Performance optimizations

Ioannis E. Venetis

University of Piraeus

Shared memory

Purpose

Understand how to use the memory hierarchy in CUDA

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

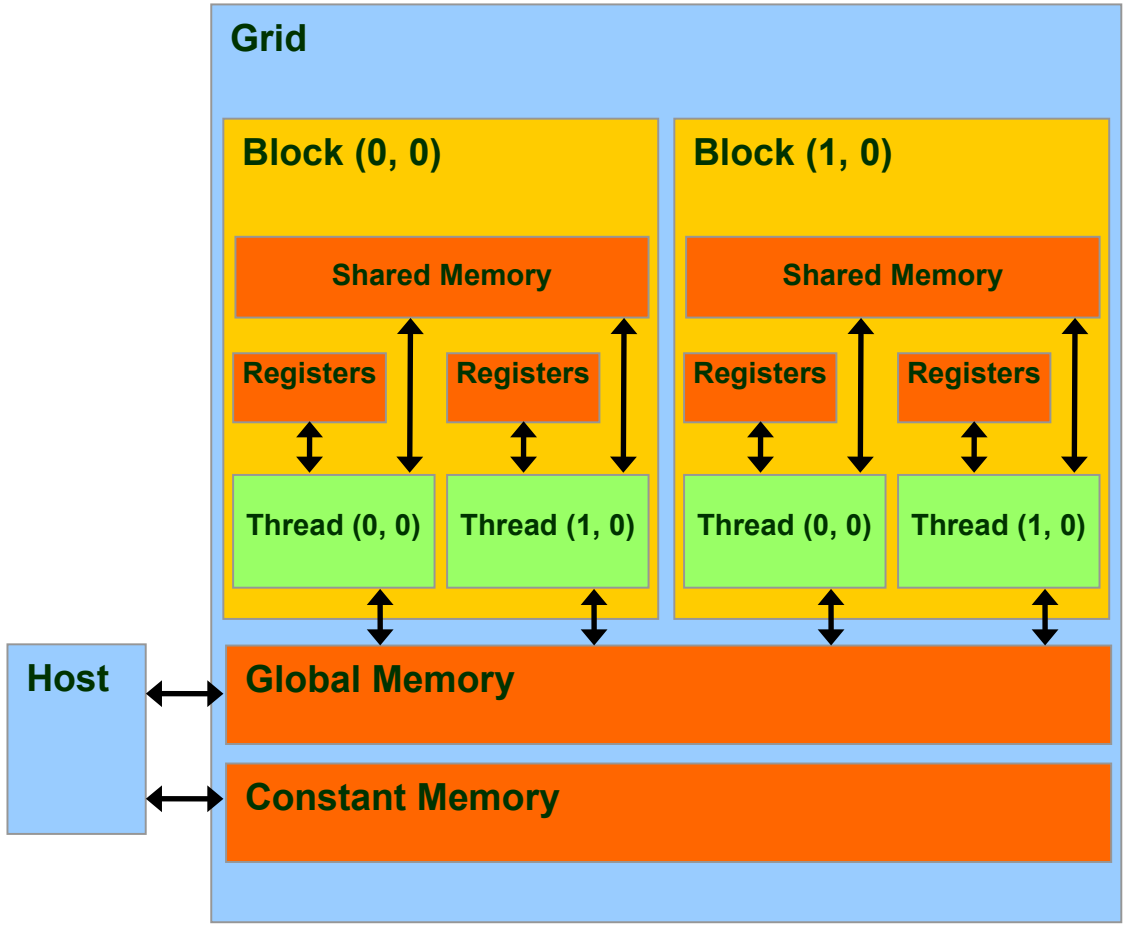
Registers vs. Memory

- Using registers is cheap (in terms of time)
  - No additional instructions to access memory
  - Very fast, but very few
- Accessing memory is slow
  - But there is plenty of it

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 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)
{

  1.   __shared__ float M_ds[TILE_WIDTH][TILE_WIDTH];
  2.   __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

Basic 2-D matrix multiplication algorithm

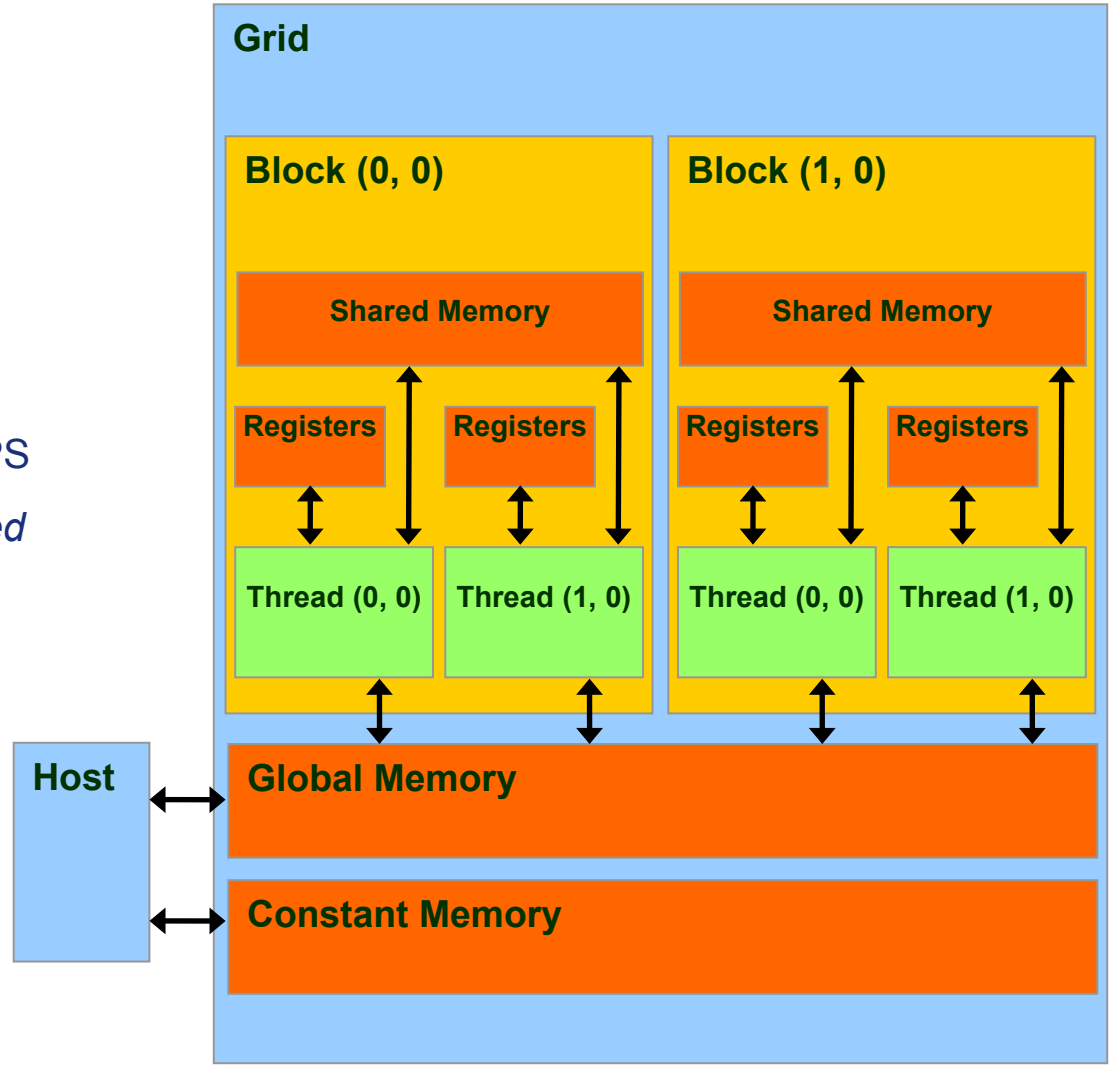
```
__global__ void MatrixMulKernel(float *M_d, float *N_d, float *P_d, int Width)
{
  // Calculate the row index of the P_d element and M
  int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  // Calculate the column index of the P_d element and N
  int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

  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;
}
```

Performance issues on the Fermi architecture

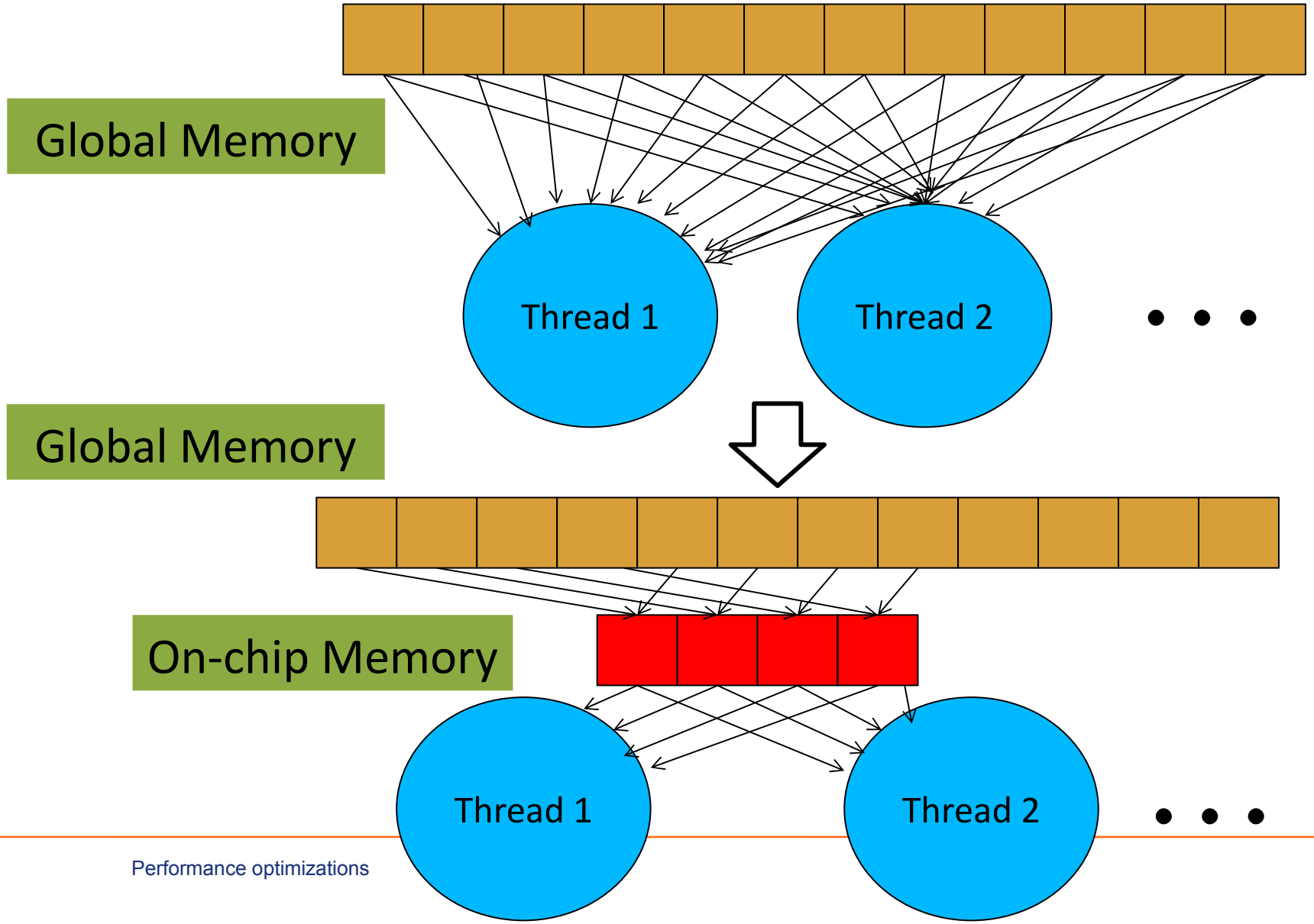
All threads access global memory to read input matrix elements

- 2 accesses (8 bytes) for one multiplication and one addition of single precision numbers
- 4B/s of memory bandwidth/FLOPS
- Max. performance of the architecture is 1000 GFLOPS
  - *4*1000 = 4000 GB/s 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

