GP-GPU and High Performances Computing

Lecture 10 – Histogram

December 1, 2023

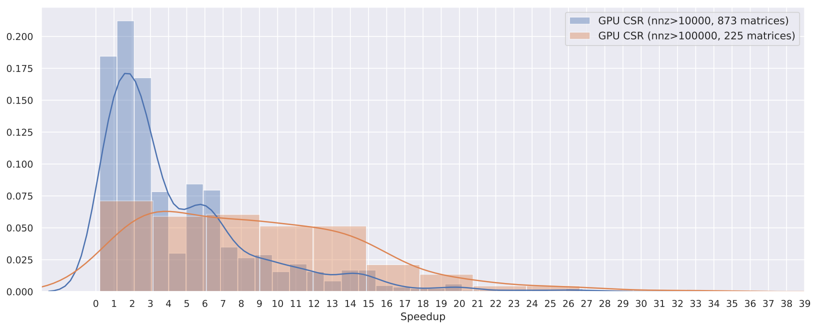
Previously

Organize storage of sparse matrices in order to

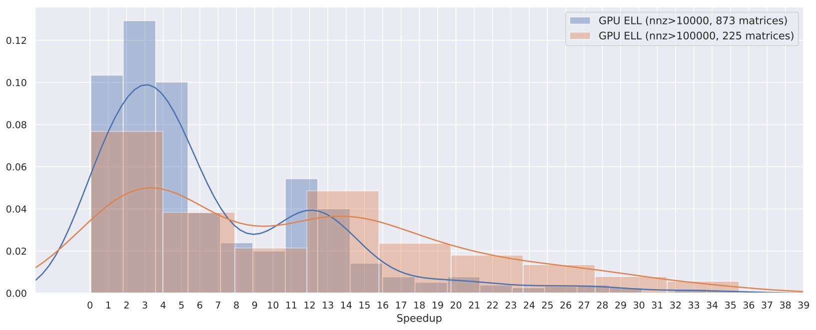
- minimize memory occupancy
- increase throughput
- limit data duplication
- limit tasks duplication

Performances comparison

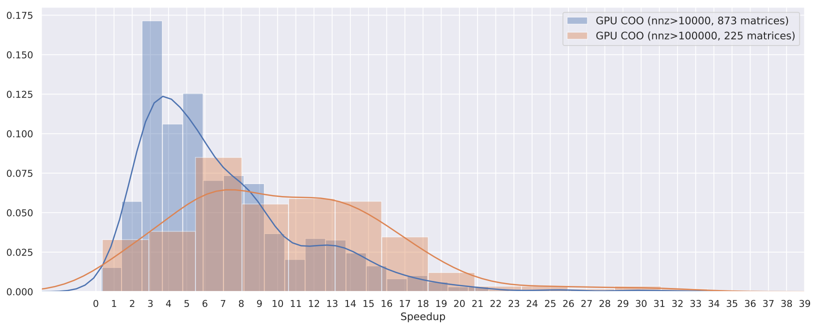
CSR


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

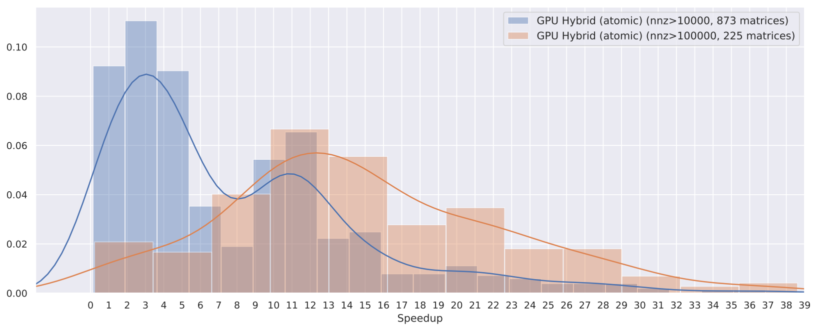
ELL


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 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[threadID] += 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 : $\times 2.5$

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 : $\times 2$.

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 : $\times 1.8$

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 : $\times 1.8$

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 : $\times 1.4$

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 = blockSize*2*gridDim.x;
  sdata[tid] = 0;
  while (i < n) {
    sdata[tid] += g_idata[i] + g_idata[i+blockSize];
    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];
}
```

Expected gain : ×1.4

Histogram computation

Objectives

The key techniques for compacting input data in parallel sparse methods for reduced consumption of memory bandwidth

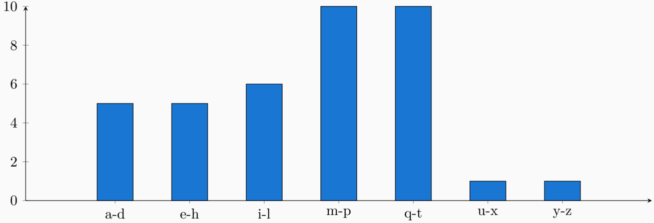
- better utilization of on-chip memory
- fewer bytes transferred to on-chip memory
- retaining regularity

Sparse data examples

- 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

Example with text data

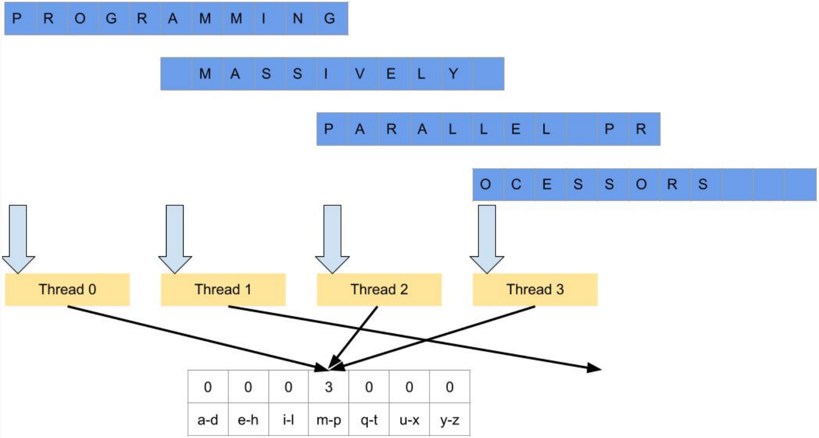
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p,
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is:



Algorithm 1: Simple binning

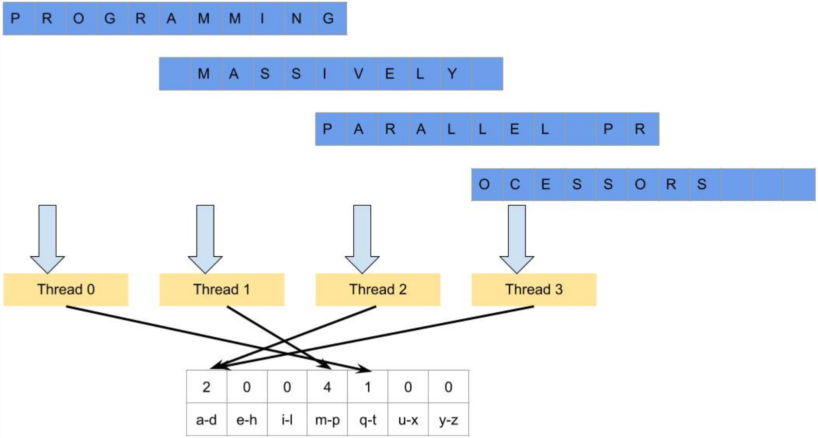
- Partition the input into sections
    - Have each thread to take a section of the input
    - Each thread iterates through its section.
    - For each letter, increment the appropriate bin counter

Algorithm 1: iteration 1



0	0	0	3	0	0	0
a-d	e-h	i-l	m-p	q-t	u-x	y-z

Algorithm 1: iteration 2



P R O G R A M M I N G

M A S S I V E L Y

P A R A L L E L P R

O C E S S O R S

Thread 0

Thread 1

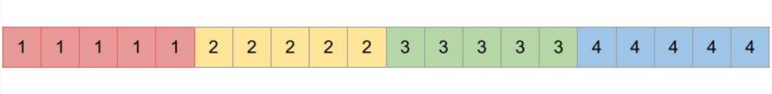
Thread 2

Thread 3

2	0	0	4	1	0	0
a-d	e-h	i-l	m-p	q-t	u-x	y-z

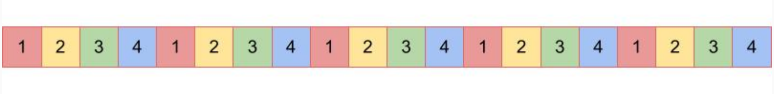
Input Partitioning Affects Memory Access Efficiency

- 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



1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4

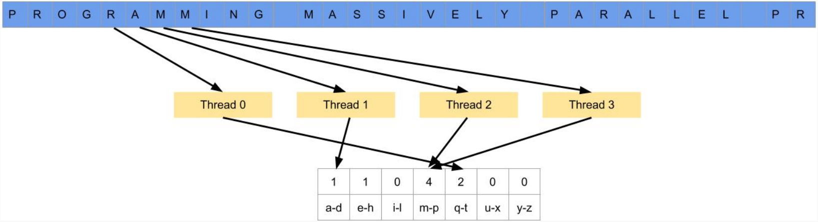
- Change to interleaved partitioning
    - All threads process a contiguous section of elements
    - They all move to the next section and repeat
    - The memory accesses are coalesced



1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

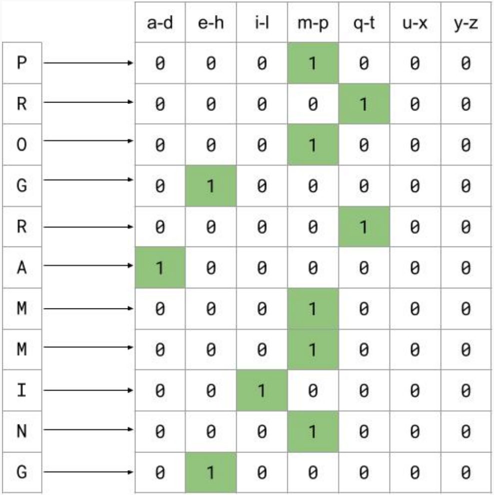
Algorithm 1b: iteration 2

Interleaved memory accesses



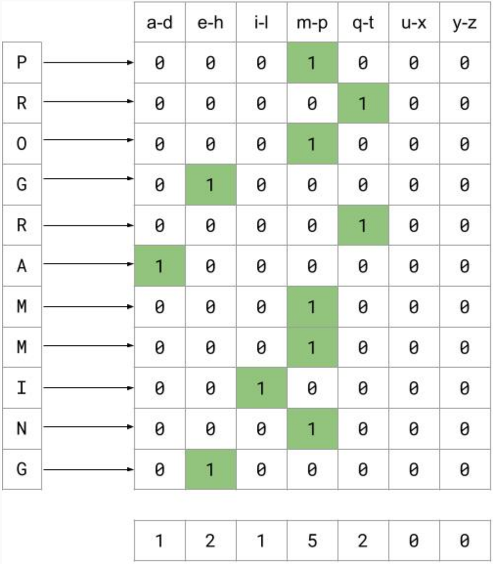
A reduction point of view

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

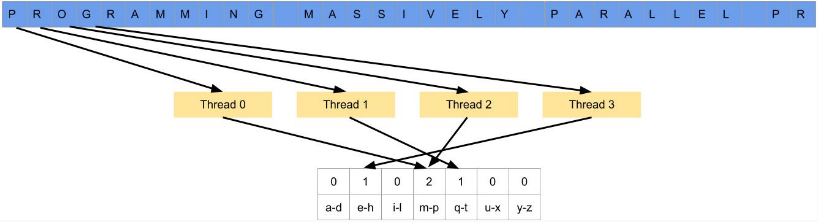
Reduction accross threads



	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

Data races

Interleaved memory accesses



Read-Modify-Write Used in Collaboration Patterns

For 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

For example, multiple customer service agents serving waiting customers

- The system maintains two numbers,
- the number to be given to the next incoming customer (I)
- the number for the customer to be served next (S)
- The system gives each incoming customer a number (read I) and increments the number to be given to the next customer by 1 modify (write I)
- A central display shows the number for the customer to be served next
- When an agent becomes available, he/she calls the number (read S) and increments the display number by 1 (modify-write S)
- Bad outcomes
  Multiple customers receive the same number, only one of them receives service
  Multiple agents serve the same number

A Common Arbitration Pattern

For example, multiple customers booking airline tickets in parallel Each

- 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 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

Execution scenario 1

Thread 0

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



```

Thread 1

```



Old = Mem[x] // 1
New = Old + 1 // 2
Mem[x] = New // 2
```

- Thread 1 : `Old = 0`
- Thread 2 : `Old = 1`
- After the sequence : `Mem[x] = 2`

Execution scenario 2

Thread 0

```



Old = Mem[x] // 1
New = Old + 1 // 2
Mem[x] = New // 2
```

Thread 1

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



```

- Thread 1 : `Old = 1`
- Thread 2 : `Old = 0`
- After the sequence : `Mem[x] = 2`

Execution scenario 3

Thread 0

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

Mem[x] = New // 1


```

Thread 1

```


Old = Mem[x] // 1

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

- Thread 1 : `Old = 0`
- Thread 2 : `Old = 0`
- After the sequence : `Mem[x] = 1`

Execution scenario 4

Thread 0

```


Old = Mem[x] // 1

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

Thread 1

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

Mem[x] = New // 1


```

- Thread 1 : `Old = 0`
- Thread 2 : `Old = 0`
- After the sequence : `Mem[x] = 1`

Atomic operation

The goal of atomic operation is to ensure that

Thread 0

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



```

Thread 1

```



Old = Mem[x] // 1
New = Old + 1 // 2
Mem[x] = New // 2
```

or

Thread 0

```



Old = Mem[x] // 1
New = Old + 1 // 2
Mem[x] = New // 2
```

Thread 1

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



```

Atomic operation

title

```
Mem[x]= 0
```

Thread 0

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

Mem[x] = New // 1


```

Thread 1

```


Old = Mem[x] // 1

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

- Both threads receive 0 in `Old`
- `Mem[x]` becomes 1

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, unsigne
```

- Unsigned 64-bit integer atomic add

```
unsigned long long int atomicAdd(unsigned long long i
```

- 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 process 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 && alpha_position < 26)
      atomicAdd(&(histo[alphabet_position/4]), 1);
    i += stride;
  }
}
```

Performances

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

title

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

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

Latency determines throughput

- Throughput of atomic operations on the same DRAM location is the rate at which the application can execute an atomic operation.
- The rate for atomic operation on a particular location is limited by the total latency of the read-modify-write sequence, typically more than 1000 cycles for global memory (DRAM) locations.
- This means that if many threads attempt to do atomic operation on the same location (contention), the memory throughput is reduced to < 1/1000 of the peak bandwidth of one memory channel!

Example :-)

1. Some customers realize that they missed an item after they started to check out
2. They run to the isle and get the item while the line waits:
   The rate of checkout is drastically reduced due to the long latency of running to the isle 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.

Privatization

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 (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!

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
  __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) {
    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] );
  }
}
```

About privatizaion

- 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.

Conclusion

Conclusions

- Themes of this class
  - Memory access
  - Race condition
  - Atomic operation
  - Use private memory