CS 6023 - GPU Programming

Histogram Computation

10/09/2018

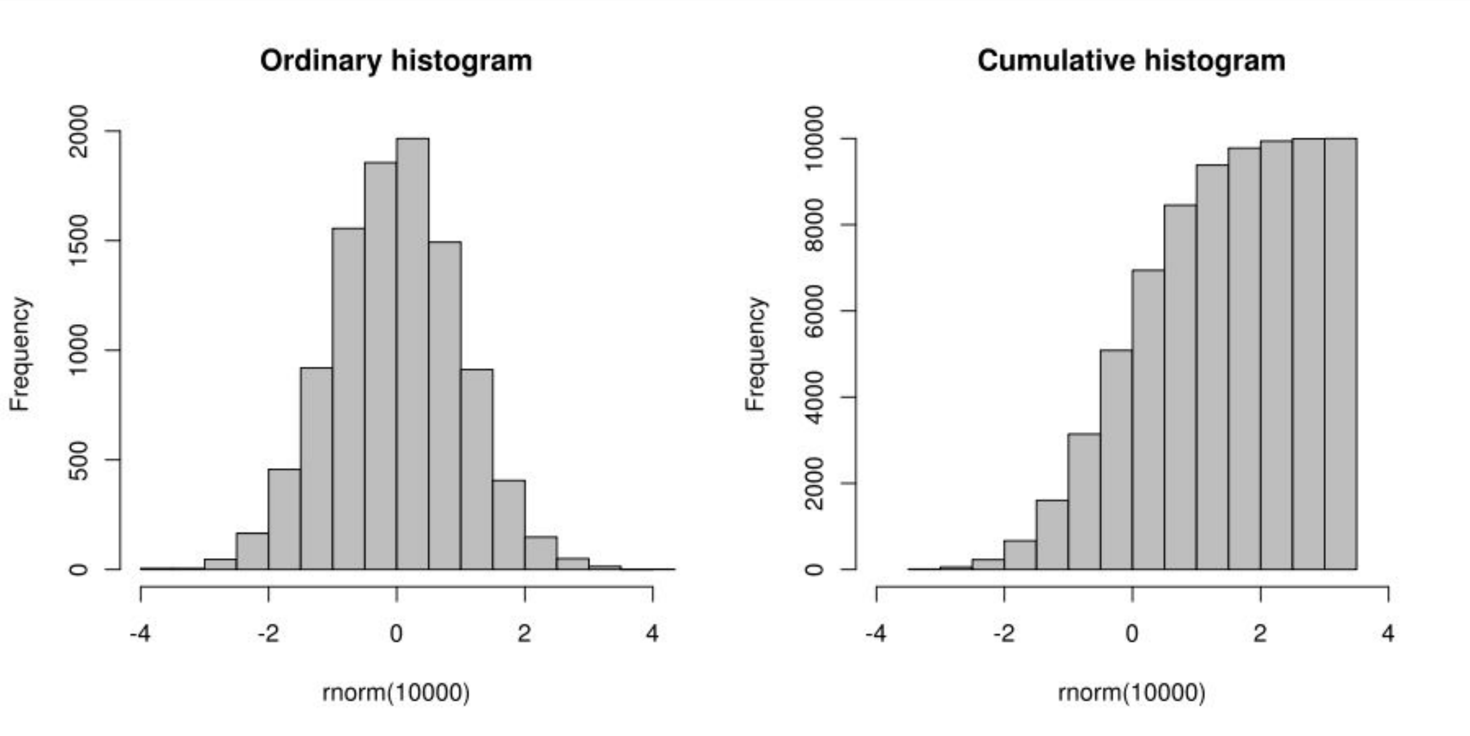
Setting and Agenda

- So far we have looked at vector addition and matrix multiplication

- Now we will move towards more realistic algorithms.
- To start with, we will look at **histogram computation**

Acknowledgement: Nvidia teaching kit

Histogram



Ordinary histogram
Frequency
0
500
1000
1500
2000
-4
-2
0
2
4
rnorm(10000)
Cumulative histogram
Frequency
0
2000
4000
6000
8000
10000
-4
-2
0
2
4
rnorm(10000)