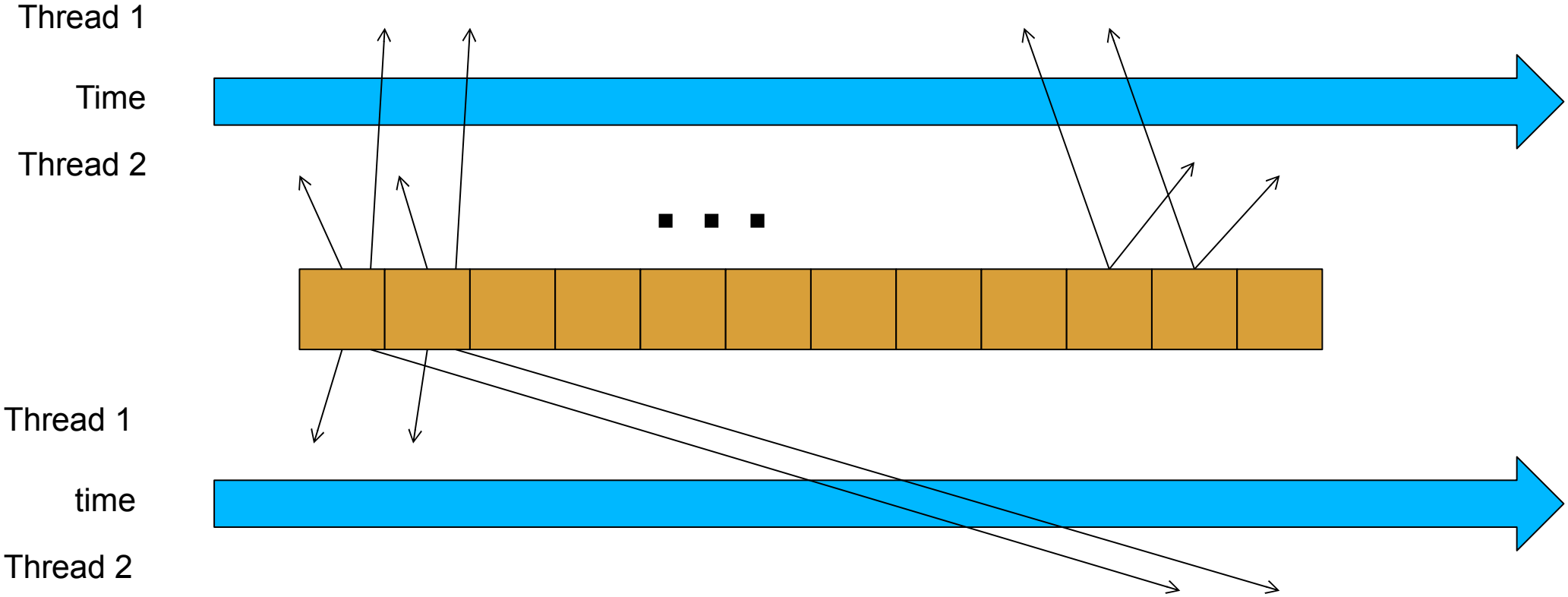


Basic idea when using tiles



When to use shared memory

Good when threads access the data with small time differences



Bad when threads access the data with large time differences

Overview of the technique

Repeat

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

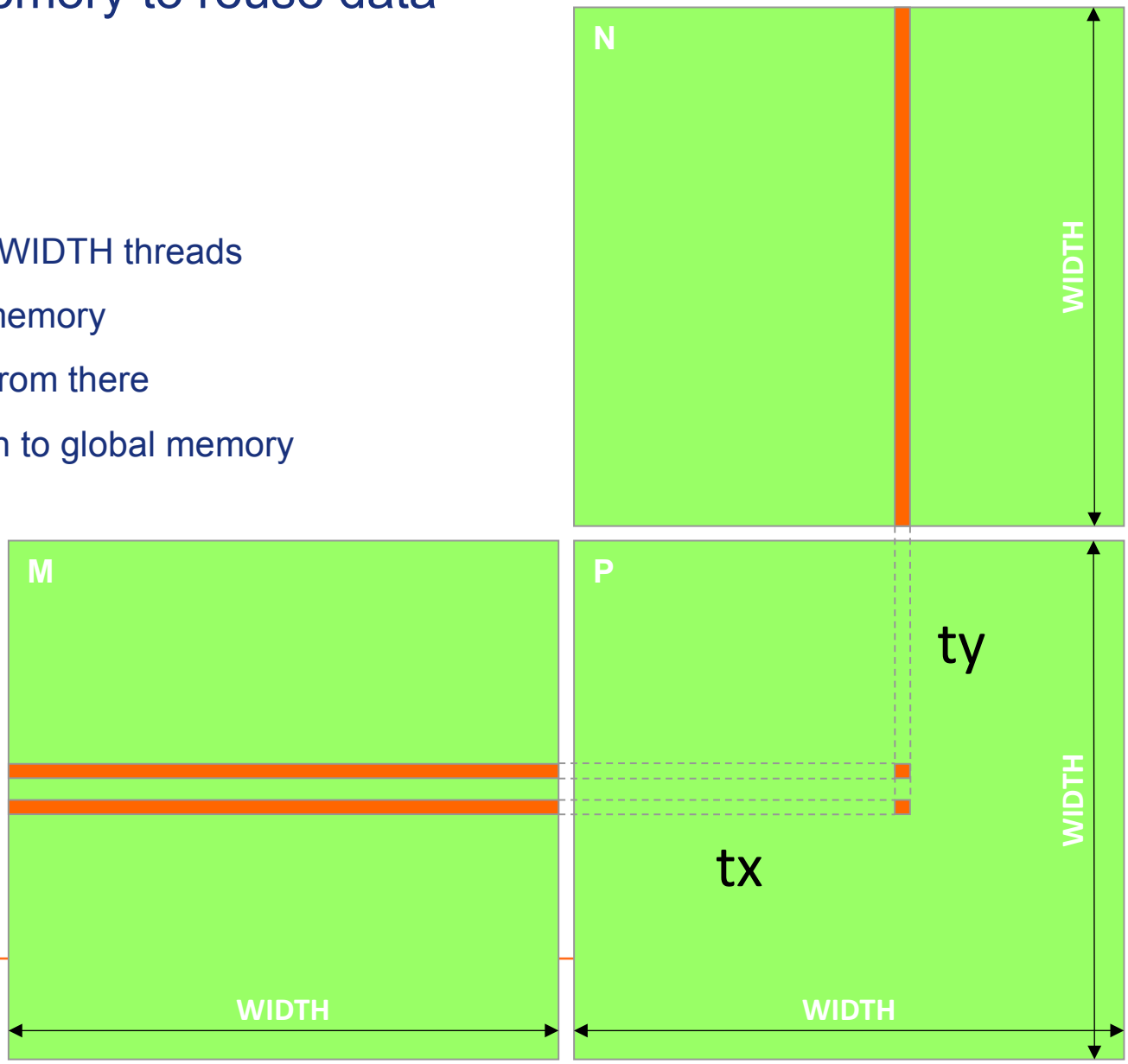
while more tiles are available

Main idea: Use shared memory to reuse data

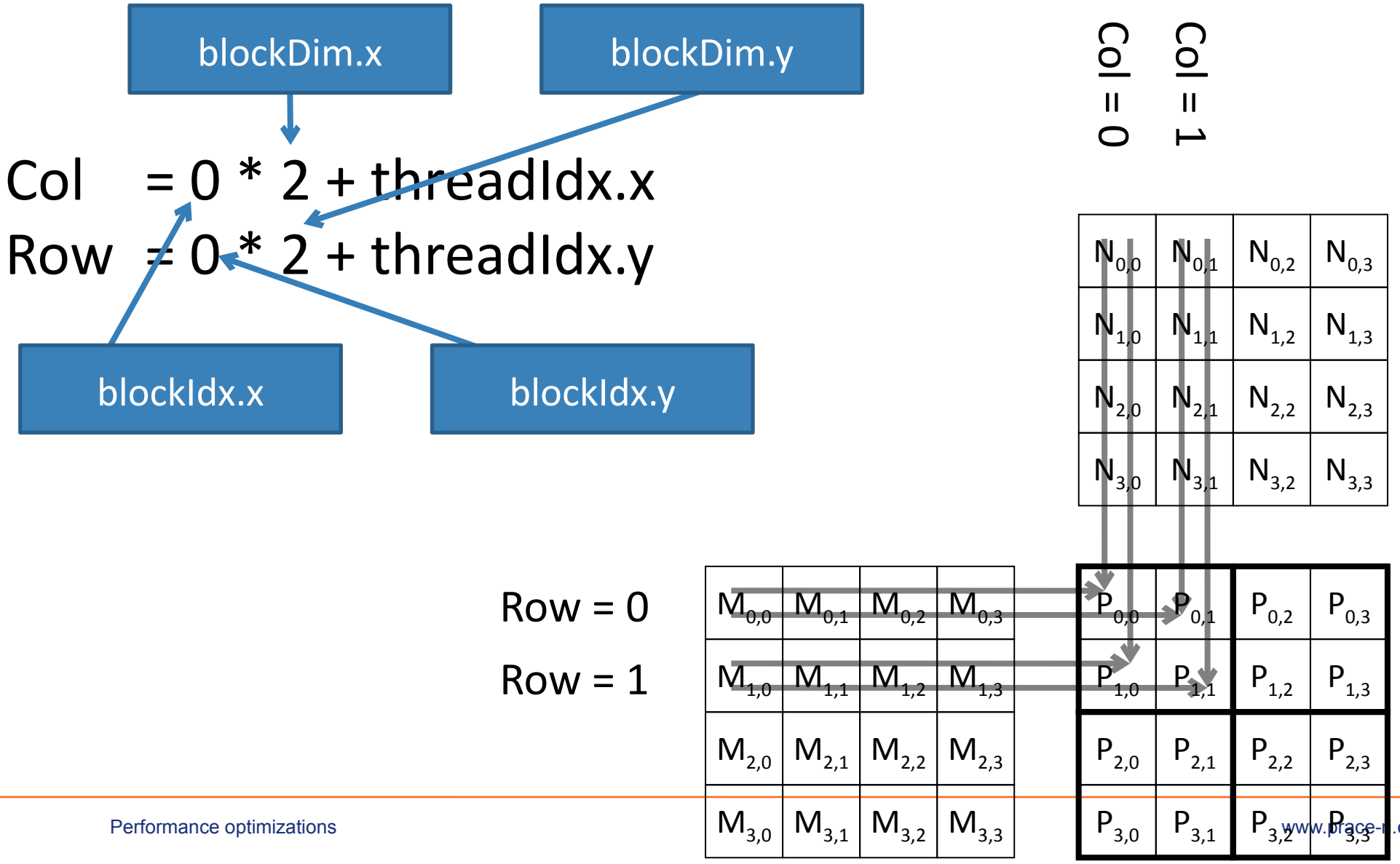
Every input element is read from WIDTH threads

Copy each element into shared memory

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



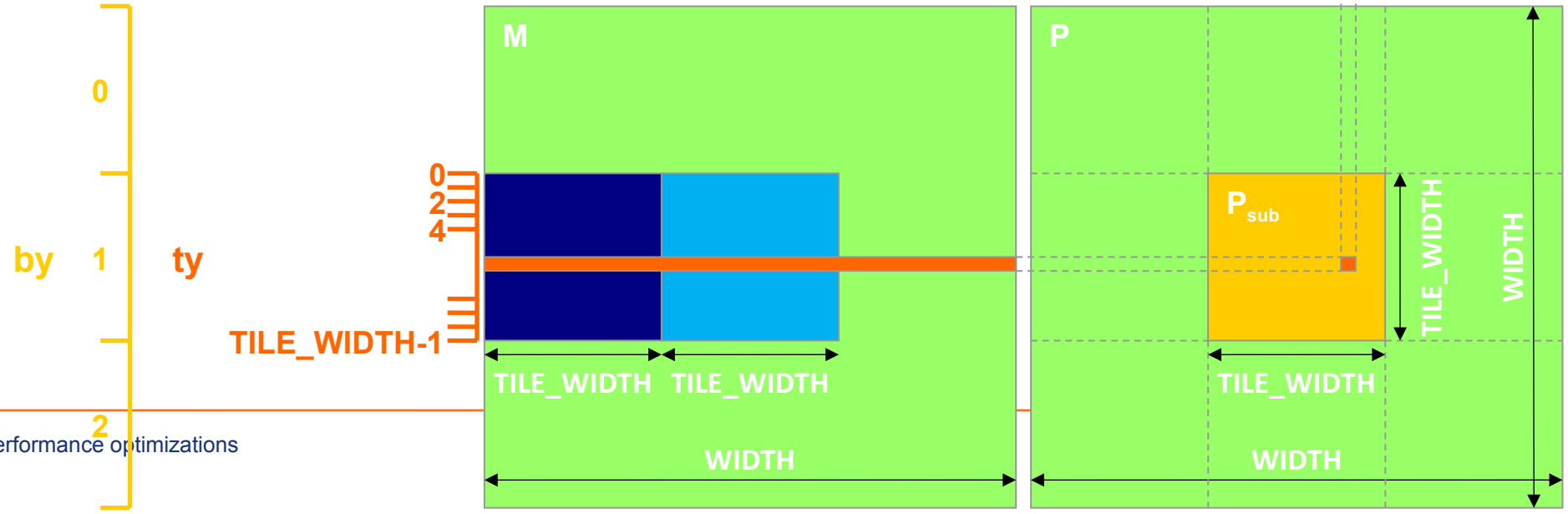
Processing block (0, 0) for TILE_WIDTH = 2



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_d and N_d



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 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

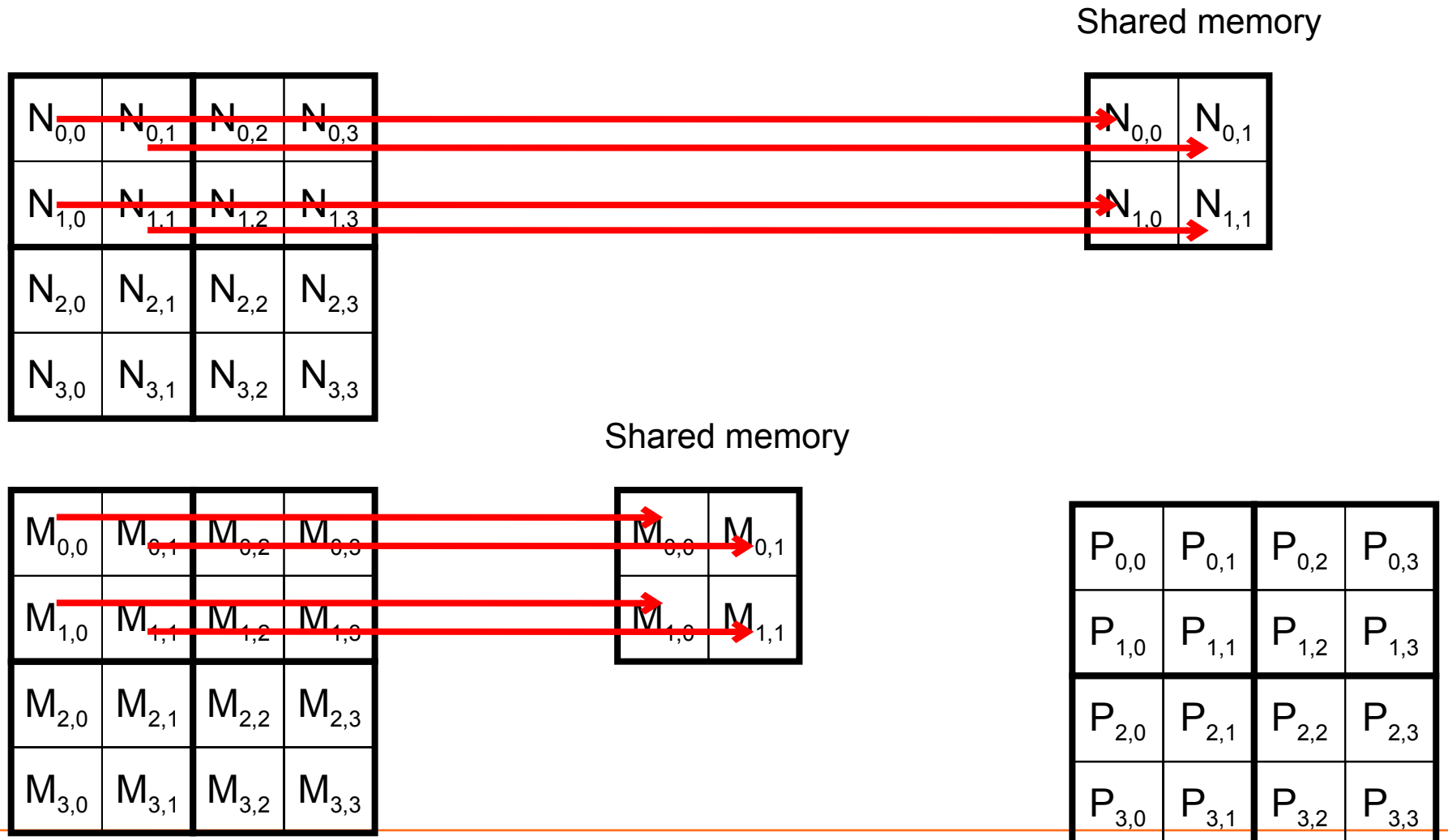
Using the elements copied into shared memory perform as many operations as possible towards the final result

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

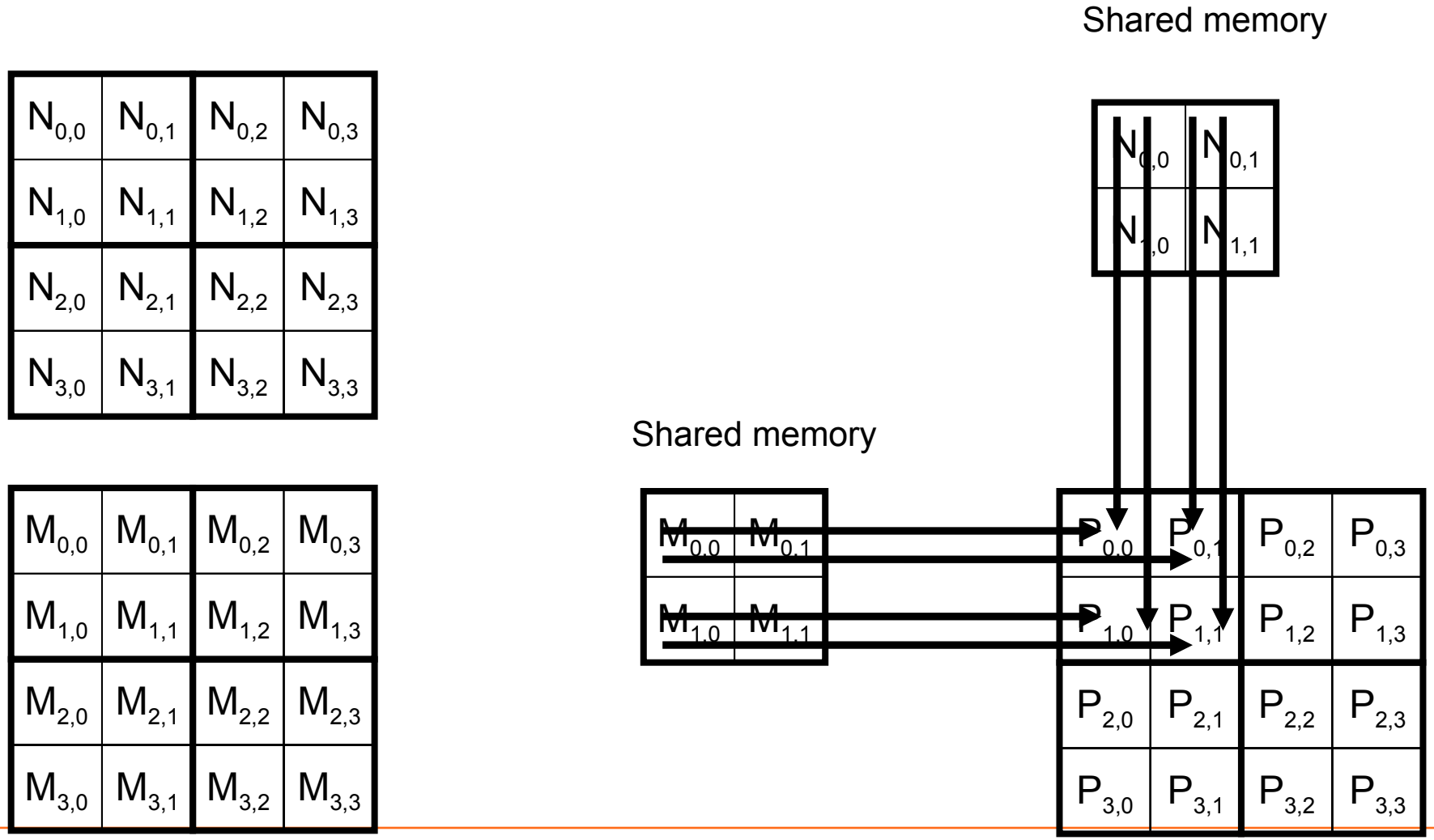
Third step: Write back final result

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

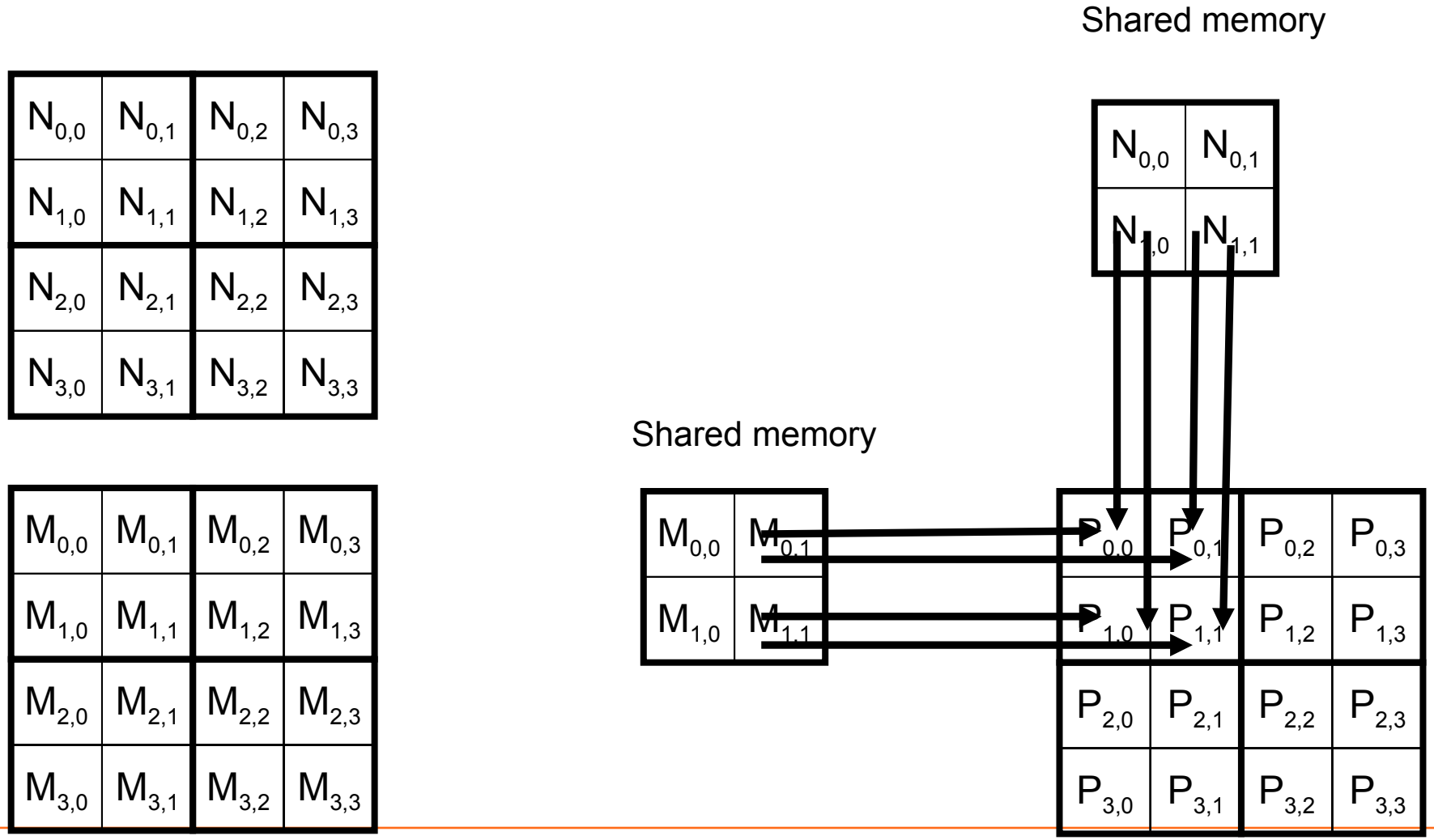
Processing block (0,0)



