Lecture 11 - GP-GPU and High Performances Computing

Histogram

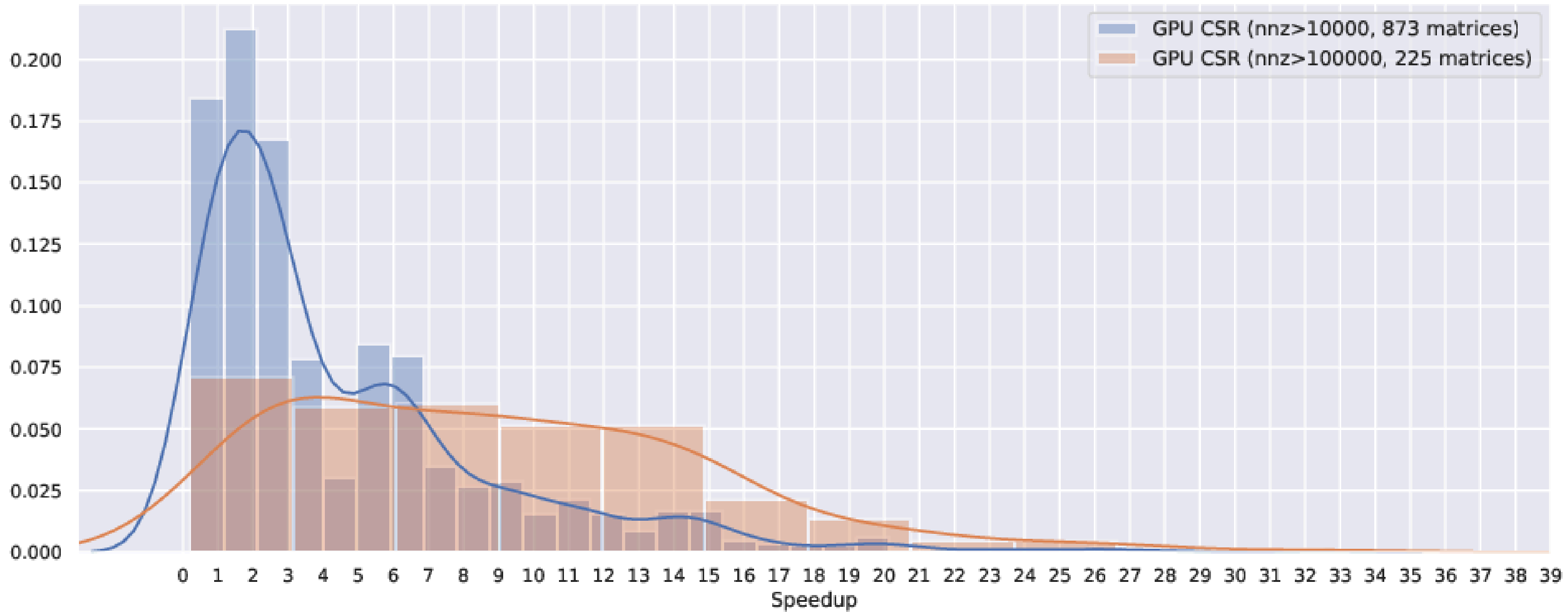
Previously

Organize storage of sparse matrices in order to:

- Minimize memory occupancy
- Increase throughput
- Limit data duplication
- Limit tasks duplication