Where do we use histograms

- Statistics
- Image processing
- As features for machine learning

Histogram - Statistics

- Long history of application
- Rules for choosing bin size
  - Sturgis rule

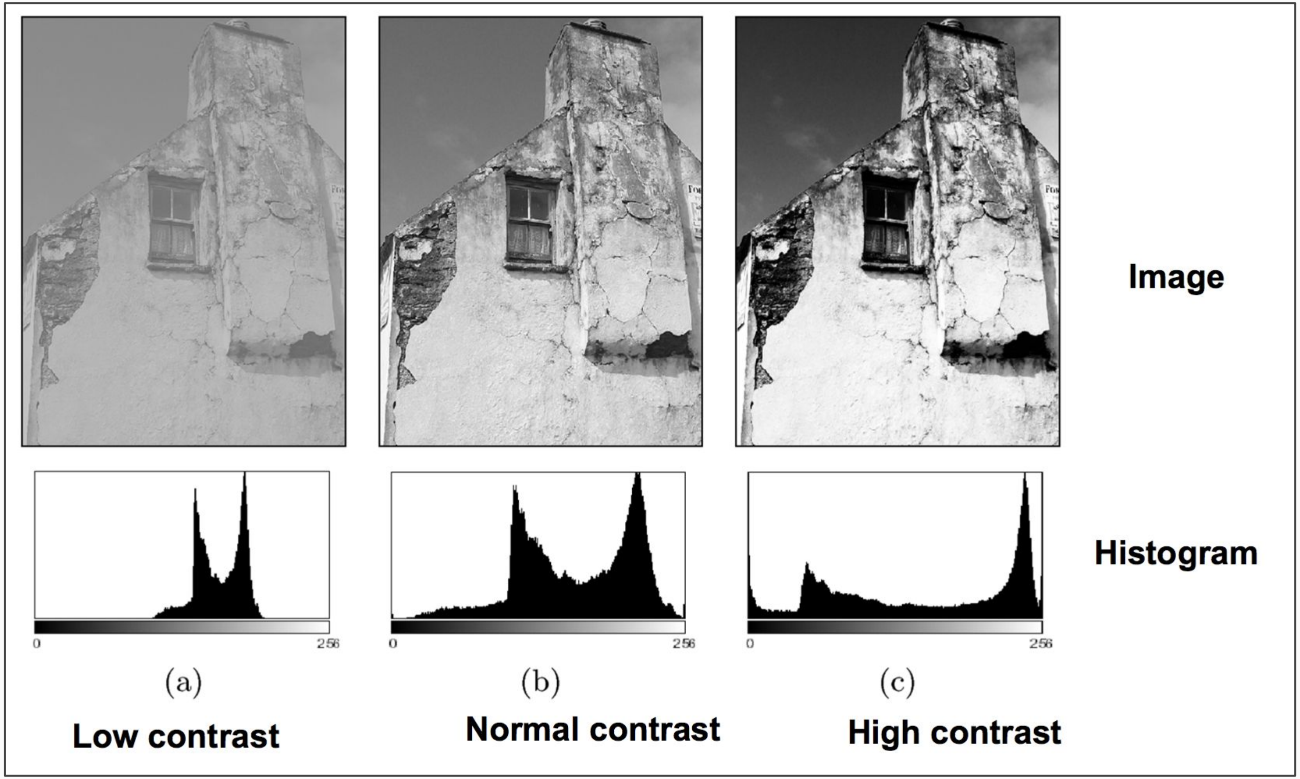
    $$J = 1 + 3.3 \log_{10} n$$

    where J is the number of bins and n is the total number of datapoints
  - Rice rule

    $$J = 2n^{1/3}$$

  - Example, if n = 20, both rules give about 6 bins

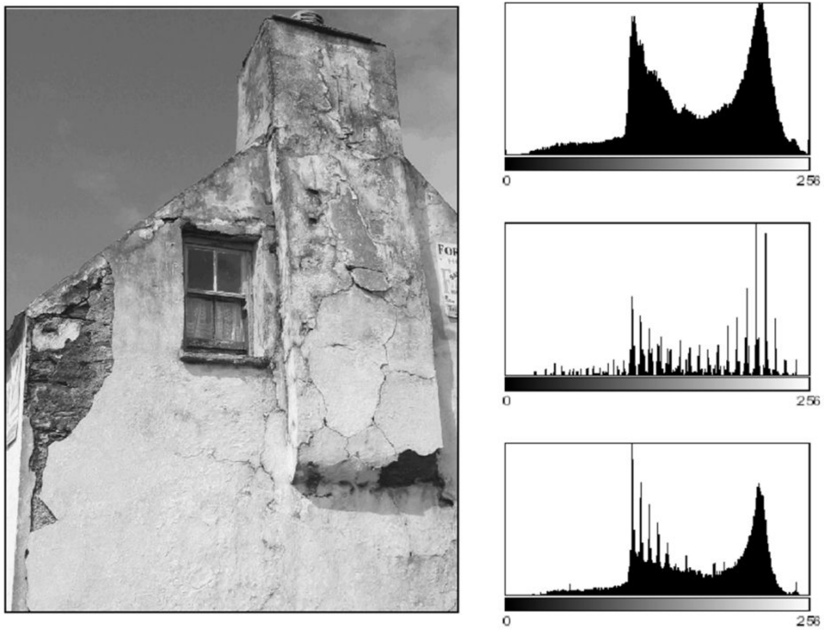
Image processing



https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture02.pdf

Image forensics

Original Image



(a) Original Histogram

(b) Histogram after GIF conversion

(c) Fix? Scaling image by 50% and Interpolating values recreates some lost colors

