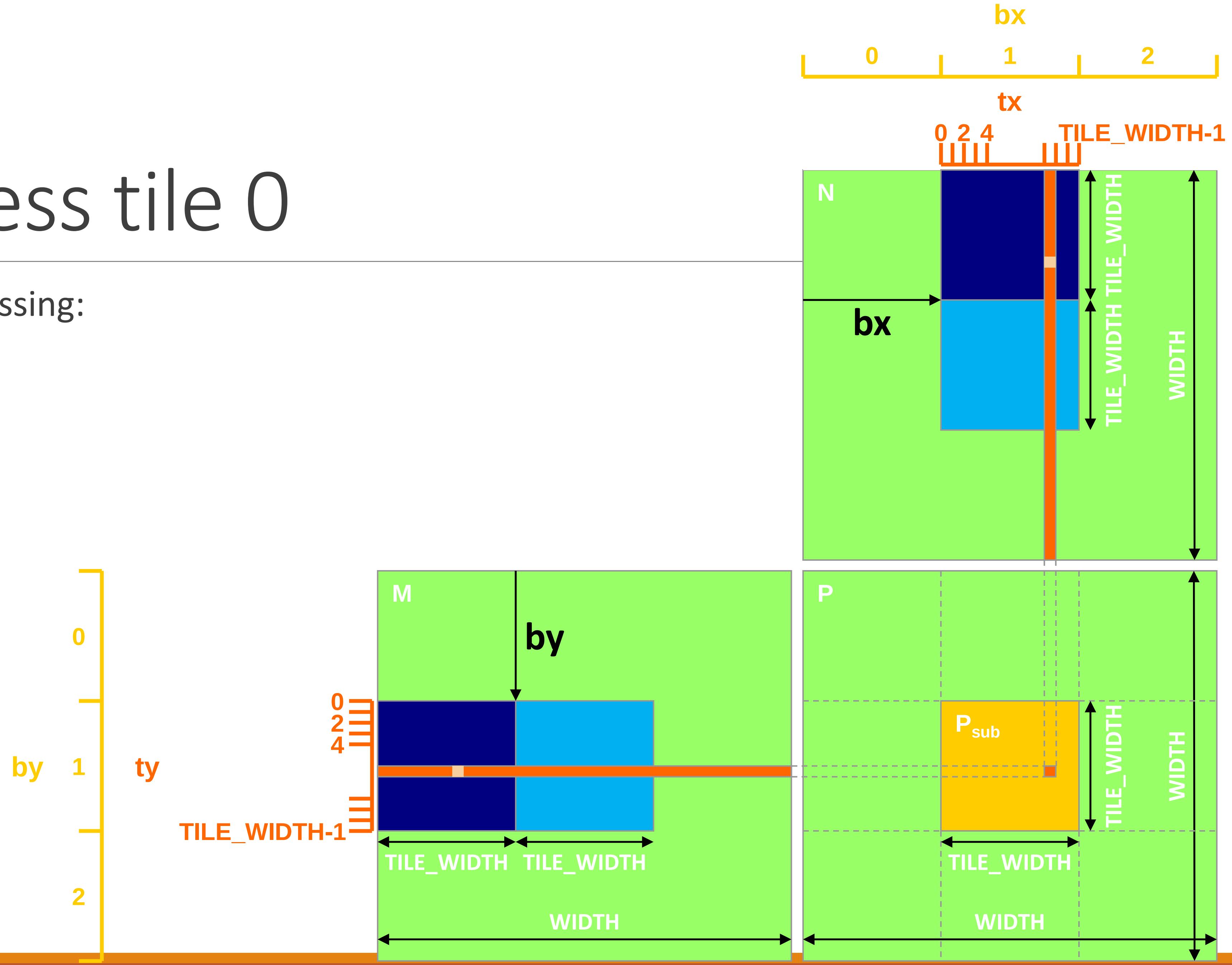
Required in algorithms that use tiles

- Ensures that all elements of the tile have been loaded into shared memory
- Ensures that all elements of the tile have been used in computations

How to access tile 0

Accessing tile 0 with 2-D addressing:

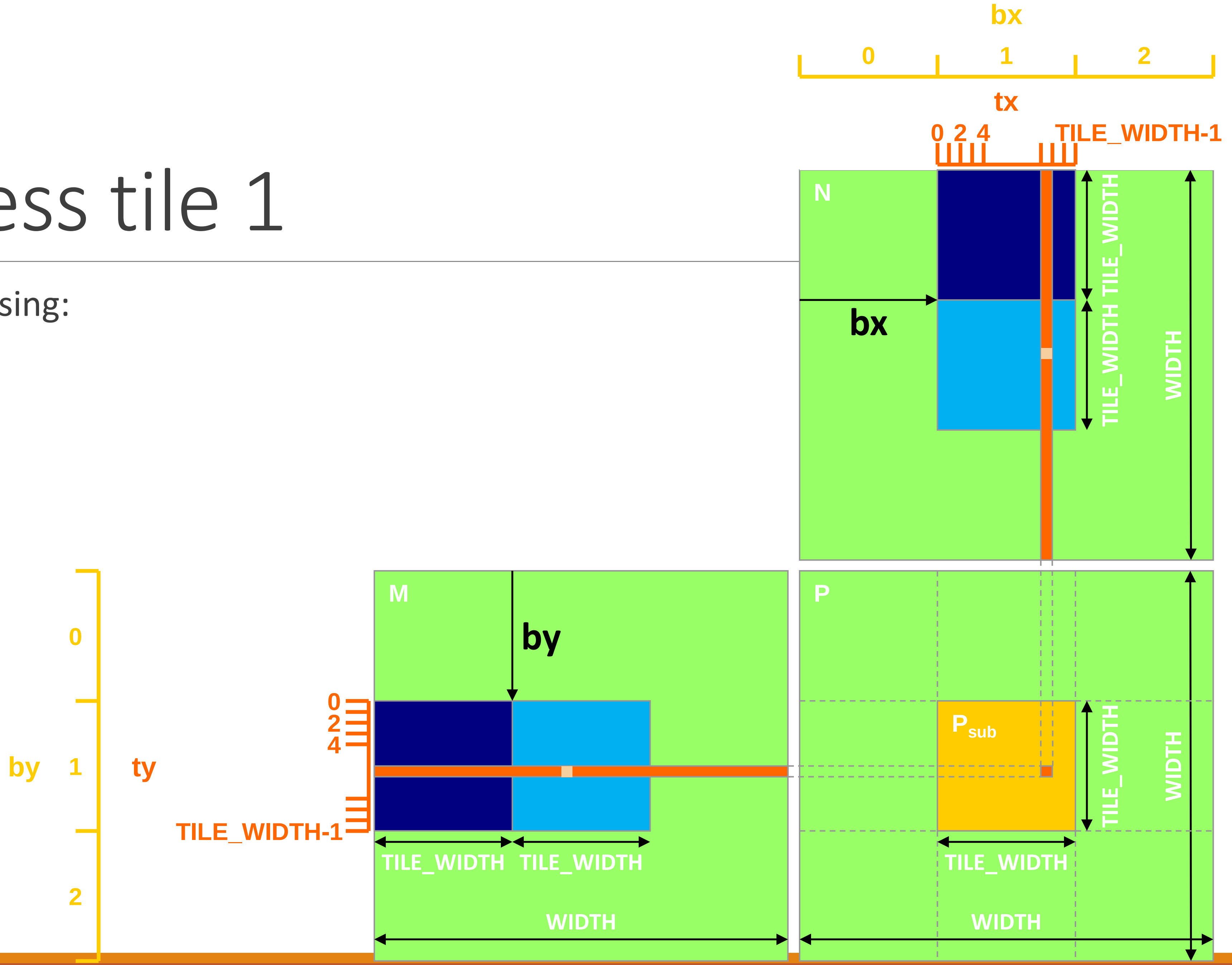
- M[Row][tx]
  - Row = by * TILE_WIDTH + ty
- N[ty][Col]
  - Col = bx * TILE_WIDTH + tx



How to access tile 1

Accessing tile 1 with 2D addressing:

- M[Row][1 * TILE_WIDTH + tx]
  - Row = by * TILE_WIDTH + ty
- N[1 * TILE_WIDTH + ty][Col]
  - Col = bx * TILE_WIDTH + tx