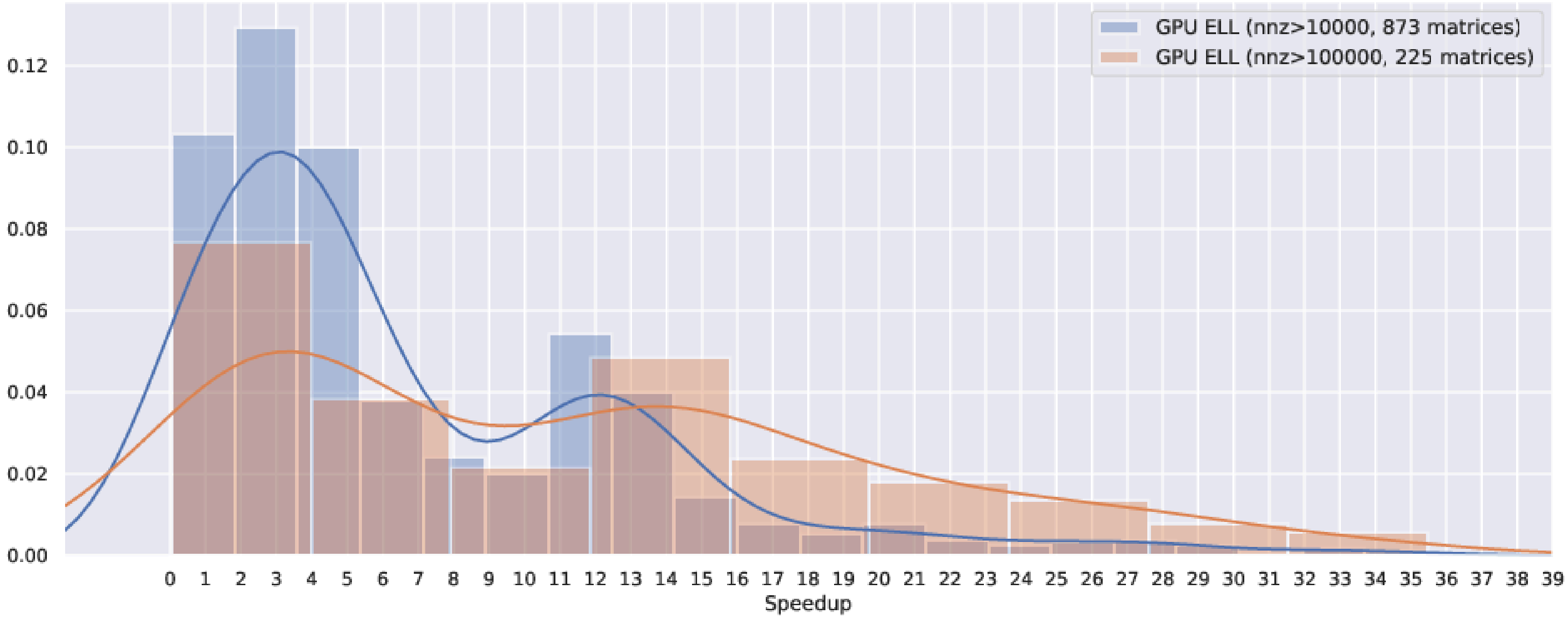
Performances comparison

CSR



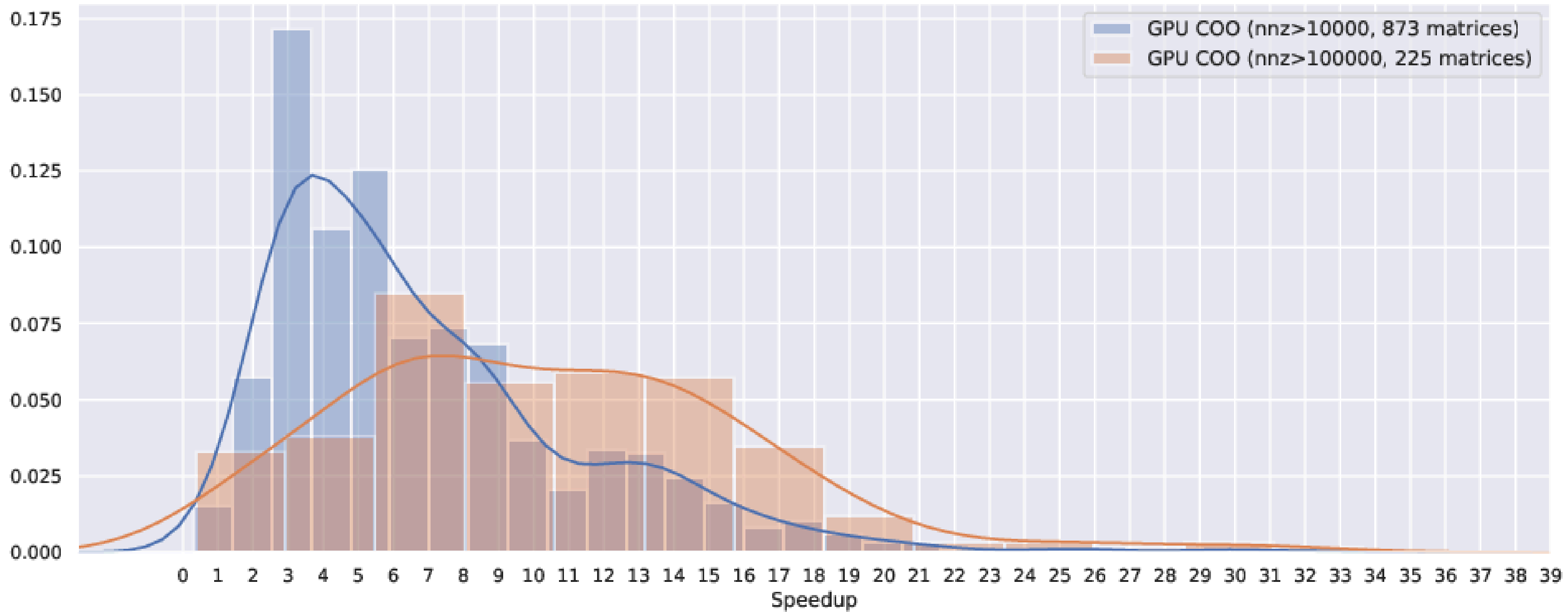
GPU CSR (nnz>10000, 873 matrices)
GPU CSR (nnz>100000, 225 matrices)
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Speedup

