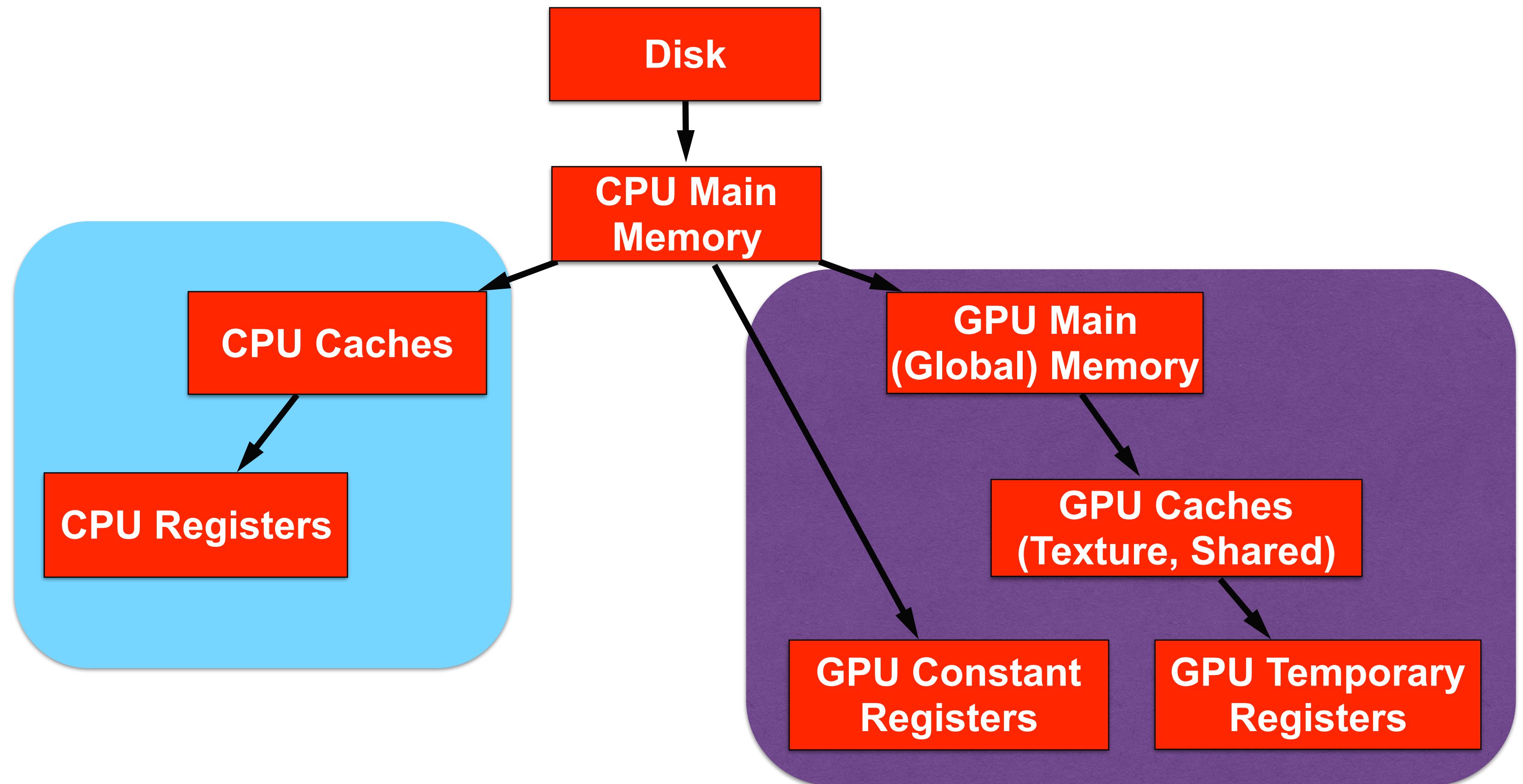


Lecture 5:

CUDA C (part 2)

Modern Parallel Computing
John Owens
EEC 289Q, UC Davis, Winter 2018

CPU/GPU Memory Hierarchy



What about hybrid (CPU/GPU same die) processors?

Memory model

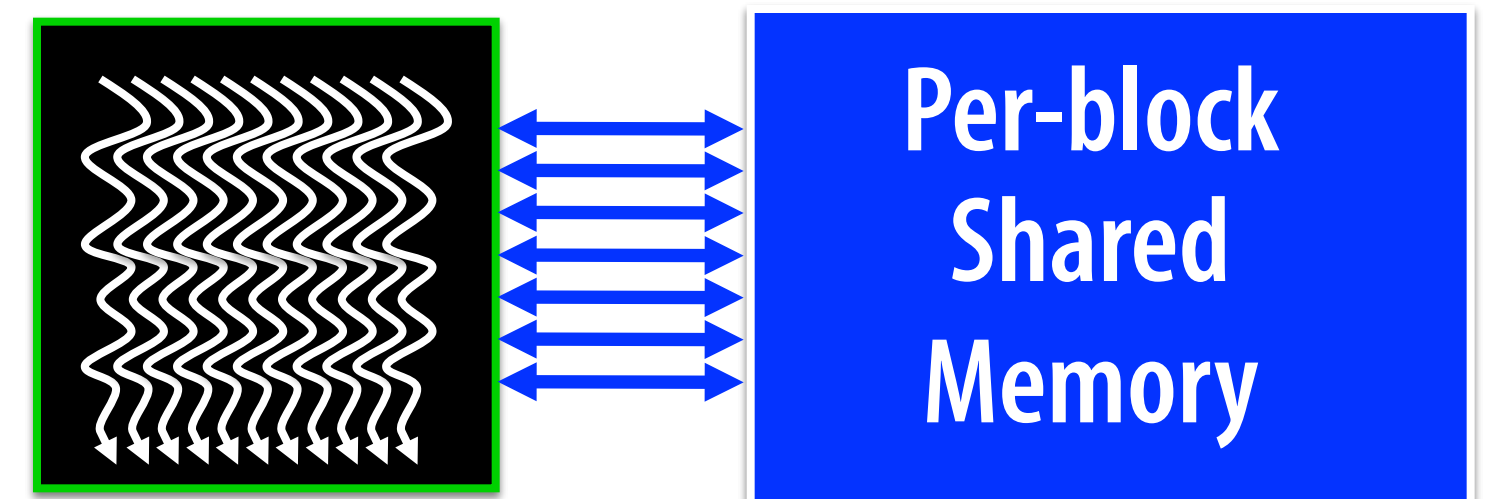
Thread



This is either

- Registers (up to 255/thread)
- Spill to DRAM (“local memory”)

Block



SM 5.0 GPUs:
48 kB/thread block

Synchronization of blocks

- Threads within block may synchronize with barriers
- Why is this necessary? Your mental model needs to be:
 - Any warp can run at any time
 - Warps can and will run until they hit a barrier
- Consider 128 threads in a block where you want thread t 's register $y = \text{thread } 127 - t$'s register x (reversal):
`shared[t] = register_x`
`register_y = shared[127-t]`

Synchronization of blocks

- Threads *within* block may synchronize with barriers

- ... Step 1 ...

- `__syncthreads();`

- ... Step 2 ...

- Blocks *coordinate* via atomic memory operations

- e.g., increment shared queue pointer with `atomicInc()`

- but don't *cooperate* between blocks!

- e.g., block 1 spins waiting for block 0 to complete

- Implicit barrier between dependent kernels

- `vec_minus<<<nblocks, blksize>>>(a, b, c);`
`vec_dot<<<nblocks, blksize>>>(c, c);`

Using per-block shared memory

- Variables shared across block

```
__shared__ int *begin, *end;
```

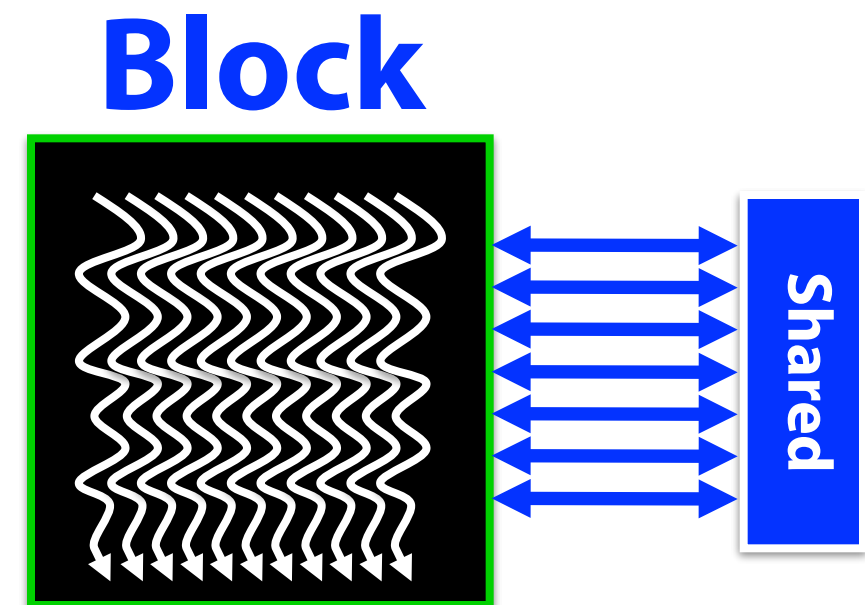
- Scratchpad memory

```
__shared__ int scratch[blocksize];
```

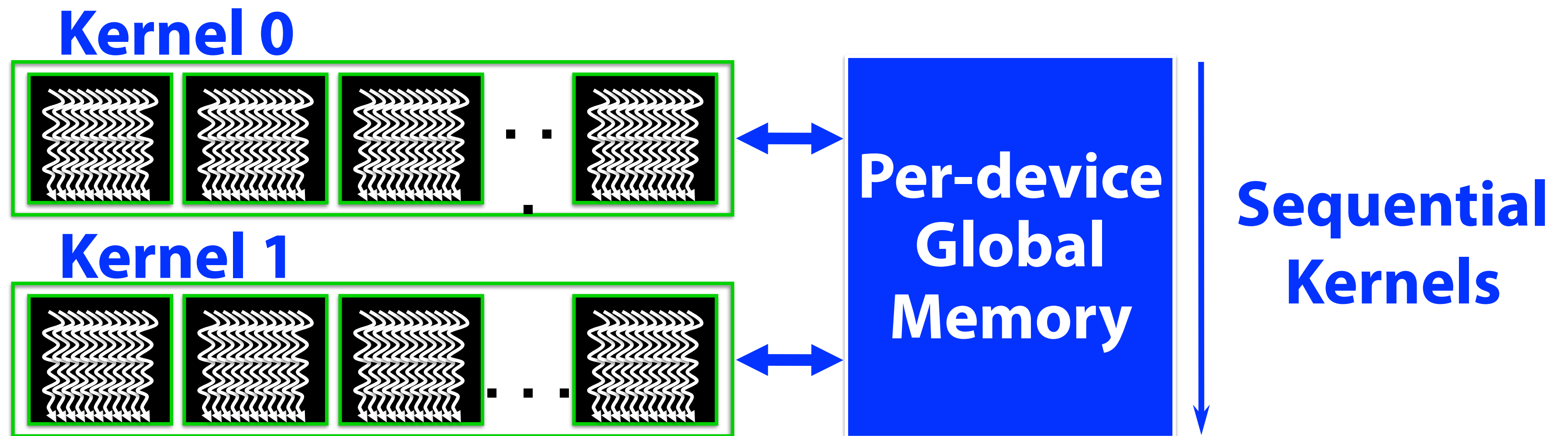
```
scratch[threadIdx.x] = begin[threadIdx.x];  
// ... compute on scratch values ...  
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

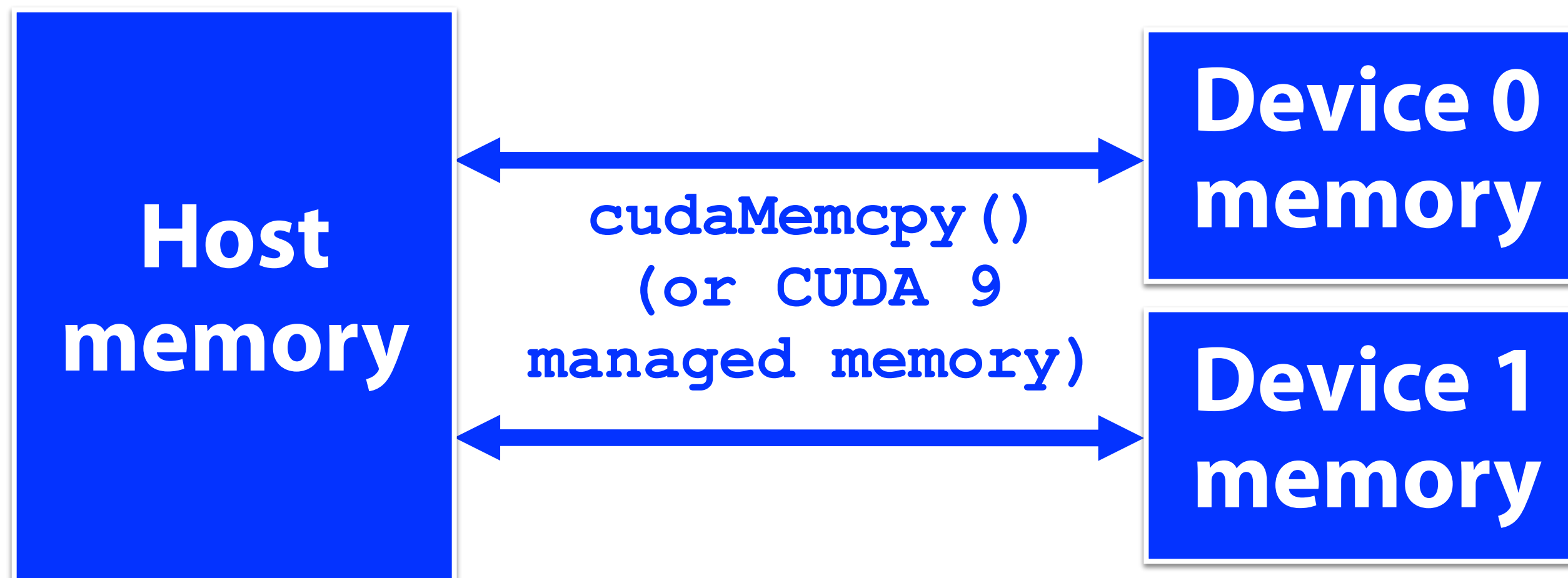
```
scratch[threadIdx.x] = begin[threadIdx.x];  
__syncthreads();  
int left = scratch[threadIdx.x - 1];
```



Memory model



Memory model



Warp-wide instructions

- `int __all(int predicate);`
- `int __any(int predicate);`
- `unsigned int __ballot(int predicate);`
- `__shfl, __shfl_up, __shfl_down, __shfl_xor`
exchange a variable between threads within a warp.

```
#include <stdio.h>
__global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)           // Note unused variable for
        value = arg;         // all threads except lane 0
    value = __shfl(value, 0);  // Get "value" from lane 0
    if (value != arg)
        printf("Thread %d failed.\n", threadIdx.x);
}
int main() {
    bcast<<< 1, 32 >>>(1234);
    cudaDeviceSynchronize();
    return 0; }
```

```
#include <stdio.h>
__global__ void warpReduce() {
    int laneId = threadIdx.x & 0x1f;
    // Seed starting value as inverse lane ID
    int value = 31 - laneId;
    // Use XOR mode to perform butterfly reduction
    for (int i=16; i>=1; i/=2)
        value += __shfl_xor(value, i, 32);
    // "value" now contains the sum across all threads
    printf("Thread %d final value = %d\n", threadIdx.x, value);
}
int main() {
    warpReduce<<< 1, 32 >>>();
    cudaDeviceSynchronize();
    return 0; }
```

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory

`cudaMalloc()` , `cudaMallocManaged()` ,
`cudaFree()`

- Explicit memory copy for host \leftrightarrow device, device \leftrightarrow device, device \leftrightarrow host

`cudaMemcpy()` , `cudaMemcpy2D()` , ...

- Texture management

`cudaBindTexture()` , `cudaBindTextureToArray()` , ...

- OpenGL & DirectX interoperability

`cudaGLMapBufferObject()` ,
`cudaD3D9MapVertexBuffer()` , ...

Example: Vector Addition Kernel

```
// Compute vector sum  $C = A+B$ 
```

```
// Each thread performs one pair-wise addition
```

```
__global__ void vecAdd(float* A, float* B, float* C) {
```

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

```
    C[i] = A[i] + B[i];
```

```
}
```

```
int main() {
```

```
    // Run N/256 blocks of 256 threads each
```

```
    vecAdd<<< N/256, 256>>>>(d_A, d_B, d_C);
```

```
}
```

Example: Host code for `vecAdd`

```
// allocate and initialize host (CPU) memory
```

```
float *h_A = ..., *h_B = ...;
```

```
// allocate device (GPU) memory
```

```
float *d_A, *d_B, *d_C;
```

```
cudaMalloc( (void**) &d_A, N * sizeof(float) );
```

```
cudaMalloc( (void**) &d_B, N * sizeof(float) );
```

```
cudaMalloc( (void**) &d_C, N * sizeof(float) );
```

```
// copy host memory to device
```

```
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice );
```

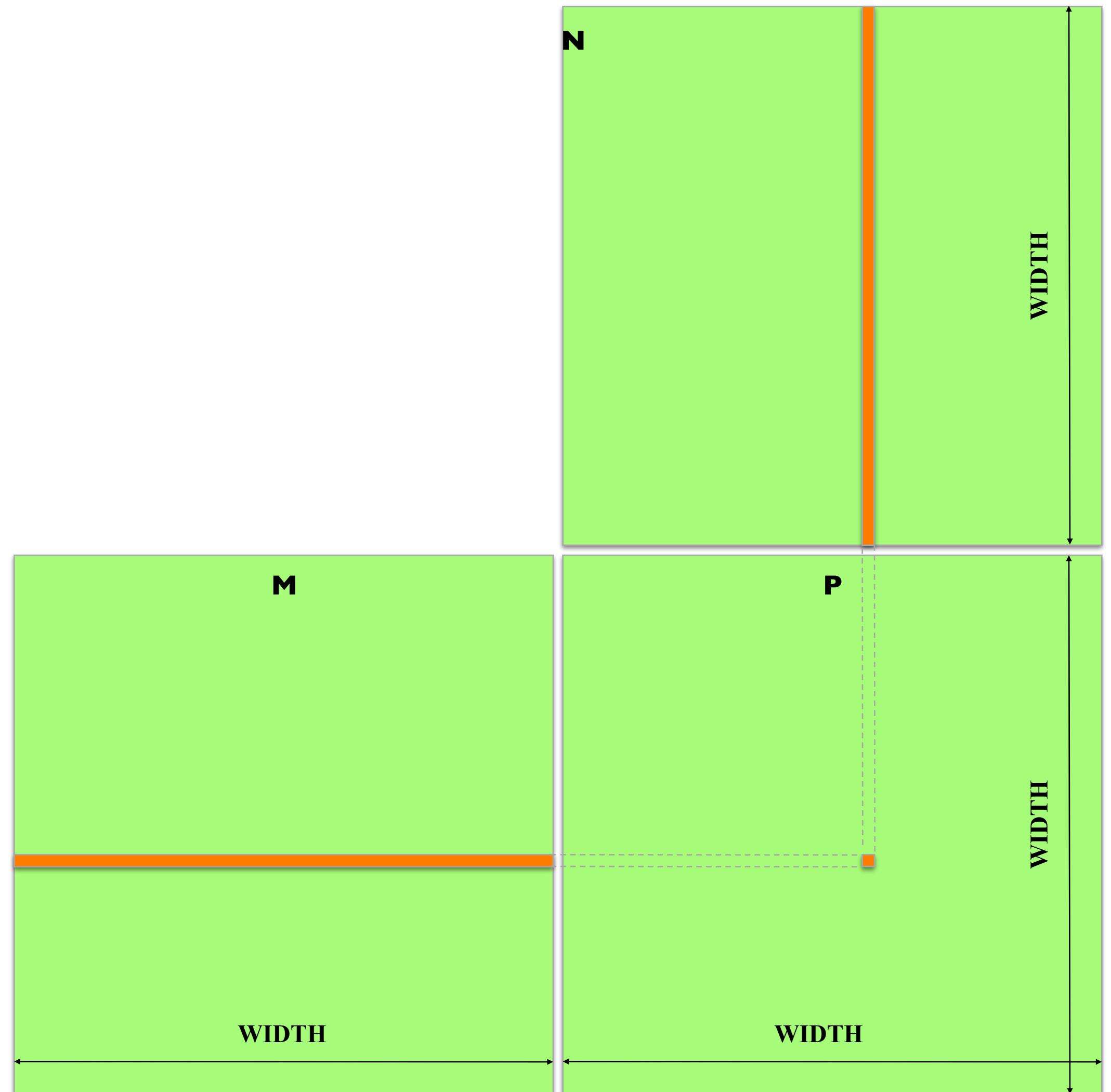
```
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice );
```

```
// execute the kernel on N/256 blocks of 256 threads each
```

```
vecAdd<<<N/256, 256>>>(d_A, d_B, d_C);
```

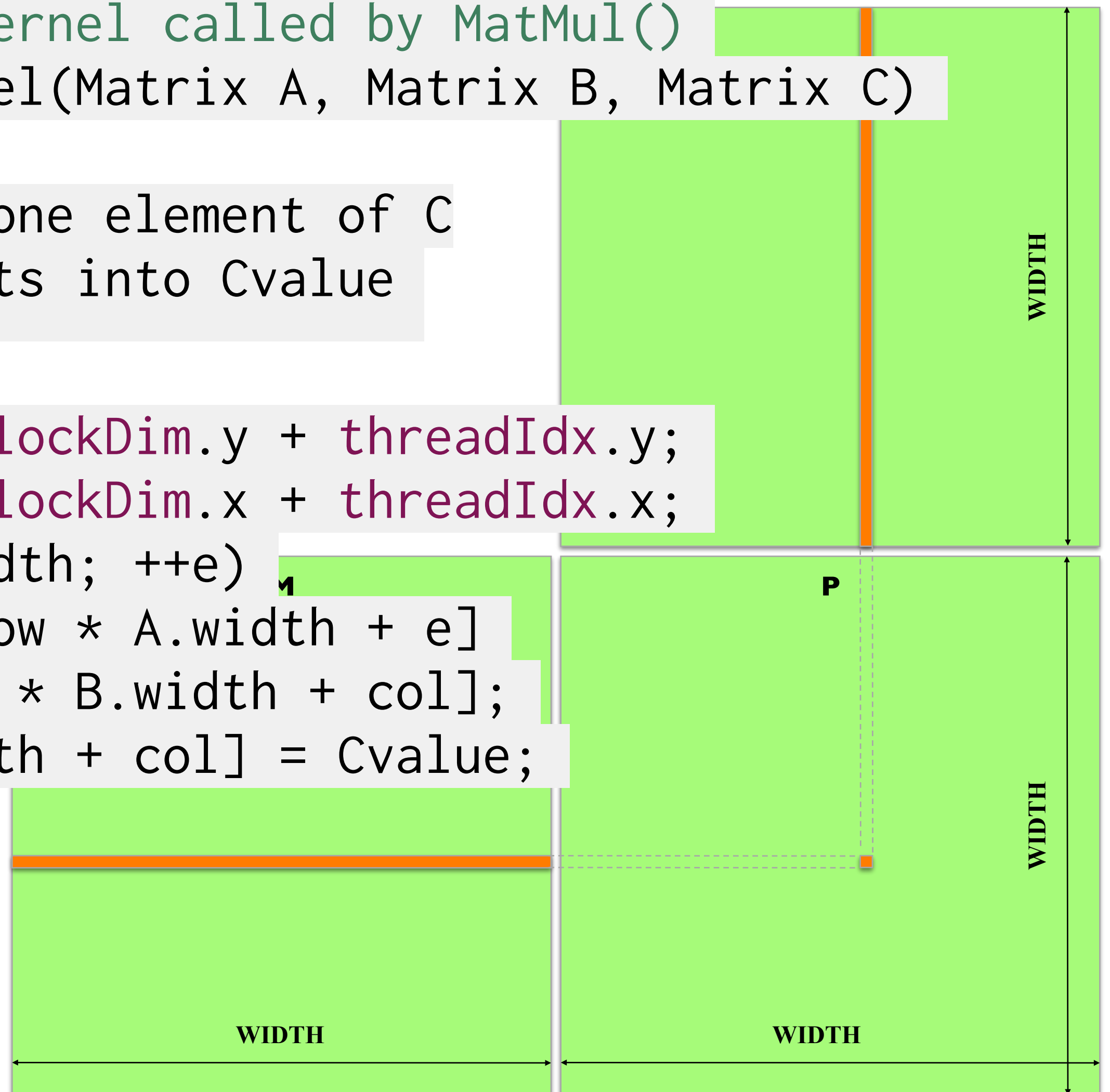
Dense Matrix Multiplication

- for all elements E in destination matrix P
 - $P_{r,c} = M_r \cdot N_c$



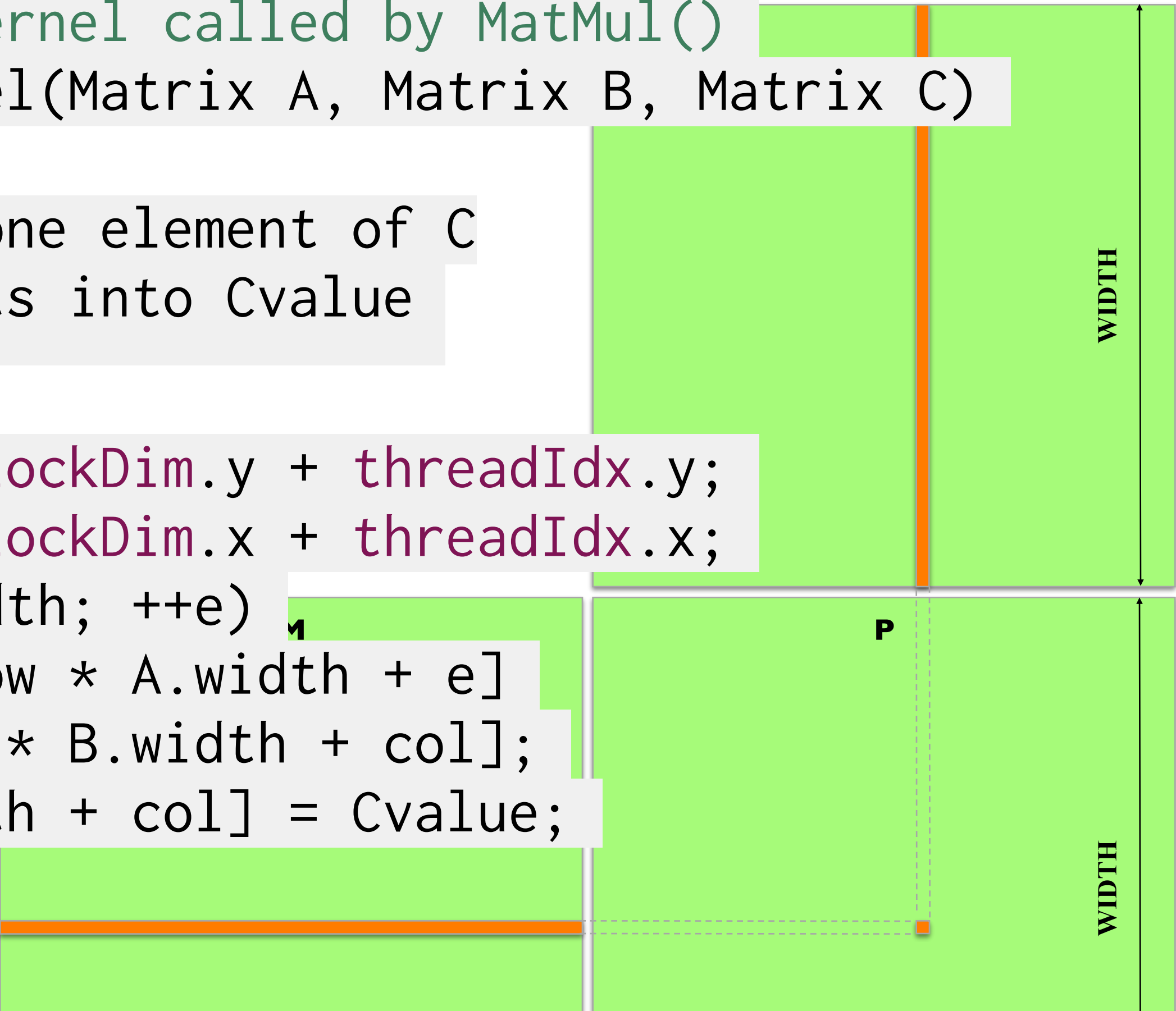
Dense Matrix Multiplication

```
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)  
{  
    // Each thread computes one element of C  
    // by accumulating results into Cvalue  
  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e]  
                 * B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Dense Matrix Multiplication

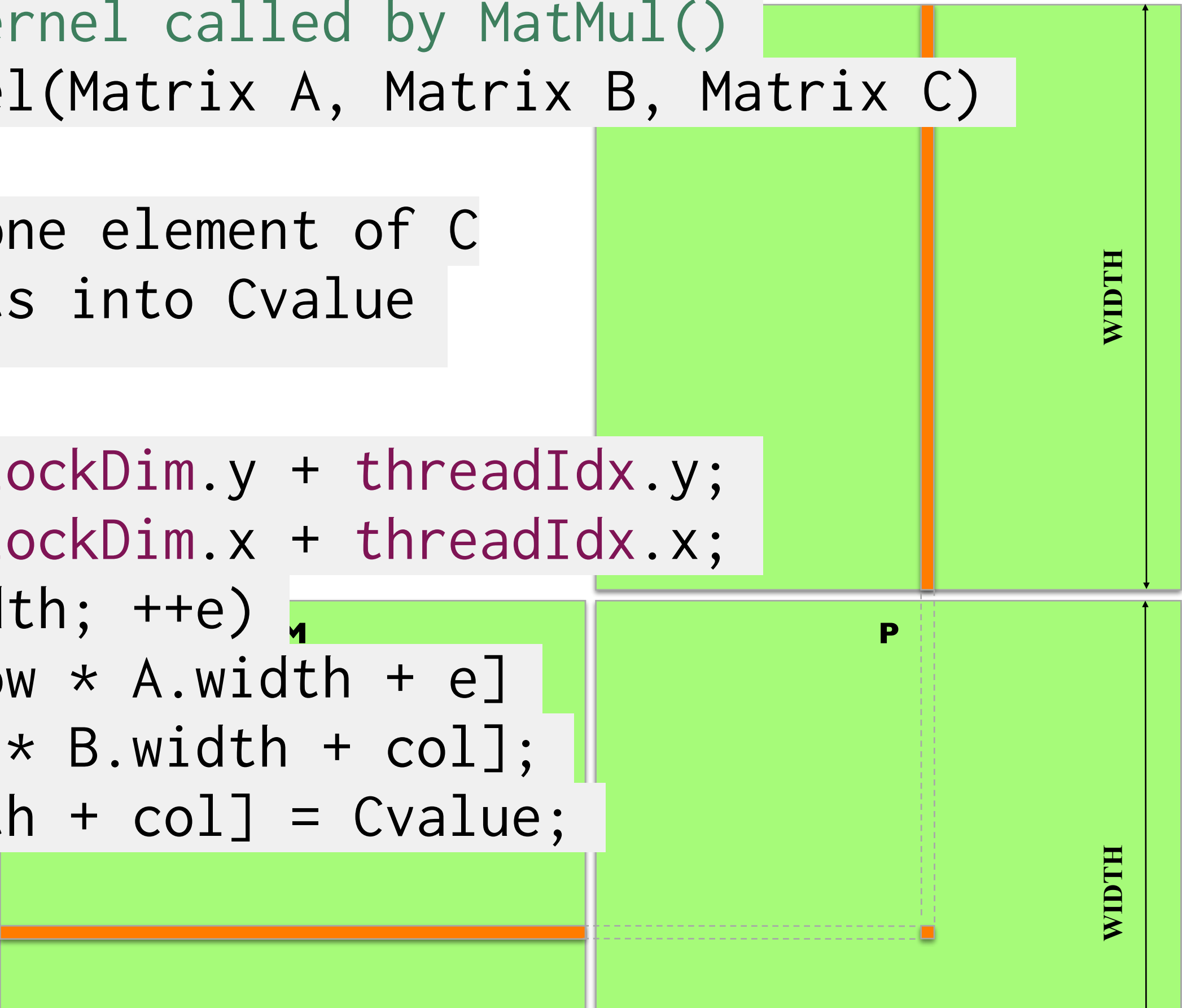
```
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)  
{  
    // Each thread computes one element of C  
    // by accumulating results into Cvalue  
  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e]  
                 * B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Q1: Discounting addressing arithmetic, how many multiplications are we going to do?

Dense Matrix Multiplication

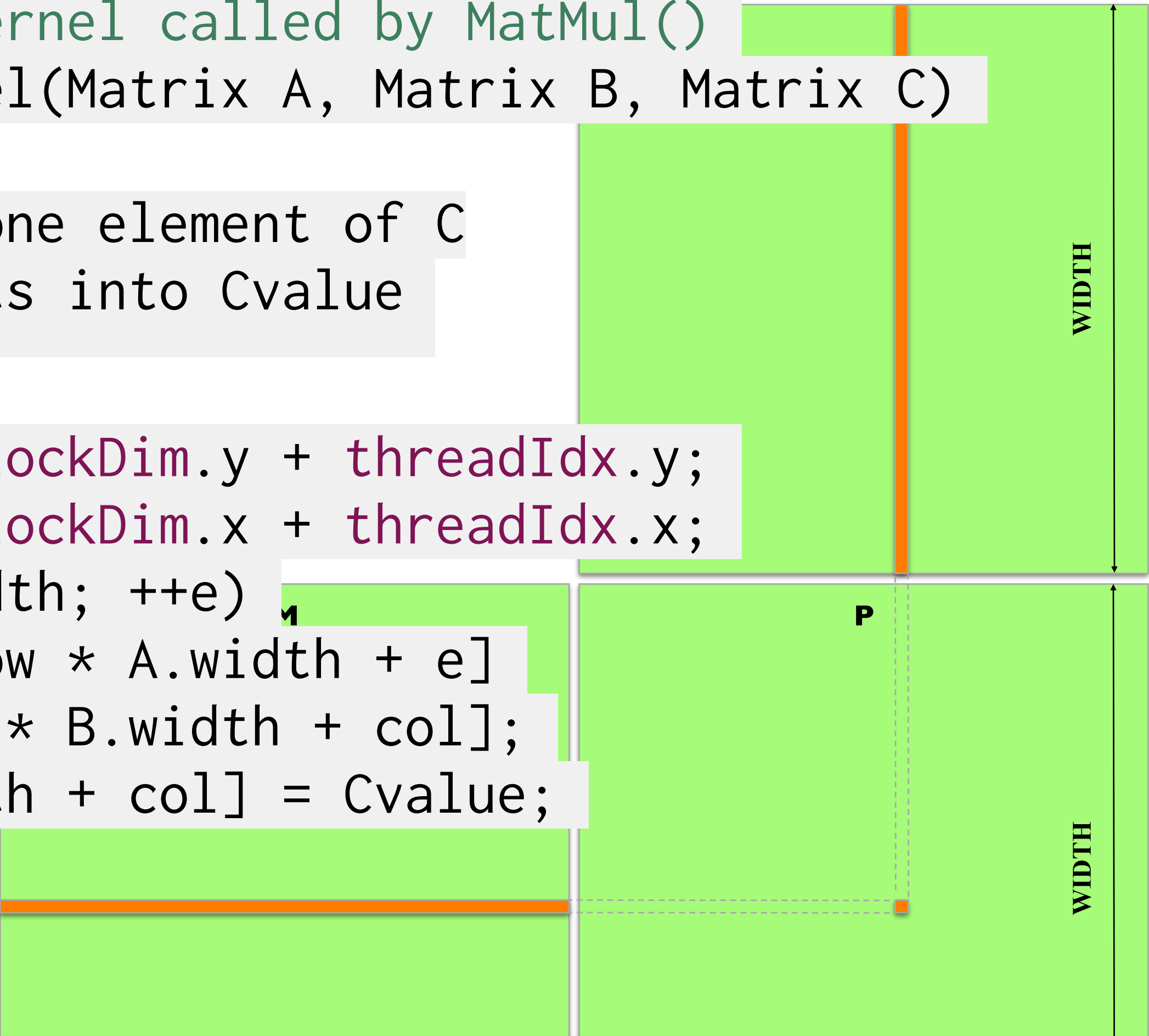
```
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)  
{  
    // Each thread computes one element of C  
    // by accumulating results into Cvalue  
  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e]  
                 * B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Q2: Out of the 3 global memory reads—
{A, B, C}.elements—which accesses are coalesced?

Dense Matrix Multiplication

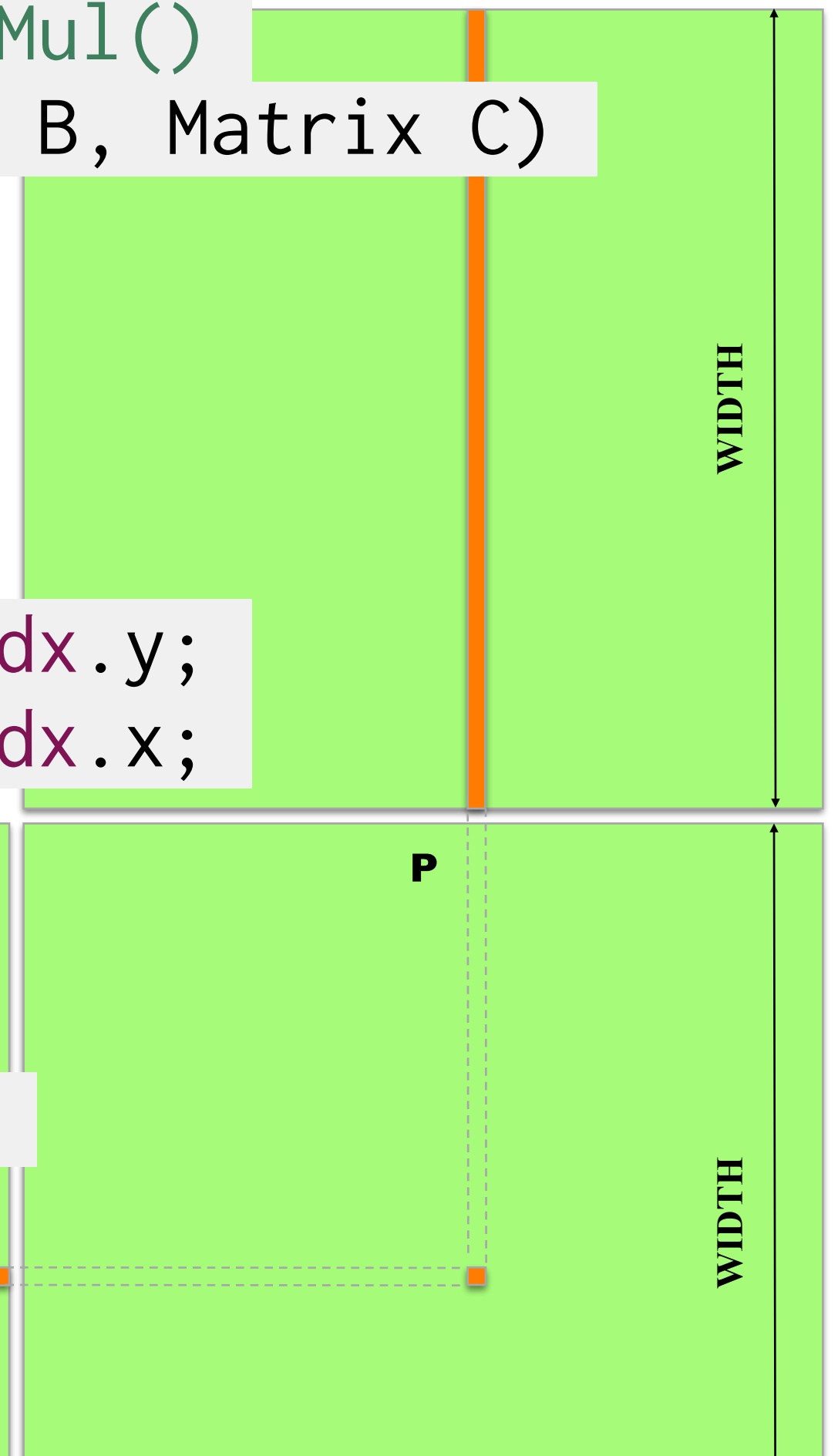
```
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)  
{  
    // Each thread computes one element of C  
    // by accumulating results into Cvalue  
  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e]  
                 * B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Q3: How do we make **A.elements** coalesced?

Dense Matrix Multiplication

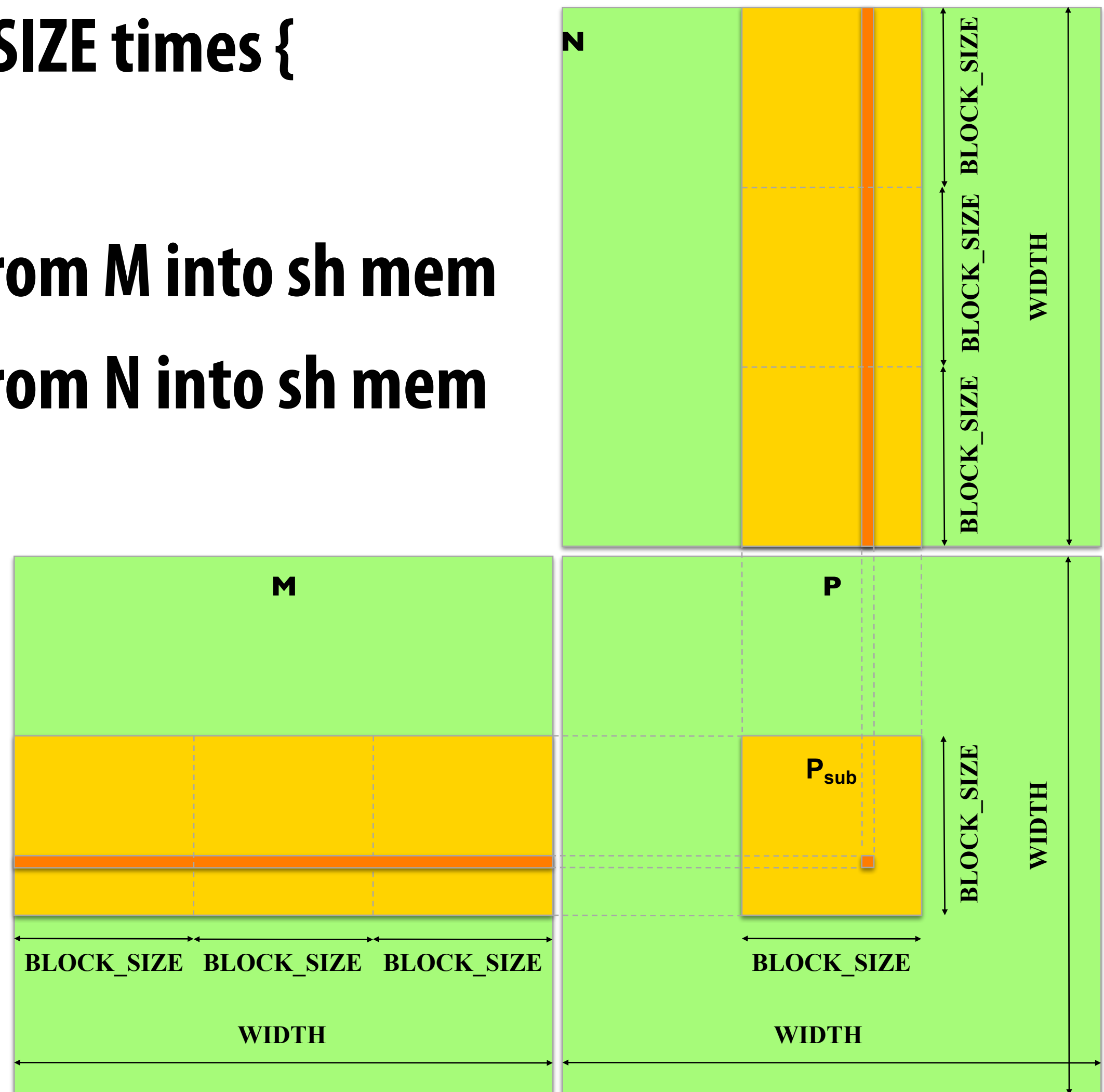
```
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)  
{  
    // Each thread computes one element of C  
    // by accumulating results into Cvalue  
  
    float Cvalue = 0;  
    int row = blockIdx.y * blockDim.y + threadIdx.y;  
    int col = blockIdx.x * blockDim.x + threadIdx.x;  
    for (int e = 0; e < A.width; ++e)  
        Cvalue += A.elements[row * A.width + e]  
                 * B.elements[e * B.width + col];  
    C.elements[row * C.width + col] = Cvalue;  
}
```



Q4: How many times do we read each element of **A**?

Dense Matrix Multiplication

- $P = M * N$ of size $WIDTH \times WIDTH$
- Repeat $WIDTH / BLOCK_SIZE$ times {
 - Each thread block:
 - Reads a subblock from M into sh mem
 - Reads a subblock from N into sh mem
 - Each thread:
 - Computes a mini-dot-product
- Output total sum



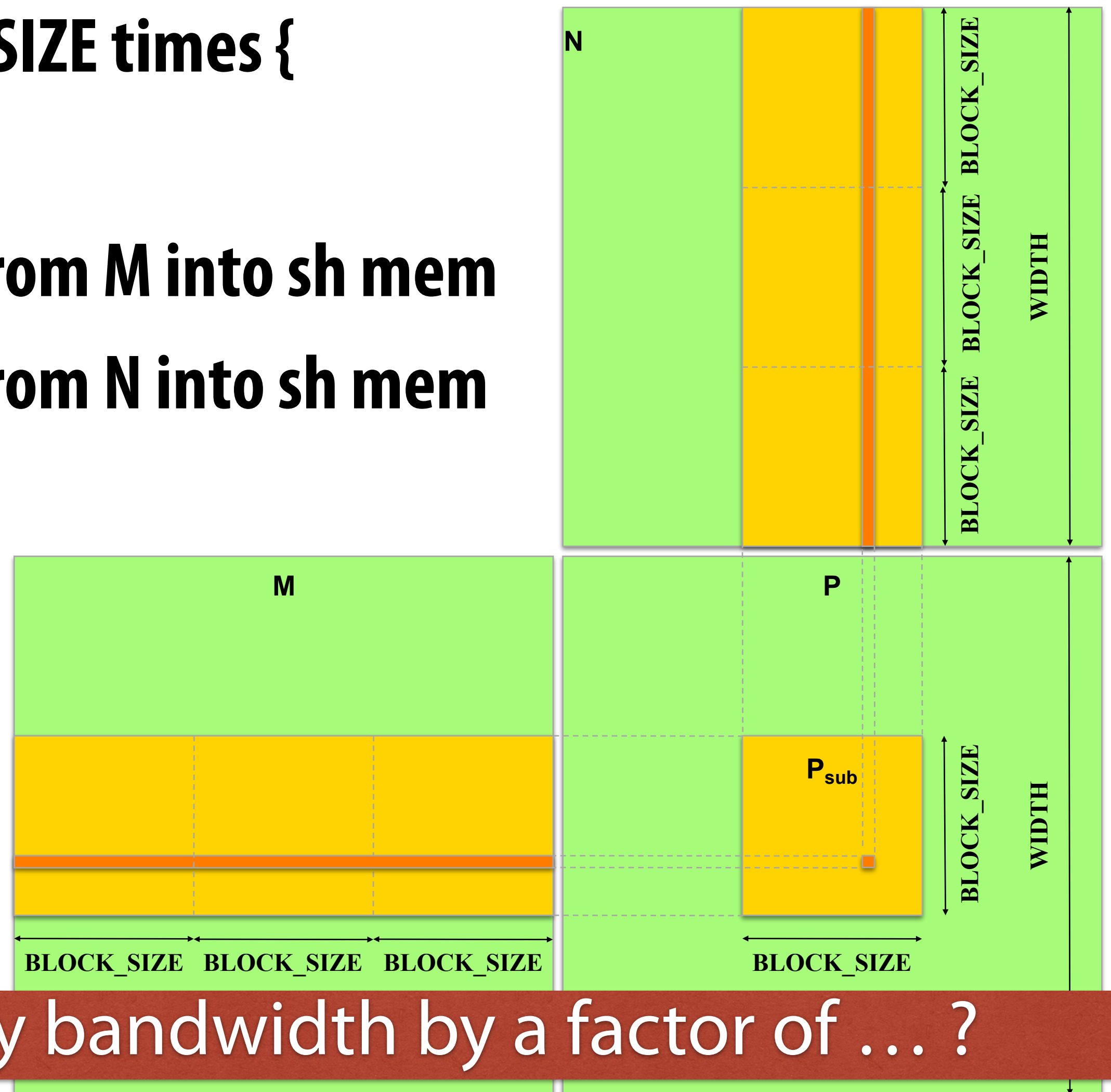
```

for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
        Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
}

```

Dense Matrix Multiplication

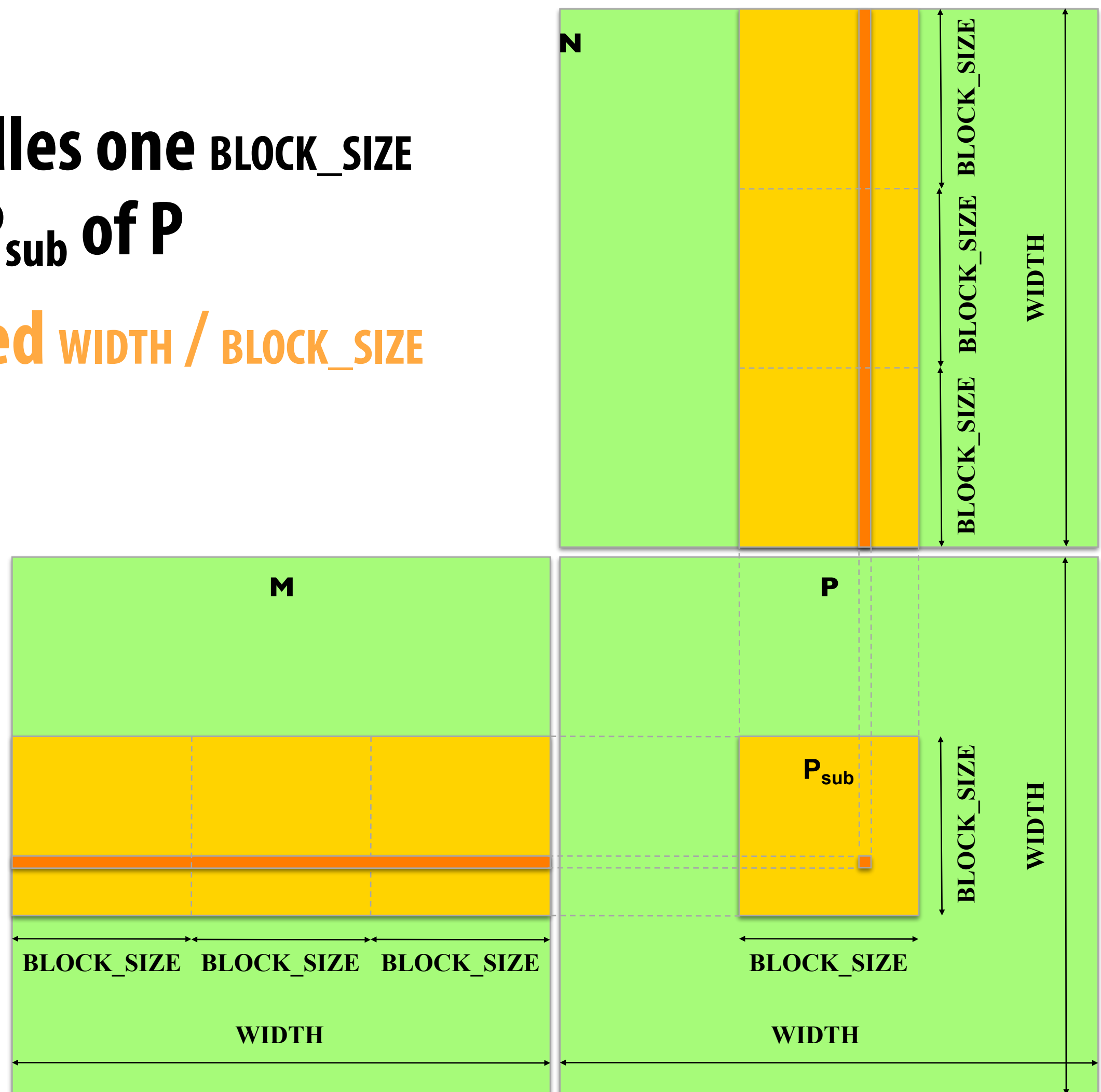
- $P = M * N$ of size **WIDTH** x **WIDTH**
- Repeat **WIDTH / BLOCK_SIZE** times {
 - Each thread block:
 - Reads a subblock from **M** into sh mem
 - Reads a subblock from **N** into sh mem
 - Each thread:
 - Computes a mini-dot-product
- Output total sum



Q5: We save memory bandwidth by a factor of ... ?

Dense Matrix Multiplication

- $P = M * N$ of size $WIDTH \times WIDTH$
- With blocking:
 - One **thread block** handles one $BLOCK_SIZE \times BLOCK_SIZE$ sub-matrix P_{sub} of P
 - **M and N are only loaded $WIDTH / BLOCK_SIZE$ times from global memory**
- Great saving of memory bandwidth!





GPU Teaching Kit

Accelerated Computing



Module 7.1 – Parallel Computation Patterns (Histogram)

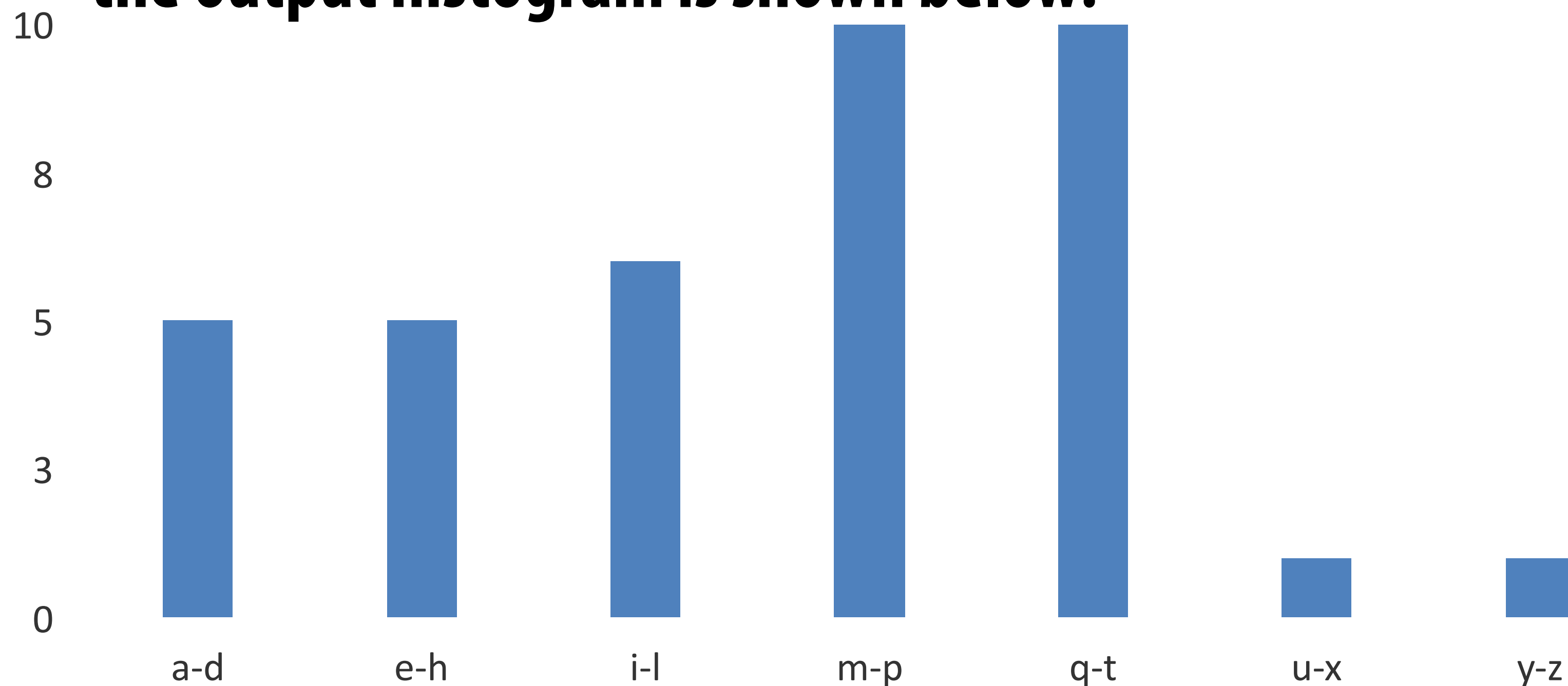
Histogramming

Histogram

- **A method for extracting notable features and patterns from large data sets**
 - **Feature extraction for object recognition in images**
 - **Fraud detection in credit card transactions**
 - **Correlating heavenly object movements in astrophysics**
 - **...**
- **Basic histograms: for each element in the data set, use the value to identify a “bin counter” to increment**

A Text Histogram Example

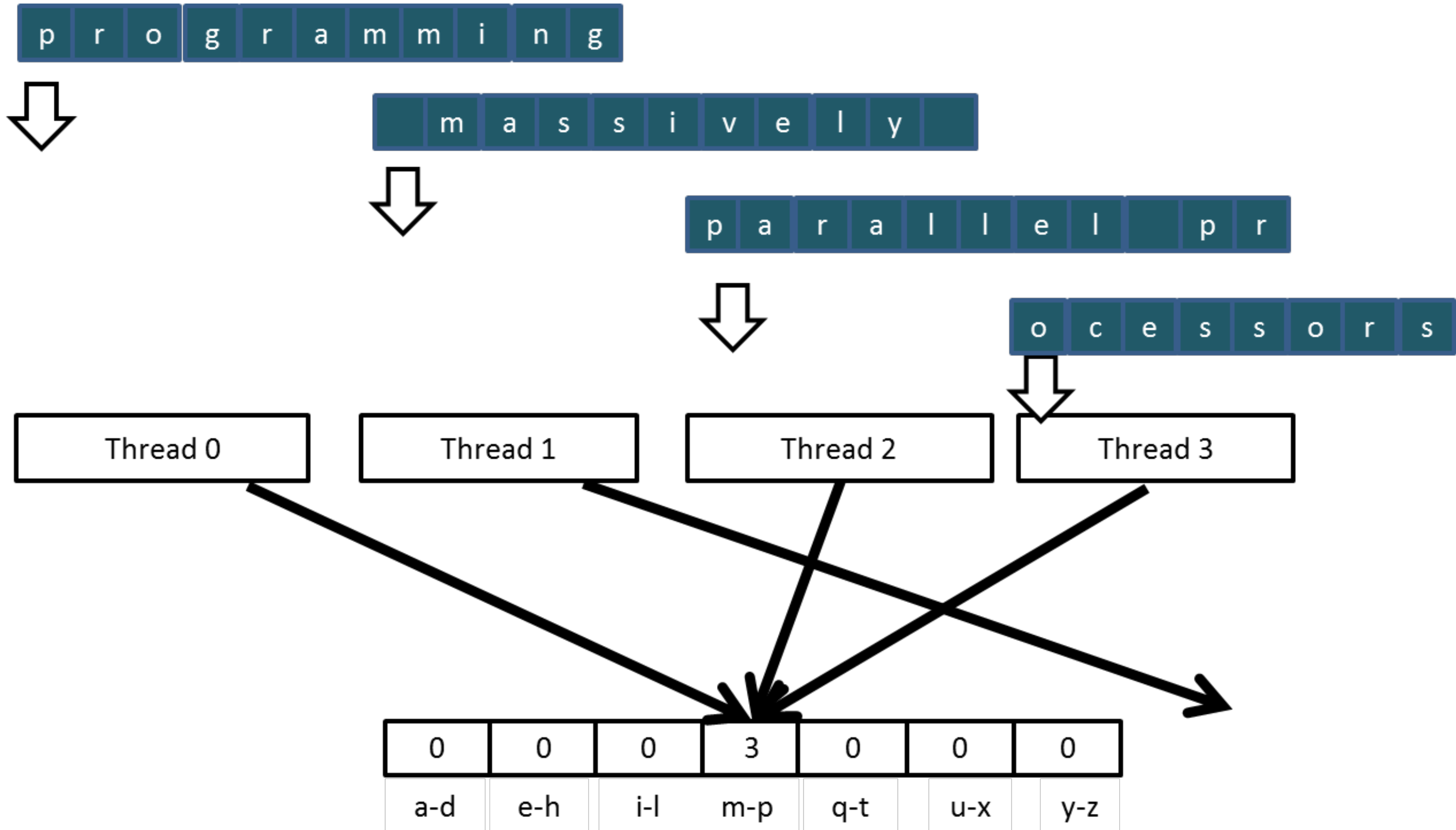
- Define the bins as four-letter sections of the alphabet: a–d, e–h, i–l, n–p, ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:



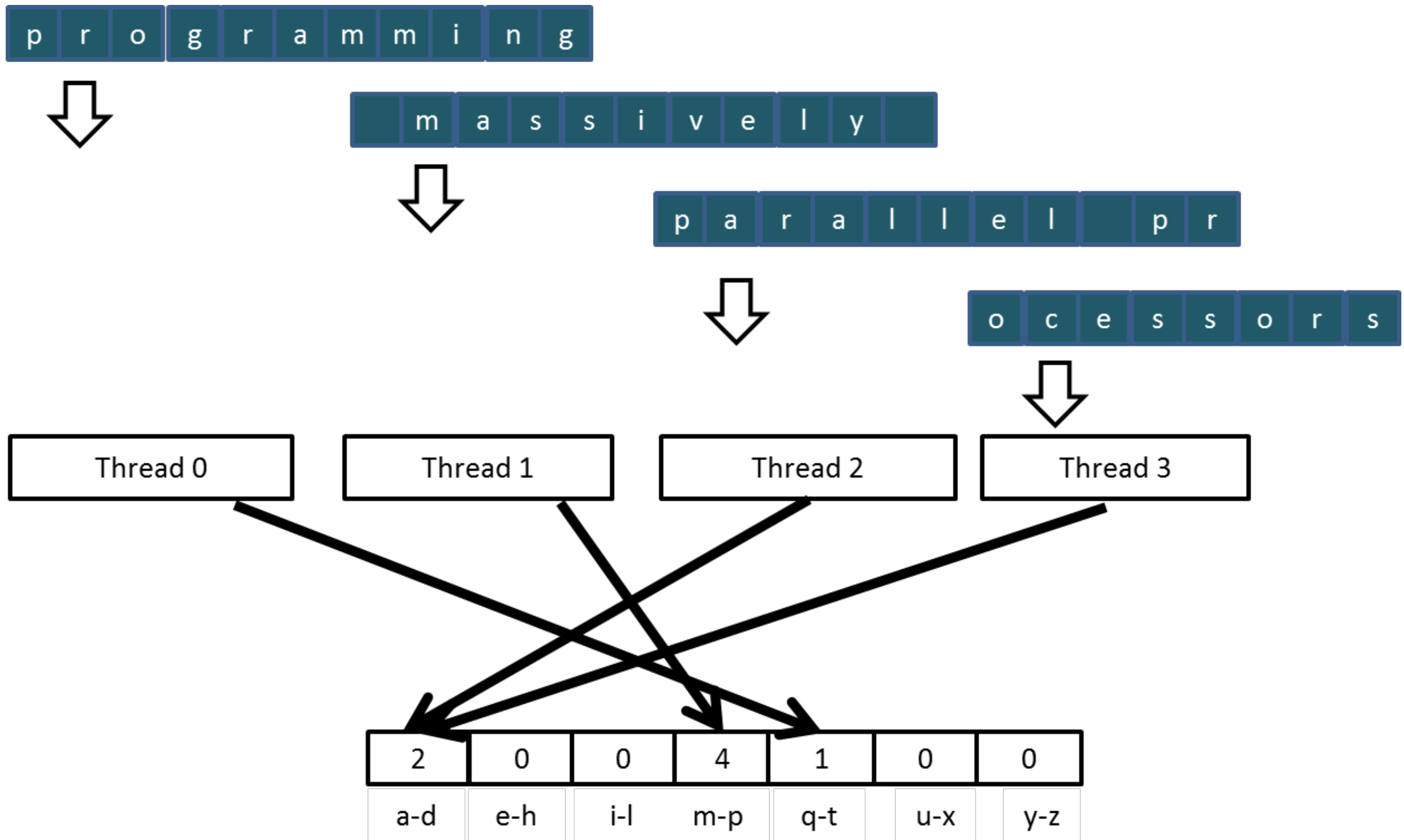
A simple parallel histogram algorithm

- **Partition the input into sections**
- **Have each thread to take a section of the input**
- **Each thread iterates through its section.**
- **For each letter, increment the appropriate bin counter**

Sectioned Partitioning (Iteration #1)

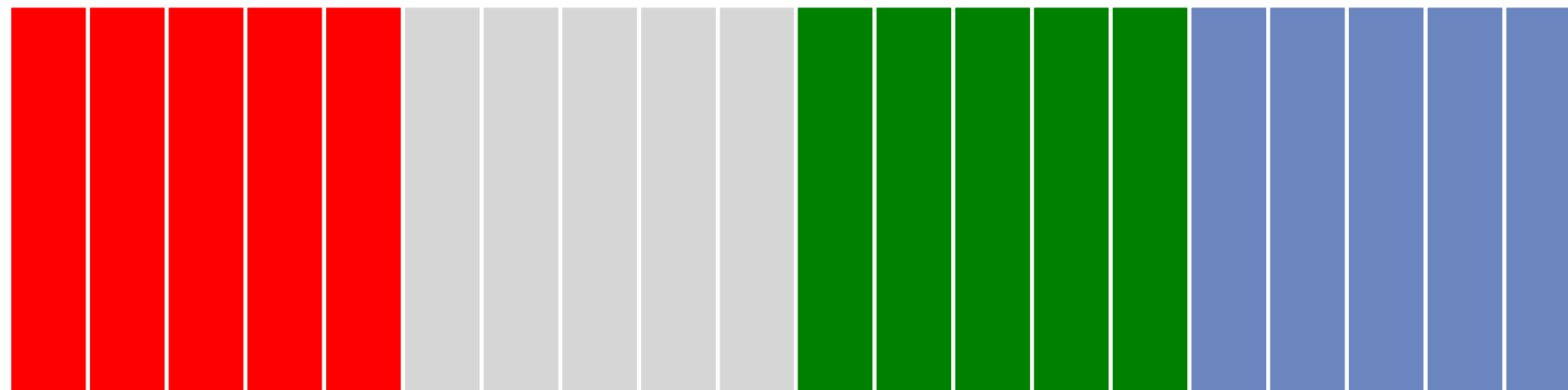


Sectioned Partitioning (Iteration #2)



Input Partitioning Affects Memory Access Efficiency

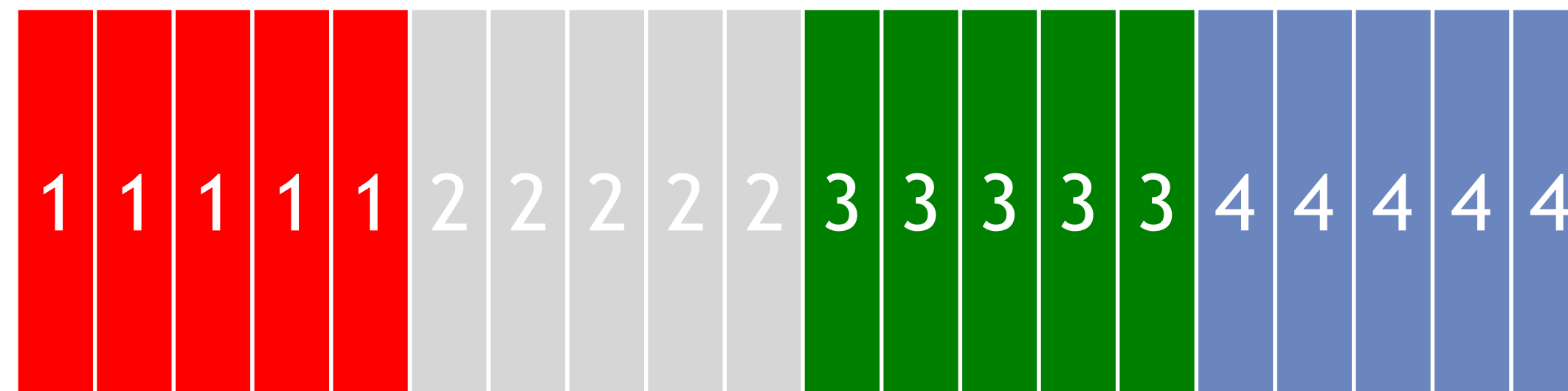
- **Sectioned partitioning results in poor memory access efficiency**
 - **Adjacent threads do not access adjacent memory locations**
 - **Accesses are not coalesced**
 - **DRAM bandwidth is poorly utilized**



Input Partitioning Affects Memory Access Efficiency

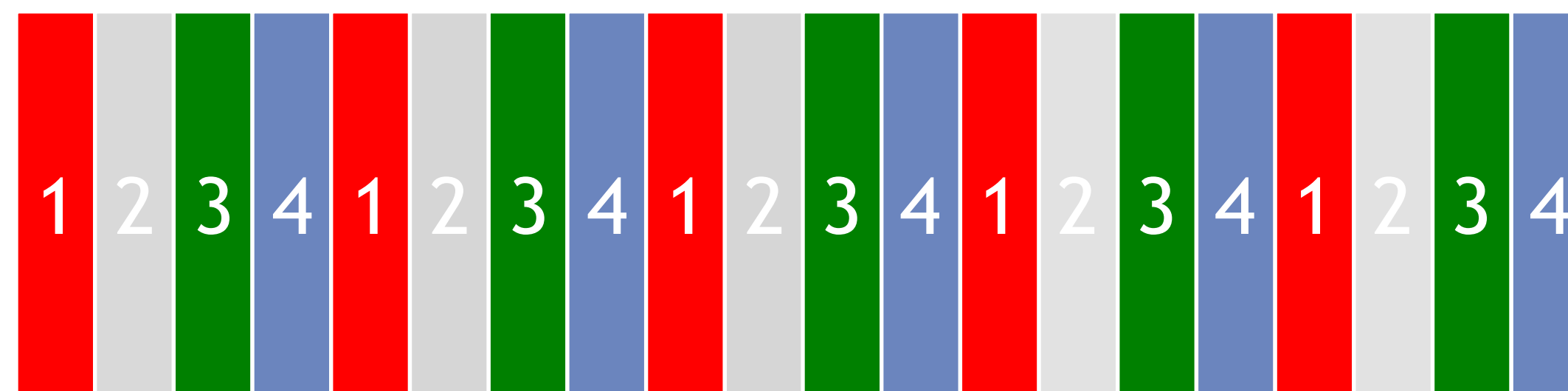
■ Sectioned partitioning results in poor memory access efficiency

- Adjacent threads do not access adjacent memory locations
- Accesses are not coalesced
- DRAM bandwidth is poorly utilized



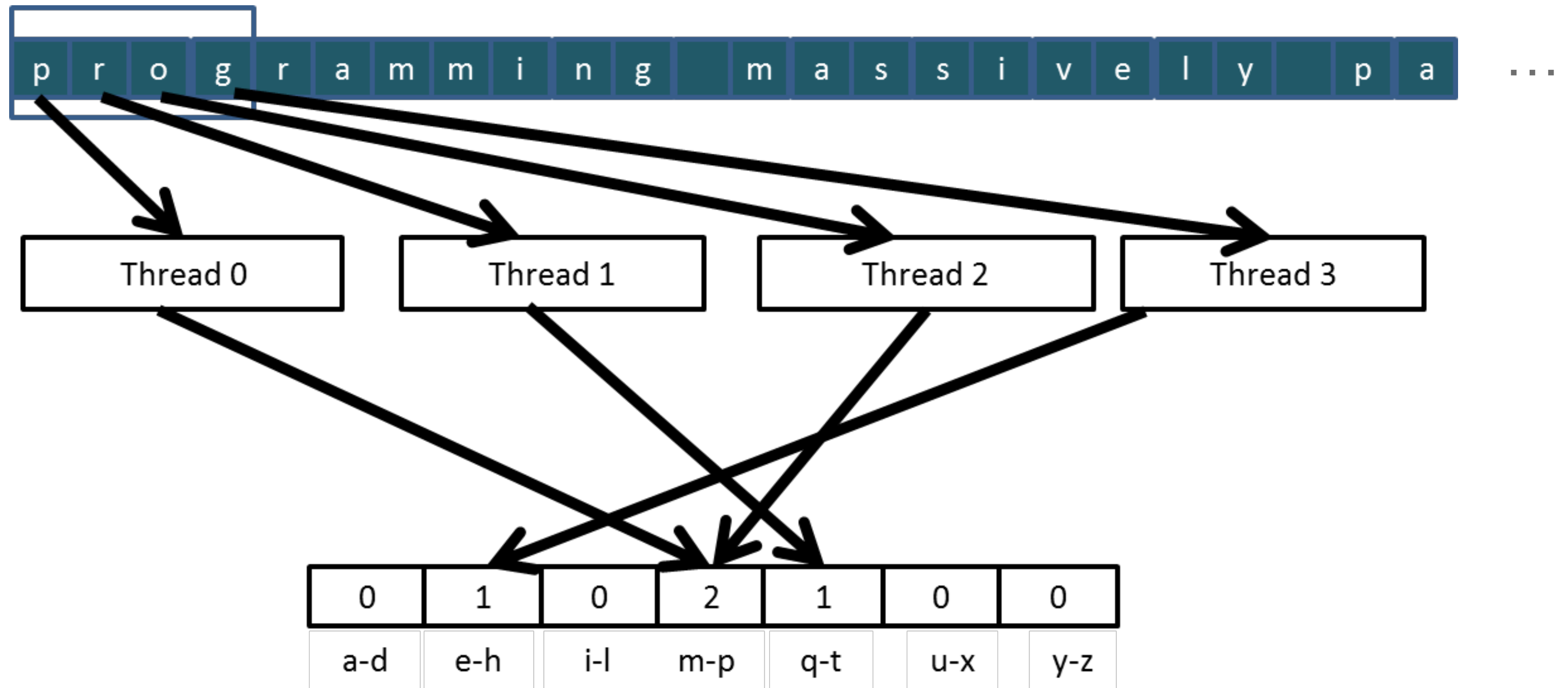
■ Change to interleaved partitioning

- All threads process a contiguous section of elements
- They all move to the next section and repeat
- The memory accesses are coalesced

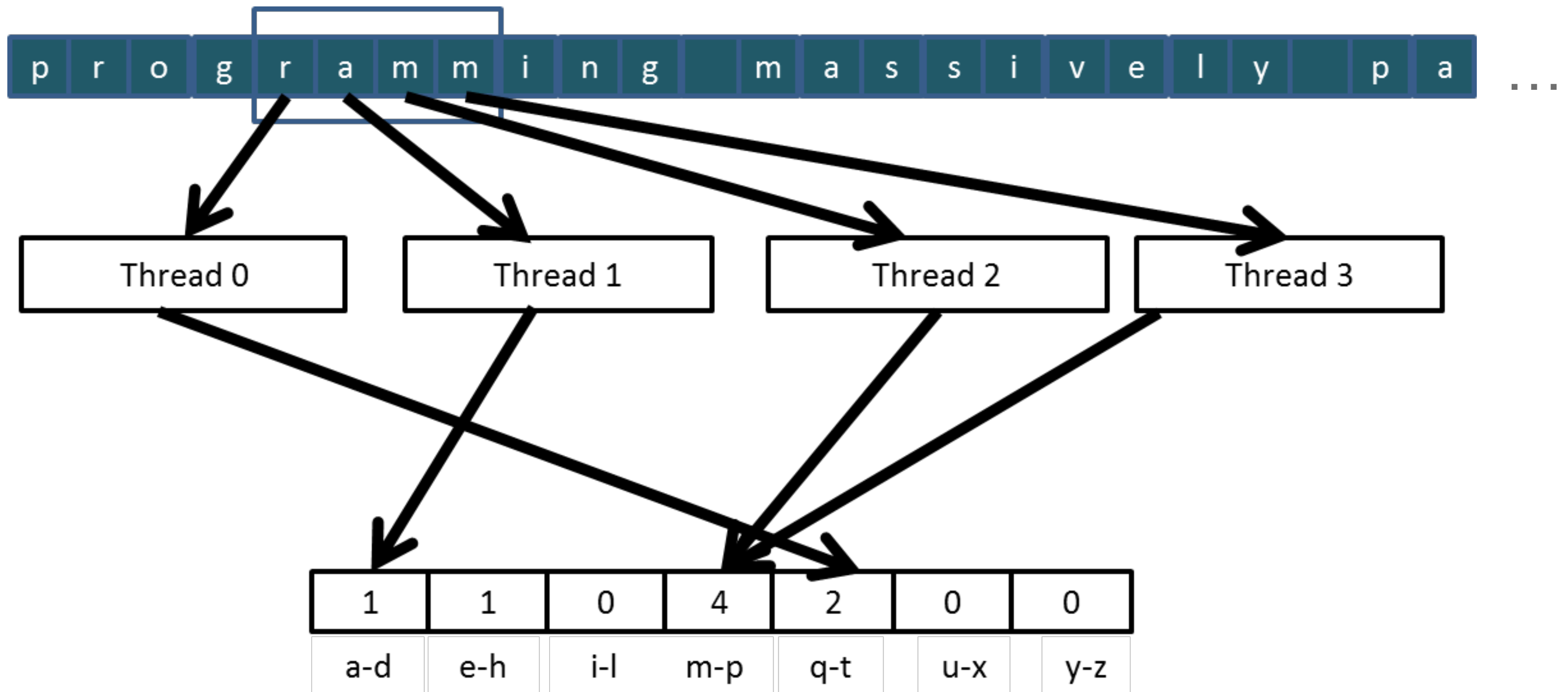


Interleaved Partitioning of Input

- For coalescing and better memory access performance



Interleaved Partitioning (Iteration 2)



Data Race in Parallel Thread Execution

thread1: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

thread2: $\text{Old} \leftarrow \text{Mem}[x]$
 $\text{New} \leftarrow \text{Old} + 1$
 $\text{Mem}[x] \leftarrow \text{New}$

- **Old and New are per-thread register variables.**
- **Question 1: If Mem[x] was initially 0, what would the value of Mem[x] be after threads 1 and 2 have completed?**
- **Question 2: What does each thread get in their Old variable?**
- **Unfortunately, the answers may vary according to the relative execution timing between the two threads, which is referred to as a data race.**

Purpose of Atomic Operations – To Ensure Good Outcomes

```
thread1:  Old ← Mem[x]  
          New ← Old + 1  
          Mem[x] ← New
```

```
thread2:  Old ← Mem[x]  
          New ← Old + 1  
          Mem[x] ← New
```

Or

```
thread1:  Old ← Mem[x]  
          New ← Old + 1  
          Mem[x] ← New
```

```
thread2:  Old ← Mem[x]  
          New ← Old + 1  
          Mem[x] ← New
```

Key Concepts of Atomic Operations

- **A read-modify-write operation performed by a single hardware instruction on a memory location address**
 - **Read the old value, calculate a new value, and write the new value to the location**
- **The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete**
 - **Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue**
 - **All threads perform their atomic operations serially on the same location**

Atomic Operations in CUDA

- Performed by calling functions that are translated into single instructions (a.k.a. *intrinsic functions* or *intrinsics*)
 - Atomic `add`, `sub`, `inc`, `dec`, `min`, `max`, `exch` (exchange), `CAS` (compare and swap)
 - Read CUDA C programming Guide 4.0 or later for details
- Atomic Add
 - `int atomicAdd(int* address, int val);`
 - reads the 32-bit word `old` from the location pointed to by `address` in global or shared memory, computes `(old + val)`, and stores the result back to memory at the same address. The function returns `old`.

A Basic Text Histogram Kernel

- The kernel receives a pointer to the input buffer of byte values
- Each thread processes the input in a strided pattern

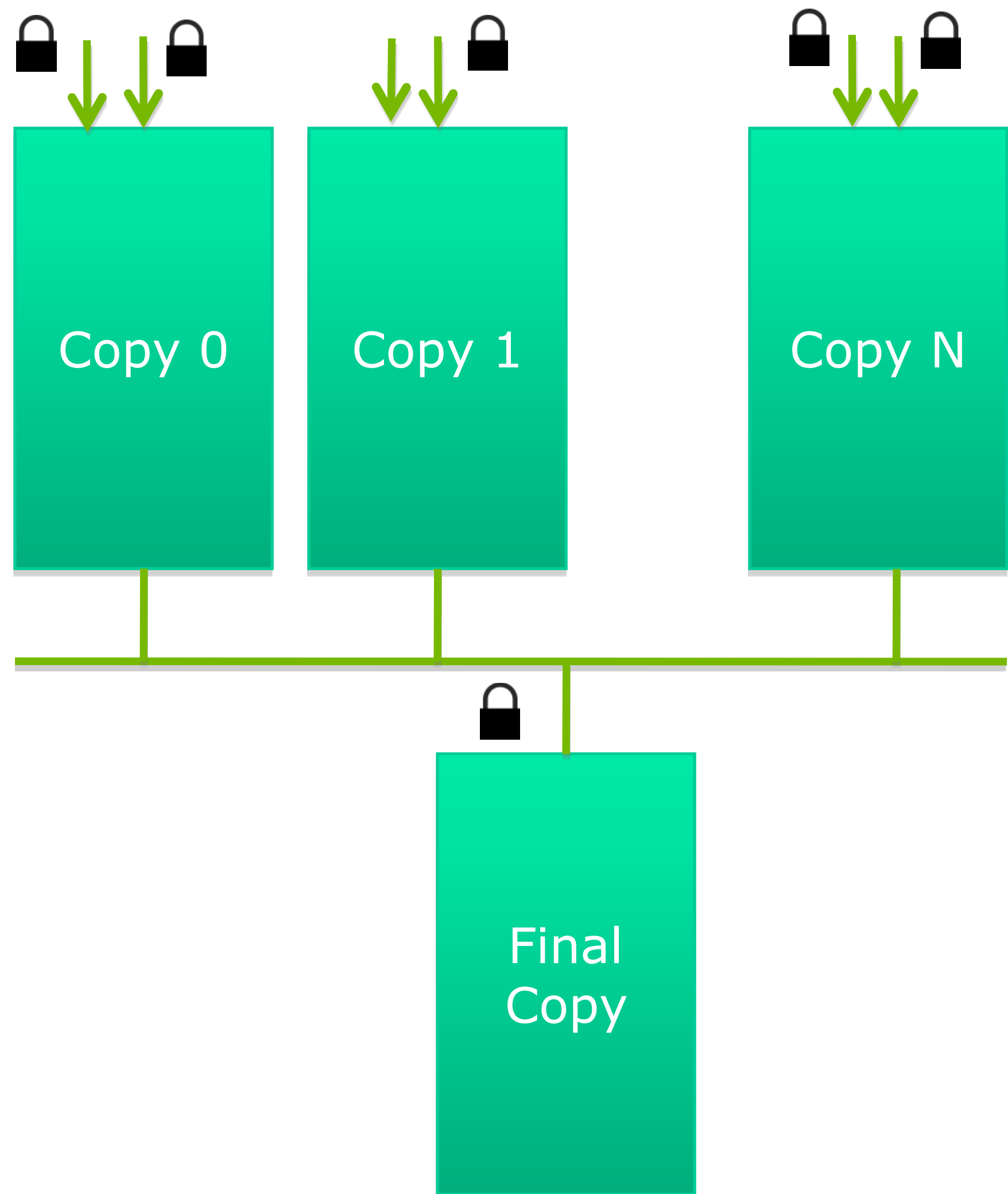
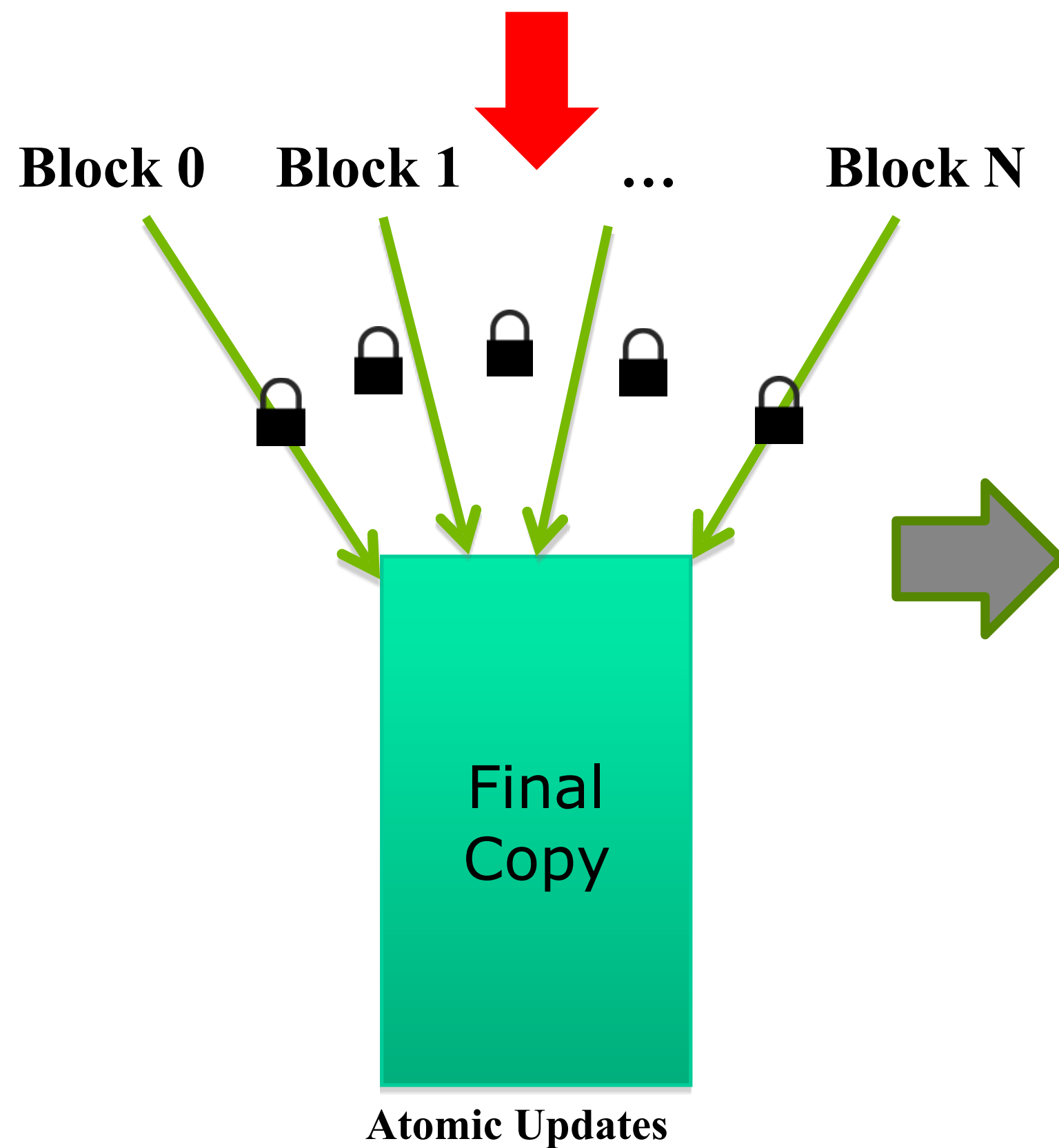
```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // 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;
    }
}
```

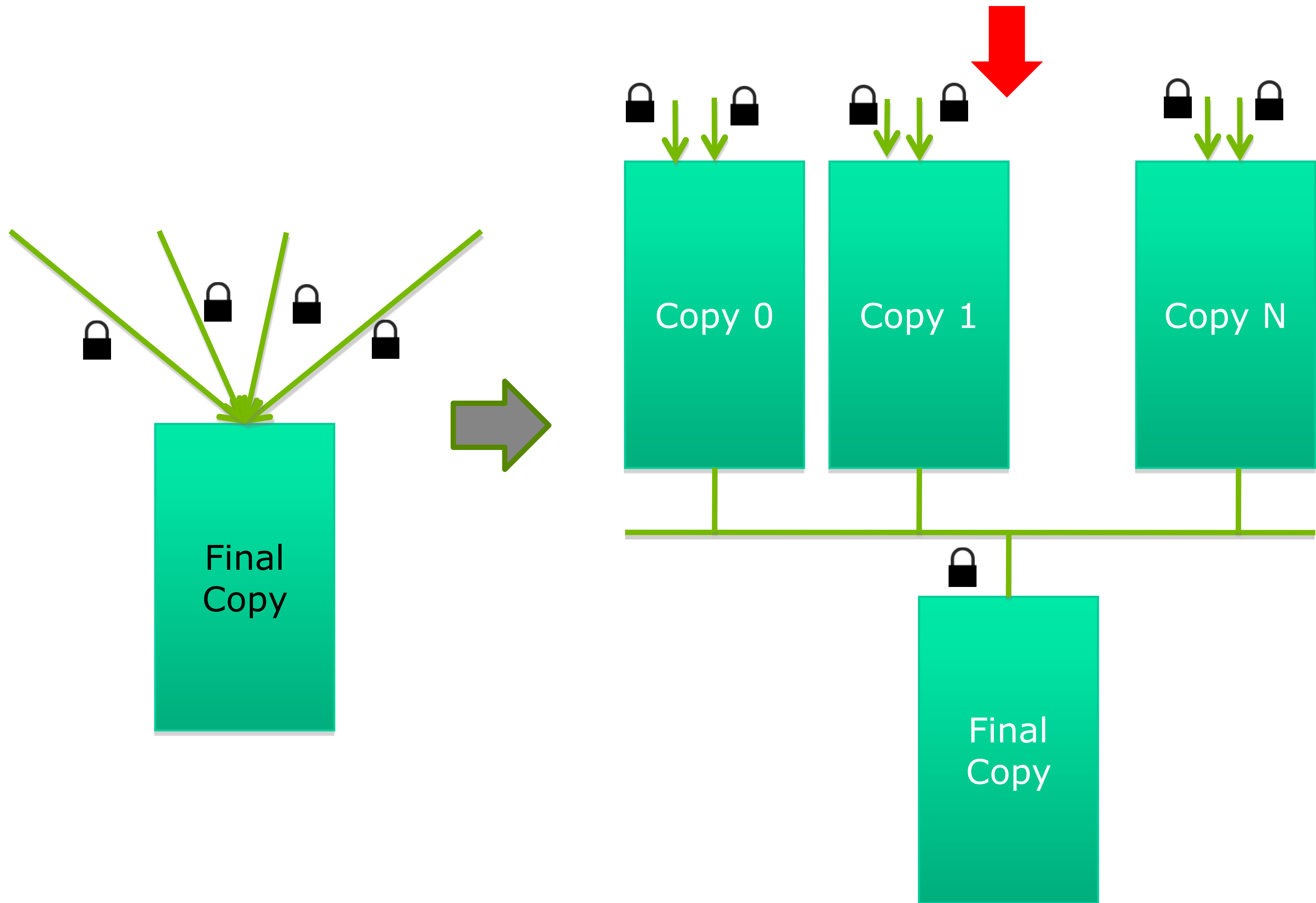
Privatization

Heavy contention and
serialization

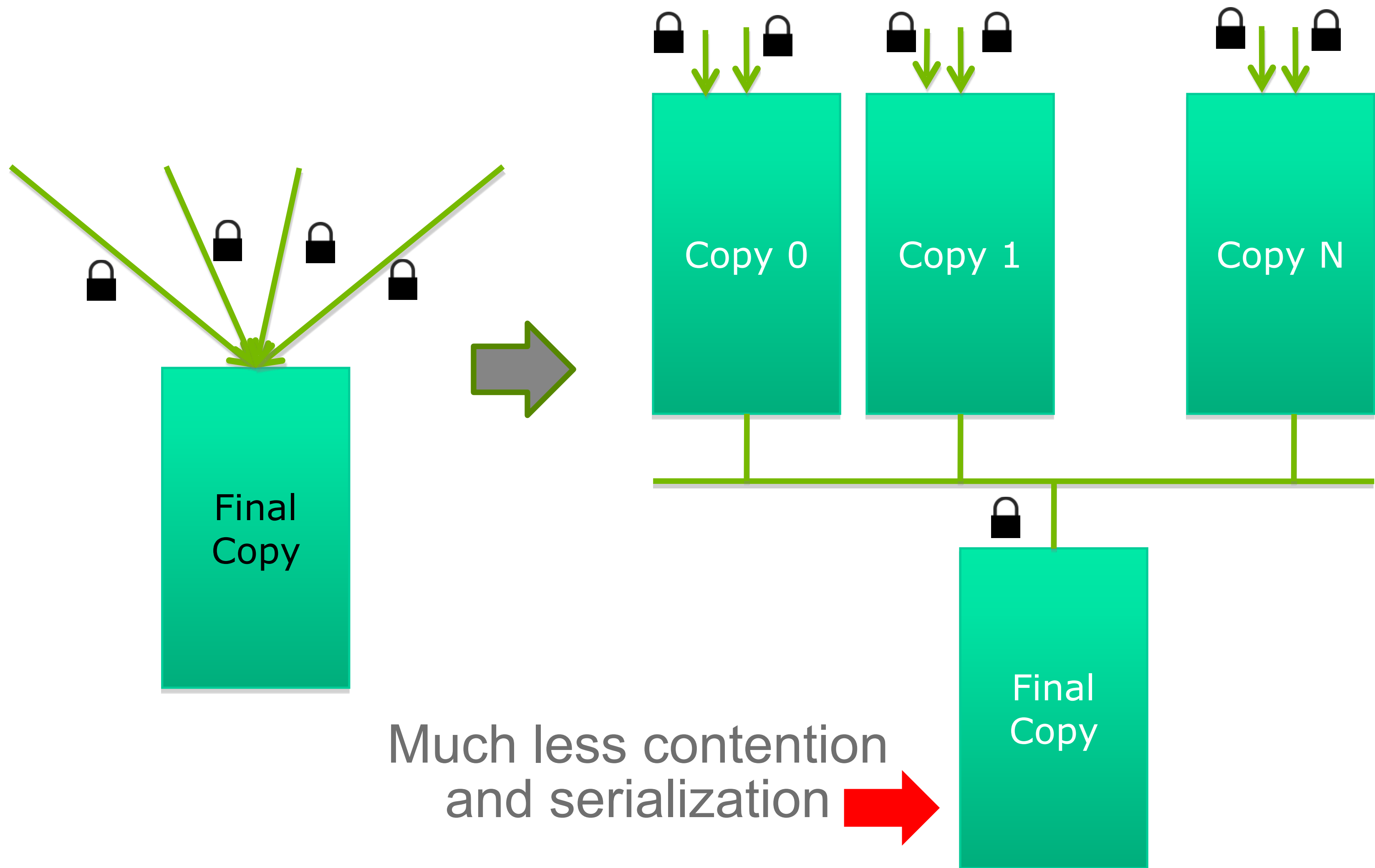


Privatization (cont.)

Much less contention and
serialization



Privatization (cont.)



Cost and Benefit of Privatization

■ Cost

- Overhead for creating and initializing private copies**
- Overhead for accumulating the contents of private copies into the final copy**

■ Benefit

- Much less contention and serialization in accessing both the private copies and the final copy**
- The overall performance can often be improved more than 10x**