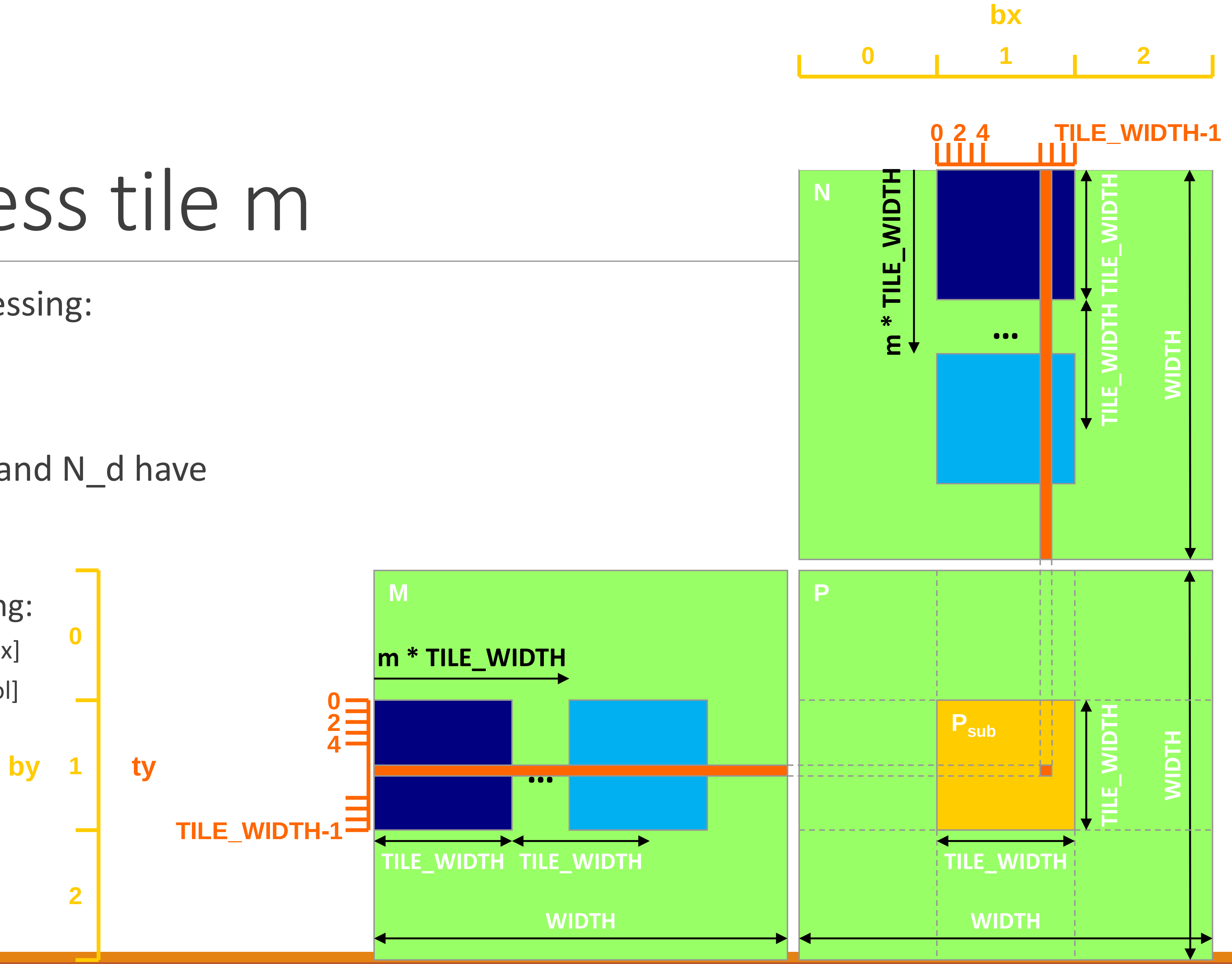
How to access tile m

Accessing tile m with 2-D addressing:

- M[Row][m * TILE_WIDTH + tx]
- N[m * TILE_WIDTH + ty][Col]

Remember that matrices M_d and N_d have been allocated dynamically

- 1-D matrices
- Converting 2-D to 1-D addressing:
  - M[Row * Width + m * TILE_WIDTH + tx]
  - N[(m * TILE_WIDTH + ty) * Width + Col]



2-D matrix multiplication with tiles

```
__global__ void MatrixMulKernel(float *M_d, float *N_d, float* P_d, int Width)
{
  __shared__ float M_ds[TILE_WIDTH][TILE_WIDTH];
  __shared__ float N_ds[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 Pd element to work on
  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Pvalue = 0.0;
```

2-D matrix multiplication with tiles

```
  // Loop over the M_ds and N_ds tiles required to compute the P_ds element
  for (int m = 0; m < Width / TILE_WIDTH; m++) {
    // Colaborative loading of M_ds and N_ds tiles into shared memory
    M_ds[ty][tx] = M_d[Row * Width + m * TILE_WIDTH + tx];
    N_ds[ty][tx] = N_d[Col + (m * TILE_WIDTH + ty) * Width];
    __syncthreads();

    for (int k = 0; k < TILE_WIDTH; k++) {
      Pvalue += M_ds[ty][k] * N_ds[k][tx];
    }
    __synchthreads();

  }
  P_d[Row * Width + Col] = Pvalue;
}
```

Size of tiles

Each block of threads should have as many threads as possible

- TILE_WIDTH = 16 gives 16*16 = 256 threads per block
- TILE_WIDTH = 32 gives 32*32 = 1024 threads per block

For 16, each block performs 2*256 = 512 transfers of float values from global memory and 256 * (2*16) = 8192 operations

- 16 operations/transfer

For 32, each block performs 2*1024 = 2048 transfers of float values from global memory and 1024 * (2*32) = 65536 operations

- 32 operations/transfer

Shared memory and threads

Each SM in Fermi has 16KB or 48KB shared memory

- The size depends on the exact model of GPU

For TILE_WIDTH = 16, each thread uses 2*16*16*4B = 2KB shared memory

- We can have up to 8 blocks of threads active
  (maximum allowed by Fermi)

For TILE_WIDTH = 32, each thread uses 2*32*32*4B= 8KB shared memory

- We can have up to 2 or 6 blocks of threads active

What have we gained?

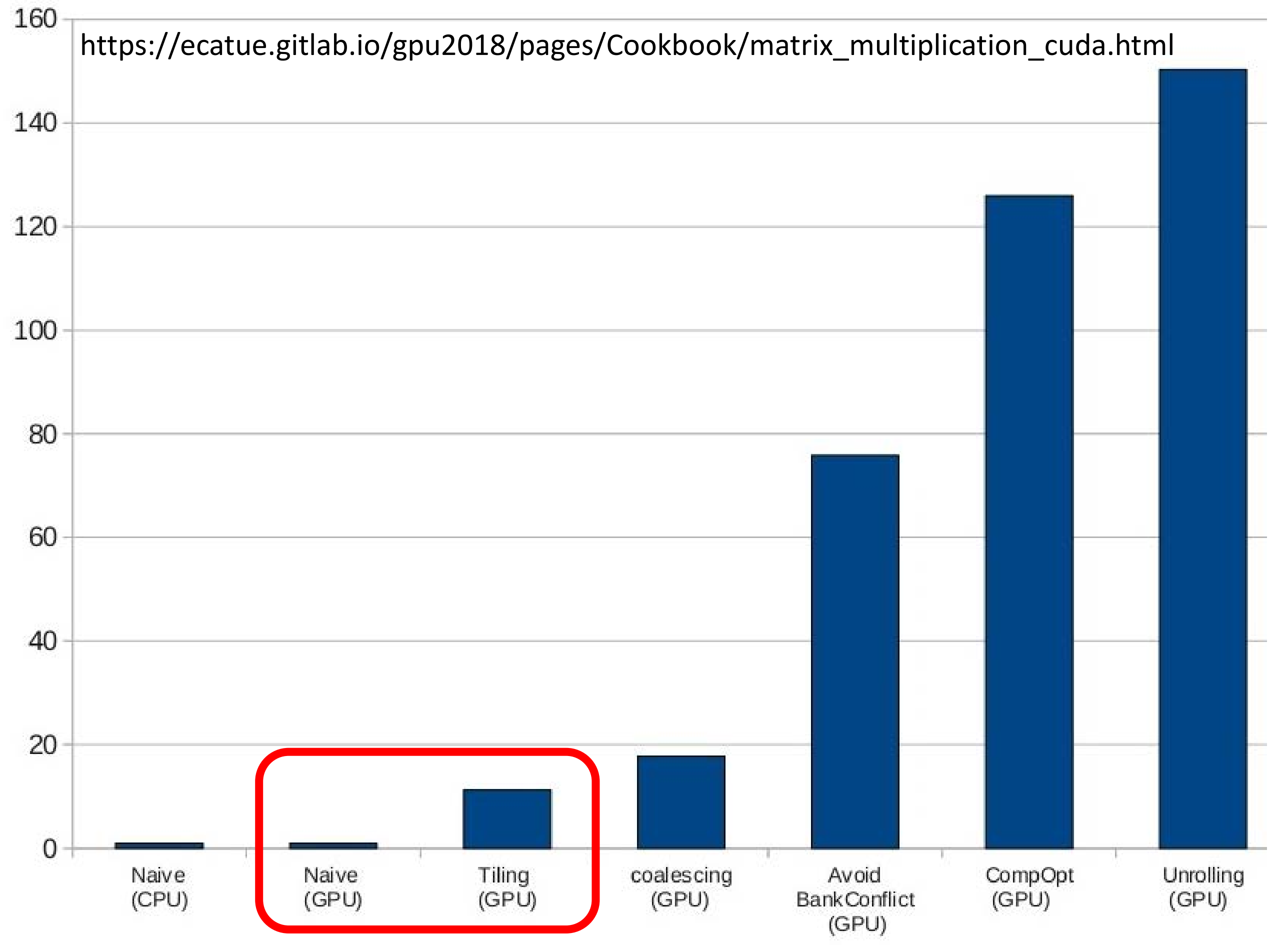
Using TILE_WIDTH = 16 we manage to reduce the number of accesses to the global memory 16 times

- The available bandwidth of 150GB/s can now support (150/4)*16 = 600 GFLOPS!
- Compare that to the 37.5 GFLOPS of the initial computational kernel