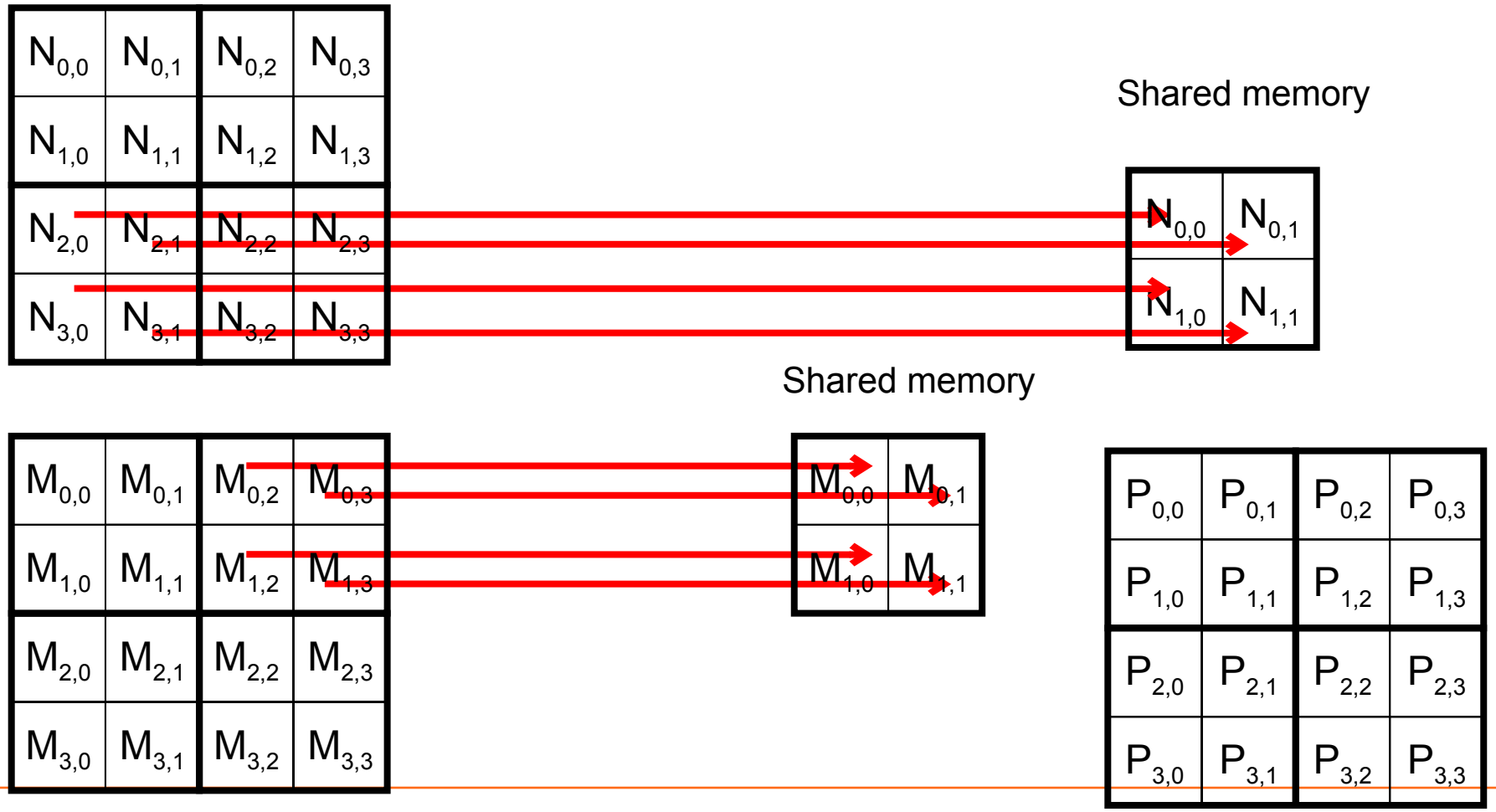
Processing block (0,0)



Processing block (0,0)



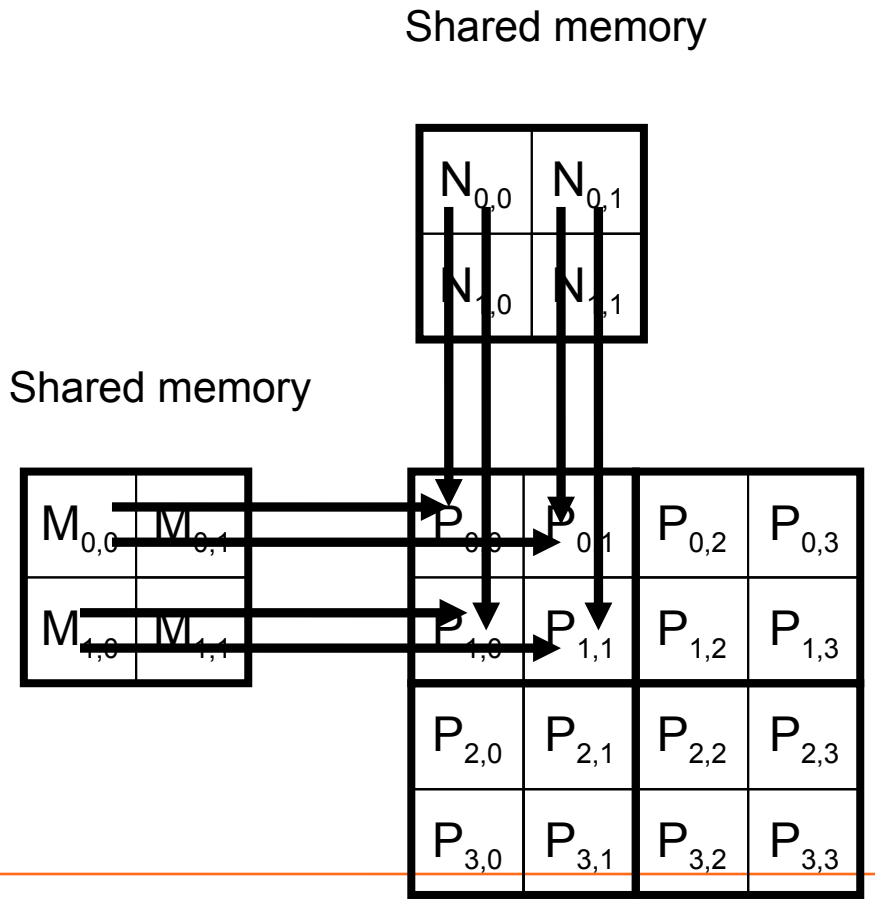
Processing block (0,0)



Processing block (0,0)

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

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



Required synchronization construct: Barrier

CUDA API function call

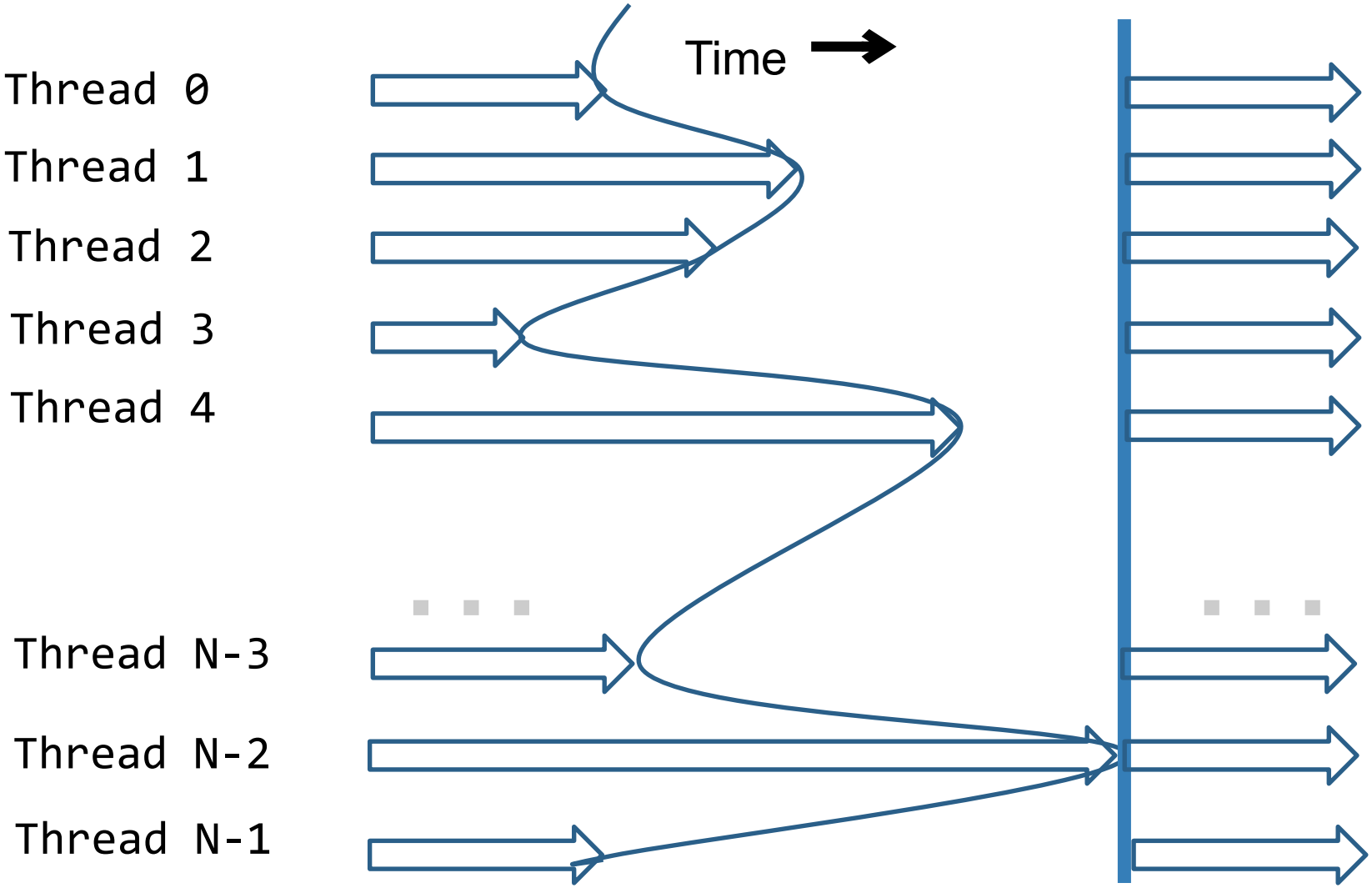
__syncthreads()

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

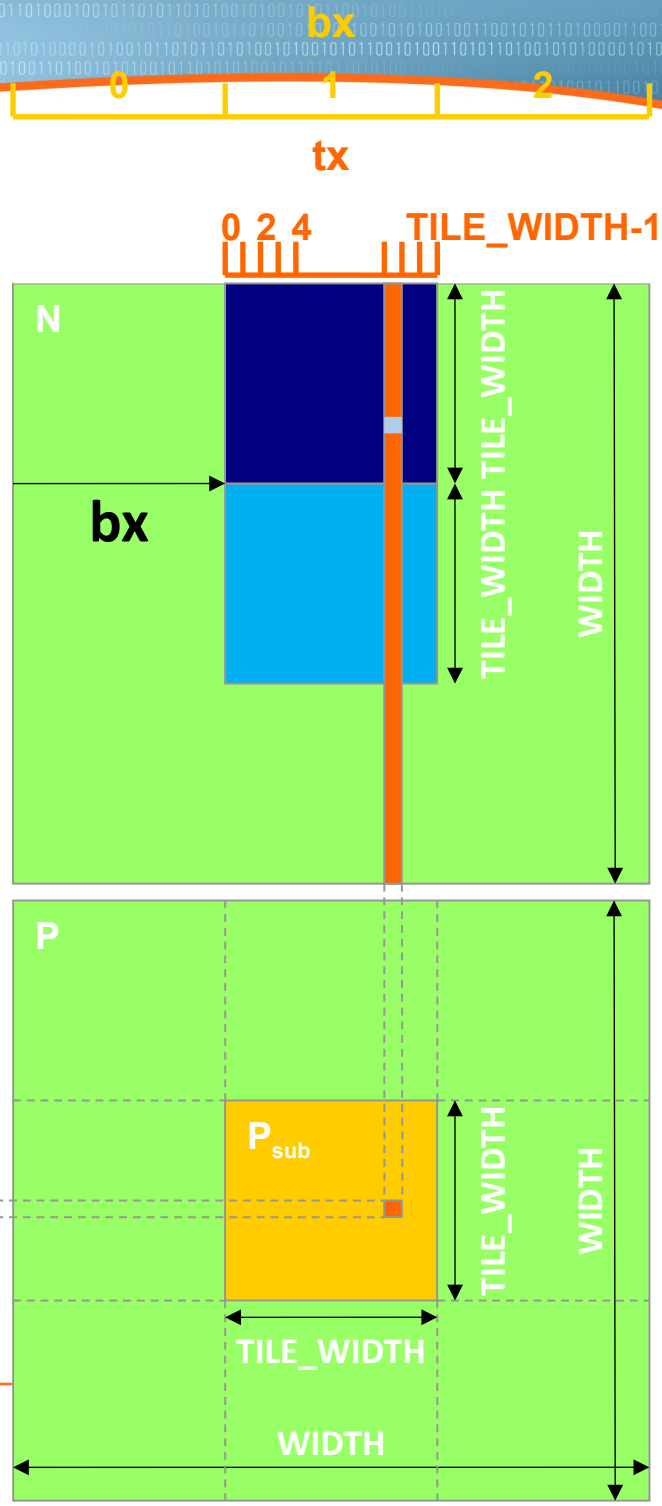
Barrier


Time
Thread 0
Thread 1
Thread 2
Thread 3
Thread 4
Thread N-3
Thread N-2
Thread N-1

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

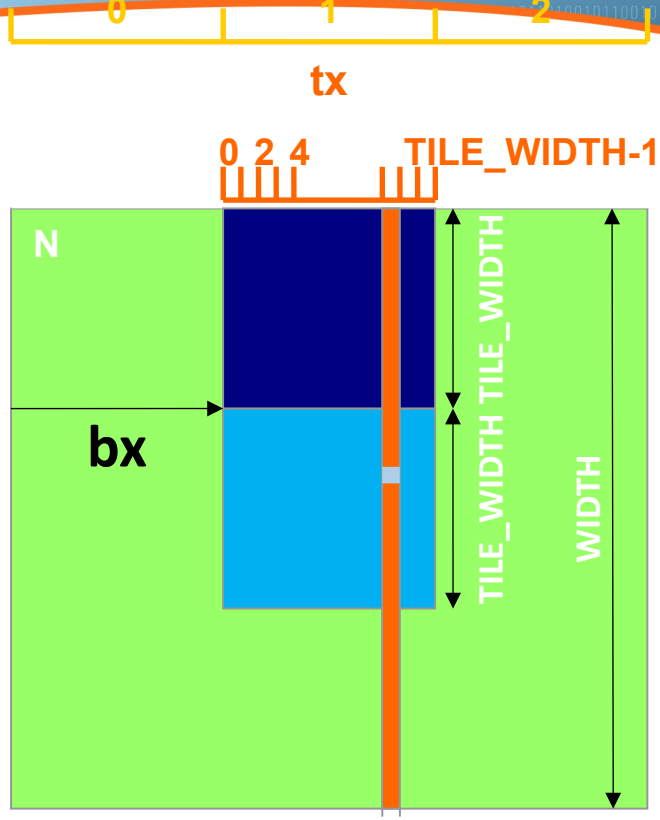
Accessing tile 1 with 2D addressing:

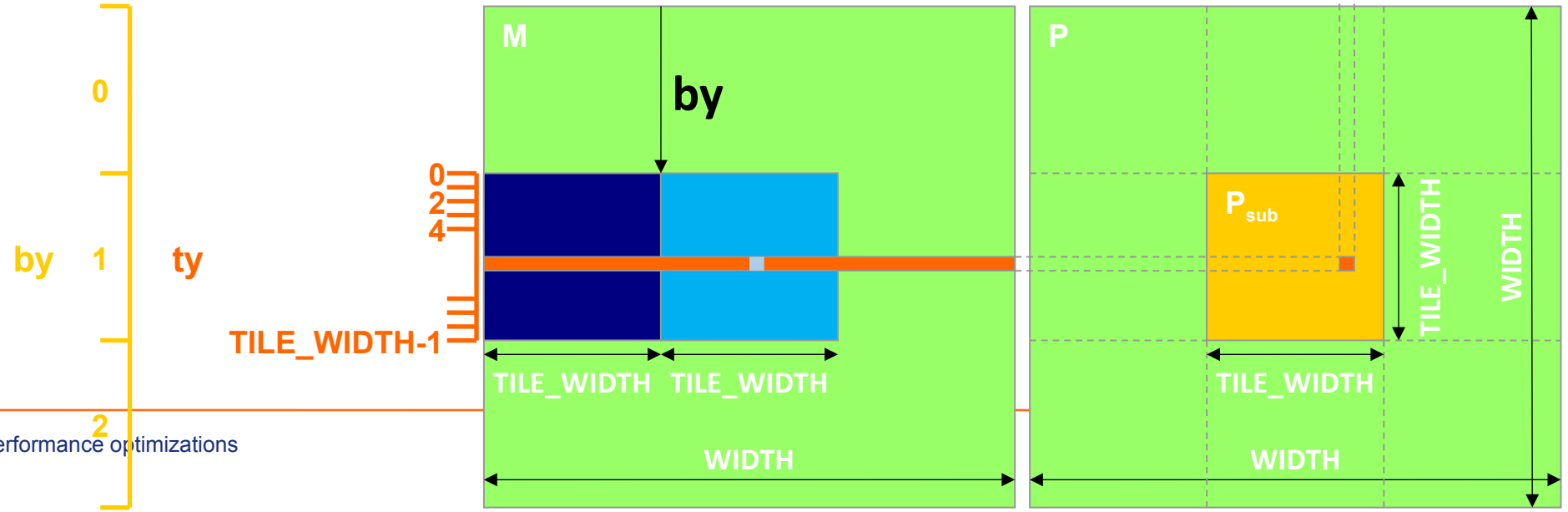
```
M[Row][1 * TILE_WIDTH + tx]
```

*Row = by * TILE_WIDTH + ty*

```
N[1 * TILE_WIDTH + ty][Col]
```

*Col = bx * TILE_WIDTH + tx*





Generalization: How to access tile m

Accessing tile m with 2-D addressing:

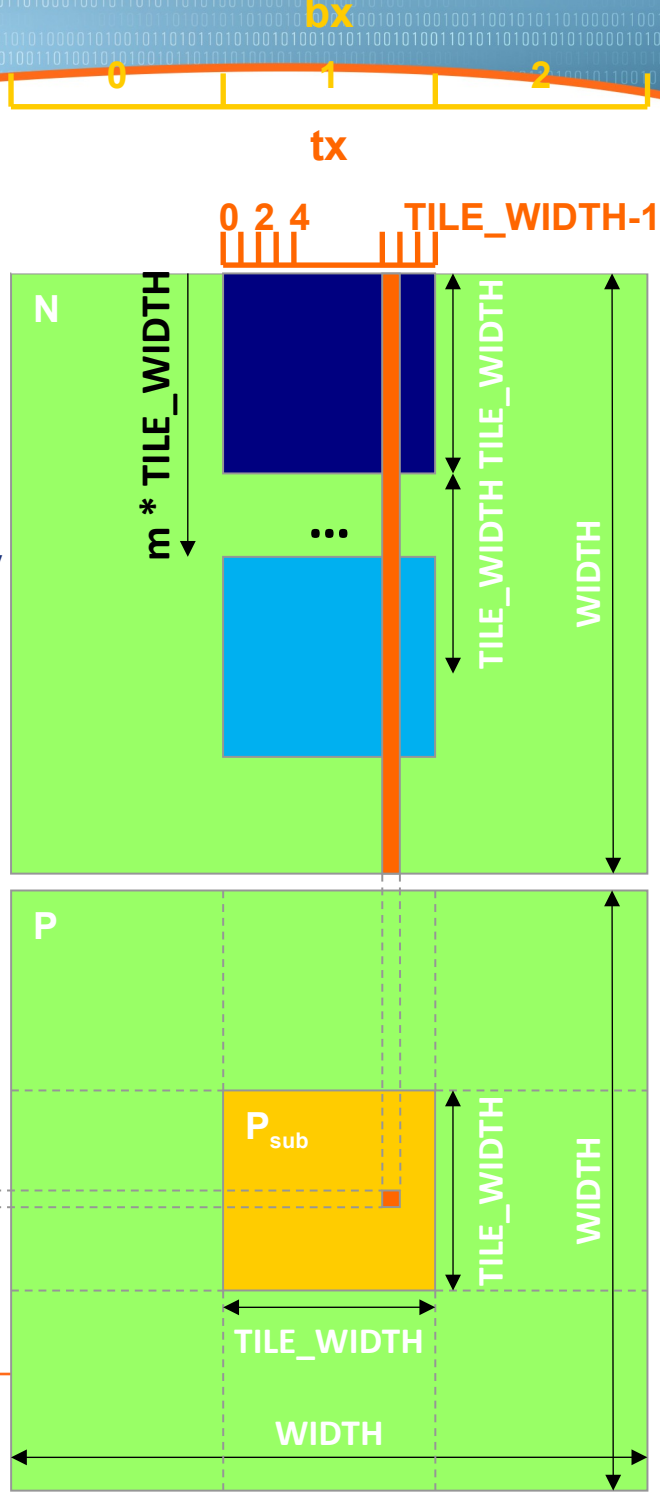
```
M[Row][m * TILE_WIDTH + tx]
N[m * TILE_WIDTH + ty][Col]
```