But GIF artifacts still visible

https://web.cs.wpi.edu/~emmanuel/courses/cs545/S14/slides/lecture02.pdf

Histograms in Machine Learning

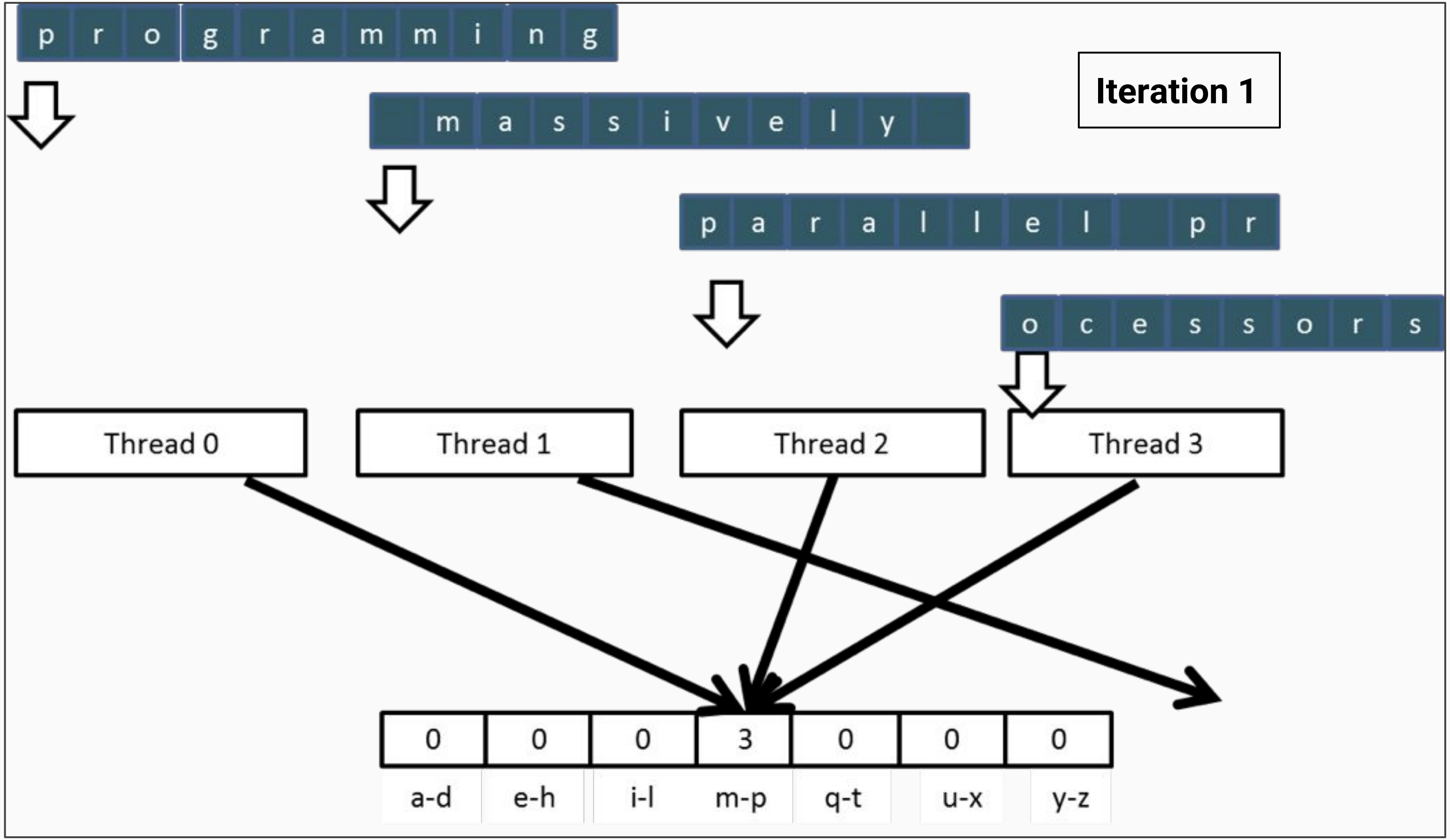
- Especially for sets with unequal sizes
  - Eg. classify groups of students based on their marks
  - Computed histograms are equal sized feature vectors
- Another common example in Natural Language Processing (NLP)
  - n-Gram histograms are common features
  - Can be used as features across sentences, paragraphs, and even documents