More results – 8800 GT GPU

Matrix multiplication

- Single precision
- Size of 4096x4096



Implications

Algorithm and source code has to be modified to take advantage of shared memory

Complicates writing and understanding of source code

Need to know the memory hierarchy of the GPU to achieve high performance

Coalesced memory accesses

Global memory bandwidth

What we need

What we get

A small 64x4 DRAM bank

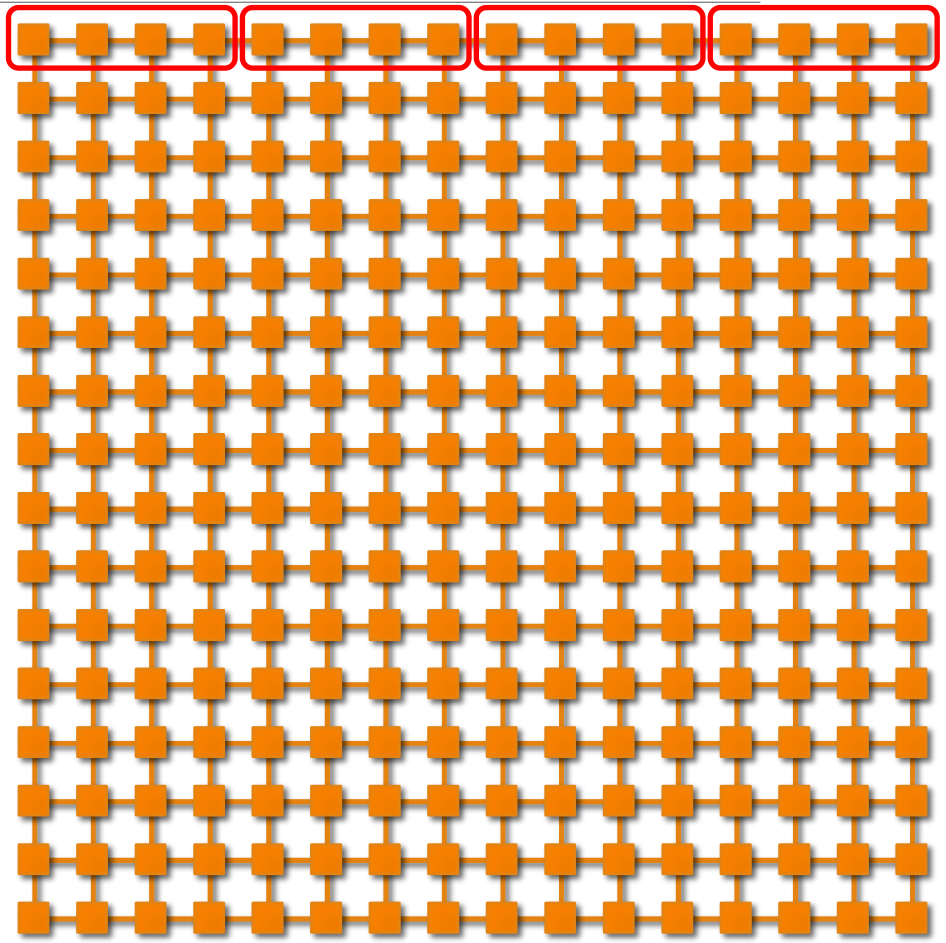
A word is 4 bits

Each row has 4 (= 2^2) words

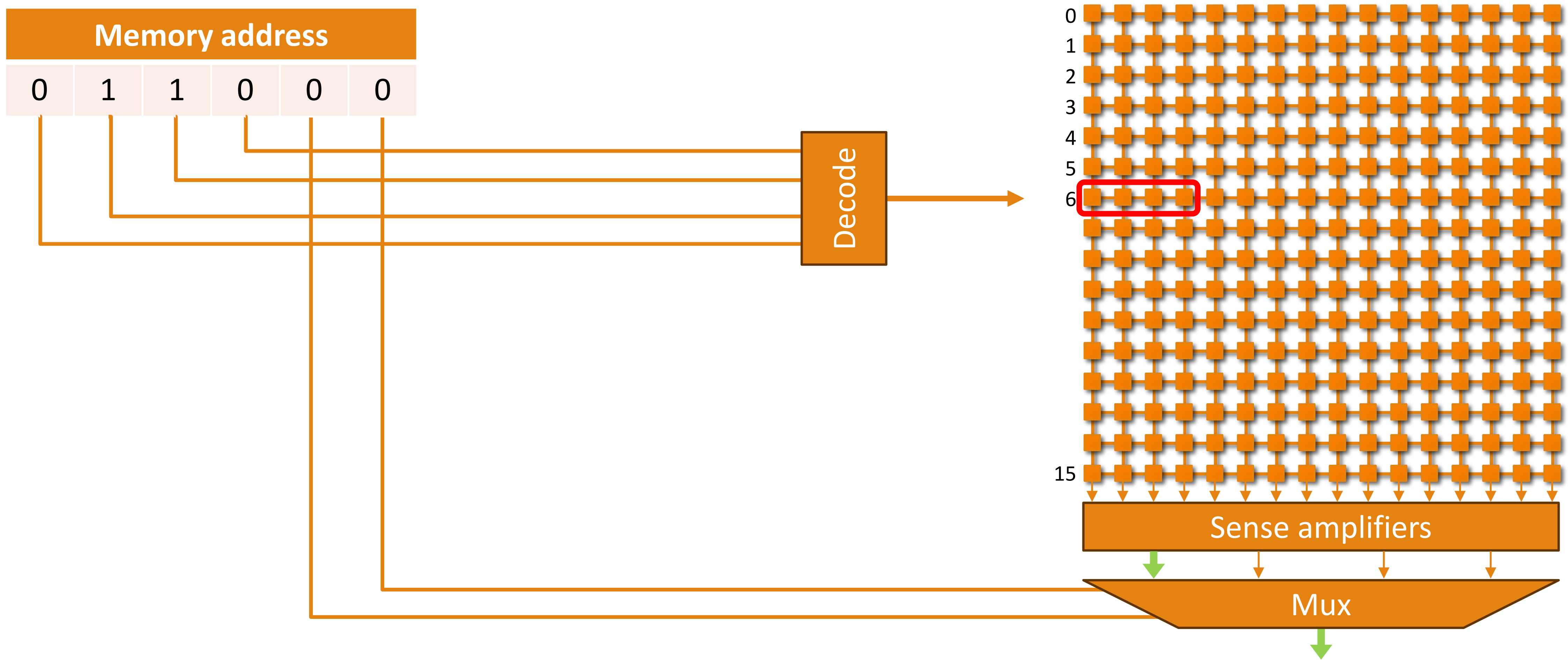
The bank has 16 (= 2^4) rows

Memory addresses are 6 bits

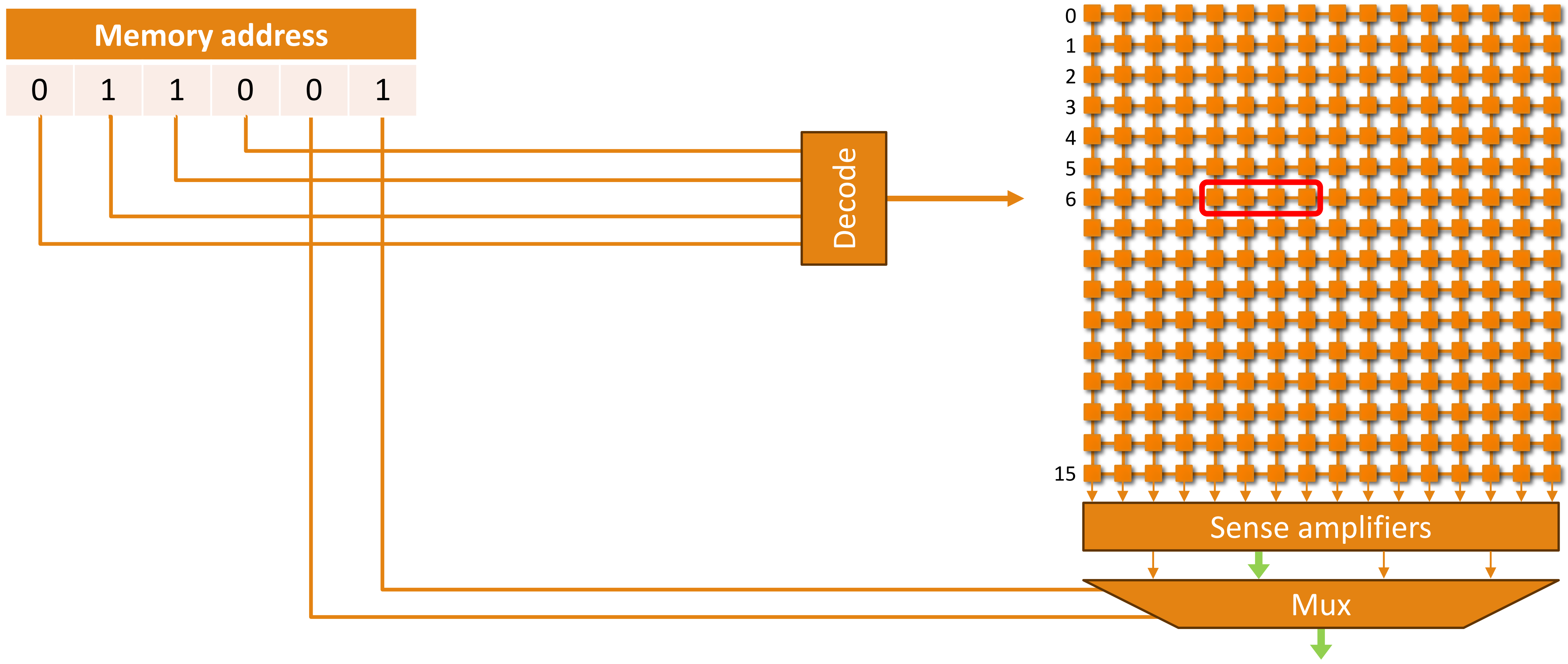
- First 4 bits identify the row
- Last 2 bits identify word within row