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)
{
1.   __shared__ float M_ds[TILE_WIDTH][TILE_WIDTH];
2.   __shared__ float N_ds[TILE_WIDTH][TILE_WIDTH];

3.   int bx = blockIdx.x;
4.   int by = blockIdx.y;
5.   int tx = threadIdx.x;
6.   int ty = threadIdx.y;

     // Identify the row and column of the Pd element to work on
7.   int Row = by * TILE_WIDTH + ty;
8.   int Col = bx * TILE_WIDTH + tx;
9.   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
10.  for (int m = 0; m < Width / TILE_WIDTH; m++) {

       // Colaborative loading of M_ds and N_ds tiles into shared memory
11.    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();

12.    for (int k = 0; k < TILE_WIDTH; k++) {
13.         Pvalue += M_ds[ty][k] * N_ds[k][tx];
14.    }

15.    __synchthreads();

16.  }

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

Comparison with the initial computational kernel

```
__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
  int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
  // Calculate the column index of the P_d element and N
  int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

  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;
}
```

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

Coalesced memory accesses

Global memory bandwidth

What we need

What we get

Memory

SRAM: (small, fast)

Registers, cache memory, …

DRAM: (large, slow)

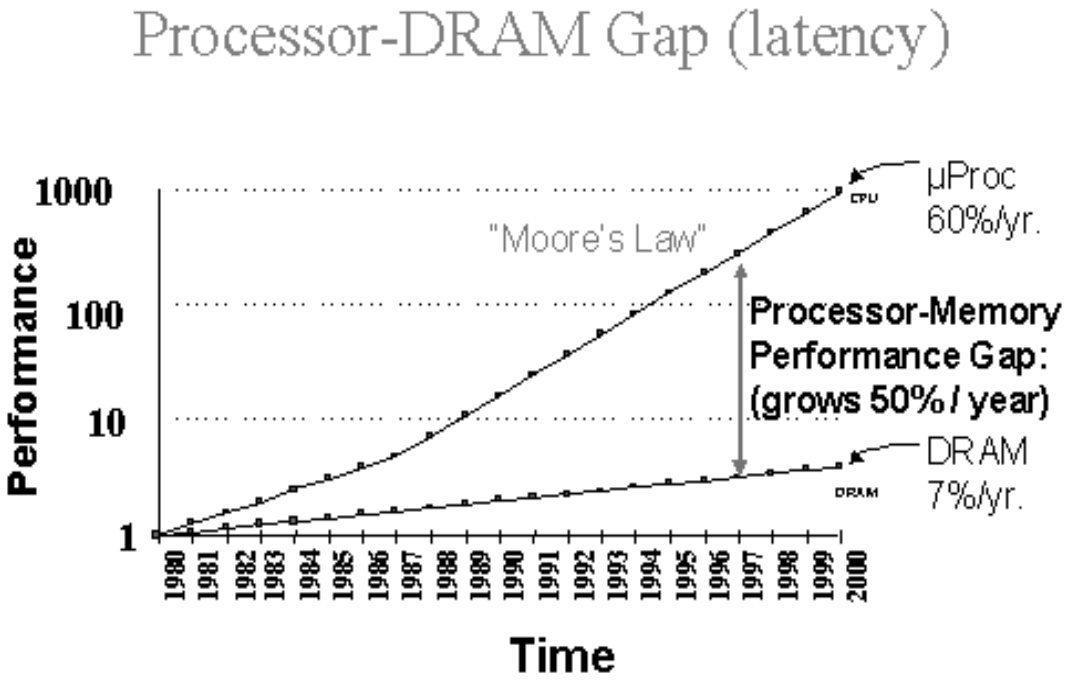
Main memory

Hiding the difference:

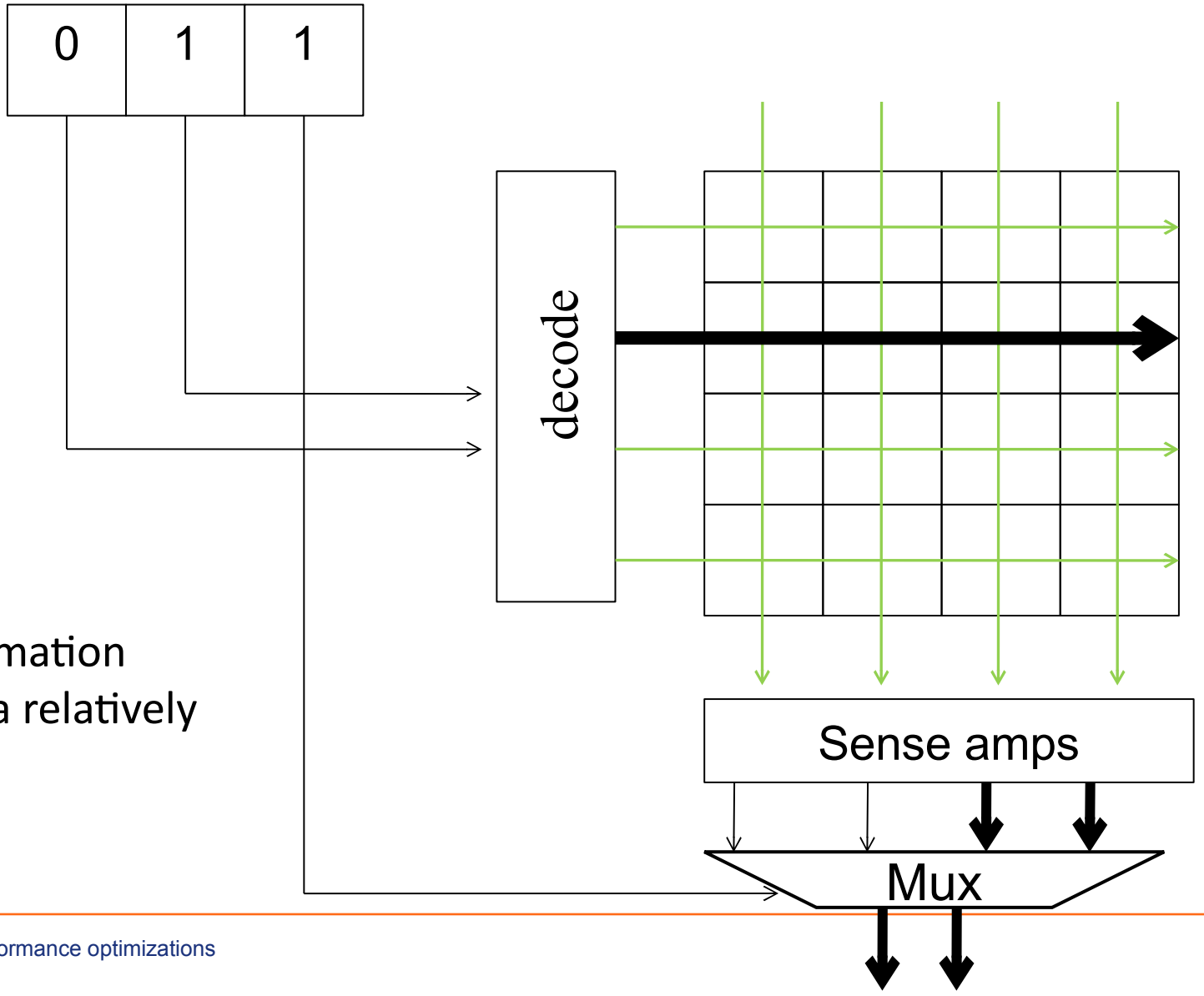
Larger caches

Breakthroughs to increase bandwidth:

- *SDRAM - synchronous DRAM*
- *RDRAM - Rambus DRAM*
- *EDORAM - extended data out SRAM*
- *multibank DRAM*

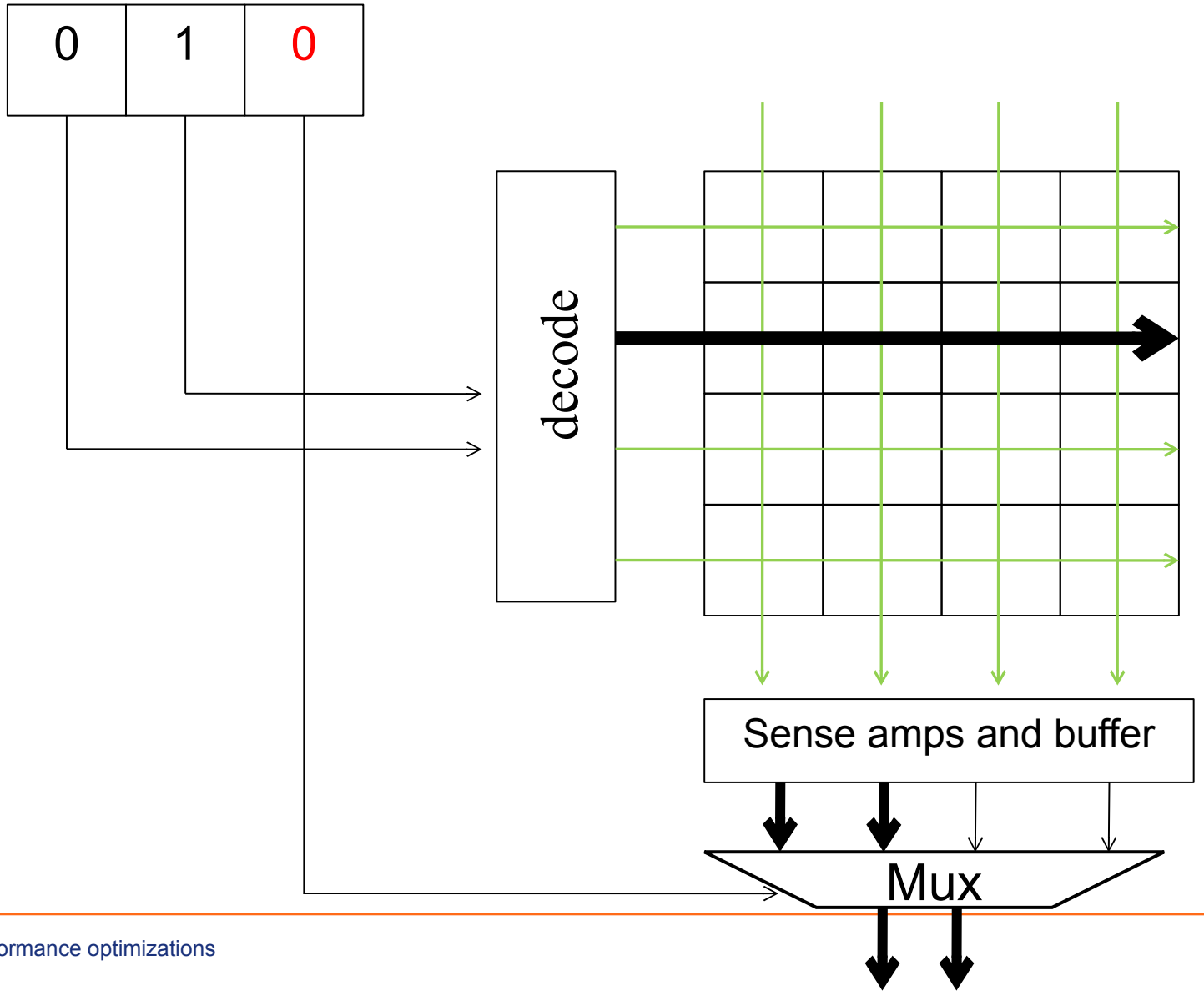


A very small (8x2 bit) DRAM Bank

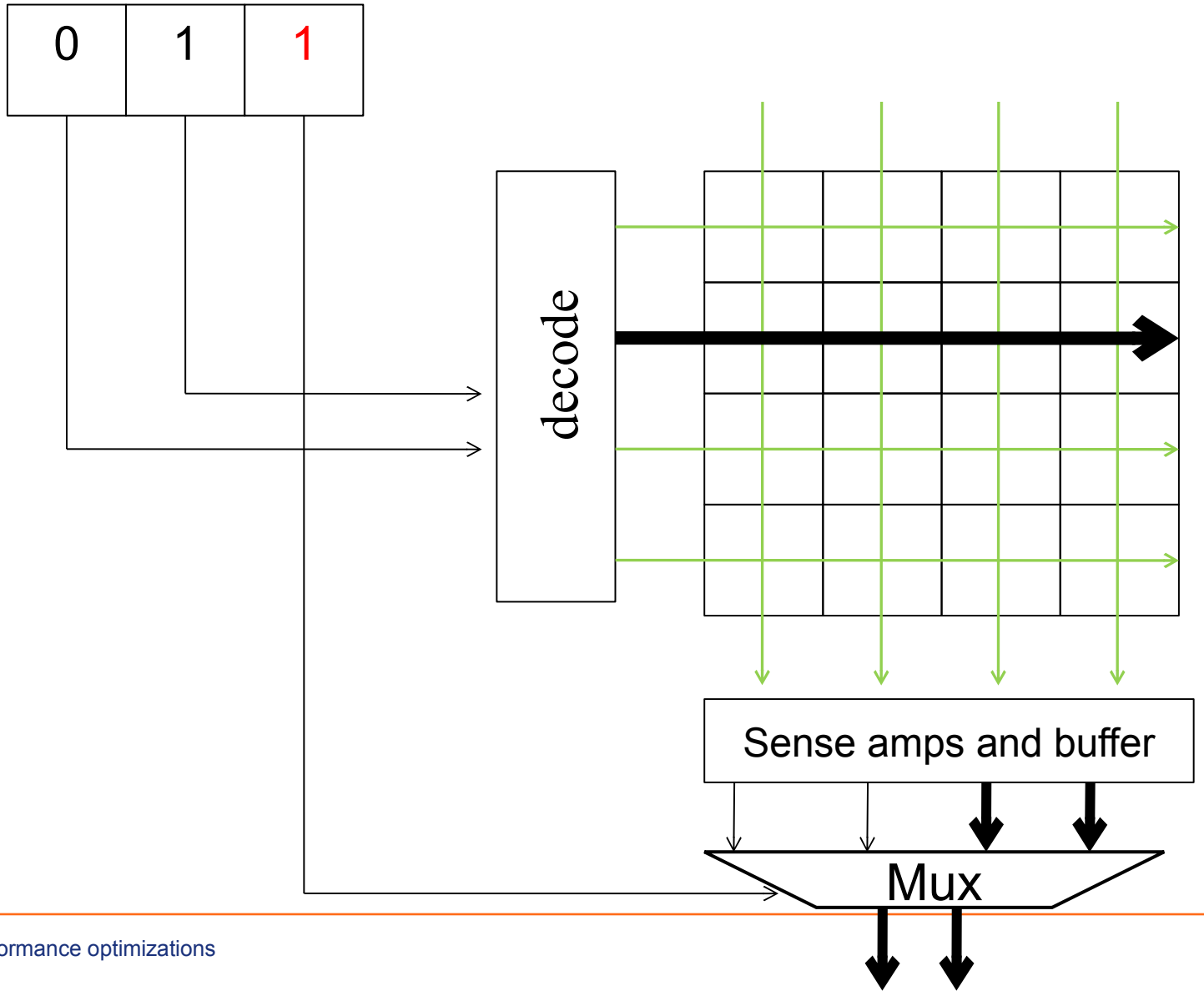


Reading information from a cell is a relatively slow process

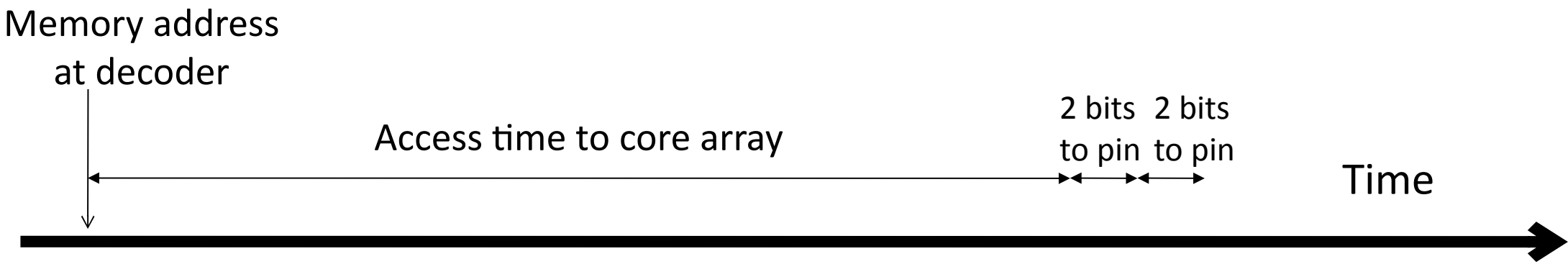
DRAM Bursting



DRAM Bursting



DRAM Bursting for a 8x2 Bank



Access 2 words – Timing without bursting

Access 2 words – Timing with bursting

Modern DRAMs are designed to always access data with bursting. Additional bytes are stored in the buffer, but discarded if accesses to memory addresses are not coalesced.