Parallel computation of histogram

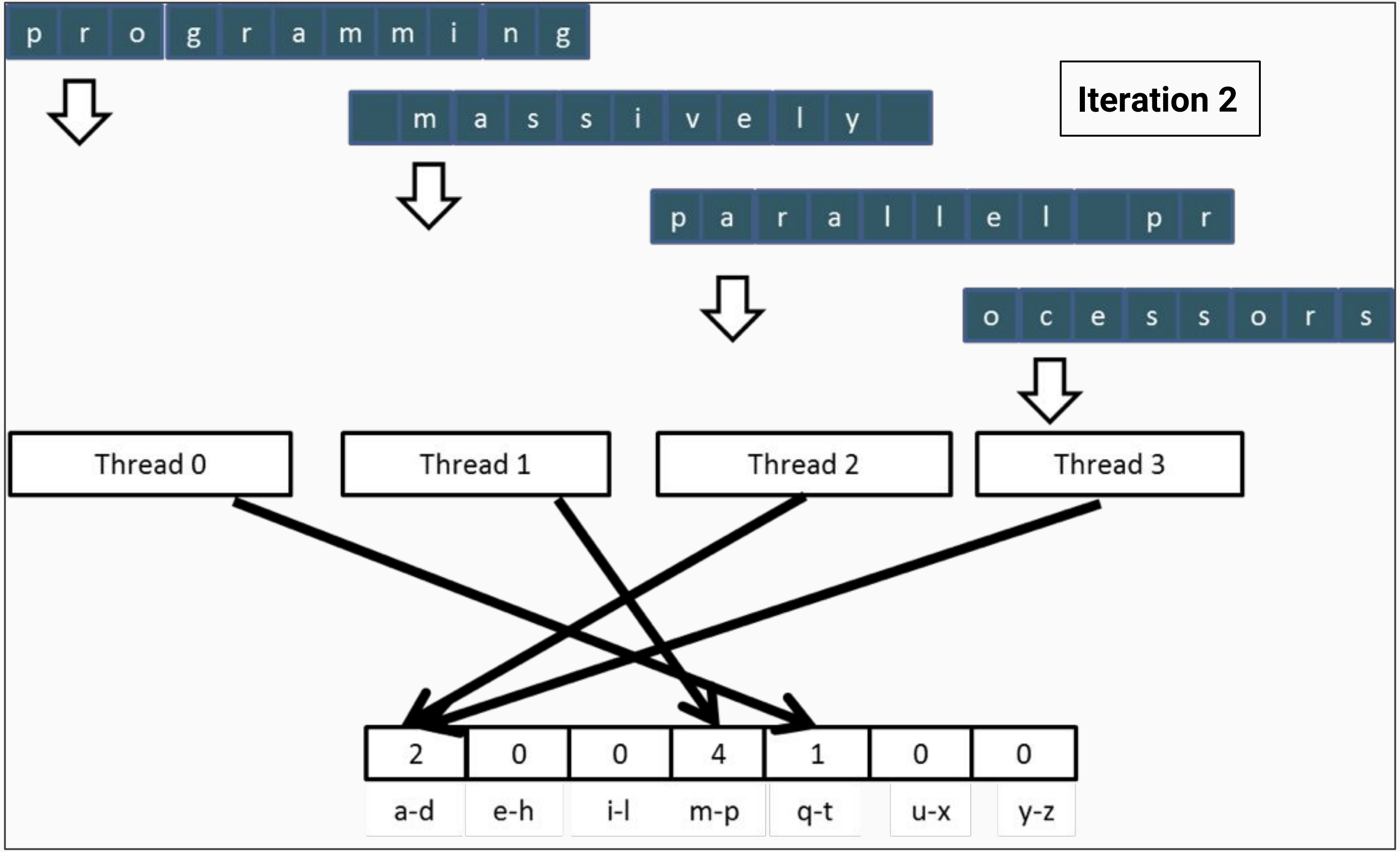
- First question of parallel programming:
  - How to divide the work amongst threads
- Answer: Divide input into equal chunks and assign each chunk to a threads

- Note that input streams (which are to be binned) are practically very large
- => One thread should process a large number of input values

Divided work amongst threads



Divided work amongst threads

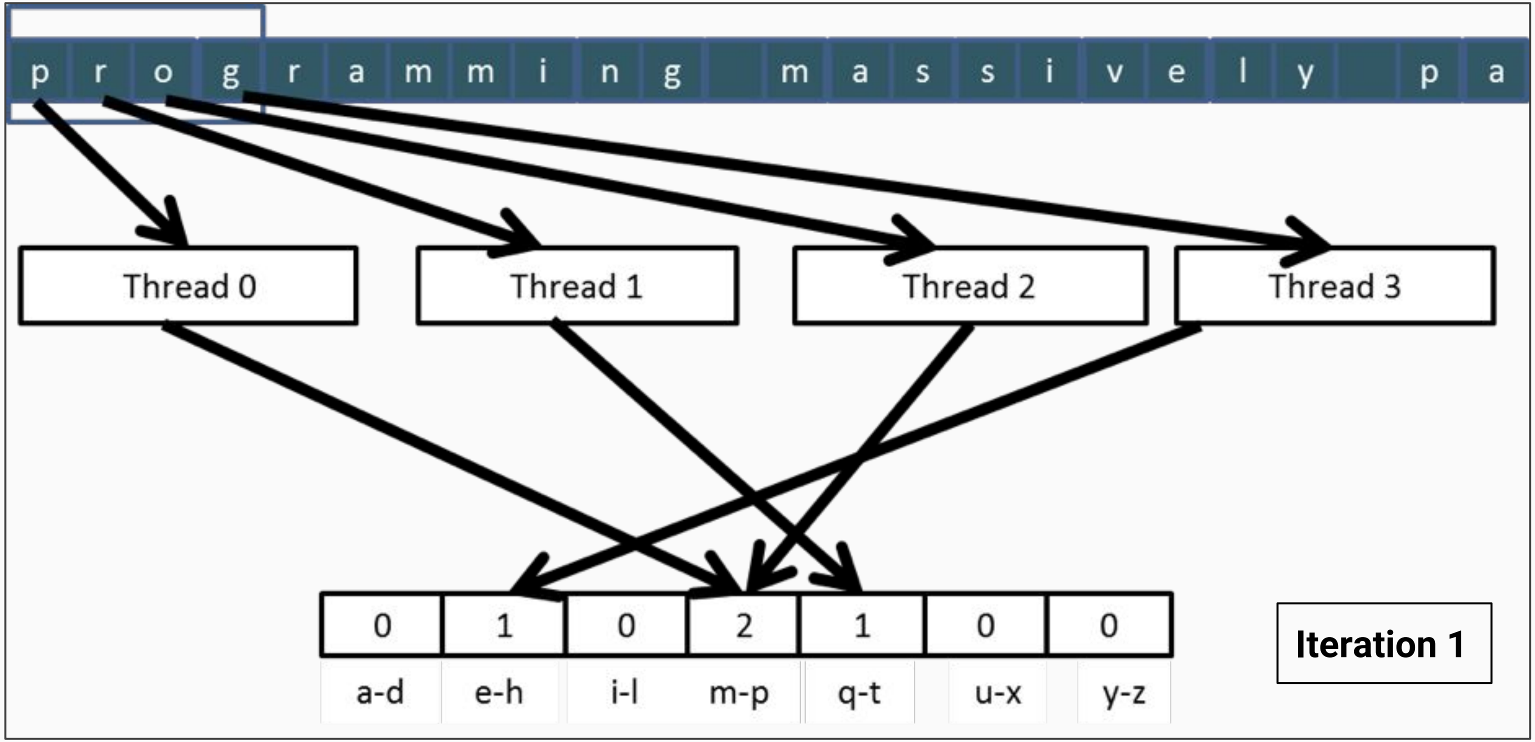


What is the memory access pattern

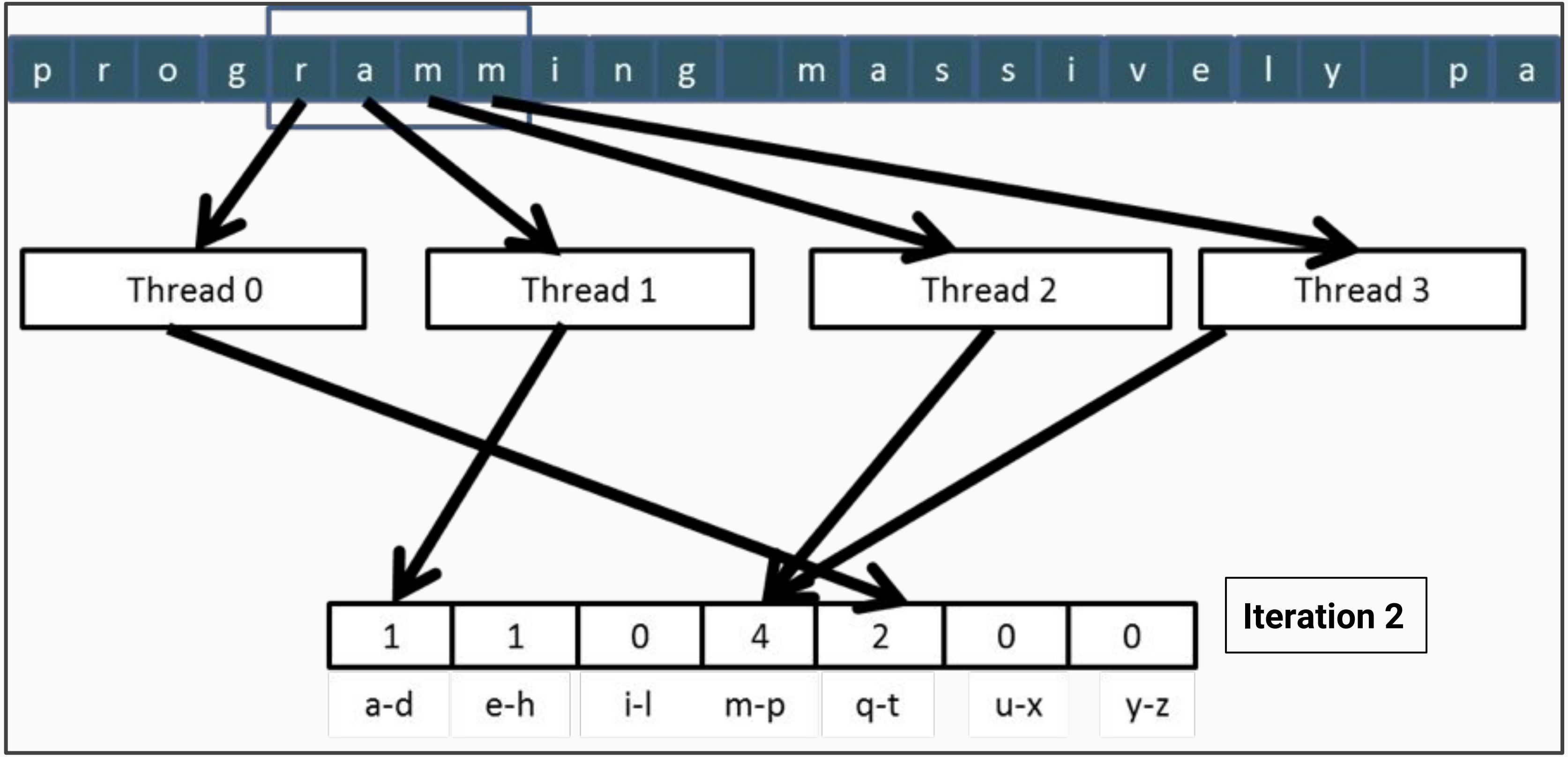
- Remember we are interested in memory access pattern from the point-of-view of DRAM
- With the above partitioning, accesses are not coalesced
- Threads in a warp will access memory locations in different burst sections