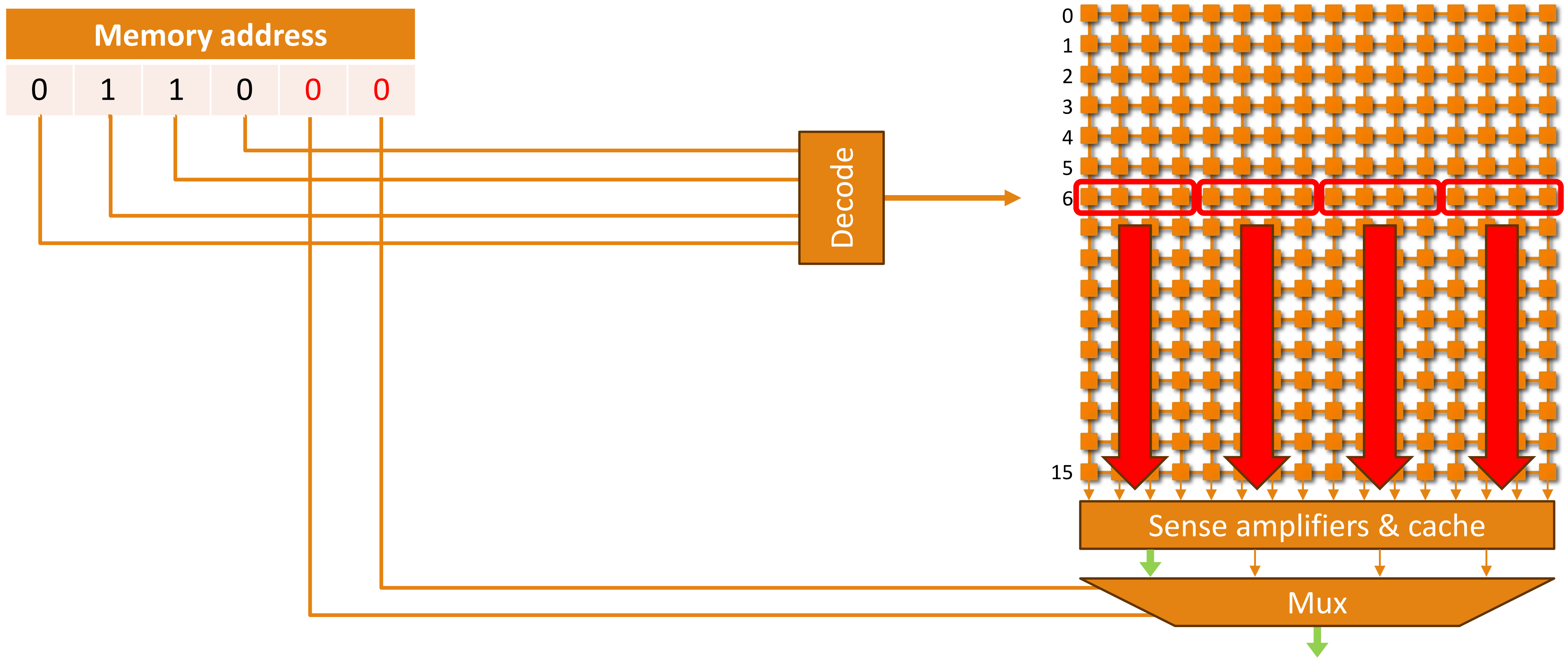
Accessing memory



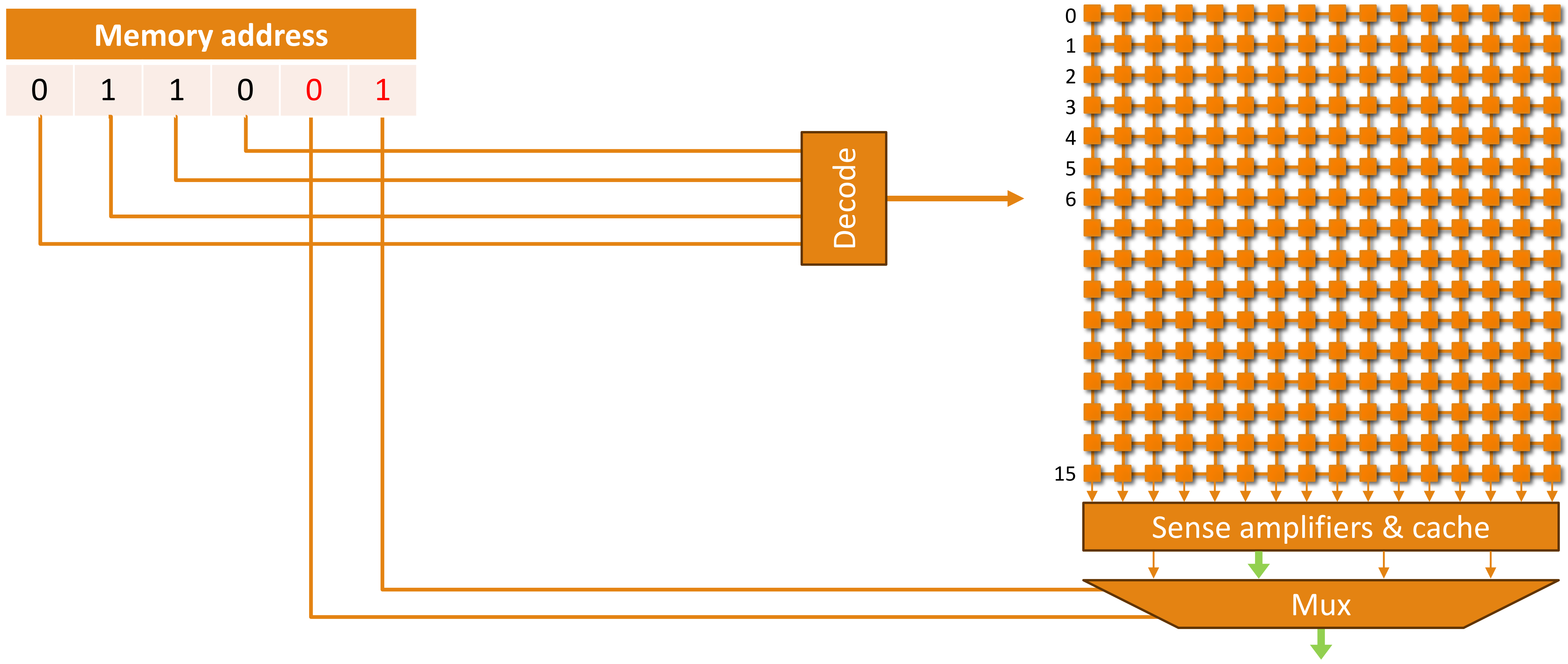
Accessing memory



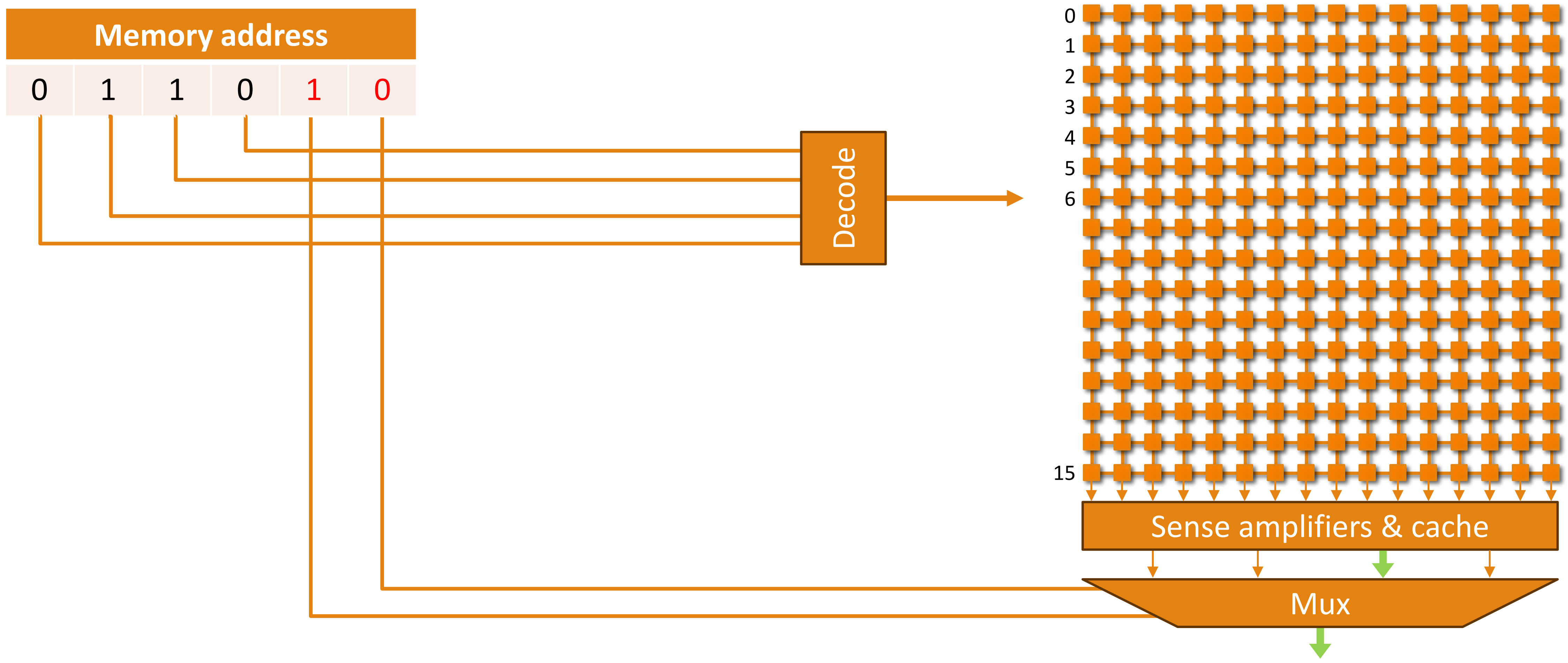
Coalesced memory accesses



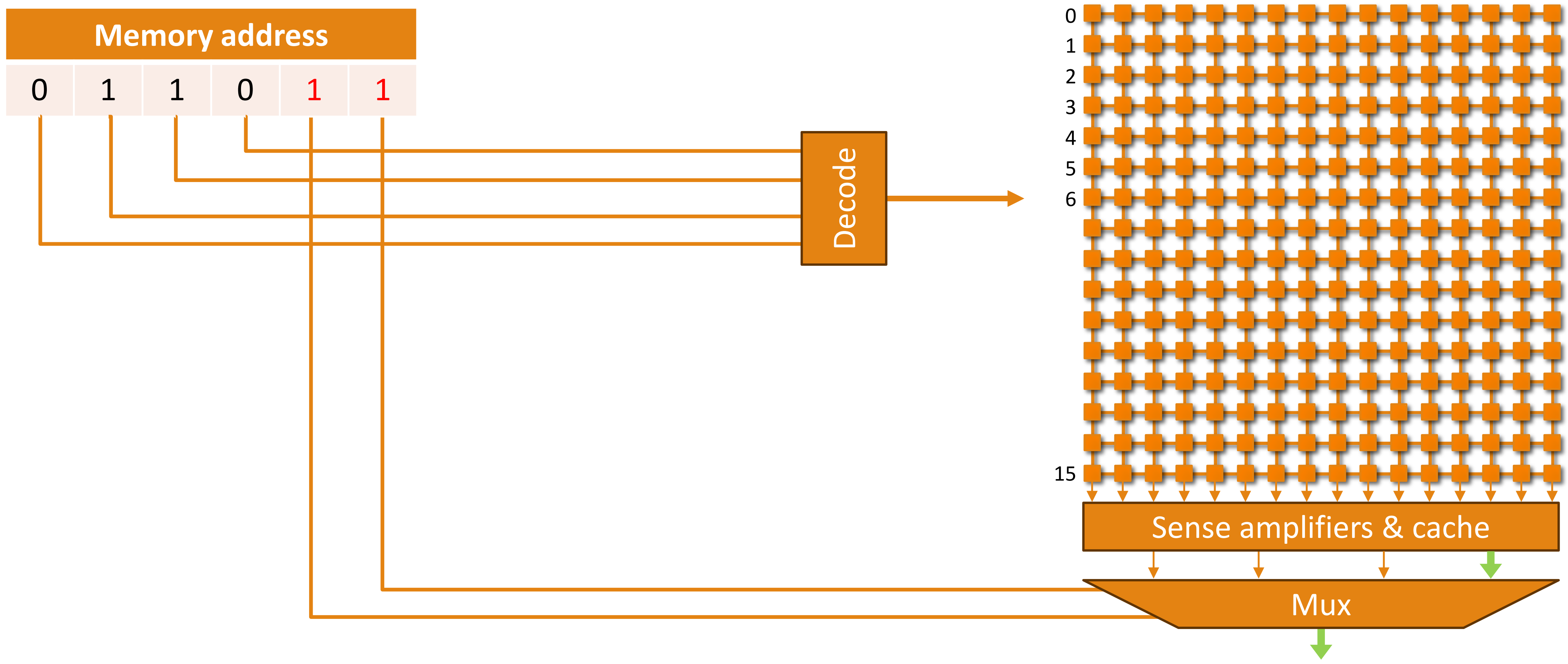
Coalesced memory accesses



Coalesced memory accesses



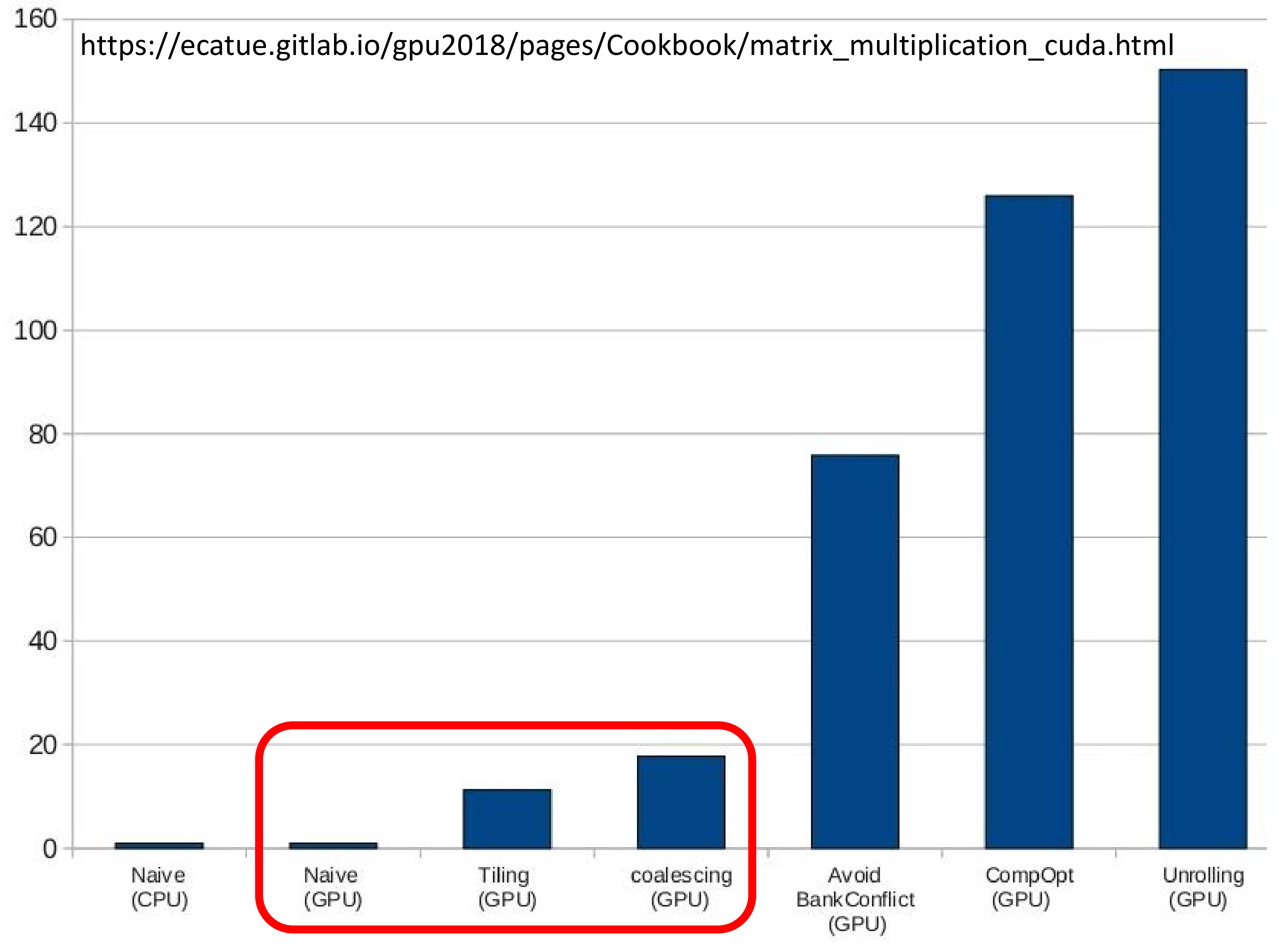
Coalesced memory accesses



More results – 8800 GT GPU

Matrix multiplication

- Single precision
- Size of 4096x4096



Implications

Algorithm and source code has to be modified to take advantage of coalesced memory accesses

Might complicate understanding of source code

Might complicate understanding of rationale behind the decision of writing the source code in that specific way

- Need to know the design of modern DRAM used for the GPU

Bank conflicts in shared memory

Shared memory

Shared memory is divided into banks
- 16 for Compute Capability 1.x
- 32 for Compute Capability ≥ 2.0

Continuous words of 32-bits are stored in continuous banks

Different banks can be accessed simultaneously

If multiple memory addresses being accessed belong to the same bank
- Bank conflict
- Access is serialized
- Bank conflicts exist only for accesses within the same bank
  - Or a half-warp for Compute Capability 1.x

For Compute Capability ≥ 2.0
- There is no bank conflict if the memory address accessed is the same for a number of threads of the warp
  - Does not hold for Compute Capability 1.x

Example

Left

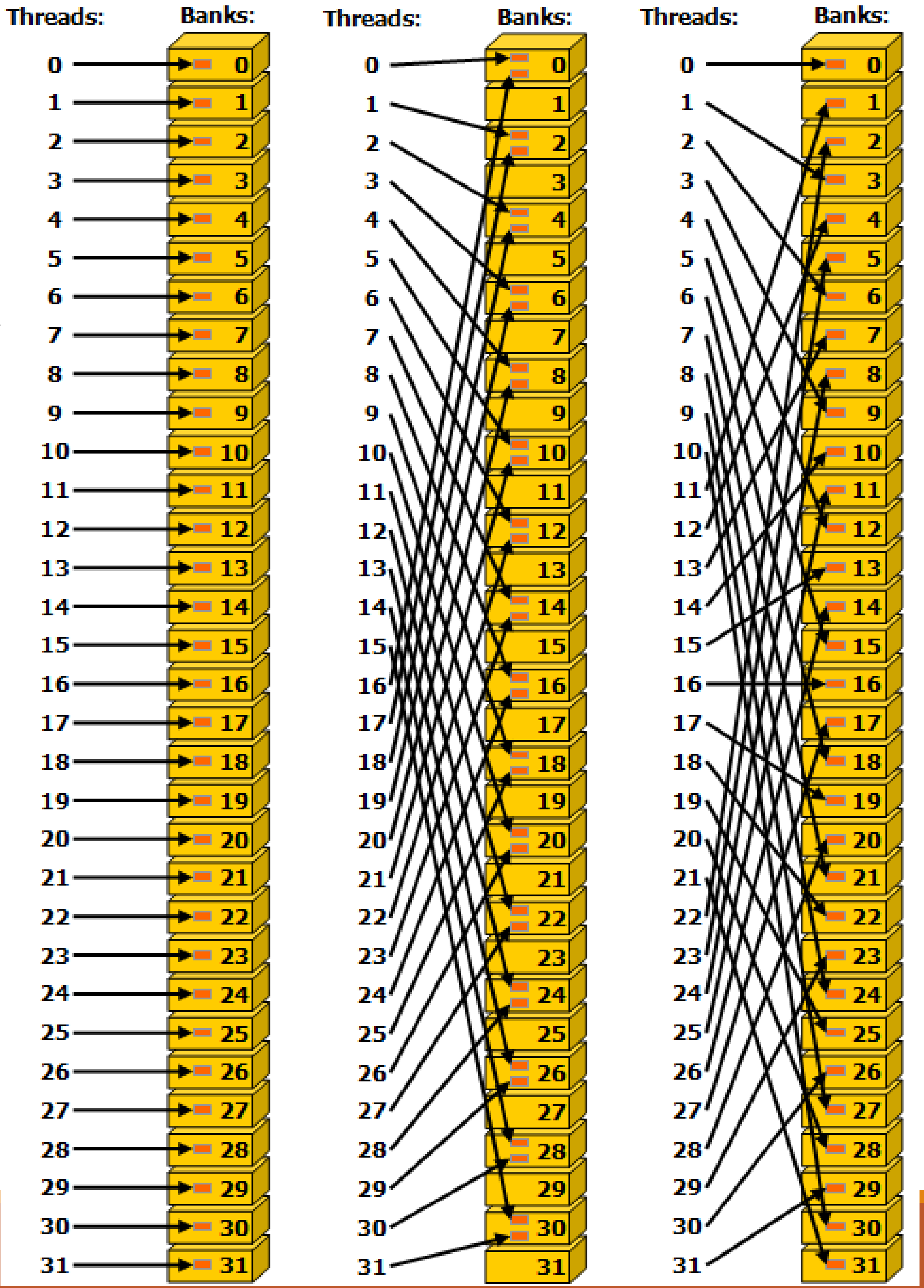
- Linear addressing with a step of 1 for 32-bit words
- No bank conflict