Efficiently accessing global memory

I/O significantly contributes to the execution time of an application

- Accessing global memory is the slowest I/O operation of the GPU

We need to access global memory as less as possible

When we access it we need to do it as efficiently as possible

- Coalesced memory accesses
  - *Access continuous memory addresses from continuous threads within a warp*

Some details about data transfers

Data transfers occur at:

- Half-warp if Compute Capability == 1.x
- Warp if Compute Capability ≥ 2.0

Accesses are carried out at blocks of 32, 64 and 128 bytes

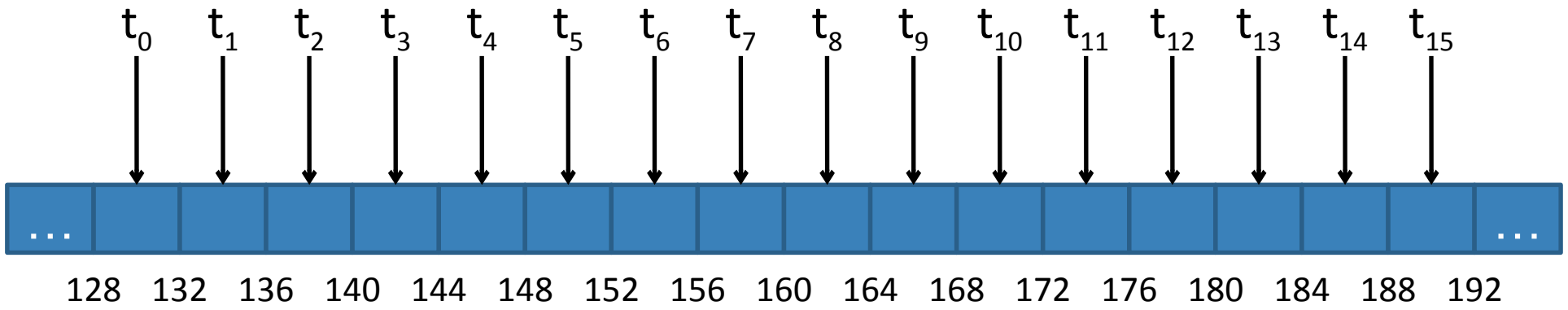
- The first memory address of the block is always aligned correspondingly

How many memory transactions will be performed depends on the Compute Capability

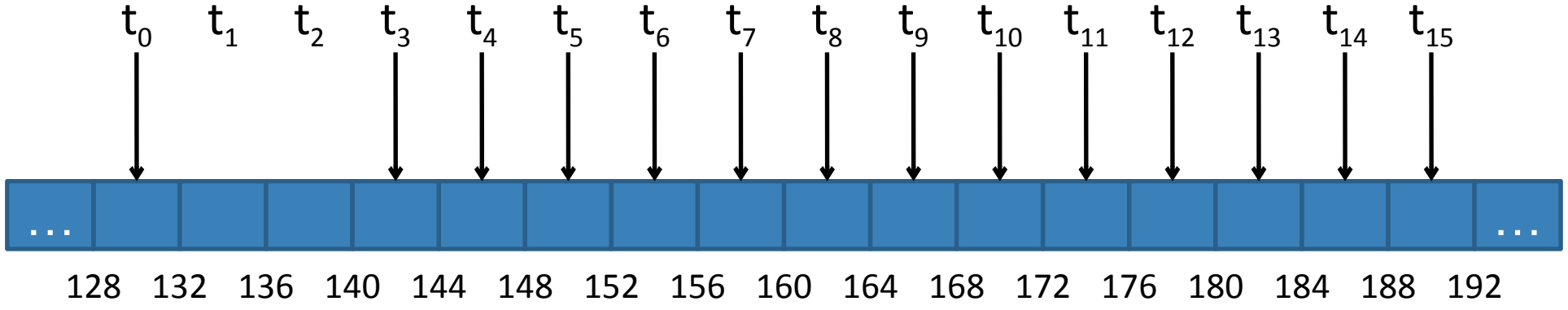
- More strict for 1.0 and 1.1
- Less strict for ≥1.2

In out examples we will use float variables (4 bytes)

Compute Capability 1.0 – 1.1: Coalesced memory accesses

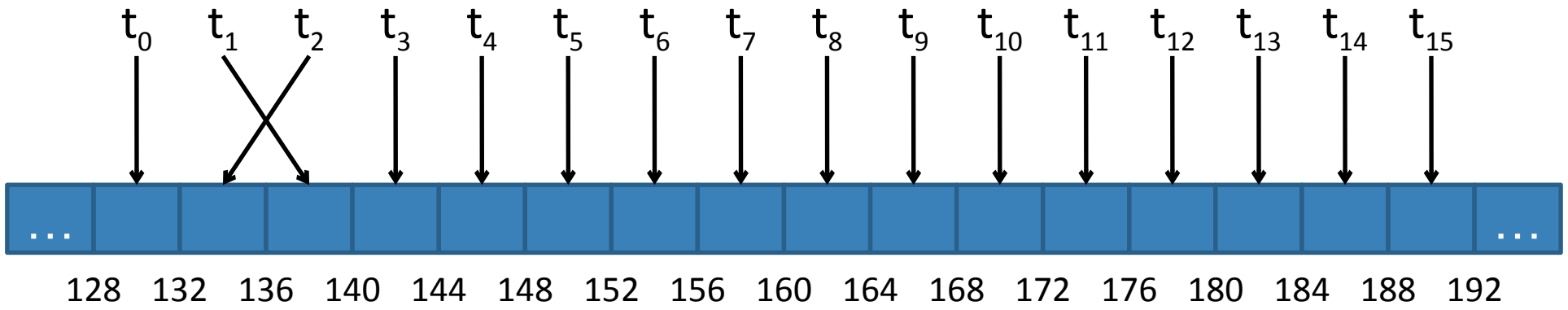


All threads participate

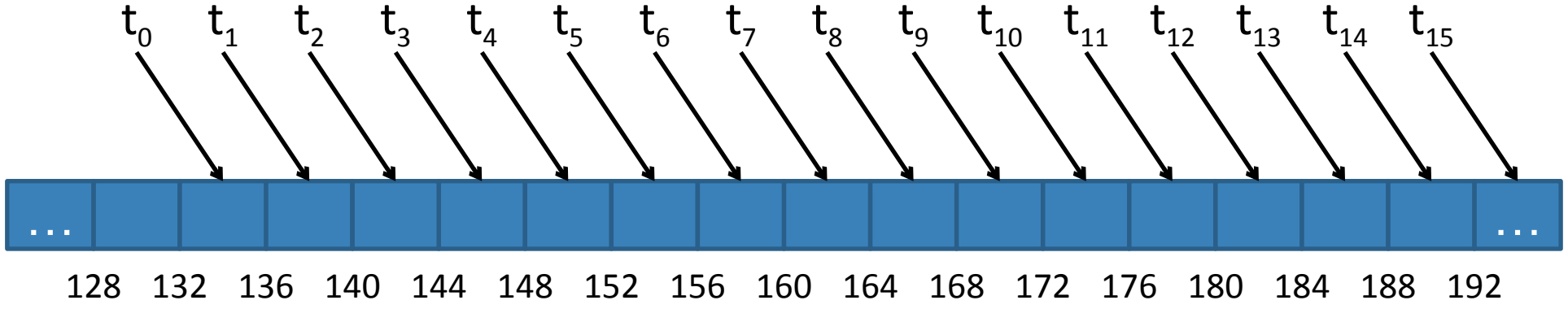


Not all threads participate

Compute Capability 1.0 – 1.1: Uncoalesced memory accesses (1/2)

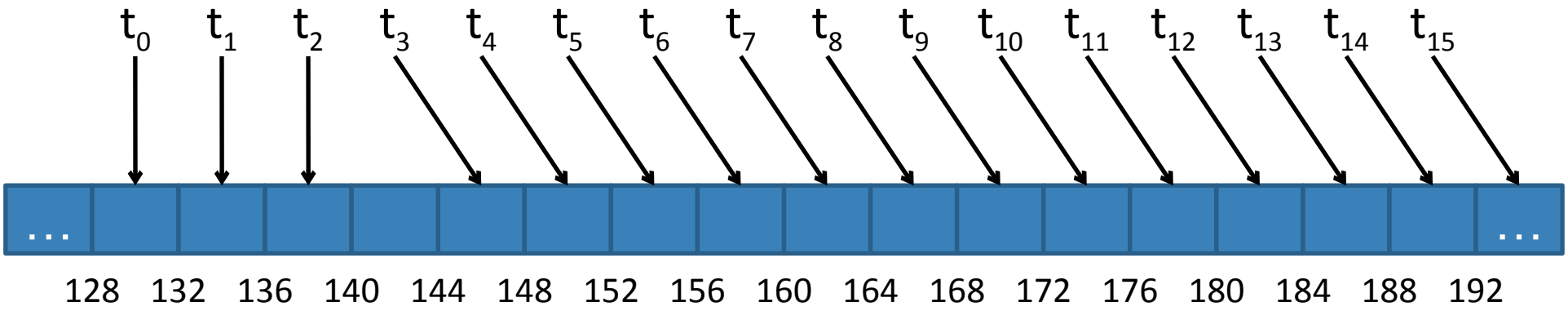


Accesses with permutations within a block – 16 transactions



Misaligned initial address – 16 transactions

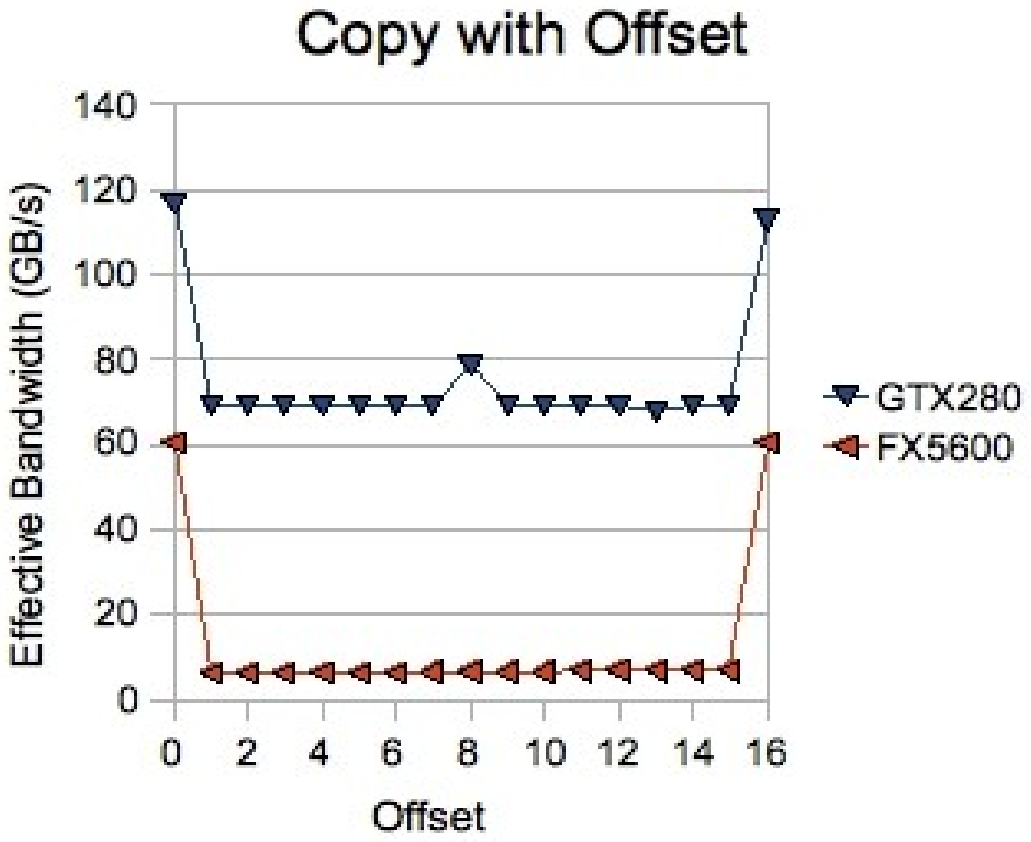
Compute Capability 1.0 – 1.1: Uncoalesced memory accesses (2/2)



Non-continuous accesses – 16 transactions

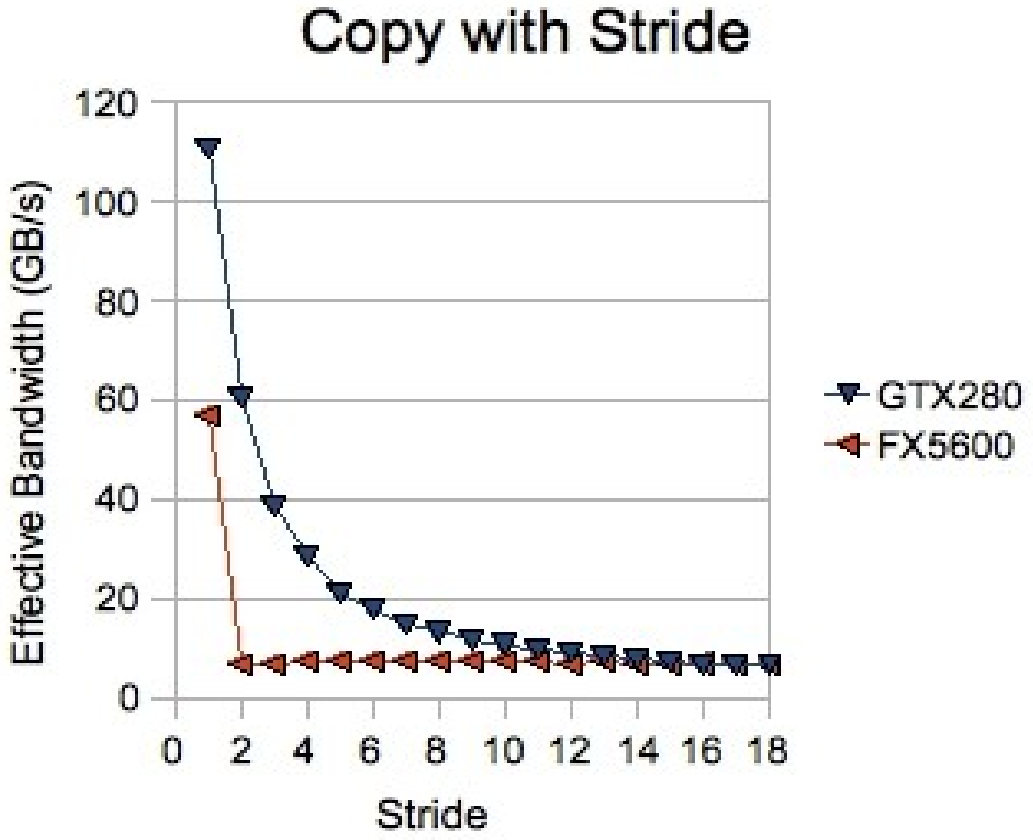
How important is coalescing?

```
__global__ void offsetCopy(float *odata, float *idata, int offset)
{
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}
```



Copying with a stride

```
__global__ void strideCopy(float *odata, float *idata, int stride)
{
    int xid = (blockIdx.x * blockDim.x + threadIdx.x) * stride;
odata[xid] = idata[xid];
}
```