- More efficient to divide the input into contiguous sections that are given to the threads together

Modified work division



Modified work division



We have a second problem

We have a second problem

- How to ensure multiple threads increment the bin counters concurrently?

thread1: Old ← Mem[x]
New ← Old + 2
Mem[x] ← New

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

- In such cases, Mem[x] can have different values based on the order of execution of these operations
- What is a race condition
  "Computational hazard that occurs when the output of a program depends on the timing of uncontrollable events such as thread scheduling"
- Can we solve this with barrier synchronization?

CUDA Atomics

Race condition - experiment

```
__global__ void my_add(int *a) {
      *a += 1;
}

int main() {
      ...
      int a = 0, *a_d;
      cudaMalloc((void**) &a_d, sizeof(int));
      cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);
      ...
      my_add<<<1000,1000>>>(a_d);
      ...
      print("a = %d\n", a)
}
```

- We expect a sequential program to output 1000*1000 = 1,000,000
- On real runs, value can be as low as 100

Atomics

- Use CUDA intrinsics (also called intrinsic functions)
- Intrinsics are single instructions which are specially handled by compilers. In most cases, they are translated into a sequence of instructions

- CUDA supports intrinsic add

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

- This 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`.

Other supported functions

```
atomicAdd()
atomicSub()
atomicMin()
atomicMax()
atomicInc()
atomicDec()
atomicExch()
atomicCAS()
atomicAnd()
atomicOr()
atomicXor()
```

The CAS atomic operation is the most versatile

```
int atomicCAS(int *address, int compare, int val) {
    old = *address;
    *address = (old == compare) ? val : old;
    return old;
}
```

- Stands for compare and swap
- Can be used to implement other primitives like locks

Modified program from earlier

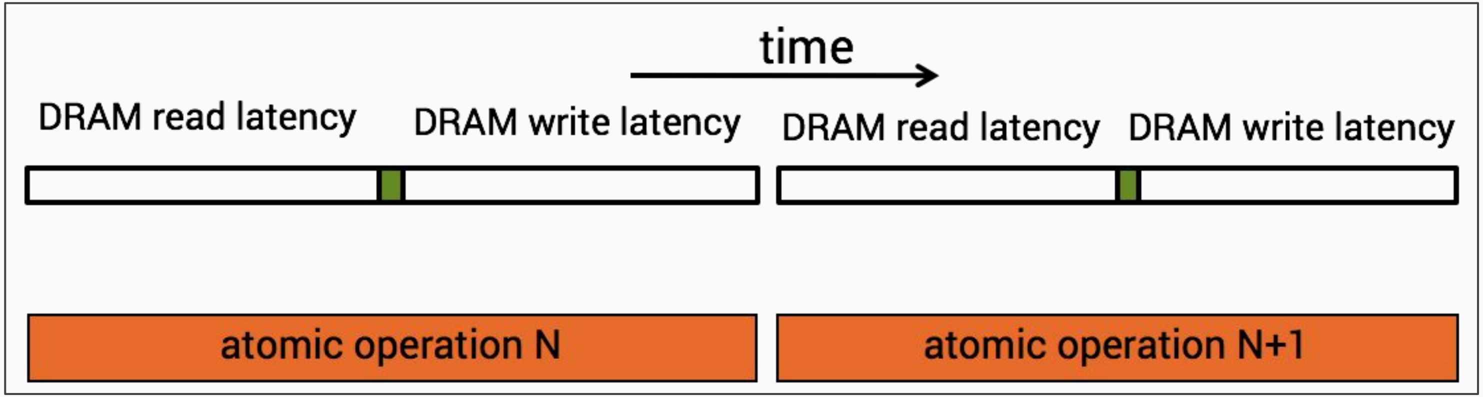
```
__global__ void my_add(int *a) {
      // *a += 1;
      atomicAdd (a, 1);
}

int main() {
      ...
      int a = 0, *a_d;
      cudaMalloc((void**) &a_d, sizeof(int));
      cudaMemcpy(a_d, &a, sizeof(int), cudaMemcpyHostToDevice);
      ...
      my_add<<<1000,1000>>>(a_d);
      ...
      print("a = %d\n", a)
}
```

- We now get the correct output 1000*1000 = 1,000,000
- But performance suffers. Can be 100x slower than earlier run (without atomics)

Atomics - Performance on DRAM

- Atomic operations suffer from very large DRAM delays
- First it starts with a DRAM read, which has latency of few hundred cycles
- Then any arithmetic / logical operations are performed
- Then it ends with a DRAM write, which has latency of few hundred cycles
- During this entire time, that DRAM location is closed to other accesses



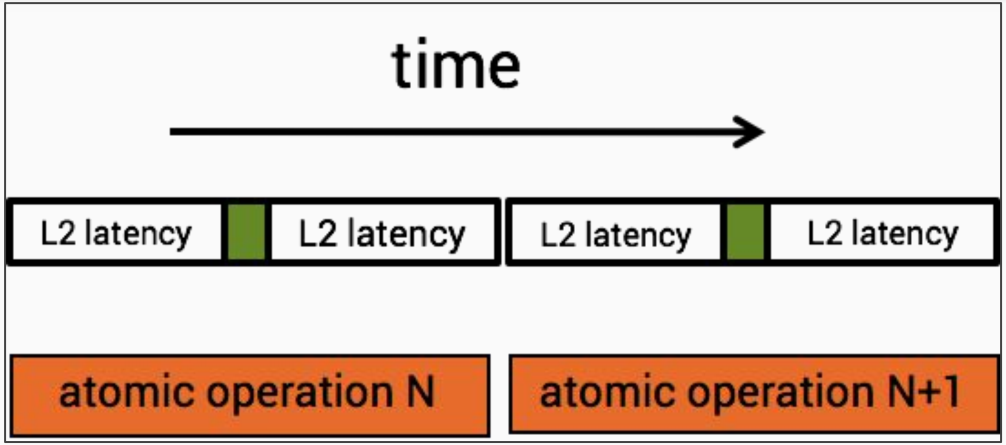
Atomics - Performance on DRAM

- Atomics can significantly increase program execution time
- If several threads would like atomic accesses to the same locations, then DRAM throughput can reduce 1000x!
- Analogy: Queue in supermarket where some customers run to the aisle

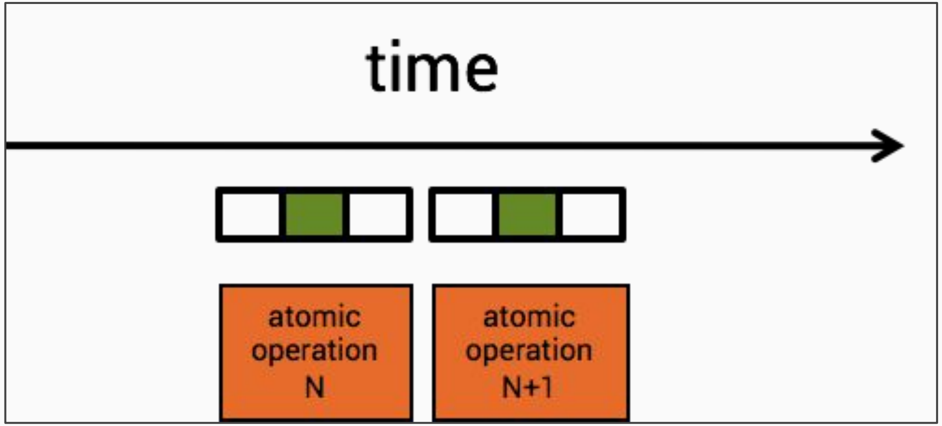
- One solution is to provide a faster memory for such operations
- Fermi whitepaper from Nvidia “Thanks to a combination of more atomic units in hardware and the addition of the L2 cache, atomic operations performance is up to 20× faster in Fermi compared to the GT200 generation”

Support in HW for atomics on other memories

On L2 cache



On shared memory



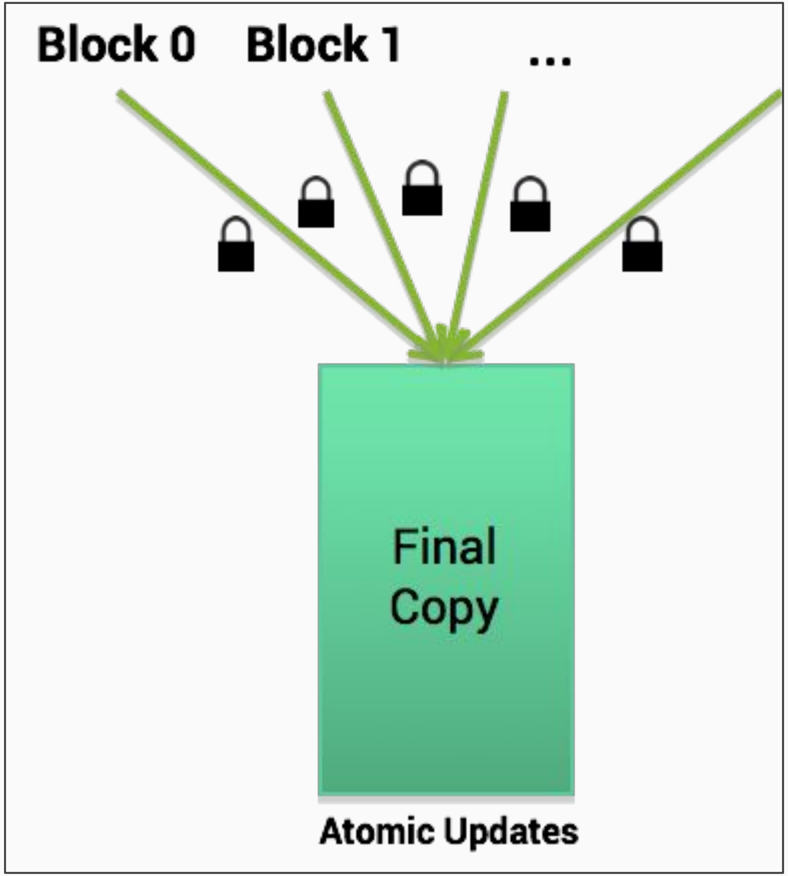
Histogram

- First we partitioned the input contiguously and shared work amongst threads
- Then we introduced atomic operations to avoid race conditions
- Then architectural improvements for faster atomics

- There is something more we can do for significant performance improvement

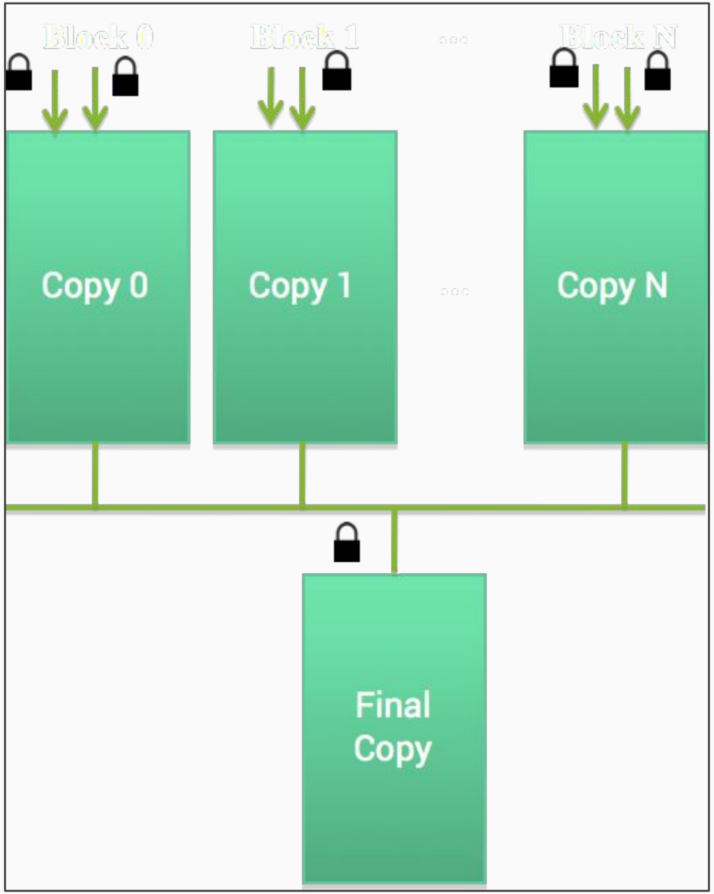
Privatization

- Currently, all blocks work on different parts of the input
- But write to the same output!
- Leads to serialization of atomic updates
- Can we do better?



Privatization

- Instead, use the concept of privatisation
- Each block updates private copies of the histogram
- Fewer threads synchronizing per copy leader to better performance
- After all threads are done, create a final copy from individual copies (in case of histogram, by adding)
- Can lead to significant performance improvement (~ 10x)



Under what conditions does privatization work?

Under what conditions does privatization work?

- Operation involved is commutative and associative
  - In case of histogram, add operation satisfies this
  - Merge sort
- The output being computed should be small so that private histograms can fit in memory
  - In case of histogram, output is typically much smaller than input
  - Ideally private copies should fit in shared memory (reduces atomics penalty 100x vs. global memory DRAM)
  - Privatization is another strong use case of shared memory naturally shared by threads in a block

Large histograms

- What if output histograms are huge
  - Eg. in NLP, consider histogram of all 3-grams

    For vocabulary of 5000 words, histogram needs to have 125×10^9 bins
- What then?

Large histograms

- What if output histograms are huge
    - Eg. in NLP, consider histogram of all 3-grams

      For vocabulary of 5000 words, histogram needs to have 125×10^9 bins
- What then?
- Can split histogram into two parts - private parts in shared memory and a common part in the global memory
- Prioritize and maintain commonly occuring 3-grams in shared memory
- Ideally should dynamically choose which 3-grams go where -> Assignment

Pseudo algo for histogram kernel with privatized shared memory output

- Declare global memory with output histogram
- For each thread:
    - Declare shared memory with private output histogram
    - Cooperatively initialize the histogram to 0
    - Synchronize
    - Identify the index/indices of the input on which to operate. For each:
        - Access each input item such that warps have coalesced access
        - Use atomic add to update appropriate bin in the output histogram
    - Cooperatively update the global output histogram with local one with atomic add