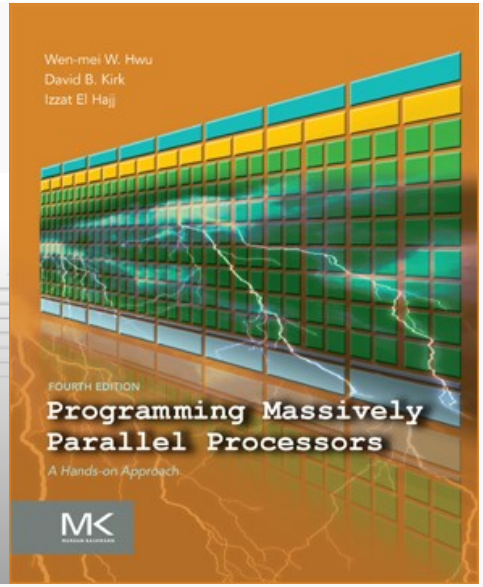
MORGAN KAUFMANN

Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 9 › Parallel Histogram



Histogram

- A **histogram** approximates the distribution of a dataset by:
  - Dividing the range of values into **bins** (or **buckets**)
  - Counting the number of values in the dataset that fall in each bin

- Example:
  - Color histogram of an image counts the number of pixels for each pixel value or range of values

Example: Color Histogram Code

- Sequential:

```
for(unsigned int i = 0; i < width*height; ++i) {
    unsigned char b = image[i];
    ++bins[b];
}
```

- Parallel:
  - Assign one thread per input element

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < width*height) {
    unsigned char b = image[i];
    ++bins[b];
}
```

What's wrong with this implementation?

Data Races

- A **data race** occurs when multiple threads access the same memory location concurrently without ordering and at least one access is a write
  - Data races may result in unpredictable program output

- Example:

Thread A	Thread B
`oldVal = bins[b]`	`oldVal = bins[b]`
`newVal = oldVal + 1`	`newVal = oldVal + 1`
`bins[b] = newVal`	`bins[b] = newVal`

  - If both threads have the same `b` and `bins[b]` is initially 0, the final value of `bins[b]` could be **2** or **1**

Data Races Example

Time	Thread A	Thread B
1	`oldVal = bins[b]`	
2	`newVal = oldVal + 1`	
3	`bins[b] = newVal`	
4		`oldVal = bins[b]`
5		`newVal = oldVal + 1`
6		`bins[b] = newVal`

In these two scenarios, the final value of bins[b] will be 2

Time	Thread A	Thread B
1		`oldVal = bins[b]`
2		`newVal = oldVal + 1`
3		`bins[b] = newVal`
4	`oldVal = bins[b]`	
5	`newVal = oldVal + 1`	
6	`bins[b] = newVal`	

Data Races Example

Time	Thread A	Thread B
1	`oldVal = bins[b]`	
2	`newVal = oldVal + 1`	
3		`oldVal = bins[b]`
4	`bins[b] = newVal`	
5		`newVal = oldVal + 1`
6		`bins[b] = newVal`

In these two scenarios (and many others), the final value of bins[b] will be 1

Time	Thread A	Thread B
1		`oldVal = bins[b]`
2		`newVal = oldVal + 1`
3	`oldVal = bins[b]`	
4		`bins[b] = newVal`
5	`newVal = oldVal + 1`	
6	`bins[b] = newVal`	

Mutual Exclusion

- To avoid data races, concurrent read-modify-write operations to the same memory location need to be made **mutually exclusive** to enforce ordering

- One way to do this on CPUs is using **locks** (mutex)
  - Example:

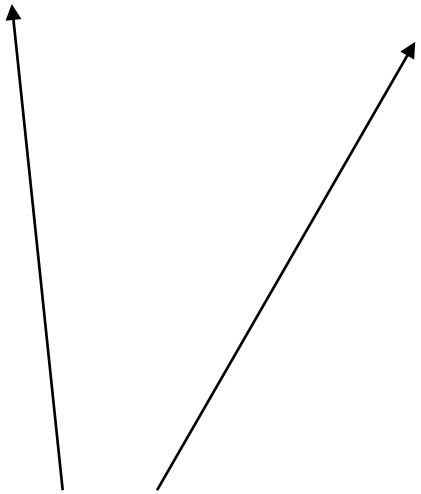
```
mutex_lock(lock);