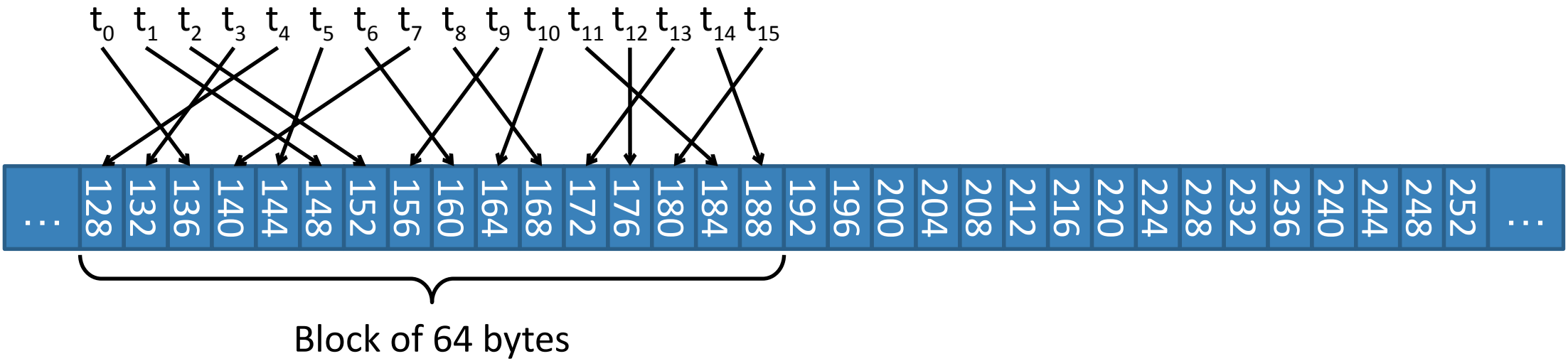


Compute Capability 1.2

Continuous or random access within a 64 byte block

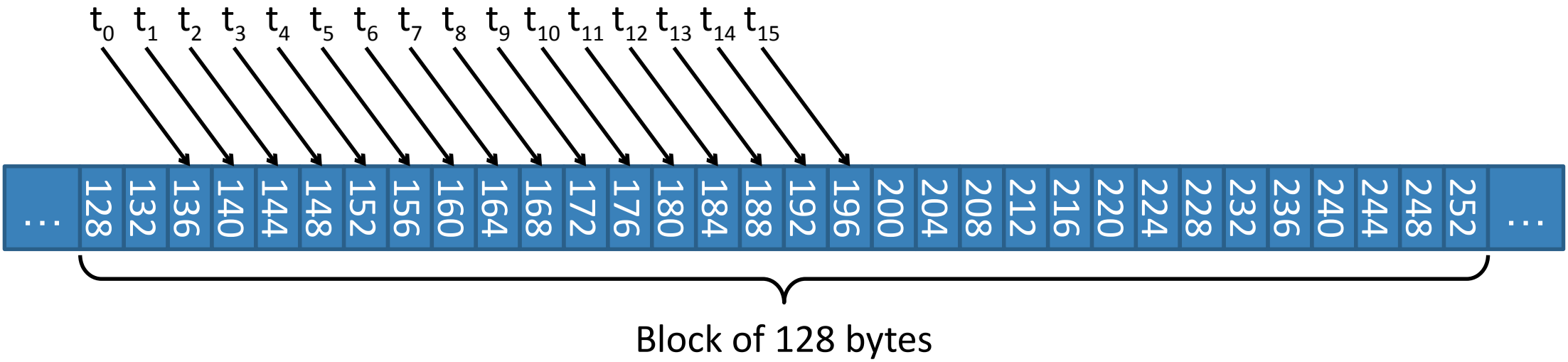
1 transaction of a 64 byte block

Same for Compute Capability ≥ 2.0 and 32 threads



Misaligned first memory address

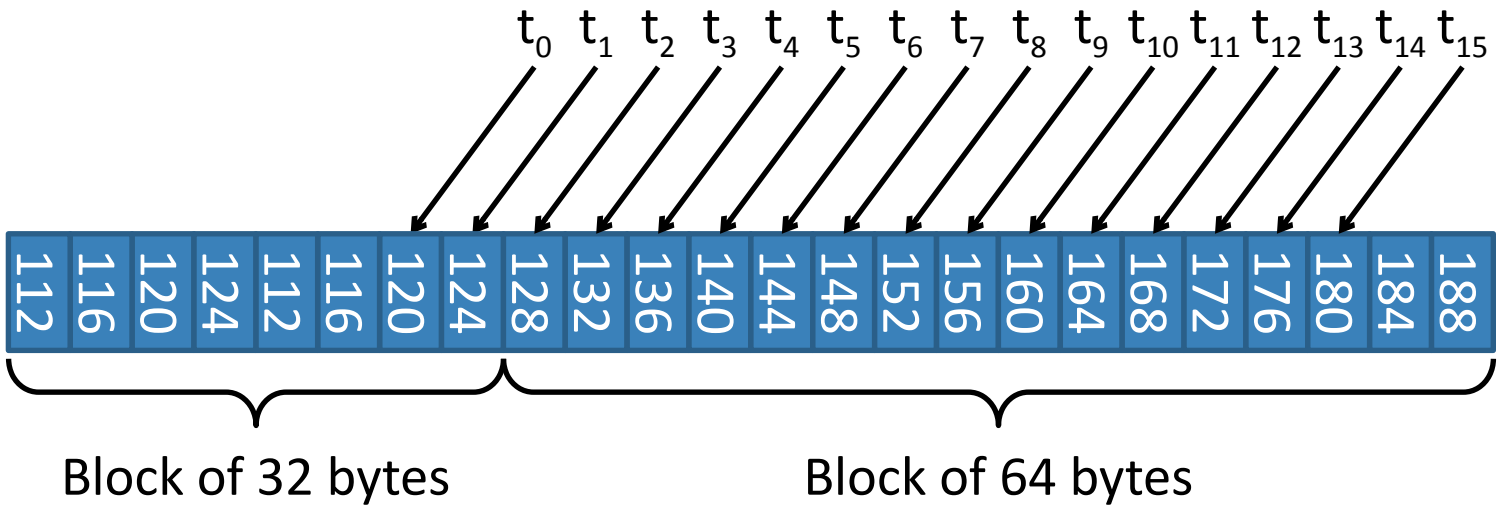
1 transaction of a 128 byte block



Compute Capability 1.2

Misaligned first memory address

- 2 transaction
  - *1 64 byte block*
  - *1 32 byte block*



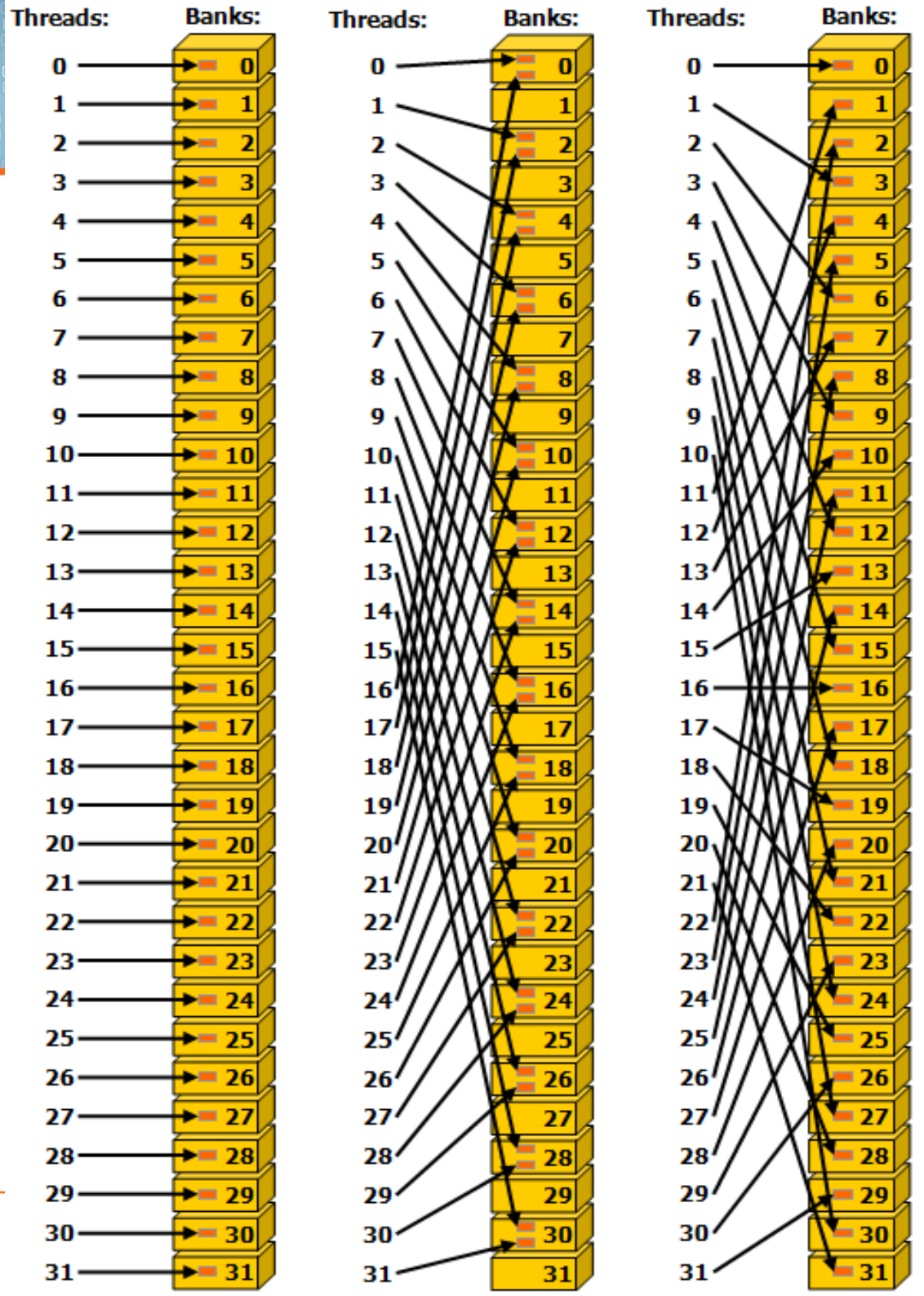
Bank conflicts in shared memory

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*

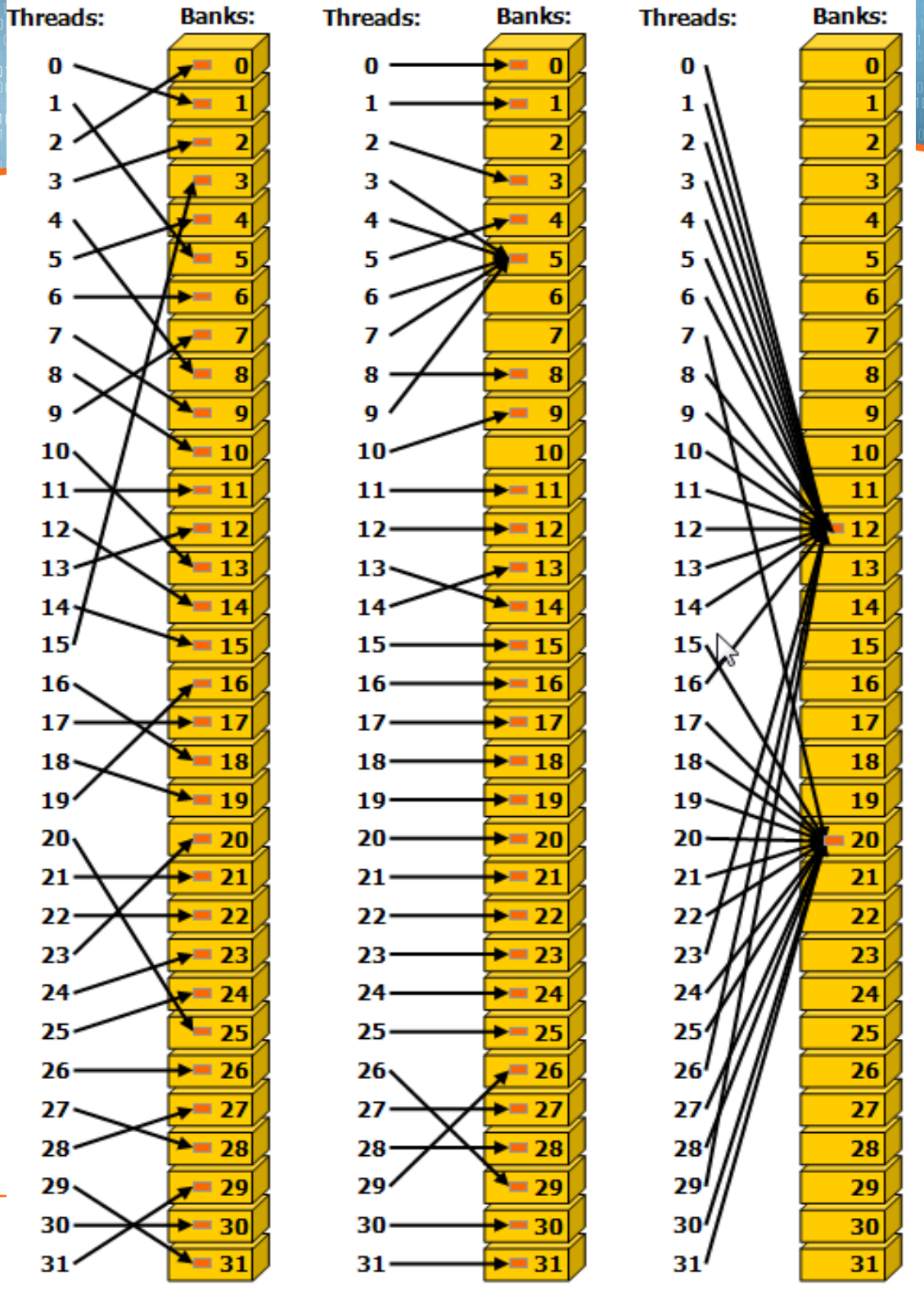
Example

- 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



Flow control

Flow control

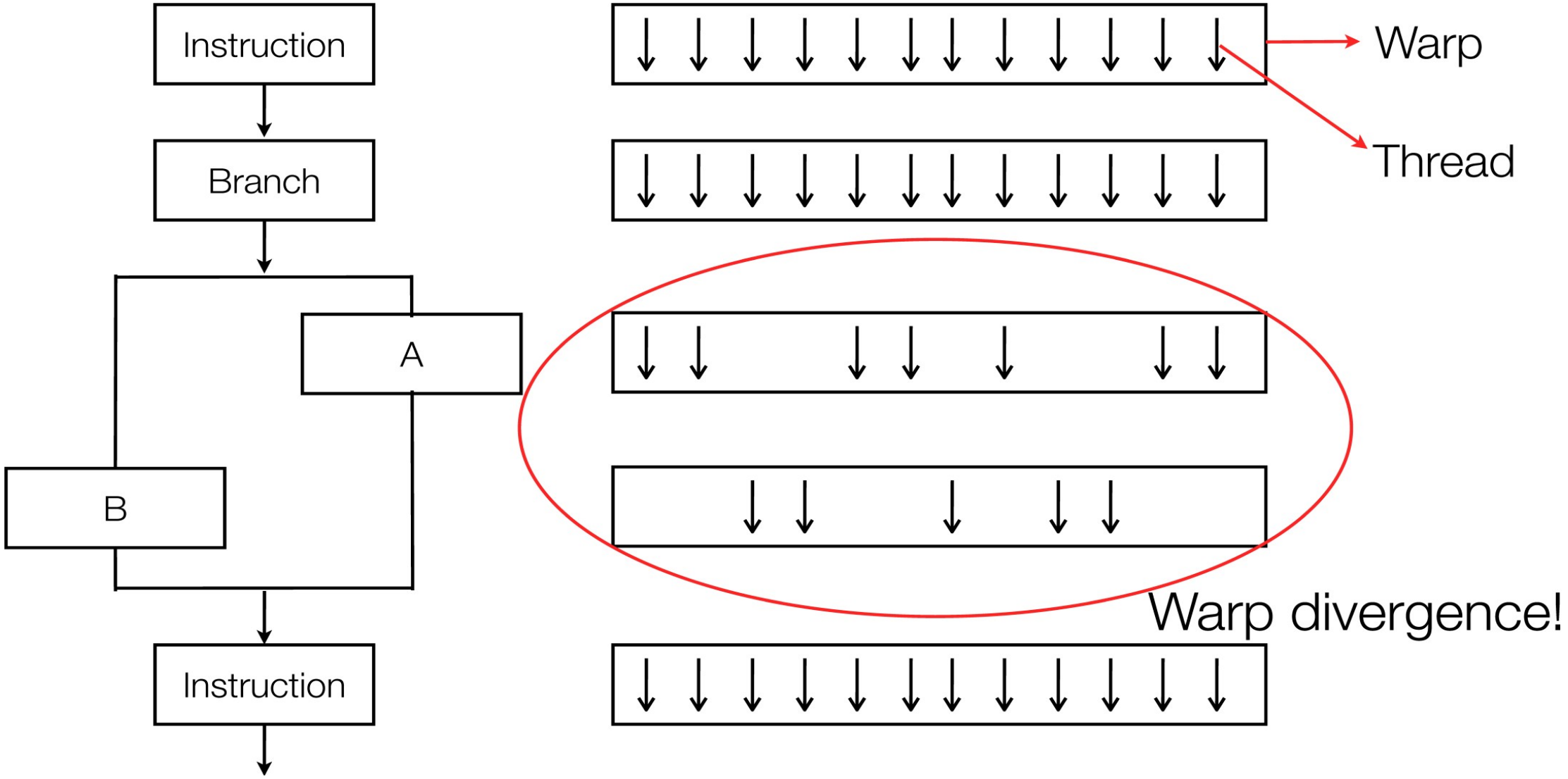
'if…else' instruction

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

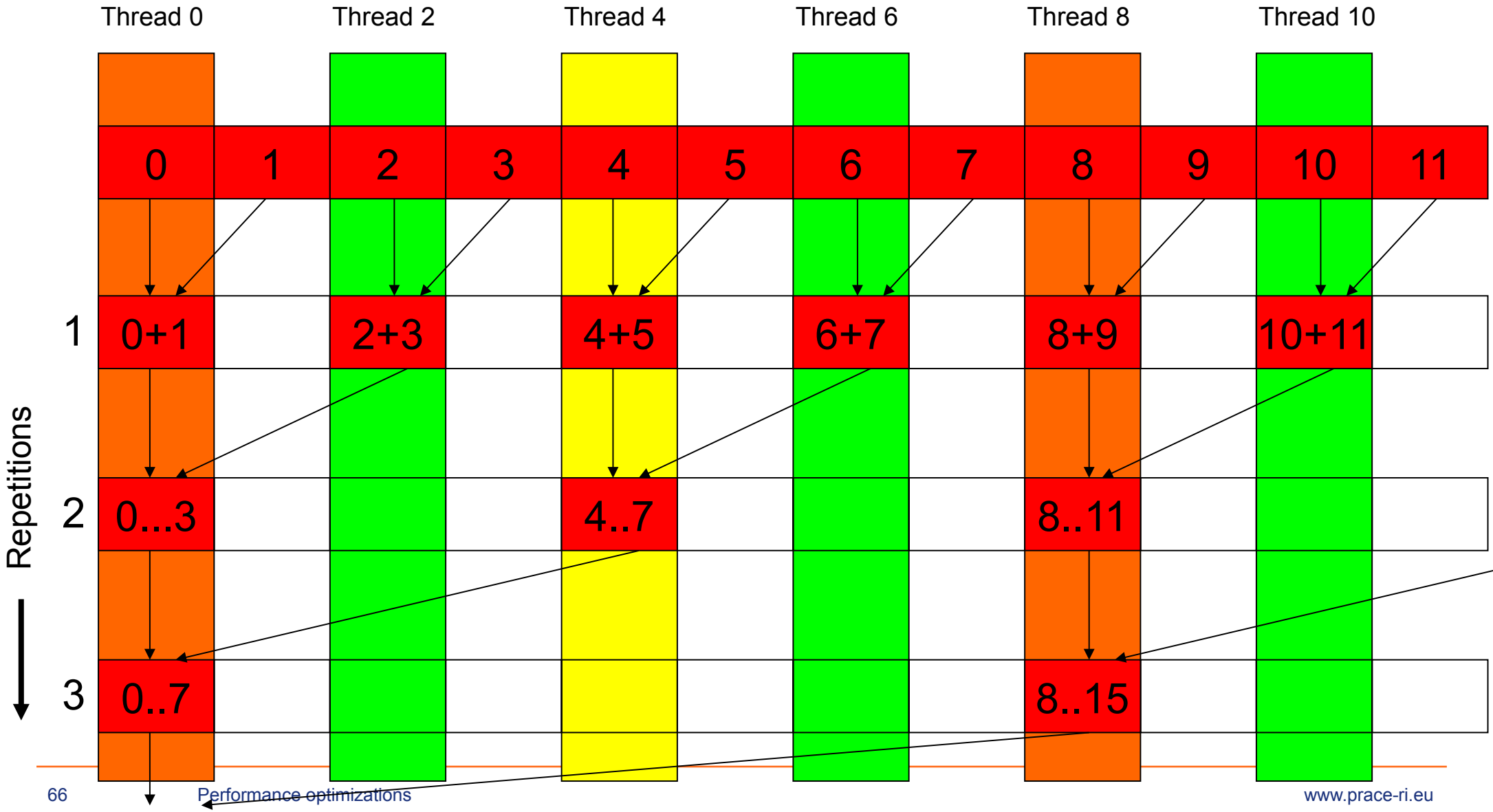
A first implementation

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*

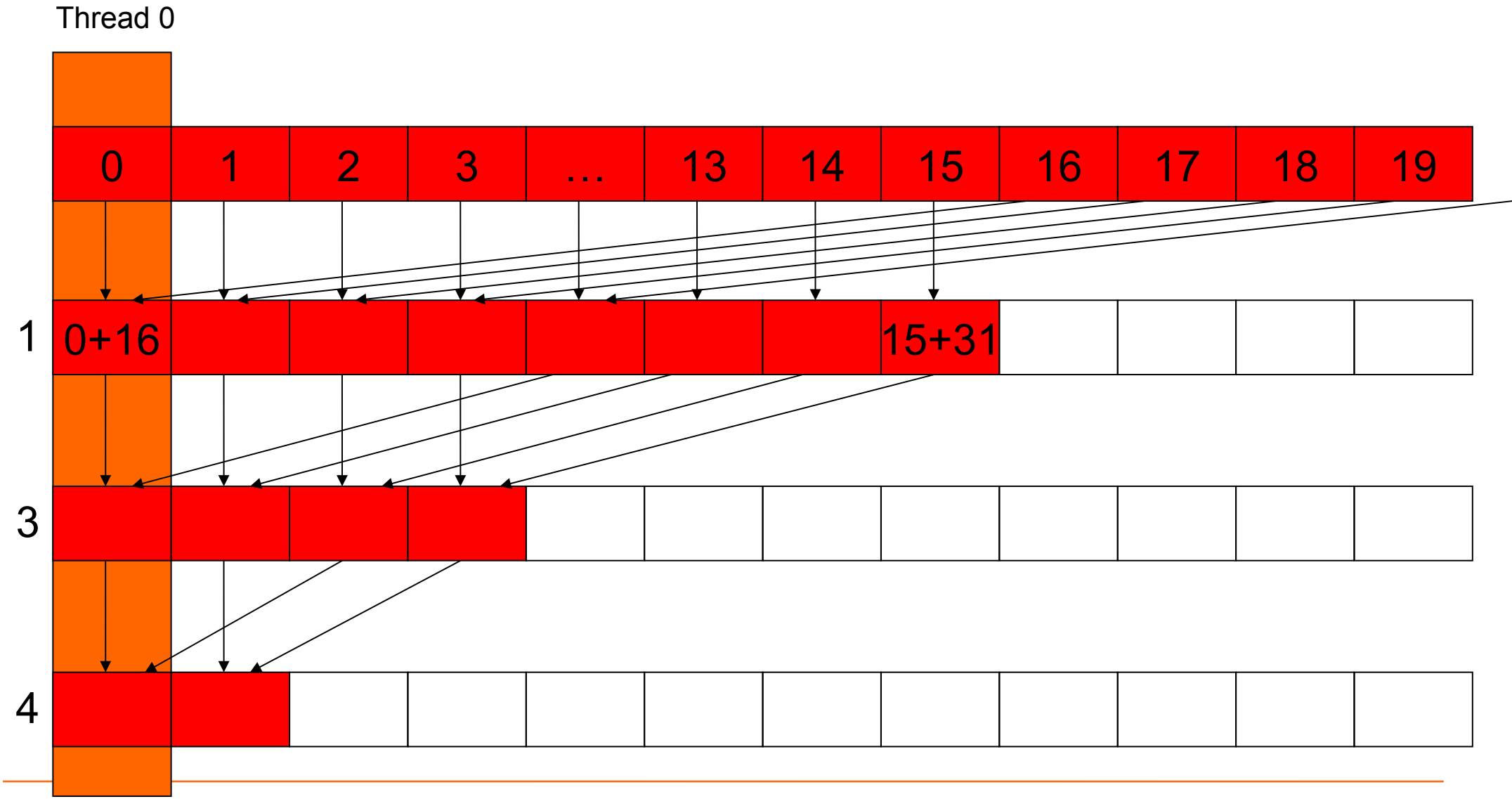
At most half of the threads in a warp contribute towards calculating the result