ELL



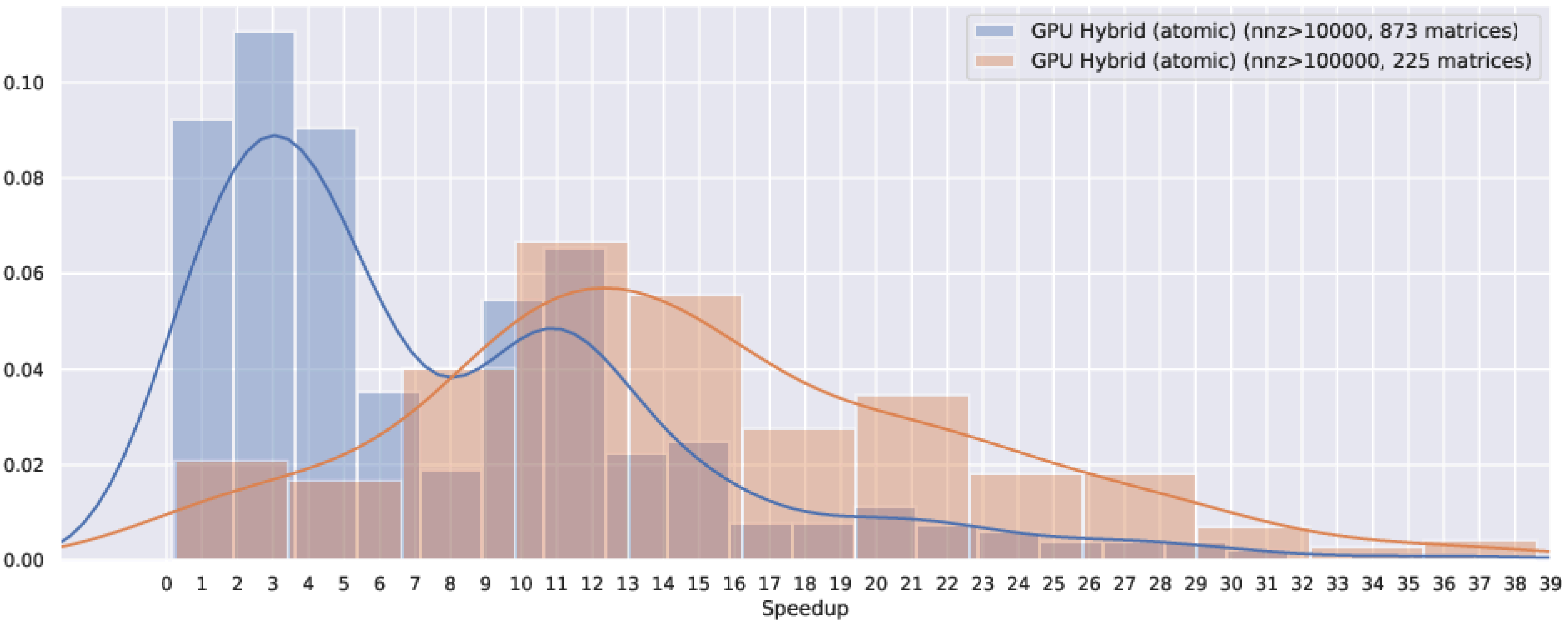
GPU ELL (nnz>10000, 873 matrices)
GPU ELL (nnz>100000, 225 matrices)
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Speedup

COO



0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
GPU COO (nnz>10000, 873 matrices)
GPU COO (nnz>100000, 225 matrices)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Speedup

Hybrid



GPU Hybrid (atomic) (nnz>10000, 873 matrices)
GPU Hybrid (atomic) (nnz>100000, 225 matrices)
0.10
0.08
0.06
0.04
0.02
0.00
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
Speedup

Reduction techniques (prerequisites)

Reduction α: interleaved addressing

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  // do reduction in shared mem
  for (unsigned int s = 1; s < blockDim.x; s *= 2) {
    __syncthreads();
    int index = 2 * s * tid;
    if (index < blockDim.x) {
      sdata[tid] = sdata[tid] + sdata[tid + s];
    }
    // Thread 0 writes result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
  }
}
```

Reduction β: strided access

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  // do reduction in shared mem
  for (int s = 1; s < blockDim.x; s *= 2) {
    __syncthreads();
    if (threadIdx.x % (2 * s) == 0)
      sdata[threadIdx.x] += sdata[threadIdx.x + s];
  }
  // Thread 0 writes result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Expected gain: **×2.5** (compared to naive baseline)

Reduction γ: Sequential Addressing

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
  sdata[tid] = g_idata[i];
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Expected gain: **×2** (compared to naive baseline)

Reduction δ: add during load

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x/2; s > 0; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Expected gain: **×1.8** (incremental over γ, total ~×3.6 vs baseline)

Reduction ε: unroll warp

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x/2; s > 32; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  if (tid < 32) warpReduce(sdata, tid);
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Expected gain: **×1.8** (incremental over δ, total ~×3.6 vs baseline) *Same total gain as δ, but with better warp utilization*

Reduction: unrolling

```
__device__ void warpReduce(volatile int* sdata, int tid) {
  sdata[tid] += sdata[tid + 32];
  sdata[tid] += sdata[tid + 16];
  sdata[tid] += sdata[tid + 8];
  sdata[tid] += sdata[tid + 4];
  sdata[tid] += sdata[tid + 2];
  sdata[tid] += sdata[tid + 1];
}
```

Reduction: compile time unrolling

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
  if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
  if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
  if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
  if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
  if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
  if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
}
```

Expected gain: **×1.4** (incremental over ε, total ~×5.0 vs baseline)

Reduction ζ: unroll warp

```
reduce(int *g_idata, int *g_odata) {
  extern __shared__ int sdata[];
  // load shared mem
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
  unsigned int gridSize = blockDim.x*2*gridDim.x;
  sdata[tid] = 0;
  while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockDim.x];
    i += gridSize;
  }
  __syncthreads();
  // do reduction in shared mem
  for (unsigned int s = blockDim.x/2; s > 32; s >>= 1) {
    if (tid < s) {
      sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
  }
  if (tid < 32) warpReduce(sdata, tid);
  // write result for this block to global mem
  if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Connecting reduction to histograms

Histograms are a form of reduction:

- Input: array of values
- Output: count per bin (category)
- Operation: associative and commutative (addition)

But with specific challenges:

- Multiple output bins (not a single sum)
- High contention (many threads updating same bins)
- Requires atomic operations or privatization

The reduction techniques we learned help optimize histogram computation!

Objectives

The key techniques for efficient parallel histogram computation:

- Better utilization of on-chip memory (shared memory)
- Reducing memory bandwidth consumption
- Minimizing contention through privatization
- Understanding atomic operations and race conditions

Histogram applications

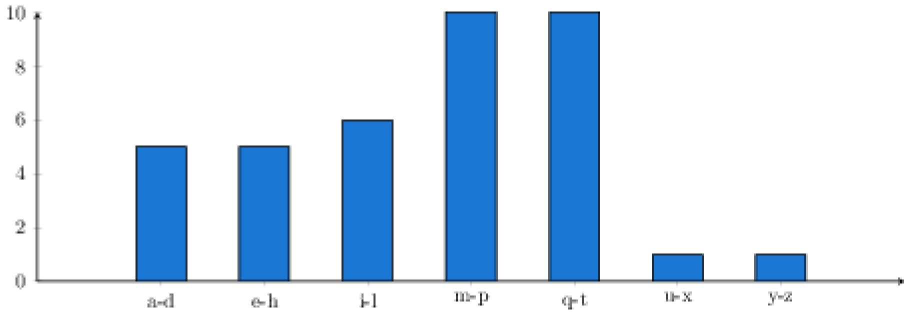
Histograms are used for extracting notable features and patterns from large data sets:

- Feature extraction for object recognition in images
- Fraud detection in credit card transactions
- Correlating heavenly object movements in astrophysics
- Statistical analysis and data visualization

Basic histogram computation: For each element in the data set, use the value to identify a "bin counter" to increment