Shared Memory Atomics for Histogram

- Each subset of threads are in the same block
- Much higher throughput than DRAM (100x) or L2 (10x) atomics
- Less contention – only threads in the same block can access a shared memory variable
- This is a very important use case for shared memory!

Shared Memory Atomics Requires Privatization

- Create private copies of the histo[] array for each thread block

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    __shared__ unsigned int histo_private[7];
```

Shared Memory Atomics Requires Privatization

- Initialize the bin counters in the private copies of histo[]

```
__global__ void histo_kernel(unsigned char *buffer,  
                             long size, unsigned int *histo)  
{  
    __shared__ unsigned int histo_private[7];  
  
    if (threadIdx.x < 7) histo_private[threadIdx.x] = 0;  
    __syncthreads();  
}
```

Build Private 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]/4) , 1) ;
    i += stride;
}
```

Build Final Histogram

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

if (threadIdx.x < 7) {
    atomicAdd(&histo[threadIdx.x], private_histo[threadIdx.x]);
}
```

More on Privatization

- **Privatization is a powerful and frequently used technique for parallelizing applications**
- **The operation needs to be associative and commutative**
 - **Histogram add operation is associative and commutative**
 - **No privatization if the operation does not fit the requirement**
- **The private histogram size needs to be small**
 - **Fits into shared memory**
- **What if the histogram is too large to privatize?**
 - **Sometimes one can partially privatize an output histogram and use range testing to go to either global memory or shared memory**

Basic Efficiency Rules

- **Develop algorithms with a data parallel mindset**
- **Minimize divergence of execution within blocks**
- **Maximize locality of global memory accesses**
- **Exploit per-block shared memory as scratchpad**
- **Expose enough parallelism**

Summing Up

- **CUDA = C + a few simple extensions**
 - **makes it easy to start writing basic parallel programs**
- **Three key abstractions:**
 - **hierarchy of parallel threads**
 - **corresponding levels of synchronization**
 - **corresponding memory spaces**
- **Supports massive parallelism of manycore GPUs**