Center

- Linear addressing with a step of 2 for 32-bit words
- 2-way bank conflict

Right

- Linear addressing with a step of 3 for 32-bit words
- No bank conflict

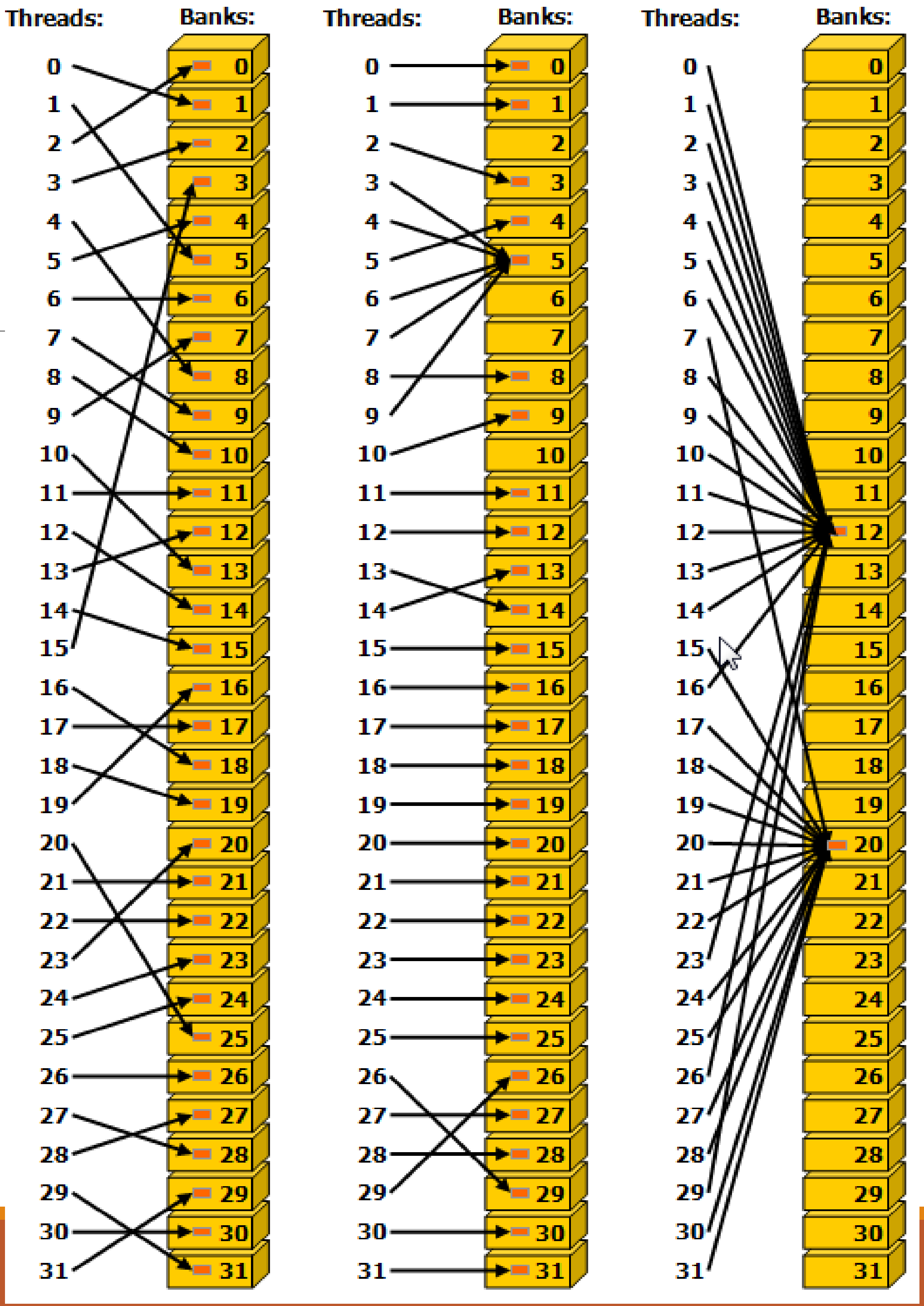


Example

Left
- Random permutation
- No bank conflict

Center
- Threads 3, 4, 6, 7, 9 access the same 32-bit word in bank 5
- No bank conflict

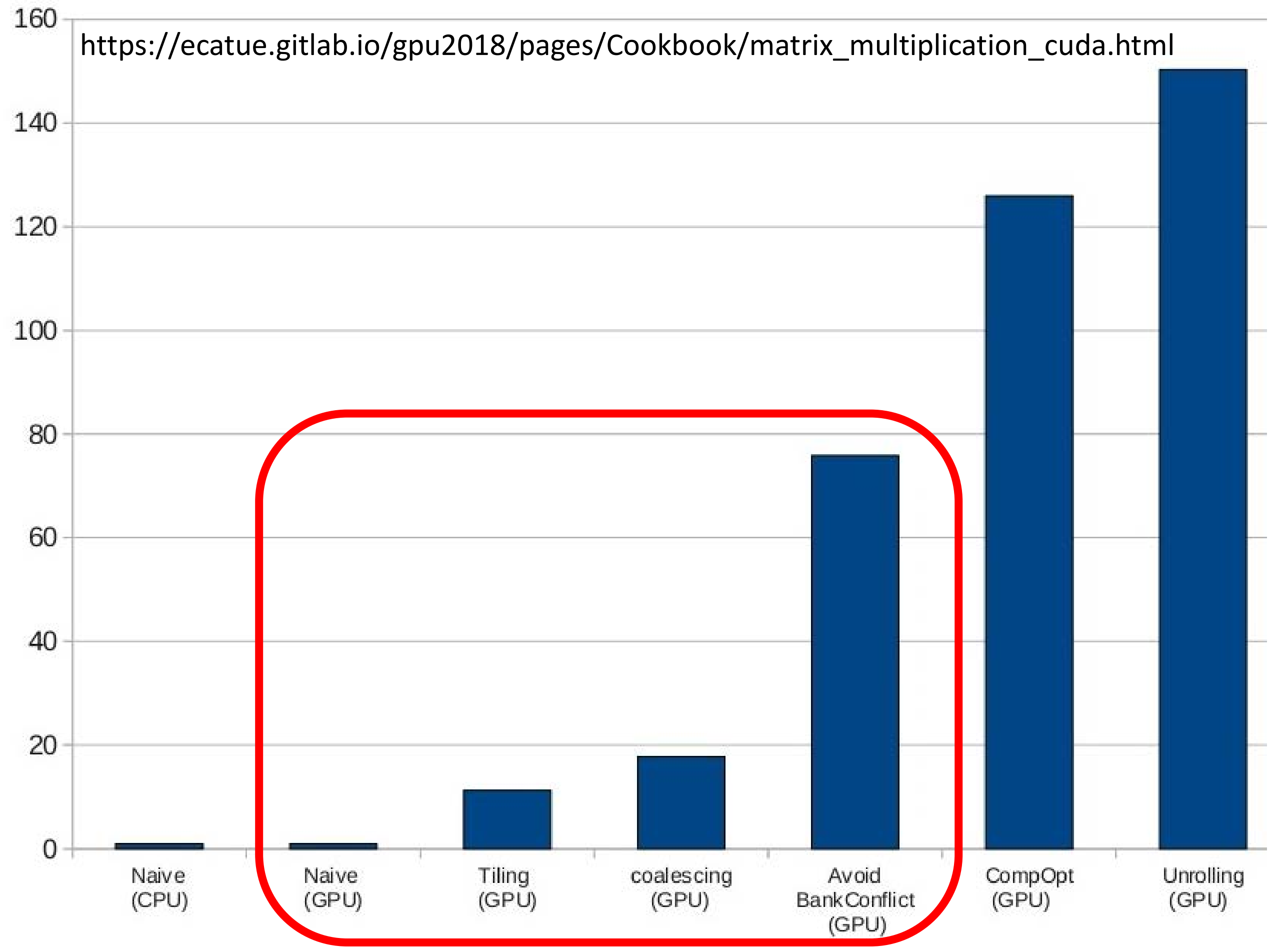
Right
- All threads access the same 32-bit word in each bank
- No bank conflict



More results – 8800 GT GPU

Matrix multiplication

- Single precision
- Size of 4096x4096



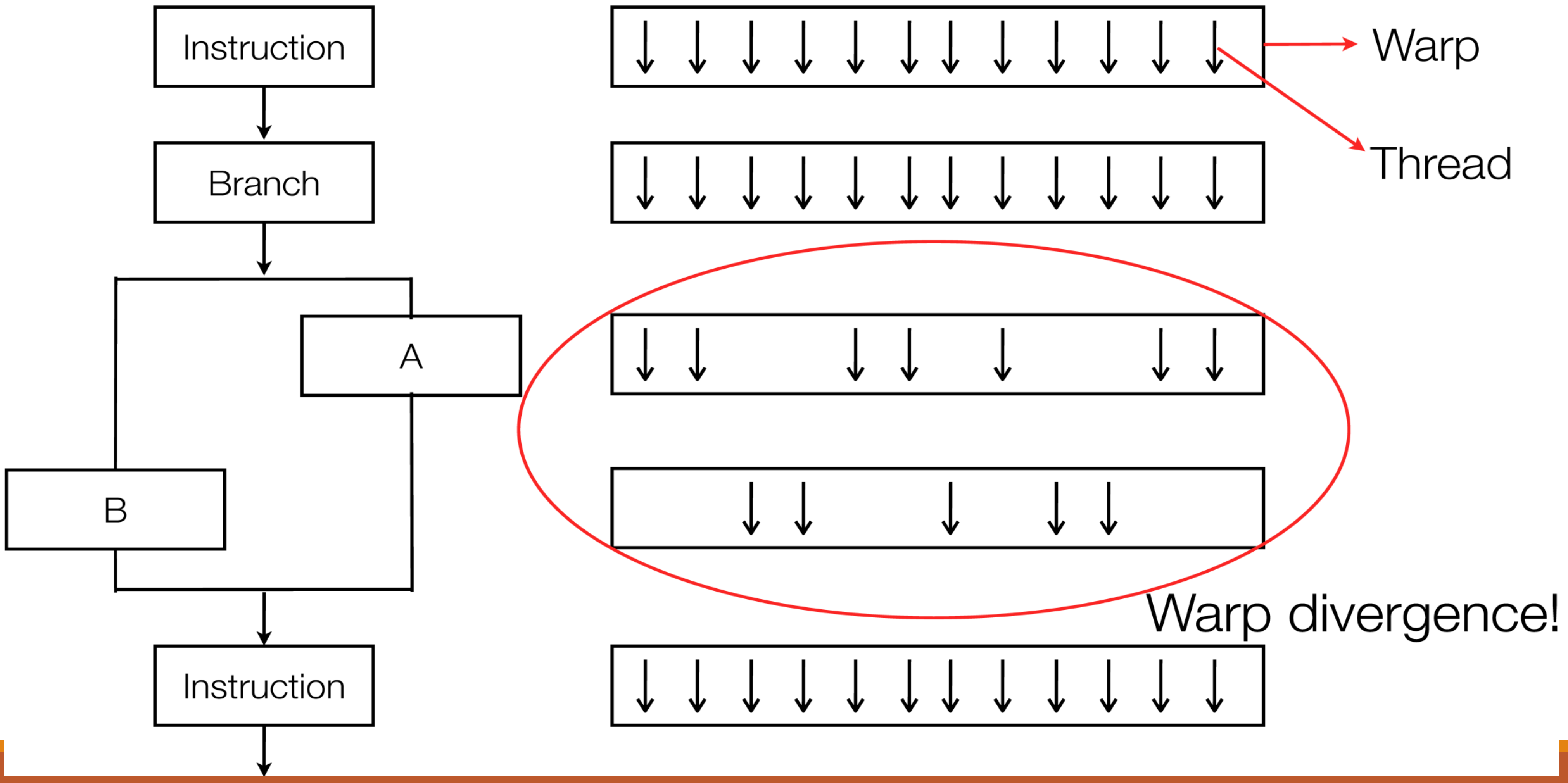
Flow control

Flow control

‘if…else’ instruction

- Threads are executed in warps
- Within each warp, the hardware cannot execute at the same time instructions of the ‘if’ and of the ‘else’ block
  - If there is no else’, neither instructions that follow the ‘if’

```
__global__ void function()
{
       ...
       if (condition) {
               ...
       } else {
               ...
       }
}
```


Instruction
Branch
A
B
Instruction
Warp
Thread
Warp divergence!

How the hardware handles the situation

The hardware serializes execution of the different execution paths

Recommendations

- All threads within a warp should execute the same instructions
    - No divergence occurs if different execution paths are followed among different warps
- If divergence cannot be completely avoided
    - Try multiple continuous threads within a warp to execute the same instructions

Summing all elements of a vector

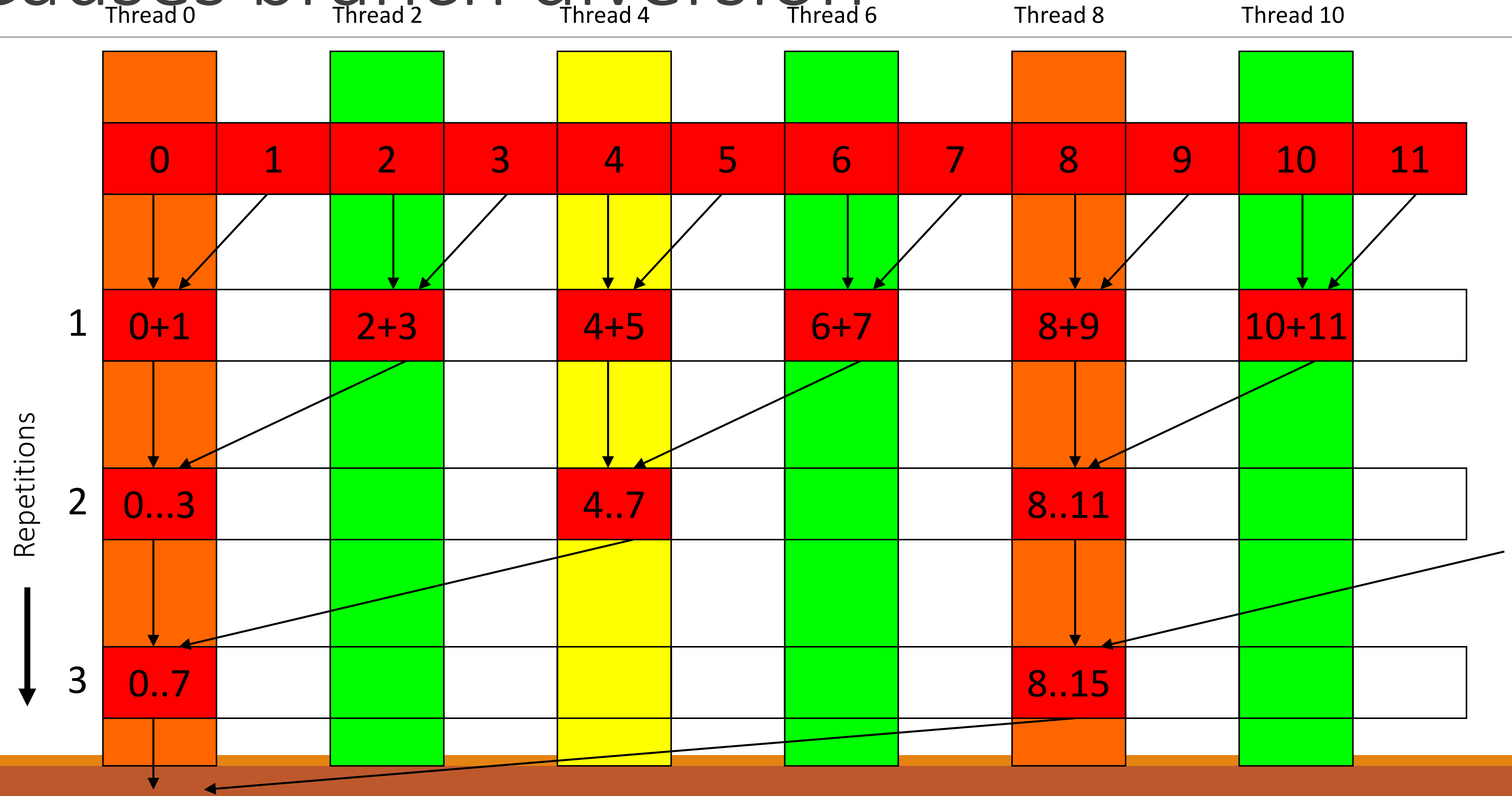
A first implementation

- Add together the closest neighboring elements that contain a partial sum
- Continue until the final result is calculated

```
__shared__ float partialSum[];
int t = threadIdx.x;

for (int stride = 1; stride < blockDim.x; stride *= 2) {
      __syncthreads();
      if (t % (2 * stride) == 0)
      partialSum[t] += partialSum[t+stride];
}
```

Causes branch diversion



Observations

During each repetition

- Two execution paths per warp
  - Threads that perform the addition and threads that don’t

At most half of the threads in a warp contribute towards calculating the result

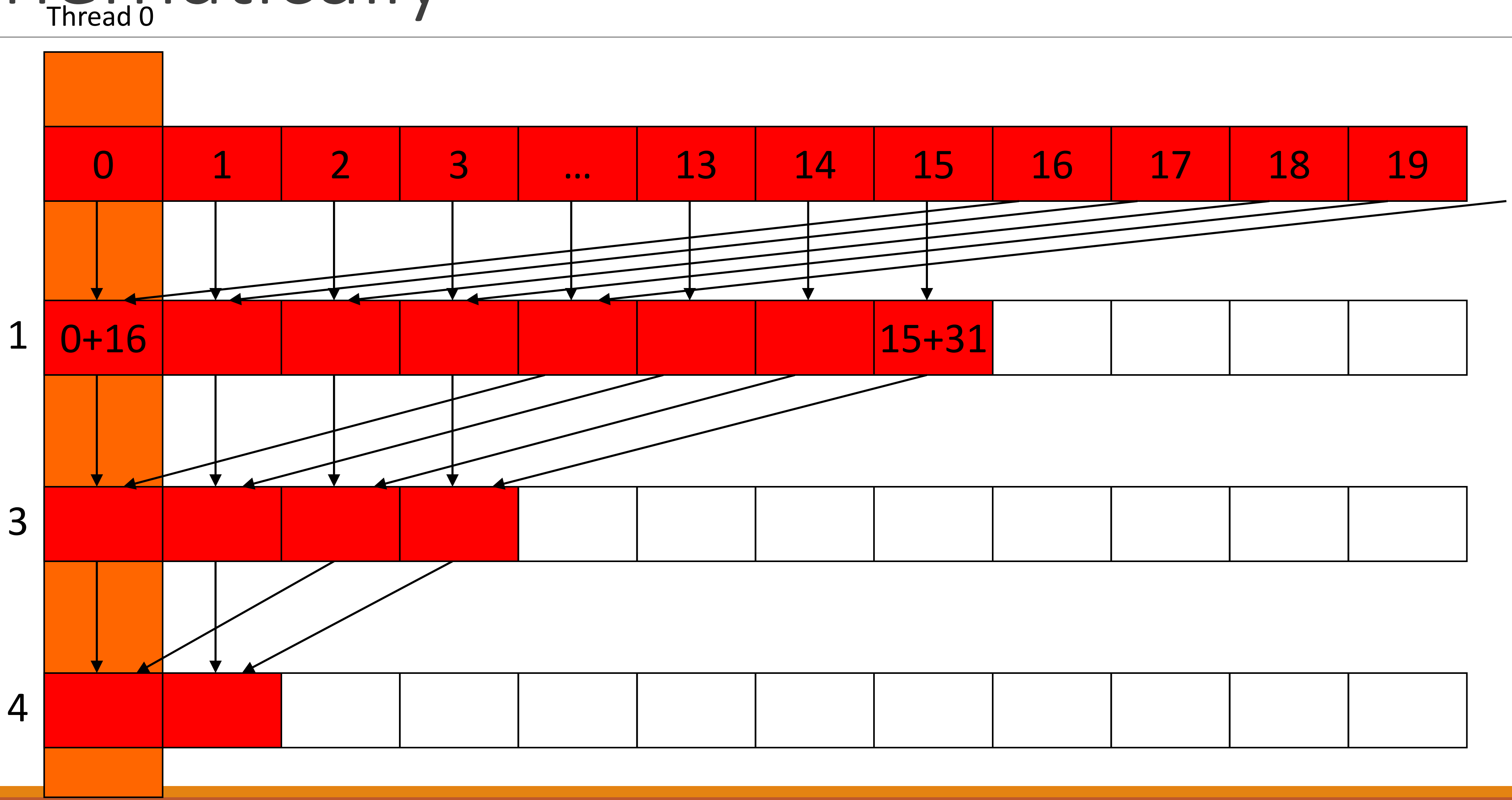
- All odd numbered threads don’t contribute already from the first iteration!
- After the 5th repetition
  - Whole warps don’t contribute
    - Although there is no branch diversion, there is poor exploitation of computational resources

A better implementation

```
__shared__ float partialSum[];
unsigned int t = threadIdx.x;

for (int stride = blockDim.x; stride > 1; stride >> 1) {
      __syncthreads();
      if (t < stride)
             partialSum[t] += partialSum[t + stride];
}
```

Schematically



Example for blocks of 512 threads

Repetition	Threads that calculate	Warps
1	256	16
2	128	8
3	64	4
4	32	2
5	16	1
6	8	1
7	4	1
8	2	1
9	1	1

Threads > Warp

Threads < Warp

Warp divergence

Optimizing Parallel Reduction in CUDA

[Optimizing Parallel Reduction in CUDA]

Goes over a total of 7(!) refinements for such a simple problem

- Refinements are tied to the architecture of the GPU

But leads to a 30x total speedup in execution time

Synchronous and asynchronous execution

Blocking and non-blocking functions

Synchronous vs. Asynchronous execution

- Synchronous:
  - Calling a function is blocking
  - Execution is performed serially
- Asynchronous:
  - Calling a function is non-blocking
  - The control of execution immediately returns to the host

Advantages of asynchronous execution

- Overlapping of processing and data transfer on different devices
  - Not only GPU and CPU
  - Accessing hard drive, transfer of data over the network, etc.

Asynchronous execution in CUDA

Most functions of the CUDA API that are called from the host are blocking

Exceptions

- Calling computational kernels
- cudaMemcpy() within the same device (cudaMemcpyDeviceToDevice())
- cudaMemcpy() from the host to the device for up to 64kB of data
- Asynchronous copying of data

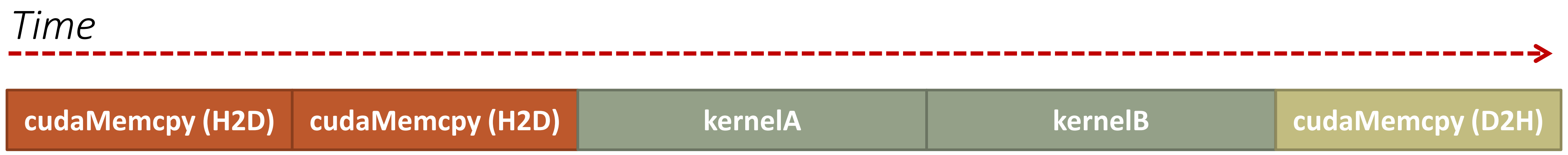
Asynchronous execution

```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernels on device
kernelA<<<blocks, threads>>>(a);
kernelB<<<blocks, threads>>>(b);

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Completely synchronous execution



Asynchronous execution

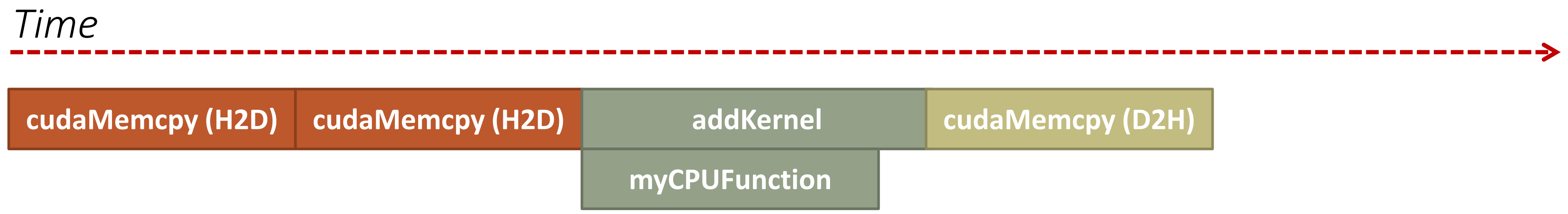
```
//copy data to device
cudaMemcpy(d_a, a, size * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size * sizeof(int), cudaMemcpyHostToDevice);

//execute kernel on device
addKernel<<<blocks, threads>>>(d_c, d_a, d_b);

//host execution
myCPUfunction();

//copy back result data
cudaMemcpy(c, d_c, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Asynchronous execution CPU-GPU



Simultaneous execution through pipelining

Latest GPU generations include subsystems for asynchronous execution of calculations and copying of data

- Allows data transfers while processing
- GPU architectures Maxwell and Kepler even have double subsystems for copying data
  - PCIe upstream (Device to Host - D2H)
  - PCIe downstream (Host to Device - H2D)
- Allows simultaneous execution of:
  - Copying part 'n-1' of results from the device to the host
  - Execution of computational kernel for part 'n' of the data
  - Copying part 'n+1' of data from the host to the device for next invocation of the computational kernel

Best strategy to achieve high performance is to overlap data transfers with computations

All GPUs with Compute Capability ≥ 2.0 have the ability to execute simultaneously multiple computational kernels

- Especially useful for problems with a relatively small problem size

CUDA Streams

Put on a queue the tasks to be performed on the GPU

- All calls to computational kernels are asynchronous
  - Control returns immediately to host for execution of next instructions
  - Can be another computational kernel invocation
- The GPU extracts task from streams when it has resources to execute them

Tasks within the same stream are executed in order(FIFO, no overlap)

Tasks among different streams don’t have a global order of execution and can overlap

```
//create a handle for the stream
cudaStream_t stream;

//create the stream
cudaStreamCreate(&stream);

//do some work in the stream...

//destroy the stream (blocks host until stream is complete)
cudaStreamDestroy(&stream);
```

Assigning tasks to streams

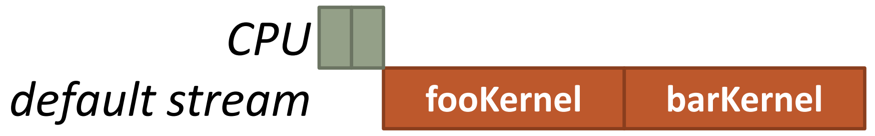
When calling a computational kernel, a 4th parameter defines the stream to be used

The default stream requires special attention

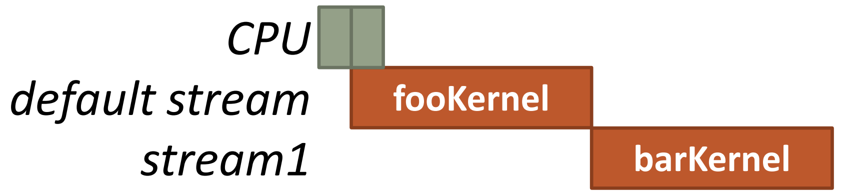
- It is synchronous with all other streams

```
//execute kernel on device in specified stream
fooKernel<<<blocks, threads, 0, stream>>>();
```

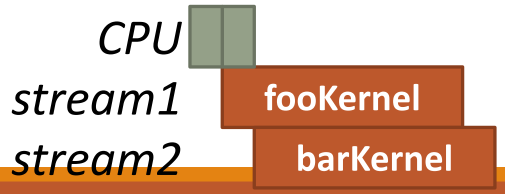
```
fooKernel<<<blocks, threads, 0>>>();
barKernel<<<blocks, threads, 0>>>();
```



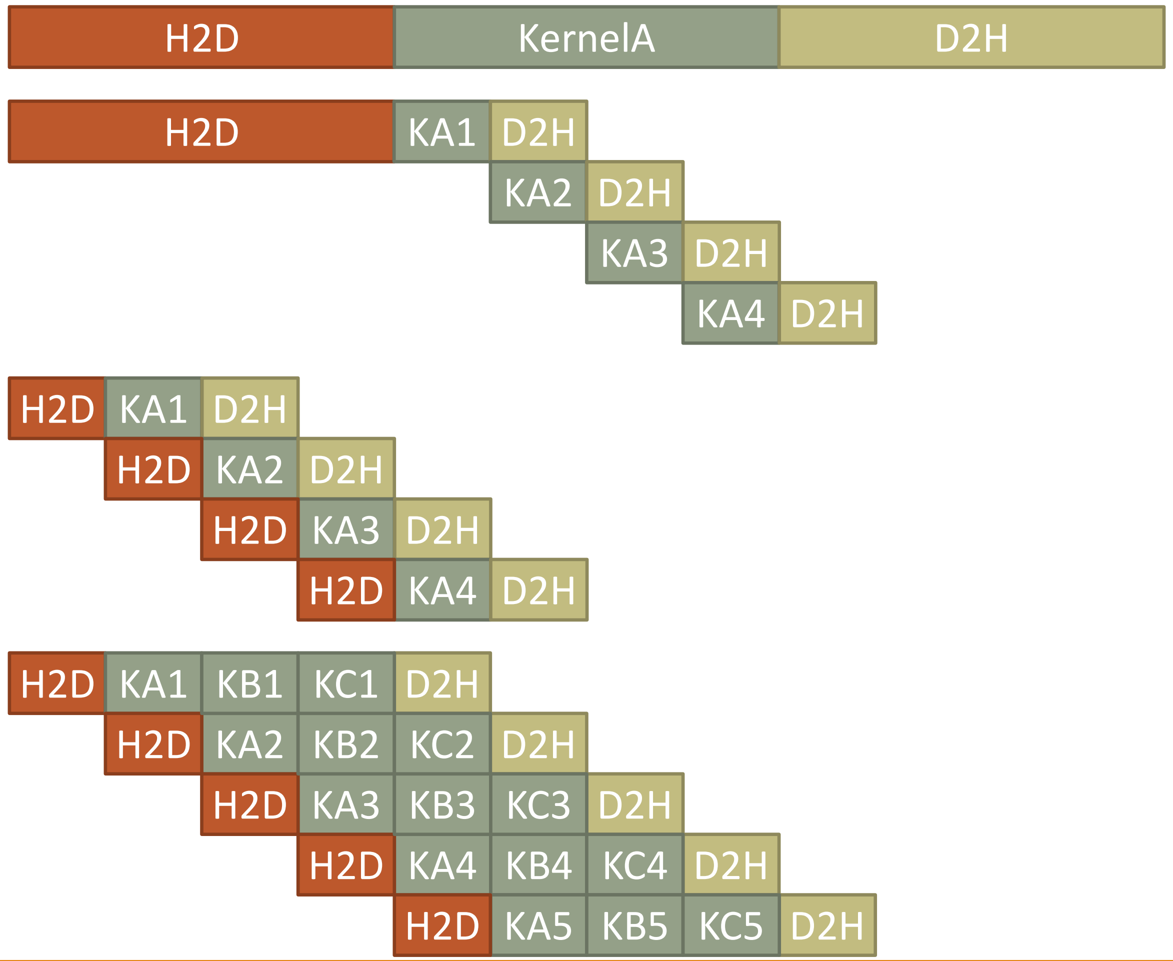
```
fooKernel<<<blocks, threads, 0>>>();
barKernel<<<blocks, threads, 0, stream1>>>();
```



```
fooKernel<<<blocks, threads, 0, stream1>>>();
barKernel<<<blocks, threads, 0, stream2>>>();
```



Levels of overlap



Synchronous execution

2-level overlapping
- H2D and D2H don’t overlap

3-level overlapping
- Both data transfer subsystems are active
- Execution subsystem active
  - Possibly underutilized

5-level overlapping
- Both data transfer subsystems are active
- Execution subsystem active
  - Larger workload => larger possibility for 100% utilization

Host can also be exploited at the same time