++bins[b];

mutex_unlock(lock);
```

Mutual Exclusion Example

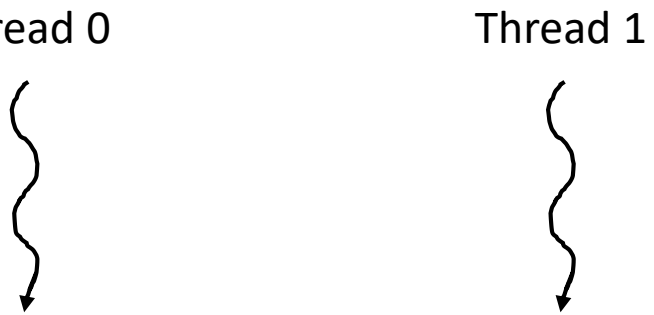
Time	Thread A	Thread B
1	`mutex_lock(lock)`	
2	`oldVal = bins[b]`	
3	`newVal = oldVal + 1`	
4	`bins[b] = newVal`	
5	`mutex_unlock(lock)`	
6		`mutex_lock(lock)`
7		`oldVal = bins[b]`
8		`newVal = oldVal + 1`
9		`bins[b] = newVal`
10		`mutex_unlock(lock)`



Thread B cannot acquire the lock before Thread A releases it (or vice versa)

Locks and SIMD Execution

Assume threads 0 and 1 are in the same warp and try to acquire the same lock




```
mutex_lock(lock)

oldVal = bins[b]

...

mutex_unlock(lock)
```

Acquires lock

Waiting for Thread 0 to release lock

Waiting for Thread 1 to complete previous instruction (SIMD)

Using locks with SIMD execution may cause deadlock

Atomic Operations

- **Atomic operations** perform read-modify-write with a single ISA instruction
- The hardware guarantees that no other thread can access the memory location until the operation completes
- Concurrent atomic operations to the same memory location are serialized by the hardware

Atomic Operations in CUDA

- Atomic Add:

```
T atomicAdd(T* address, T val)
```

  - `T` can be int, unsigned int, float, double, etc.
  - Reads the value stored at `address`, adds `val` to it, stores the new value at `address`, and returns the old value originally stored
  - Function call translated to a single ISA instruction
    - Such special functions are called *intrinsics*

- Other operations available:
  - sub, min, max, inc, dec, and, or, xor, exchange, compare and swap

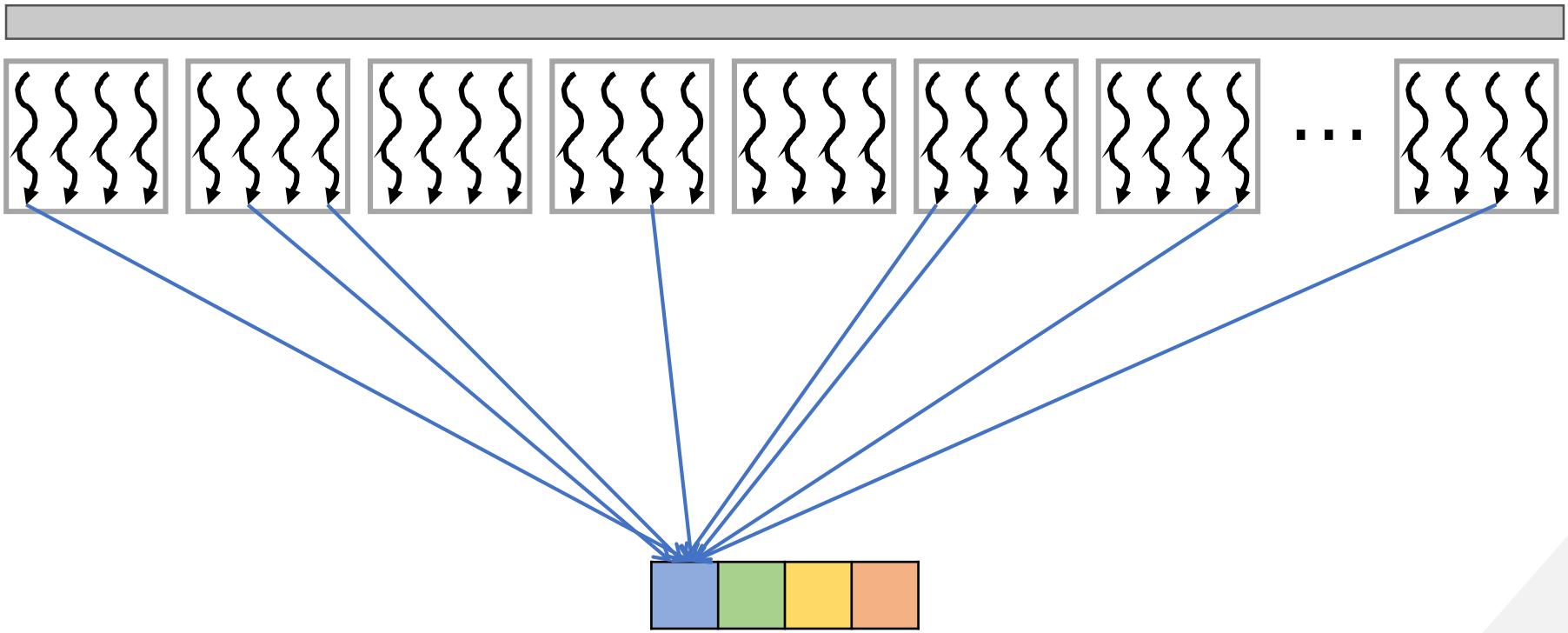
Code with Atomic Operations

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < width*height) {
    unsigned char b = image[i];
    ++bins[b];
}
```

Race condition

```
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
if(i < width*height) {
    unsigned char b = image[i];
    atomicAdd(&bins[b], 1);
}
```

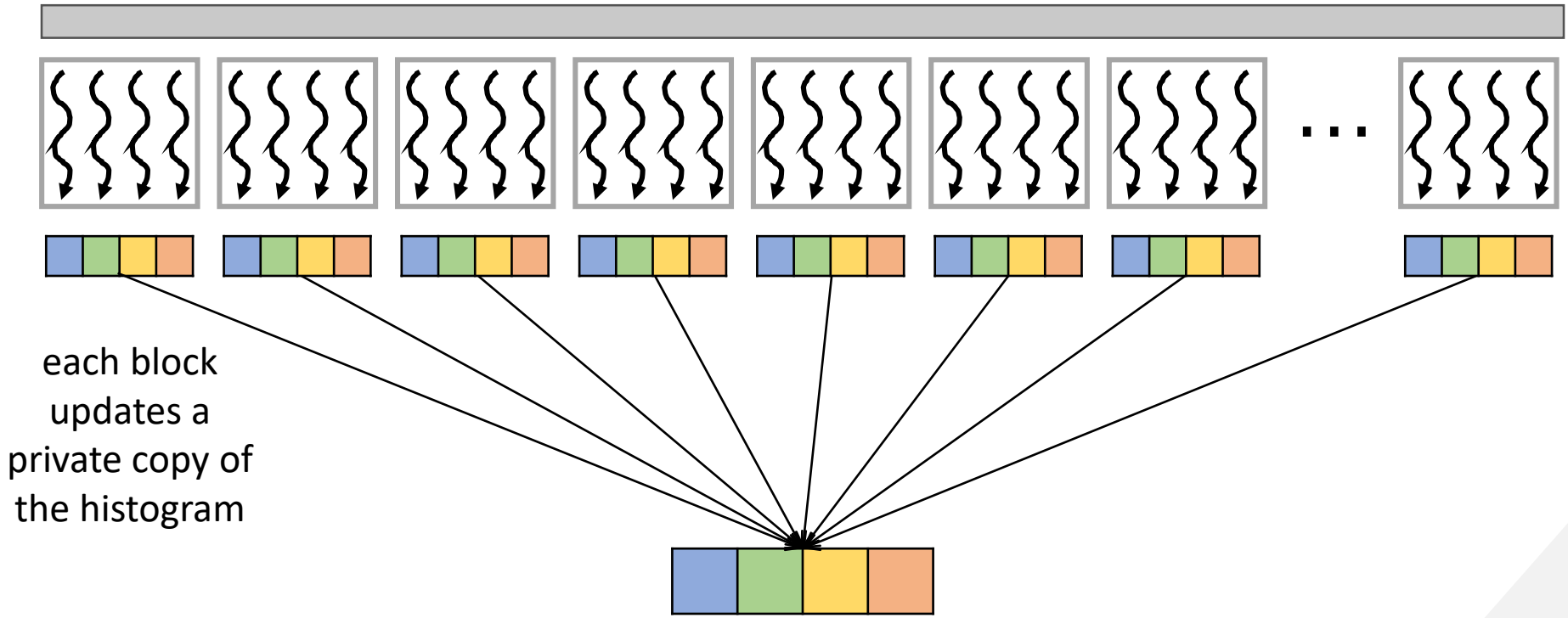
High Latency



Atomic operations on global memory have high latency

- Need to wait for both read and write to complete
- Need to wait if there are other threads accessing the same location (high probability of contention)

Privatizing the Histogram



Privatization

- **Privatization** is an optimization where multiple private copies of an output are maintained, then the global copy is updated on completion
  - Operations on the output must be associative and commutative because the order of updates has changed

- Advantage: reduces contention on the global copy

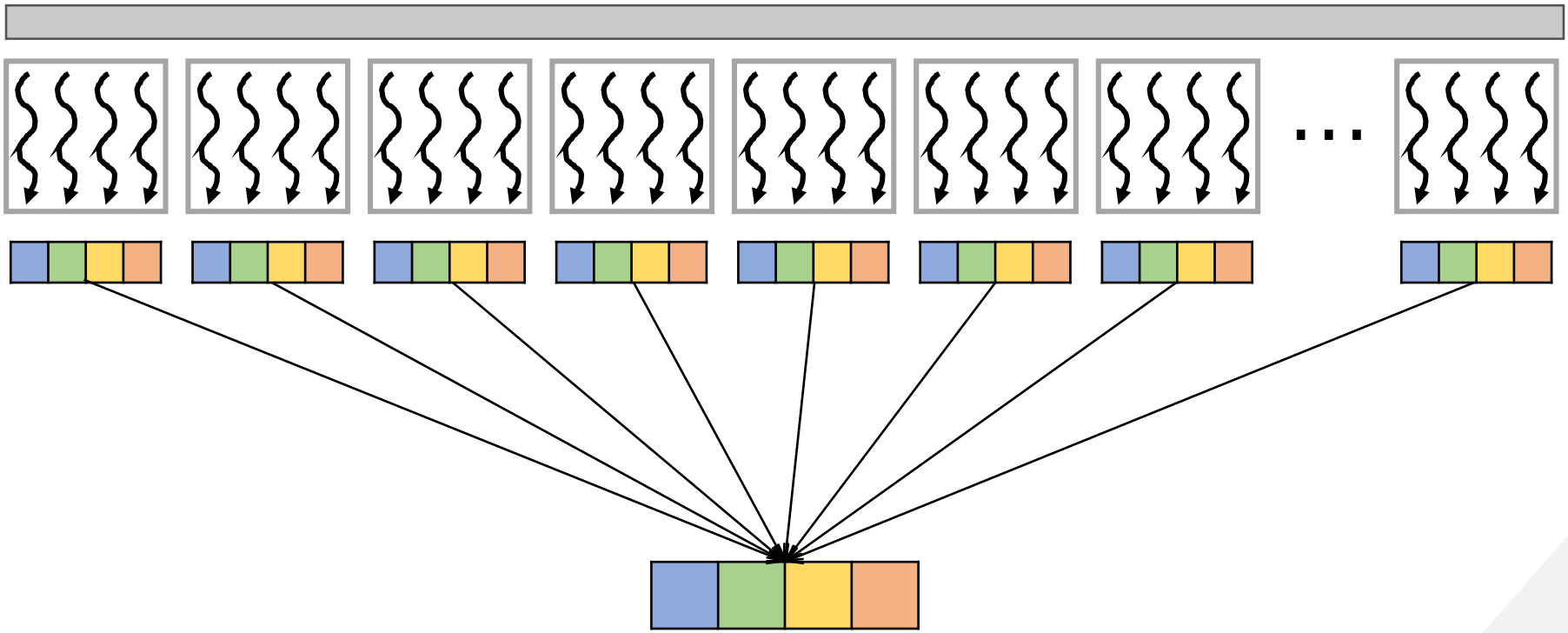
Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (minimizing idle cores during SIMD execution)	-	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread in order to reduce the “price of parallelism” when it is incurred unnecessarily