- 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

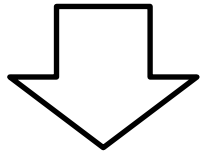


Example for blocks of 512 threads

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

Atomic instructions

A typical real-life example

- Many employees serve customers
  - Every customer receives a number
    - *We assume there are multiple machines that provide numbers*
  - A central display shows the number of the customer who will be served next
- When an employee is free, the next customer is called by adding 1 to the number on the display
- Possible problems
  - Many customers might receive the same number
  - Manu employees might call the same number

Atomic instructions

Thread 1:	Old ← Mem[x]		Thread 2:	Old ← Mem[x]
	New ← Old + 1			New ← Old + 1
	Mem[x] ← New			Mem[x] ← New

If memory location Mem[x] contains initially 0, what will be the value of Mem[x] when both threads complete execution?

The result depends on the relative execution order of instructions between the two threads

To get the correct result we must use atomic instructions

- If this is possible
- Otherwise other mechanisms must be used (mutex, etc.)
  - *Which typically are implemented using atomic instructions*

1st execution scenario

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

Thread 1: Old = 0

Thread 2: Old = 1

Mem[x] = 2

2nd execution scenario

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3		(1) Mem[x] ← New
4	(1) Old ← Mem[x]	
5	(2) New ← Old + 1	
6	(2) Mem[x] ← New	

Thread 1: Old = 1

Thread 2: Old = 0

Mem[x] = 2

3rd execution scenario

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Thread 1: Old = 0

Thread 2: Old = 0

Mem[x] = 1

4th execution scenario

Time	Thread 1	Thread 2
1		(0) Old ← Mem[x]
2		(1) New ← Old + 1
3	(0) Old ← Mem[x]	
4		(1) Mem[x] ← New
5	(1) New ← Old + 1	
6	(1) Mem[x] ← New	

Thread 1: Old = 0

Thread 2: Old = 0

Mem[x] = 1

Atomic instructions to ensure correct results

Old ← Mem[x]	
New ← Old + 1	
Mem[x] ← New	
	Old ← Mem[x]
	New ← Old + 1
	Mem[x] ← New

OR

	Old ← Mem[x]
	New ← Old + 1
	Mem[x] ← New
Old ← Mem[x]	
New ← Old + 1	
Mem[x] ← New	

Atomic instructions

A special assembly instruction that performs an operation on a memory address

- Read the old value at the memory address, calculate the new value, store the new value at the memory address

The hardware ensures that no other thread can access the memory address while the atomic instruction is executed

- Every other thread that tries to access the memory address will suspend execution
- In the end, all threads execute serially the atomic instruction

Atomic instructions in CUDA

Implemented as function calls, that are translated into simple assembly instructions (intrinsics)

- Arithmetic instructions
  - *Add, Sub, Inc, Dec, Min, Max,*
  - *Exch (Exchange), CAS (Compare And Swap)*
- Bitwise operations
  - *And, Or, Xor*
- More details can be found in the CUDA C Programming Guide
  - *https://docs.nvidia.com/cuda/cuda-c-programming-guide*

Atomic addition on a signed 32-bit integer

```
int atomicAdd(int *address, int val);
```

Reads the current 32-bit value pointed to by **address** (global or shared memory)

Calculate the value **(old + val)**

Store the new value at the same memory address

Return the previous value that was stored at the address

More atomic instruction for addition in CUDA

Atomic addition on an unsigned 32-bit integer

```
unsigned int atomicAdd(unsigned int *address,
                       unsigned int val);
```

Atomic addition on a signed 64-bit integer

```
unsigned long long int atomicAdd(
                    unsigned long long int *address,
                    unsigned long long int val);
```

Atomic addition on a single precision floating point number (**Compute capability > 2.0**)

```
float atomicAdd(float *address, float val);
```

Atomic addition on a double precision floating point number (**Compute capability > 6.0**)

```
double atomicAdd(double *address, double val);
```

Atomic instructions and Compute Capability

As already made clear with atomicAdd(), different atomic instructions are supported for GPUs with a different Compute Capability

What if I need an atomic instruction that is not available on the GPU I have access to?

The atomic instruction atomicCAS() can be used to implement any other atomic instruction!