Example with text data

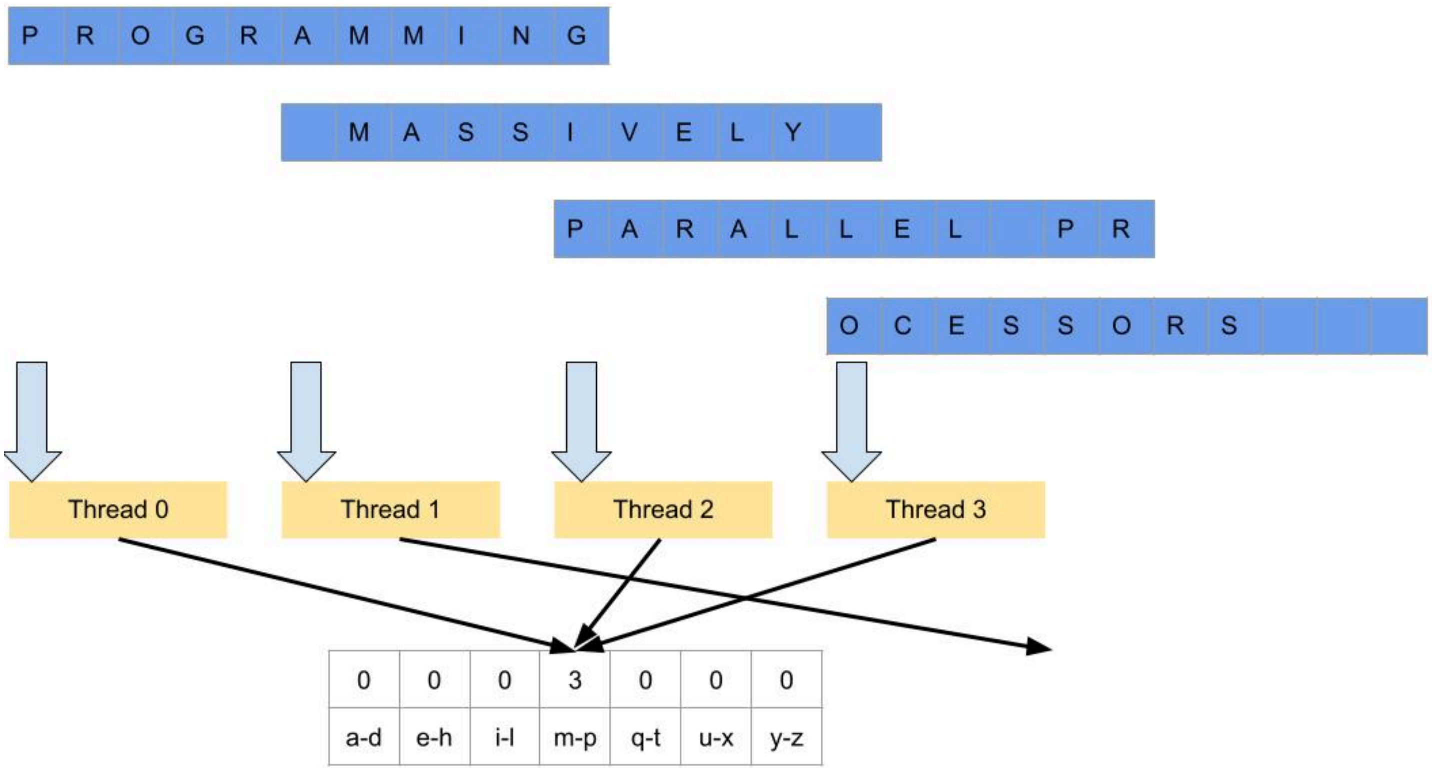
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, m-p, q-t, u-x, y-z
- For each character in an input string, increment the appropriate bin counter
- In the phrase "Programming Massively Parallel Processors" the output histogram is:



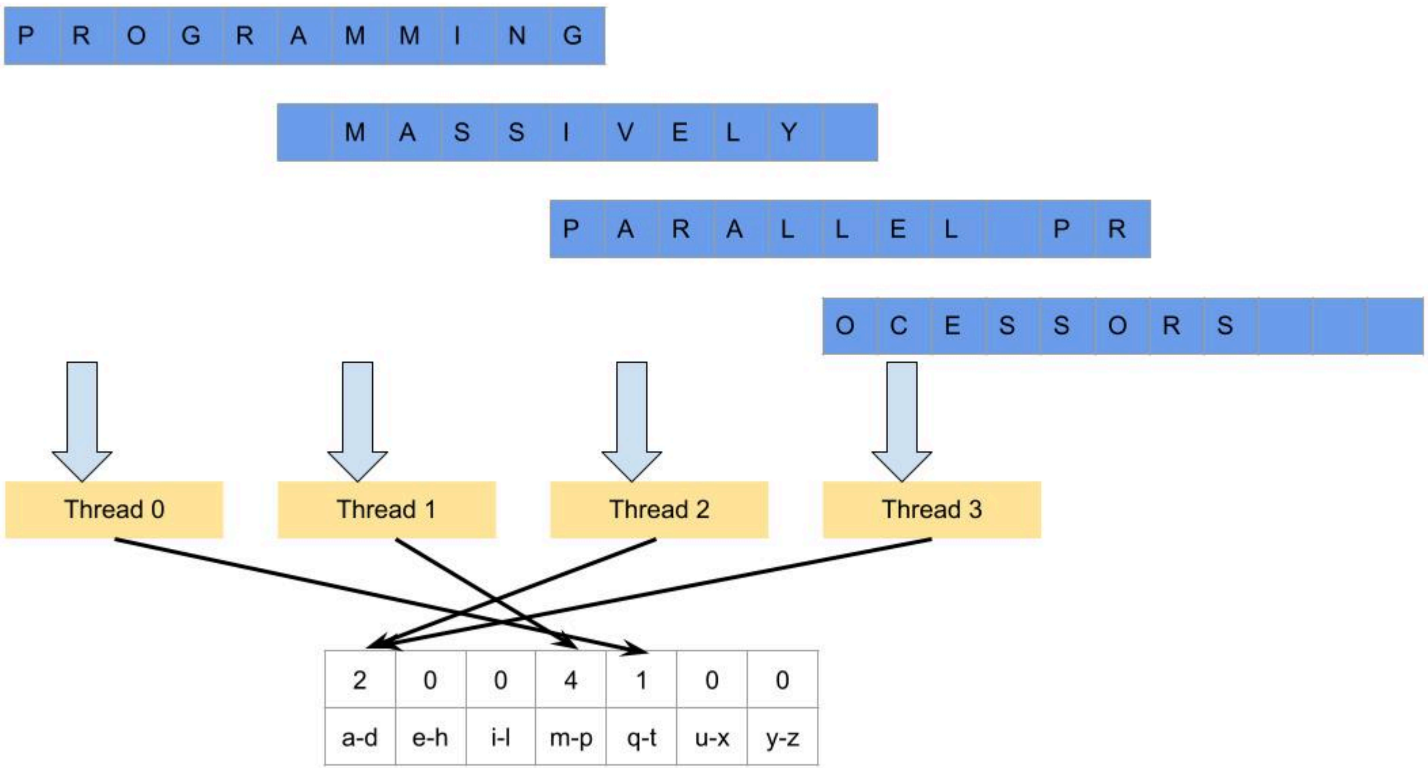
Algorithm 1: Simple binning

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

Algorithm 1: iteration 1

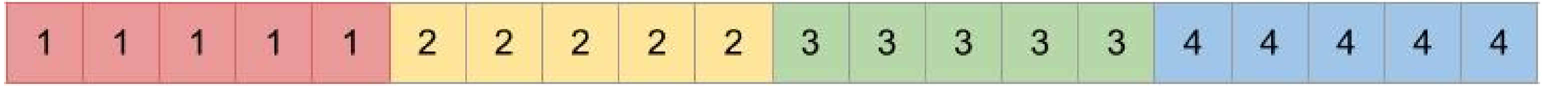


Algorithm 1: iteration 2



Sectioned partitioning: poor efficiency

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

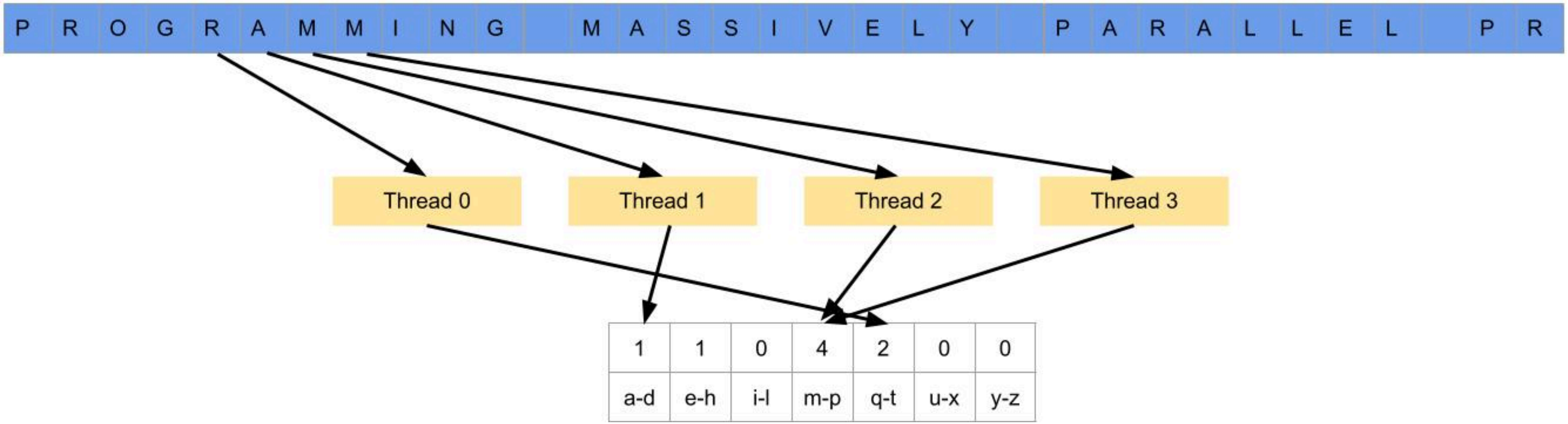


Interleaved partitioning: better efficiency

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

Algorithm 1b: iteration 2

Interleaved memory accesses



Mapping histogram count to reduce

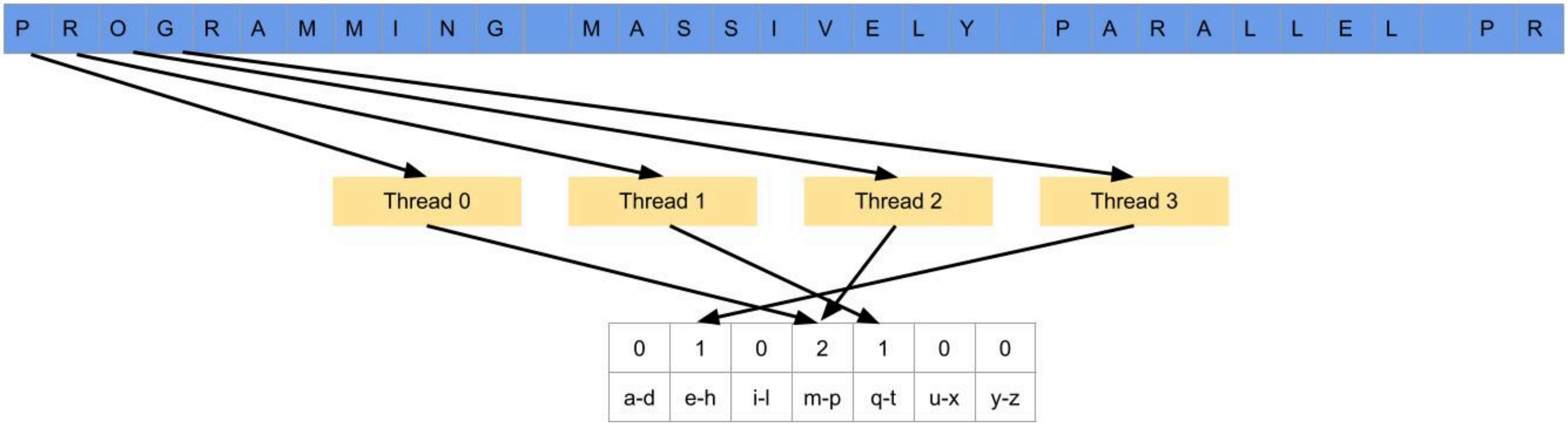
	a-d	e-h	i-l	m-p	q-t	u-x	y-z
P	0	0	0	1	0	0	0
R	0	0	0	0	1	0	0
O	0	0	0	1	0	0	0
G	0	1	0	0	0	0	0
R	0	0	0	0	1	0	0
A	1	0	0	0	0	0	0
M	0	0	0	1	0	0	0
M	0	0	0	1	0	0	0
I	0	0	1	0	0	0	0
N	0	0	0	1	0	0	0
G	0	1	0	0	0	0	0

Reduction across threads

	a-d	e-h	i-l	m-p	q-t	u-x	y-z
P	0	0	0	1	0	0	0
R	0	0	0	0	1	0	0
O	0	0	0	1	0	0	0
G	0	1	0	0	0	0	0
R	0	0	0	0	1	0	0
A	1	0	0	0	0	0	0
M	0	0	0	1	0	0	0
M	0	0	0	1	0	0	0
I	0	0	1	0	0	0	0
N	0	0	0	1	0	0	0
G	0	1	0	0	0	0	0

1	2	1	5	2	0	0

Interleaved memory accesses



Read-Modify-Write Used in Collaboration Patterns

Example: Multiple bank tellers count the total amount of cash in the safe

- Each grab a pile and count
- Have a central display of the running total
- Whenever someone finishes counting a pile, read the current running total (read) and add the subtotal of the pile to the running total (modify-write)

A bad outcome:

- Some of the piles were not accounted for in the final total

A Common Parallel Service Pattern

Example: Multiple customer service agents serving customers

- System maintains: next customer number (I), next to serve (S)
- Each customer: read I, increment I (modify-write)
- Each agent: read S, increment S (modify-write)

Bad outcomes:

- Multiple customers get same number
- Multiple agents serve same customer

A Common Arbitration Pattern

Example: Multiple customers booking airline tickets in parallel

- Brings up a flight seat map (read)
- Decides on a seat
- Updates the seat map and marks the selected seat as taken (modify-write)

A bad outcome:

- Multiple passengers ended up booking the same seat

Data Race in Parallel Thread Execution

Thread 0

```
Old = Mem[x]
New = Old + 1
Mem[x] = New
```

Thread 1

```
Old = Mem[x]
New = Old + 1
Mem[x] = New
```

Old and New are per-thread register variables.

Question 1: If Memx was initially 0, what would the value of Memx be after Thread 0 and Thread 1 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**.

Correct execution scenarios

When operations don't overlap, both threads correctly update the value:

Scenario A: Thread 0 executes first, then Thread 1

- Thread 0: `Old = 0`, writes `Mem[x] = 1`
- Thread 1: `Old = 1`, writes `Mem[x] = 2`
- Result: ✅ `Mem[x] = 2` (correct)

Scenario B: Thread 1 executes first, then Thread 0

- Thread 1: `Old = 0`, writes `Mem[x] = 1`
- Thread 0: `Old = 1`, writes `Mem[x] = 2`
- Result: ✅ `Mem[x] = 2` (correct)

Both scenarios produce correct results when operations are serialized.

Execution scenario with data race

When read operations overlap, both threads read the same initial value:

Thread 0

```
Old = Mem[x] // 0
New = Old + 1 // 1

Mem[x] = New // 1
```

Thread 1

```


Old = Mem[x] // 0

New = Old + 1 // 1
Mem[x] = New // 1
```

Key problem: Both threads read the same initial value (0) because their read operations overlap.

- Thread 0: `Old = 0` (reads before Thread 1 writes)
- Thread 1: `Old = 0` (reads before Thread 0 writes)
- Both compute `New = 1` and write it
- Result: ⚠️ `Mem[x] = 1` (incorrect! Should be 2)

This is a **data race**: the final value depends on execution timing.

Atomic operation: goal

The goal of atomic operation is to ensure serialized access:

- Only one thread can execute the read-modify-write sequence at a time
- Operations on the same location are queued and executed serially
- Result: consistent final value regardless of execution order

Atomic operation: example

With atomic operations, both execution orders produce correct results:

Thread 0 first

```
Old = atomicAdd(&Mem[x], 1) // 0
Mem[x] = 1
```

Thread 1 second

```
Old = atomicAdd(&Mem[x], 1) // 1
Mem[x] = 2
```

Problem illustration

`Mem[x] = 0`

Thread 0

```
Old = Mem[x] // 0
New = Old + 1 // 1

Mem[x] = New // 1
```

Thread 1

```
Old = Mem[x] // 0

New = Old + 1 // 1
Mem[x] = New // 1
```

- Both threads receive 0 in `Old`
- `Mem[x]` becomes 1 (incorrect!)

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

Performed by calling functions that are translated into single instructions:

`add`, `sub`, `inc`, `dec`, `min`, `max`, `exch` (exchange), `CAS` (compare and swap)

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.

More Atomic Adds in CUDA

- Unsigned 32-bit integer atomic add

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

- Unsigned 64-bit integer atomic add

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

- Single-precision floating-point atomic add (capability > 2.0)

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

Basic kernel for histogram

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) {
    int alphabet_position = buffer[i] - 'a';
    if (alphabet_position >= 0 && alphabet_position < 26)
      // Group letters into bins of 4: a-d (bin 0), e-h (bin 1), i-l (bin 2), etc.
      atomicAdd(&(histo[alphabet_position/4]), 1);
    i += stride;
  }
}
```

Atomic Operations on Global Memory

- An atomic operation on a DRAM location starts with a read, which has a latency of a few hundred cycles
- The atomic operation ends with a write to the same location, with a latency of a few hundred cycles
- During this whole time, no one else can access the location

Serialization effect

Each Read-Modify-Write has two full memory access delays

⚠️ All atomic operations on the same variable (DRAM location) are serialized.

Latency determines throughput

Atomic operations on DRAM locations are limited by latency:

- Read-modify-write sequence: **> 1000 cycles** per operation
- All operations on same location are **serialized**
- Throughput reduced to **< 1/1000 of peak bandwidth** under high contention

Example 😊

1. Some customers realize that they missed an item after they started to check out
2. They run to the aisle and get the item while the line waits:
   - The rate of checkout is drastically reduced due to the long latency of running to the aisle and back
3. Imagine a store where every customer starts the check out before they even fetch any of the items:
   - The rate of the checkout will be 1 / (entire shopping time of each customer)

Improvements

Atomic operations on Shared Memory:

- Very short latency
- Private to each thread block
- Need algorithm work by programmers

⚠️ Avoid atomic operations as much as possible.

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 10×**

Shared Memory Atomics

- Each subset of threads are in the same block
- Much higher throughput than DRAM (100×) or L2 (10×) 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!

Kernel with privatization

```
__global__ void histo_kernel(unsigned char *buffer, long size,
        unsigned int *histo)
{
  /// Create private copies of the histo[] array for each thread block
  // 7 bins needed: 26 letters / 4 = 6.5, rounded up to 7 bins
  // (a-d, e-h, i-l, m-p, q-t, u-x, y-z)
  __shared__ unsigned int histo_private[7];
  // Initialize the bin counters in the private copies of histo[]
  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) {
    // Group letters into bins of 4: a-d (bin 0), e-h (bin 1), etc.
    atomicAdd( &(histo_private[buffer[i]/4]), 1);
    i += stride;
  }
  /// Build Final Histogram
  // wait for all other threads in the block to finish
  __syncthreads();
```

About 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
- If the histogram is too large to privatize: partially privatize an output histogram and use range testing to go to either global memory or shared memory

Comparison of histogram approaches

Approach	Memory	Contention	Performance	Use Case
Simple binning	Global	Very high	Poor	Small bins, low contention
Global atomics	Global	High	Moderate	Small histograms
Privatization	Shared + Global	Low	High (10×+)	Small-medium histograms

Key insight: Privatization dramatically reduces contention by localizing updates to shared memory first.

Performance summary

Key improvements:

- **Global atomics → Privatization**: **10× or more**
  - Shared memory: 100× faster than DRAM
  - Less contention: only threads in same block compete
- **Memory access optimizations**: 2-5× additional
  - Coalesced memory access
  - Proper warp utilization

Takeaway: Privatization + good memory patterns = best performance

Conclusions

Themes of this class:

- Memory access patterns
- Race conditions
- Atomic operations
- Use of private memory (privatization)