Other Optimization Opportunities

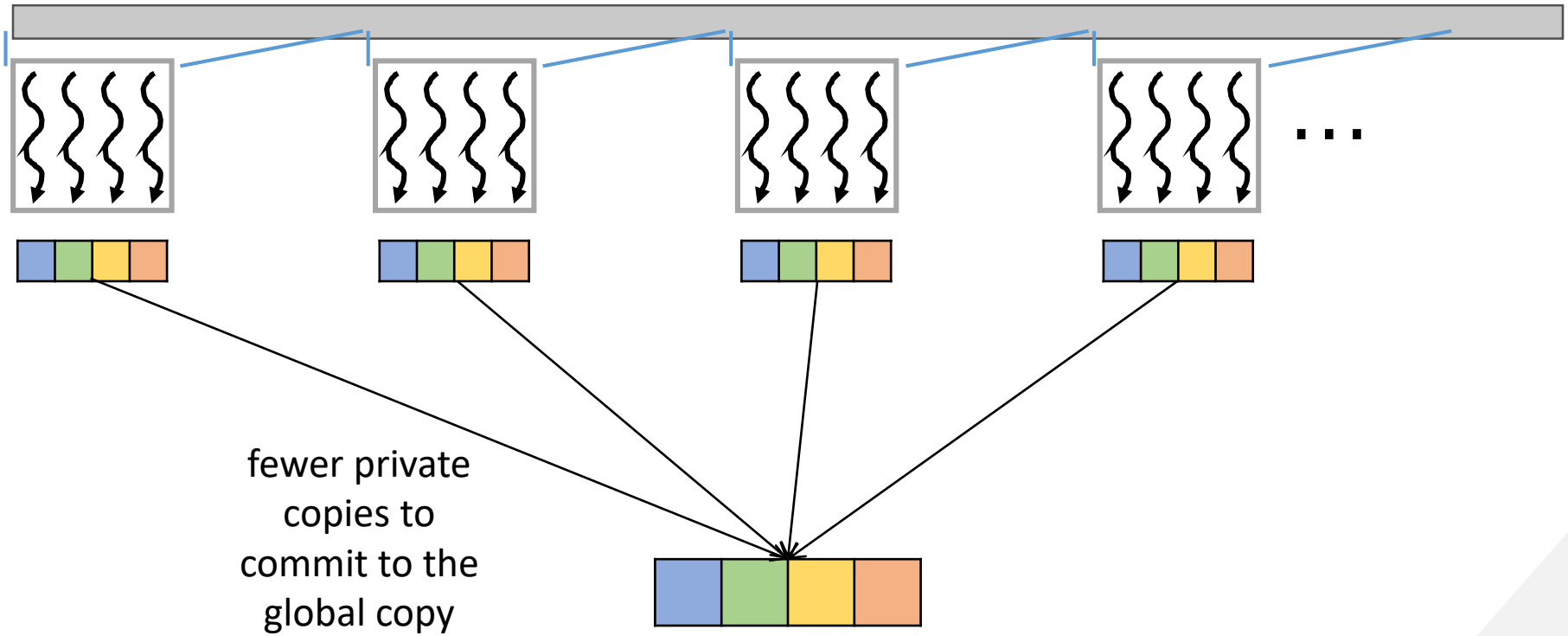
- Are memory accesses coalesced?
  - Yes (for input data)
- Is there control divergence?
  - Negligible (only at the boundary check)
- Is there data reuse?
  - Input data: No
  - Output data: Yes
    - Multiple threads in a block may update the same bin in the private histogram
    - Place the private histogram in shared memory (if it fits)
- Is there a price for parallelizing the work?
  - Atomic updates from the private copies to the global copy happen once per block
  - Use thread coarsening to reduce the number of blocks/updates

Before Coarsening



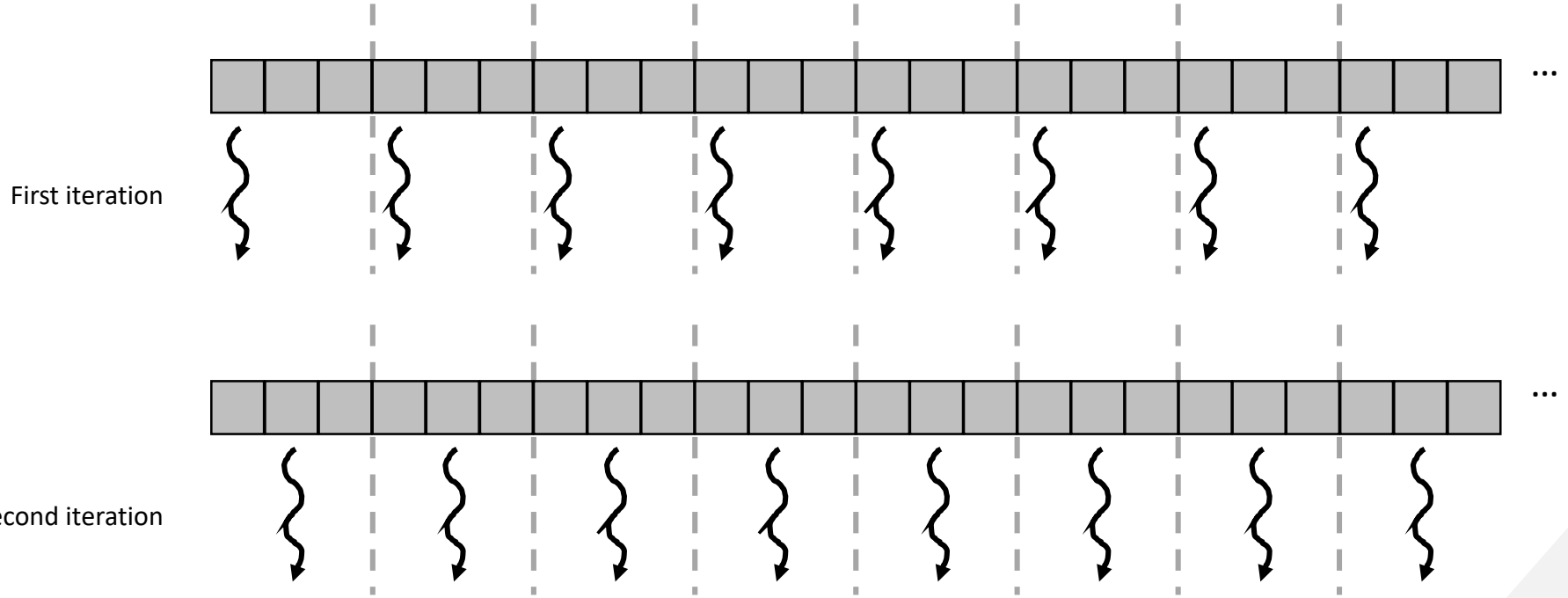
After Coarsening

each block takes a larger input segment



Coarsening Strategy

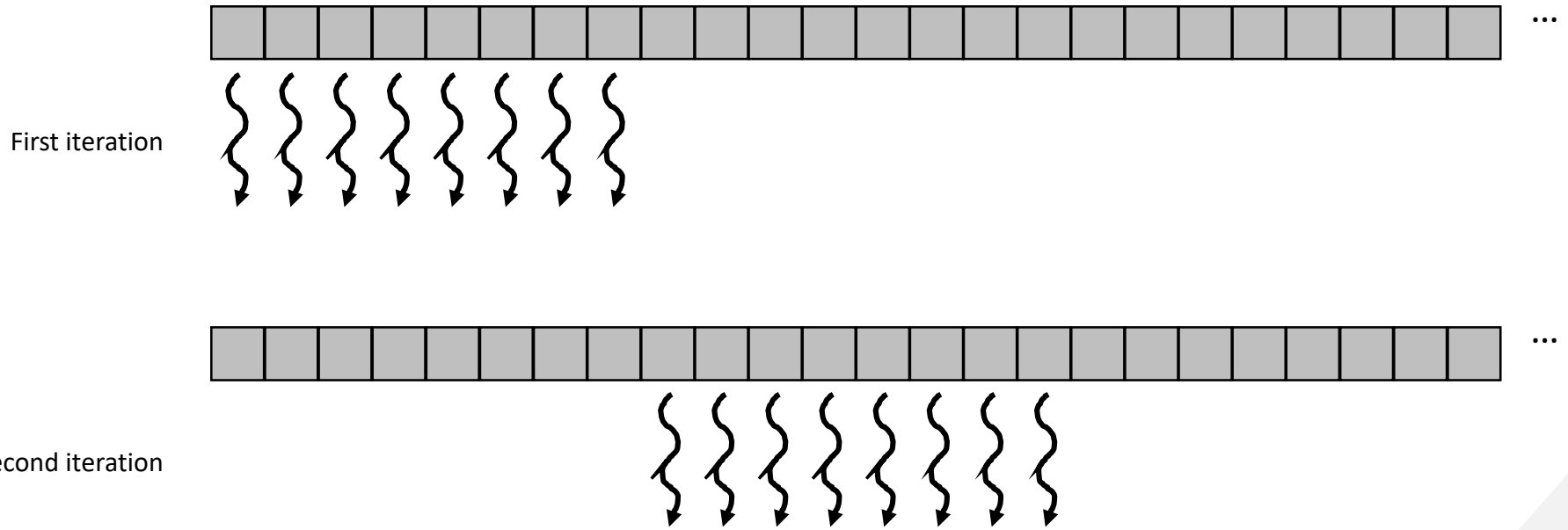
- How to assign multiple inputs to each thread?



Accesses are not coalesced

Coarsening Strategy

- How to assign multiple inputs to each thread?



Accesses are coalesced

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.