atomicCAS()

```
int atomicCAS(int *address, int compare, int val);
unsigned int atomicCAS( unsigned int *address,
                        unsigned int compare,
                        unsigned int val);
unsigned long long int atomicCAS(unsigned long long int *address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

Atomically:

- Read the current 32-bit or 64-bit value (`old`) stored at the memory address pointed to by `address`
- Evaluate the expression (`old == compare ? val : old`)
- Store the result at the same memory address
- Return the value `old`

Calculation of histogram

Μέθοδος για την εξαγωγή χρήσιμων χαρακτηριστικών και μοτίβων από μεγάλα σύνολα δεδομένων

- Εξαγωγή χαρακτηριστικών για την αναγνώριση αντικειμένων σε εικόνες
- Ανίχνευση απάτης σε συναλλαγές με πιστωτικές κάρτες
- Συσχέτιση κινήσεων ουράνιων σωμάτων στην αστροφυσική
- …

Βασικός αλγόριθμος

- Χρησιμοποίησε την τιμή κάθε στοιχείου του συνόλου δεδομένων ως αναγνωριστικό ενός «δοχείου», του οποίου την τιμή θα αυξήσεις κατά ένα

An example of an histogram

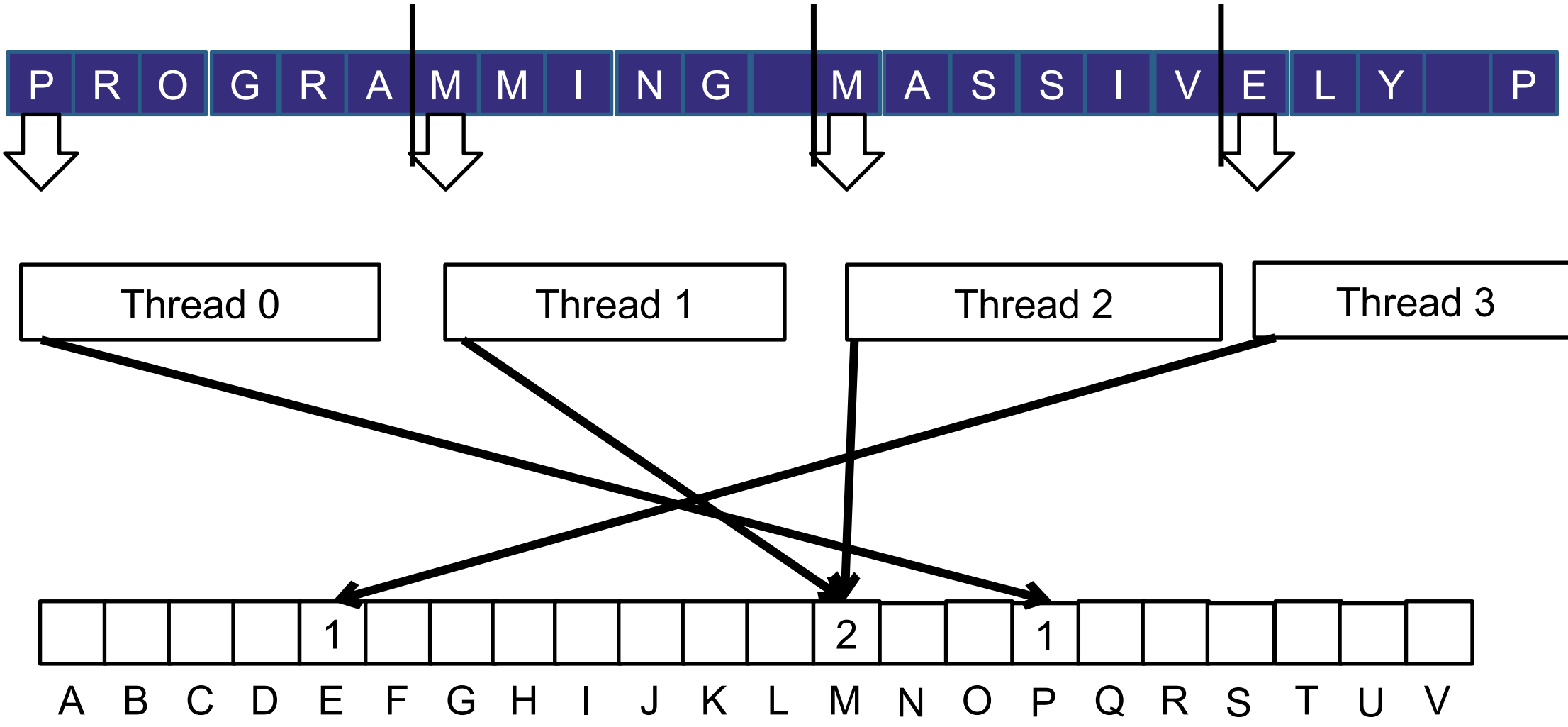
How are we calculating a histogram in parallel?

- Assign to each thread a part of the input data
- For each element, use atomic instructions to build the histogram

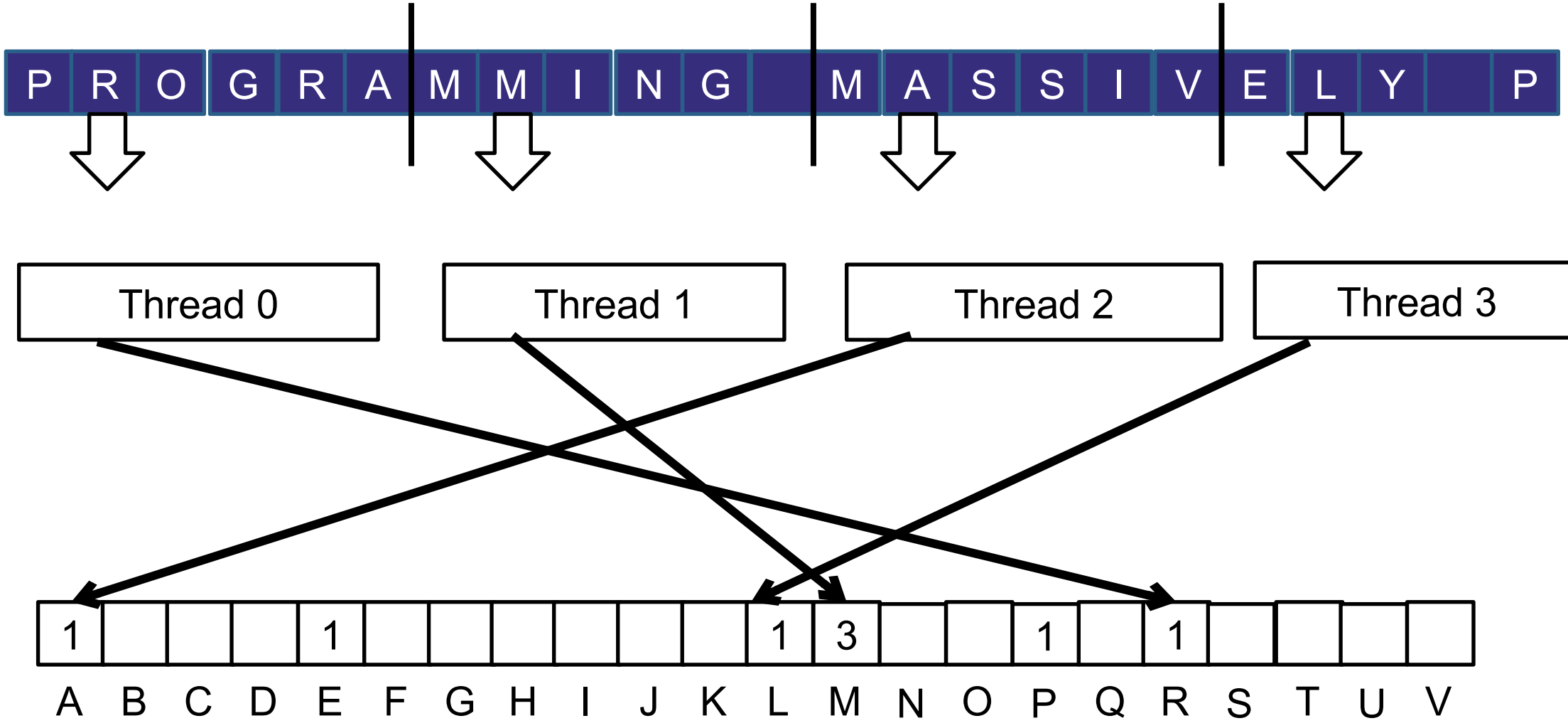
For the phrase “Programming Massively Parallel Processors” build a histogram with the frequency of each character in it

A(4), C(1), E(3), G(1), …

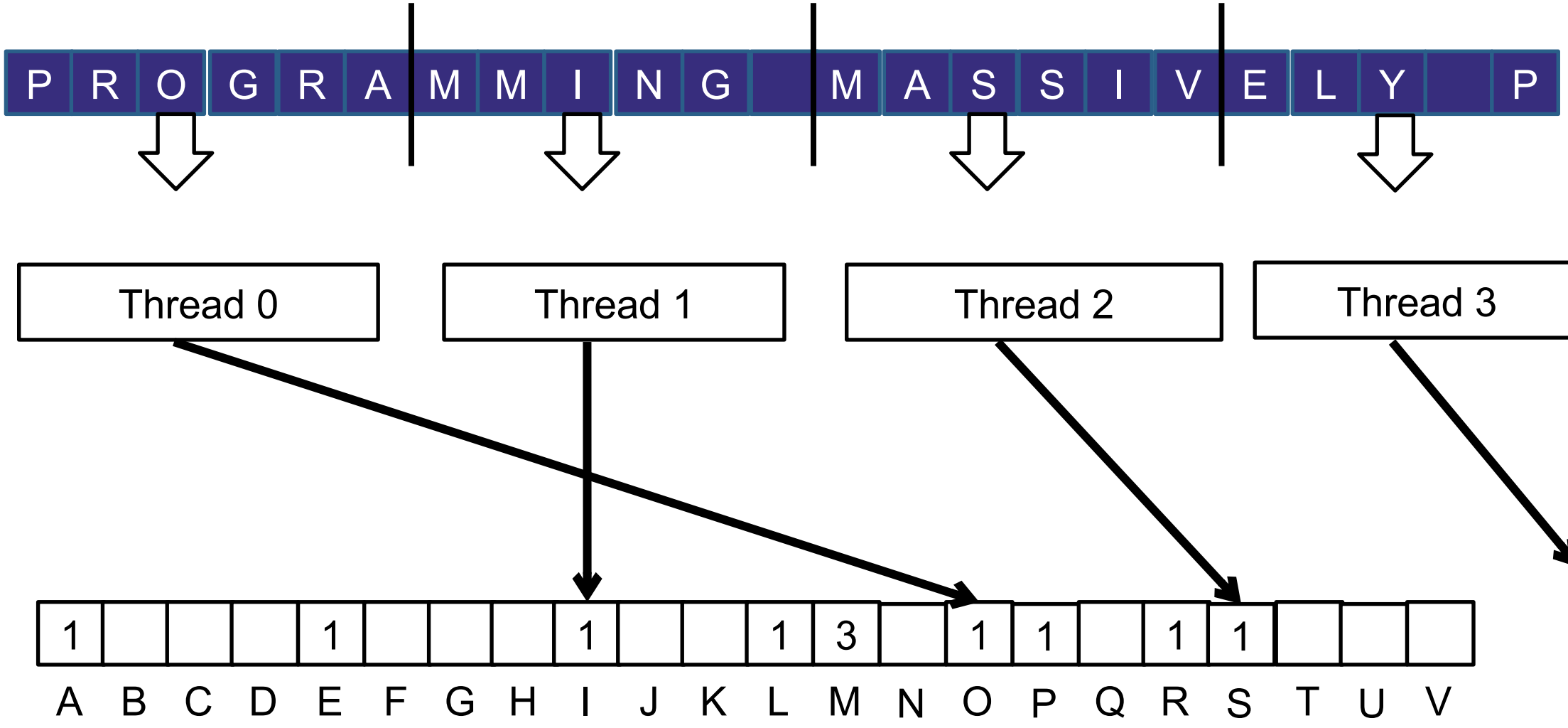
1st repetition – 1st letter in each part



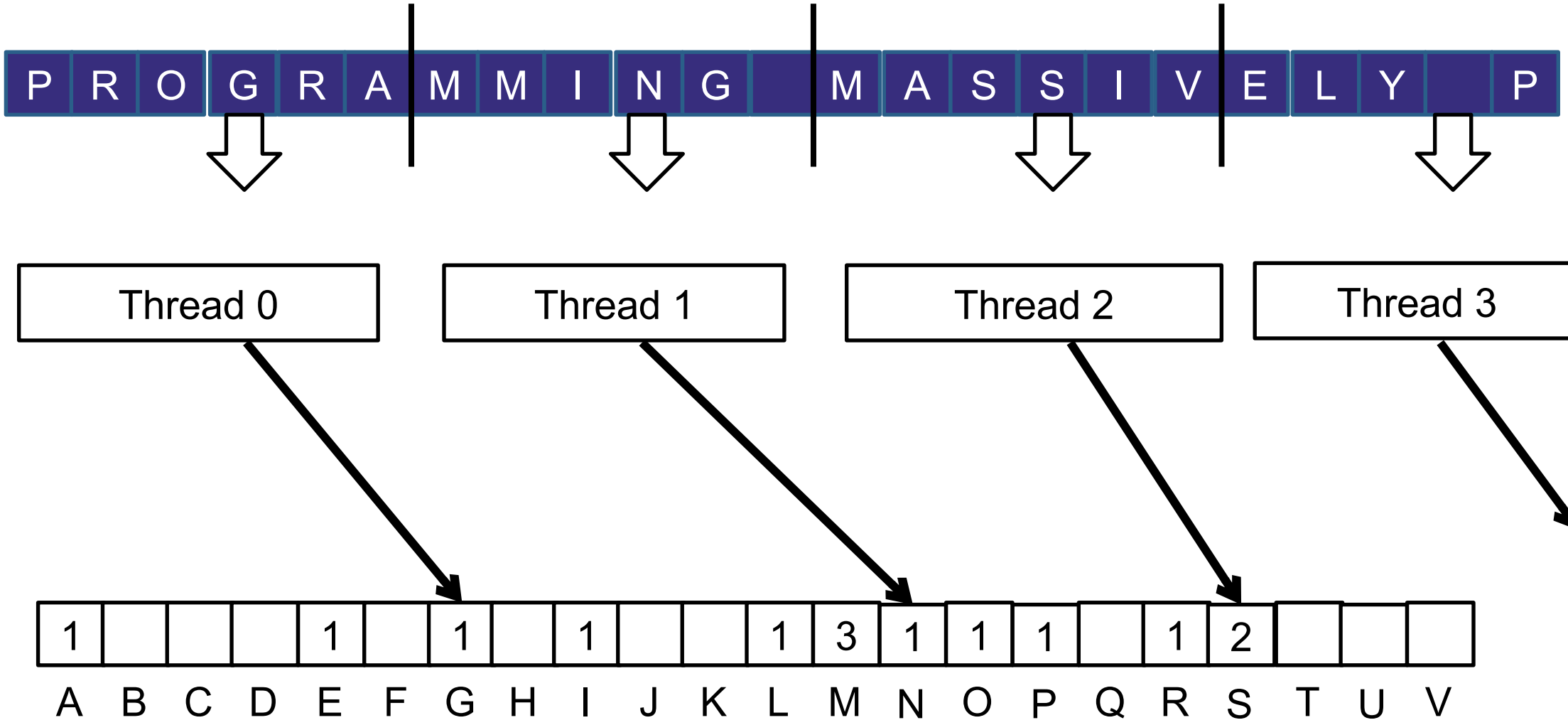
2nd repetition – 2nd letter in each part



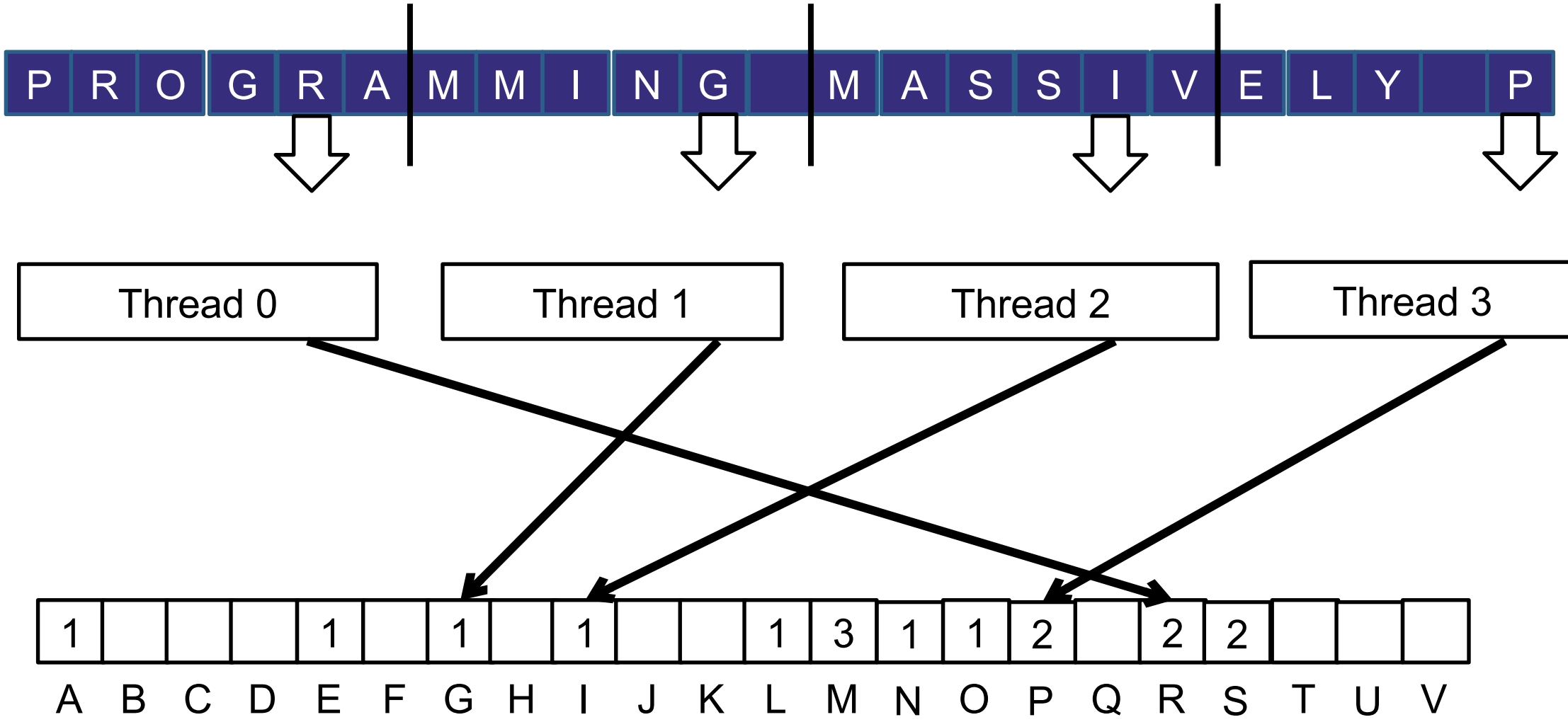
3rd repetition – 3rd letter in each part



4th repetition – 4th letter in each part



5th repetition – 5th letter in each part



Where is the problem with this algorithm?

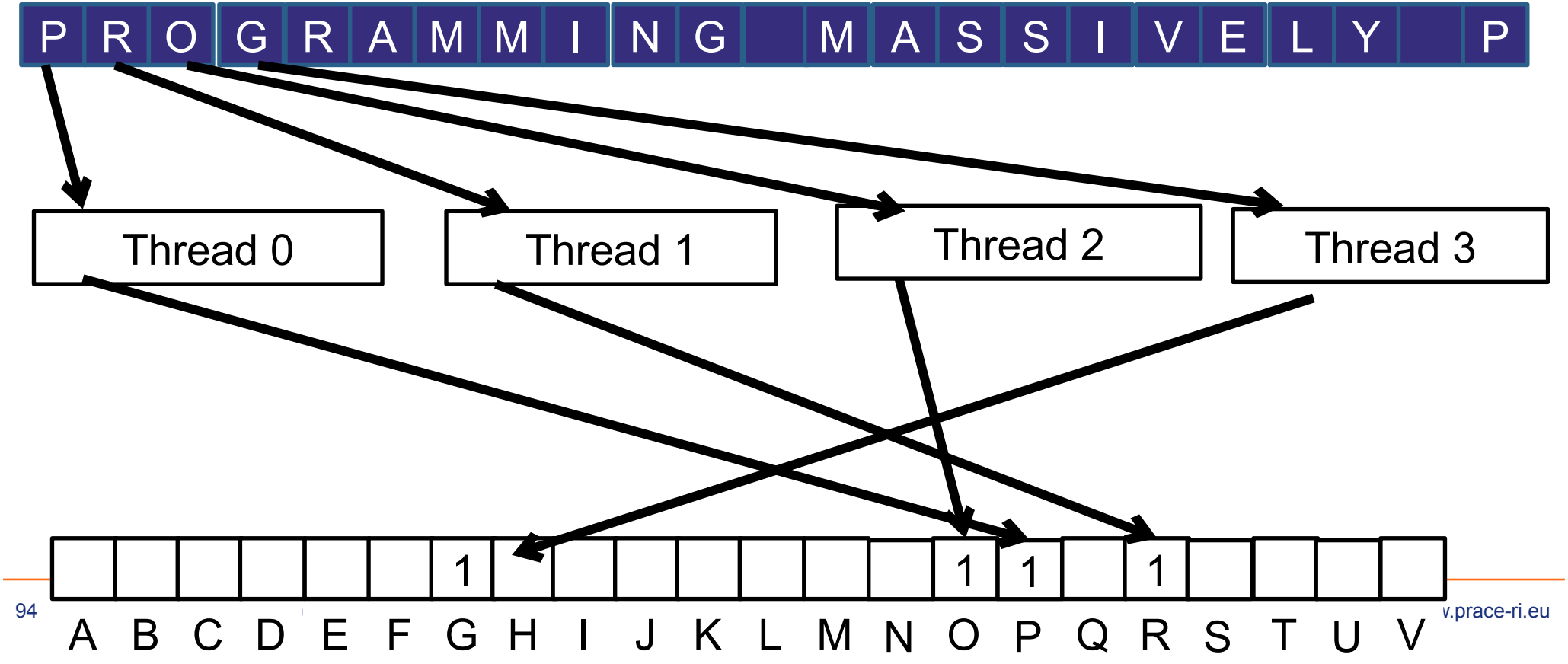
Problems of the algorithm

It reads input from non-continuous memory addresses

Assignment of elements to threads in strides

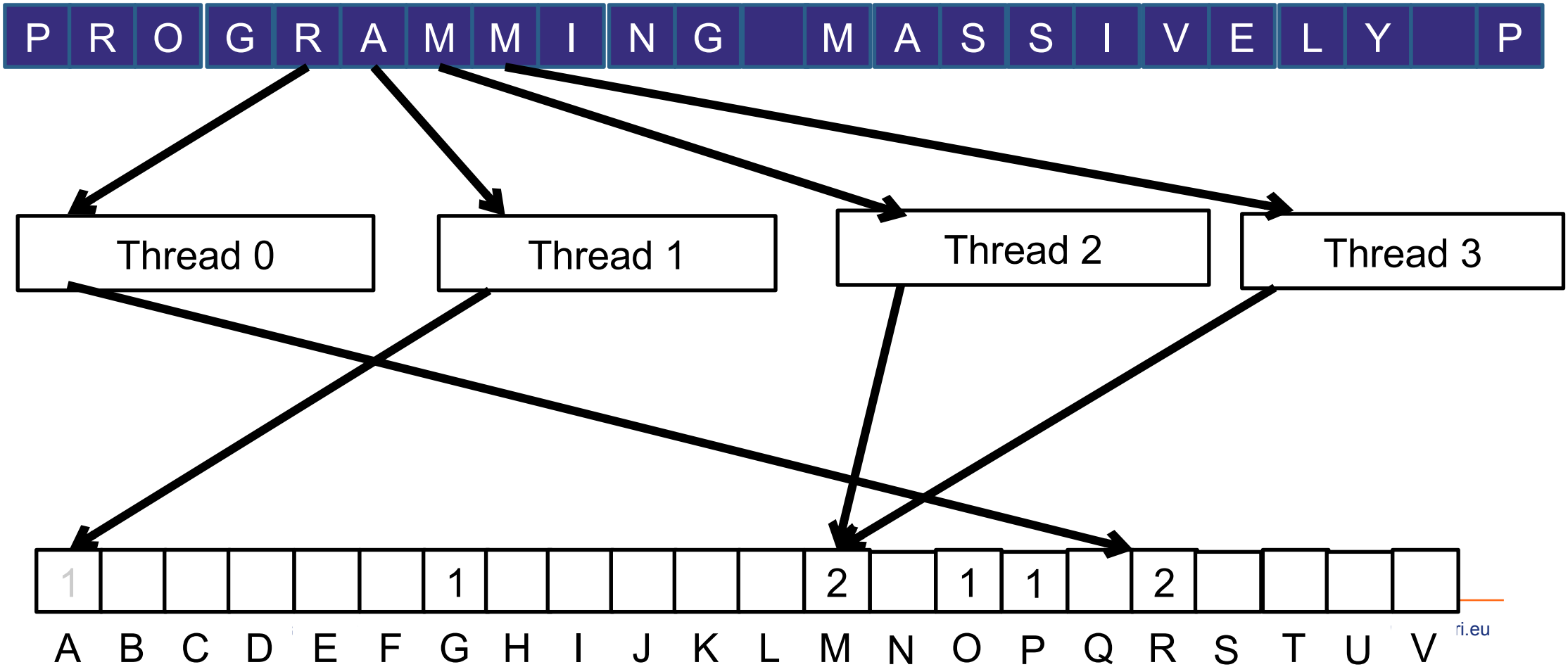
Better approach

Continuous threads should access continuous memory addresses



2nd repetition

All threads move on to the next part of the input sequence



Histogram computational kernel

Every thread processes elements in strides

```
__global__ void histo_kernel(unsigned char *buffer, long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
       atomicAdd( &(histo[buffer[i]]), 1);
       i += stride;
    }
}
```

Observation

Atomic instructions in previous example read/write memory in global memory

- Very slow to access data

Can we improve on that?

- Atomic instructions in shared memory

Privatization of variables

Local vectors of `histo[]` per block of threads

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    // Each character is 1 byte => 256 different characters
    __shared__ unsigned int histo_private[256];

    if (threadIdx.x < 256)
        histo_private[threadidx.x] = 0;
   __syncthreads();
```

First create local histogram

```
int i = threadIdx.x + blockIdx.x * blockDim.x;

// stride is total number of threads
int stride = blockDim.x * gridDim.x;

while (i < size) {
    atomicAdd( &(private_histo[buffer[i]), 1);
    i += stride;
}
```

Finally create global histogram

```
    // wait for all other threads in the block to finish
    __syncthreads();

    if (threadIdx.x < 256)
        atomicAdd( &(histo[threadIdx.x]),
                   private_histo[threadIdx.x] );

}
```

Variable privatization

Very often used when parallelizing an application

The size of the privatized variables should be small

- They must fit into shared memory

What to do if they don’t fit into shared memory?

- Tiling

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

Time

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

Time

CUDA Streams

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

Simultaneous execution through pipelining

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

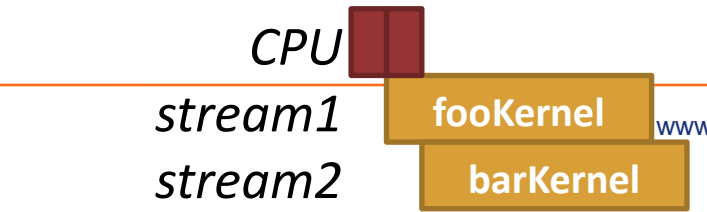
It is synchronous with all other streams!

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

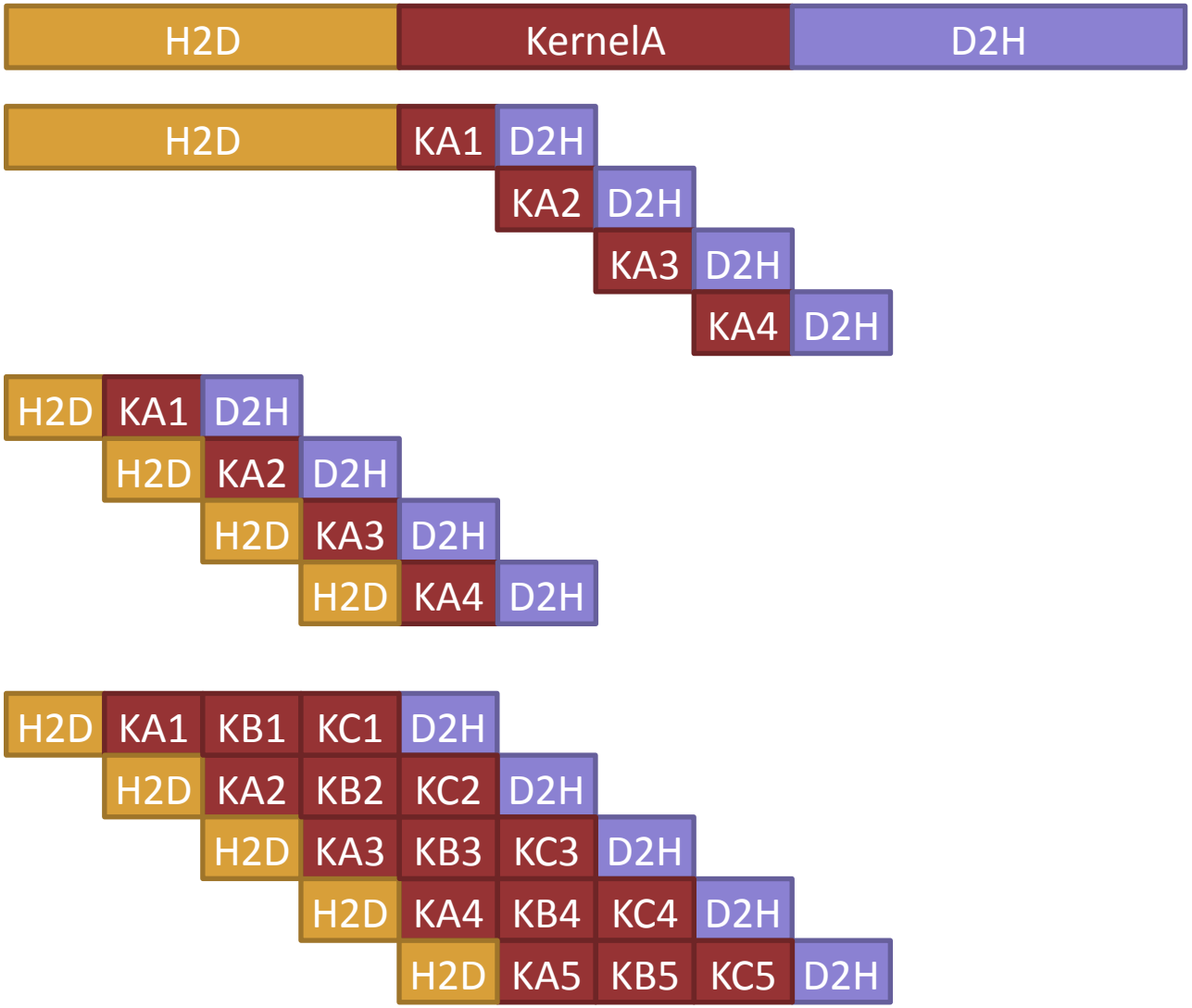
```
fooKernel<<<blocks, threads, 0>>>();
barKernel<<<blocks, threads, 0>>>();
```


```
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.

THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu