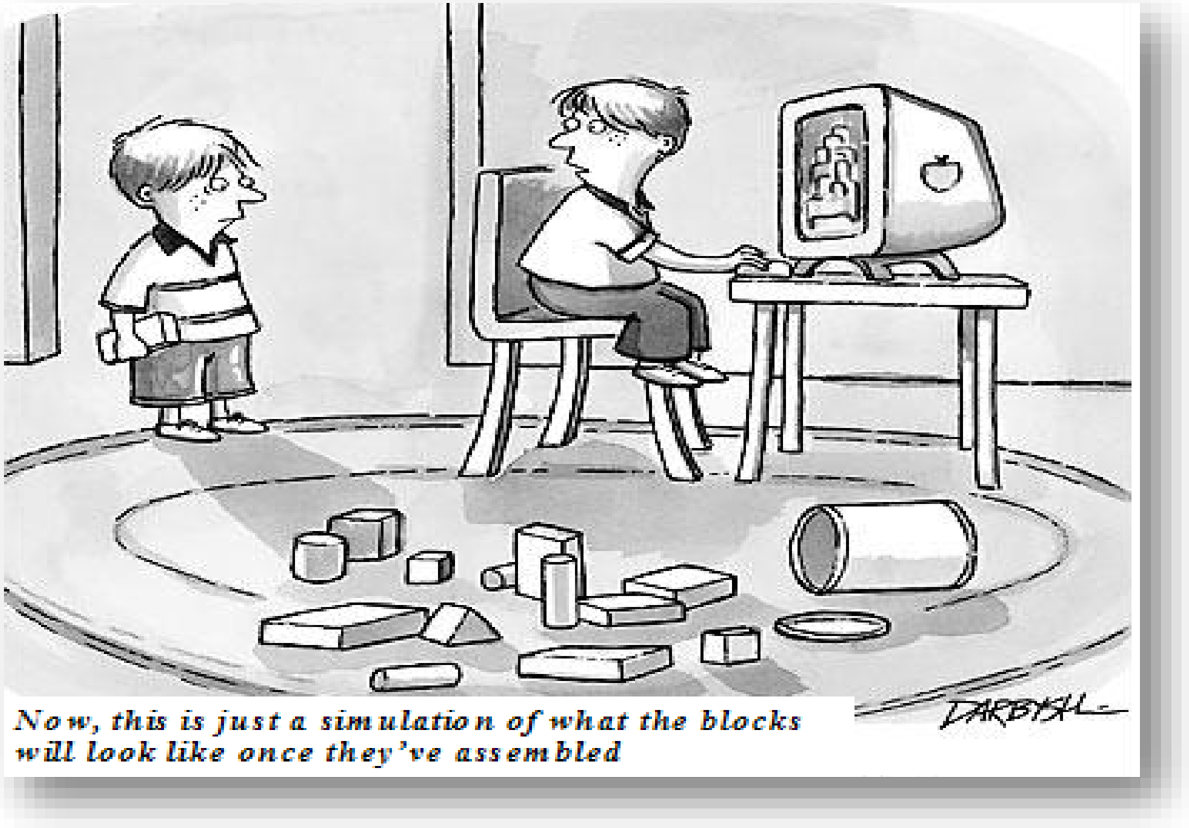
ECE569
Module 28



- Atomic Operations

A Common Arbitration Pattern

- Multiple customers booking air tickets, each
  - Brings up a flight seat map
  - Decides on a seat
  - Update the seat map, mark the seat as taken
- *A bad outcome*
  - Multiple passengers ended up booking the same seat

A Common Collaboration Pattern

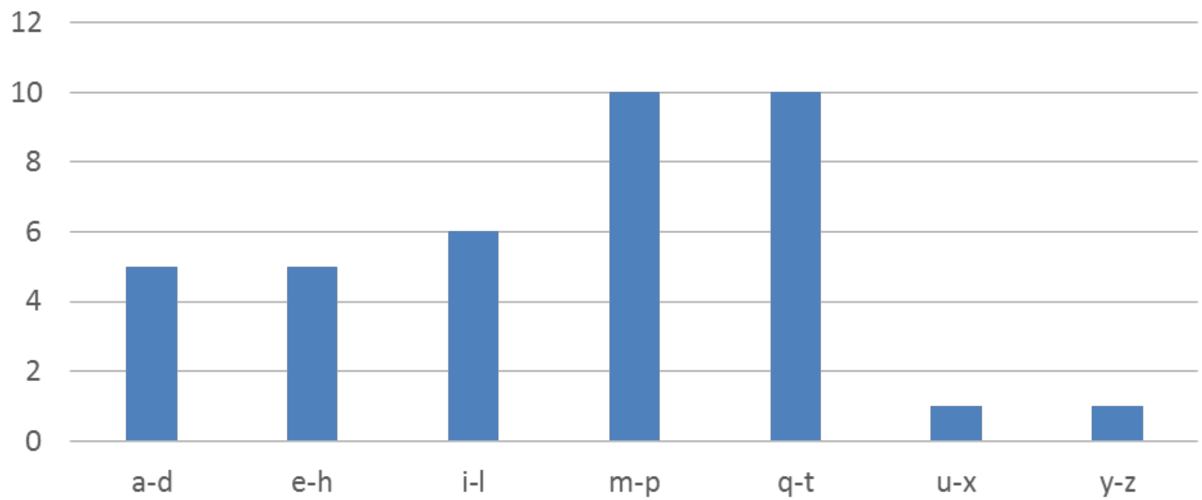
- **Multiple bank tellers count the total amount of cash in the safe**
  - Each grab a pile and count
  - Have a central display of the running total
  - Whenever someone finishes counting a pile, add the subtotal of the pile to the running total
- **A bad outcome**
  - Some of the piles were not accounted for.

Histogram Generation

- 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
    - Unusual patterns (just like object recognition): Type and location of purchases
- Basic histograms - for each element in the data set, use the value to identify a “bin” to increment
  - With a cumulative distribution function you can quickly analyze the data

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:



Serial Code

```
sequential_Histogram(char *data, int length, int *histo) {

    for (int i = 0; i < length; i++) {

        int alphabet_position = data[i] – 'a';

        if (alphabet_position >= 0 && alphabet_position < 26) {

            histo[alphabet_position/4]++
        }
    }
}
```

Data Race in Parallel Thread Execution

- 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.

thread1:		thread2:	
	Old $\leftarrow$ Mem[x]		Old $\leftarrow$ Mem[x]
	New $\leftarrow$ Old + 1		New $\leftarrow$ Old + 1
	Mem[x] $\leftarrow$ New		Mem[x] $\leftarrow$ New

Data Race in Parallel Thread Execution

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3	(1) Mem[x] ← New	
4		(1) Old ← Mem[x]
5		(2) New ← Old + 1
6		(2) Mem[x] ← New

Time	Thread 1	Thread 2
1	(0) Old ← Mem[x]	
2	(1) New ← Old + 1	
3		(0) Old ← Mem[x]
4	(1) Mem[x] ← New	
5		(1) New ← Old + 1
6		(1) Mem[x] ← New

Purpose of Atomic Operation – Correct Outcome

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

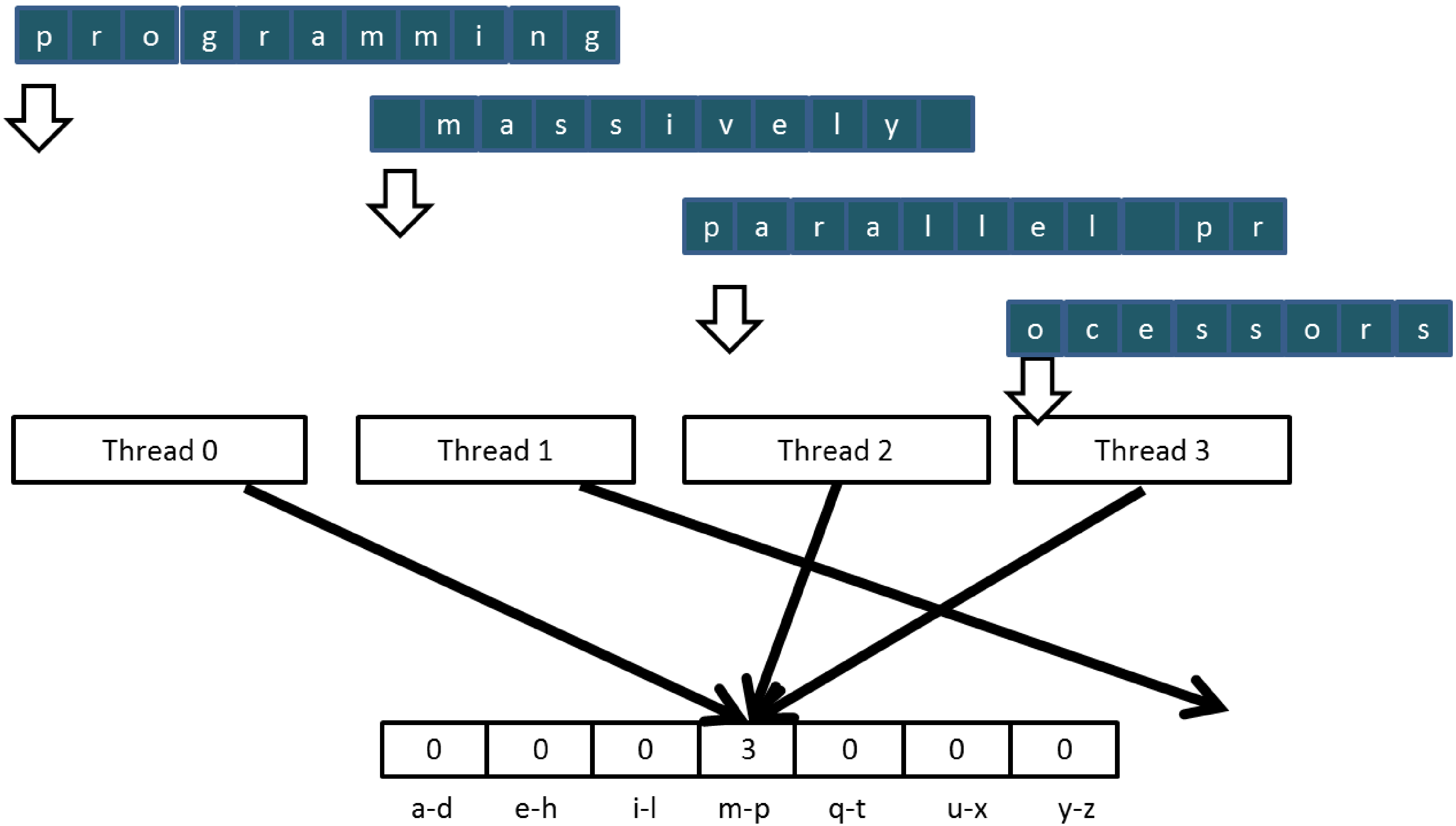
thread2: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

Or

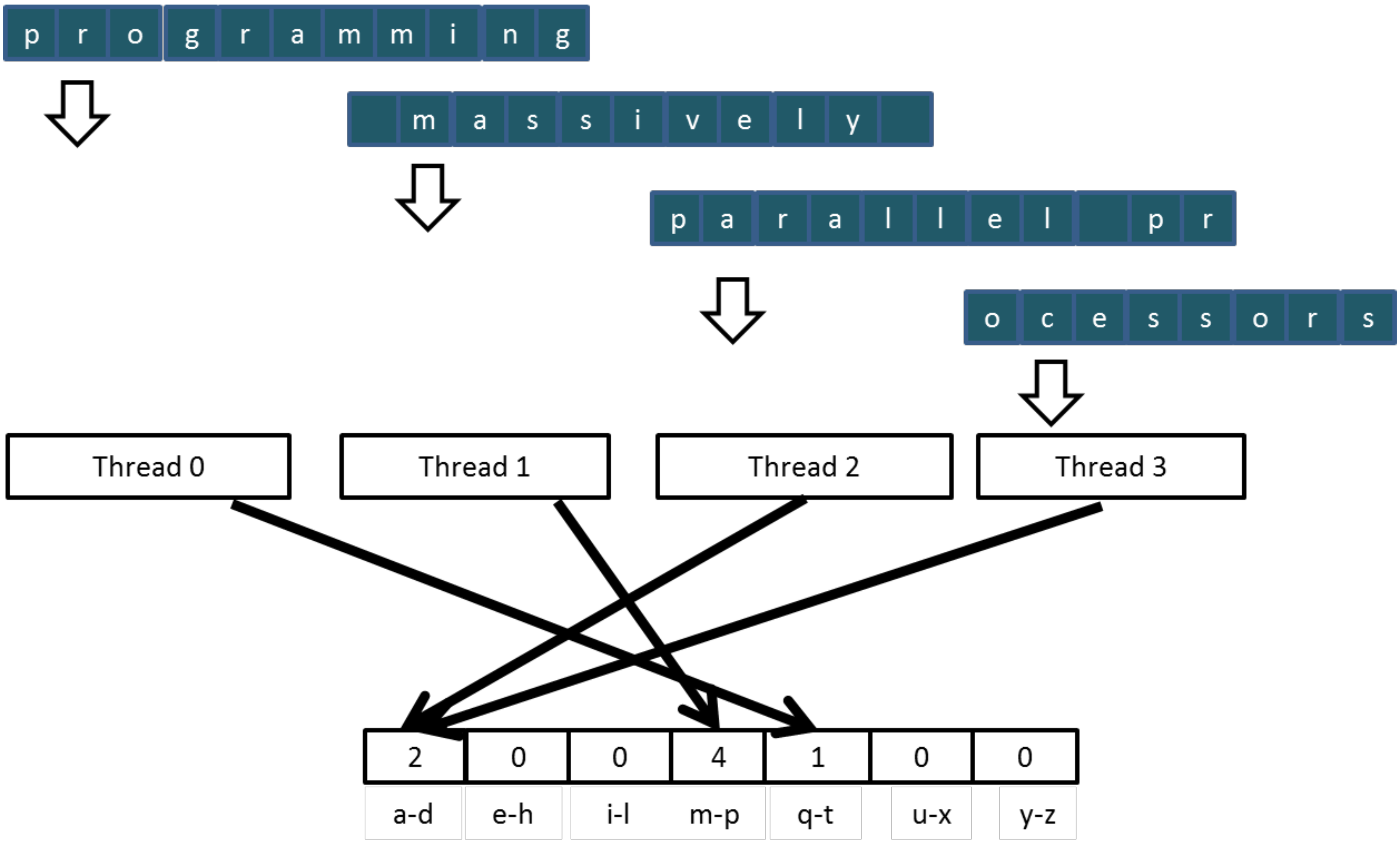
thread1: Old ← Mem[x]
New ← Old + 1
Mem[x] ← New

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

Histogram Generation Example – Iteration 1



Histogram Generation Example – Iteration 2



Atomic Operations

- Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide for details
- Atomic Add
  - int atomicAdd(int* address, int val);
    - reads 32-bit word from the location pointed to by address in global or shared memory,
    - computes (old + val),
    - stores the result back to memory at the same address.
    - returns old value
  - These operations are performed in one atomic transaction.

Atomic Add

- 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 (Compute capability 2.x+)
  - float atomicAdd(float* address, float val);
- Double-precision floating-point atomic add (Compute capability 6.x+)
  - double atomicAdd(double* address, double val);
- 16-bit floating-point atomic add (Compute capability 7.x+)
  - __half atomicAdd(__half* address, __half val);

Atomic Add Demo

- **Refer to D2L**
  - Content ➔ Demo ➔ 6.Atomic

```
#include <stdio.h>
#define NUM_THREADS 1000000
#define ARRAY_SIZE  10
#define BLOCK_WIDTH 1000
__global__ void increment_naive(int *g){
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   i = i % ARRAY_SIZE;
   g[i] = g[i] + 1; }

int main(int argc,char **argv){
    // declare and allocate host memory
    int h_array[ARRAY_SIZE];
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);
     // declare, allocate, and zero out GPU memory
    int * d_array;
    cudaMalloc((void **) &d_array, ARRAY_BYTES);
    cudaMemset((void *) d_array, 0, ARRAY_BYTES);

    increment_naive<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);

    // copy back the array of sums from GPU and print
    cudaMemcpy(h_array, d_array, ARRAY_BYTES, cudaMemcpyDeviceToHost);
    // What will be values in each element in h_array?
    // free GPU memory allocation and exit
    cudaFree(d_array);
    return 0;}
```

```
#include <stdio.h>
#define NUM_THREADS 1000000
#define ARRAY_SIZE  10
#define BLOCK_WIDTH 1000
__global__ void increment_atomic(int *g){
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   i = i % ARRAY_SIZE;
   atomicAdd(& g[i], 1);}

int main(int argc,char **argv){
    // declare and allocate host memory
    int h_array[ARRAY_SIZE];
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);
     // declare, allocate, and zero out GPU memory
    int * d_array;
    cudaMalloc((void **) &d_array, ARRAY_BYTES);
    cudaMemset((void *) d_array, 0, ARRAY_BYTES);

    increment_atomic<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);

    // copy back the array of sums from GPU and print
    cudaMemcpy(h_array, d_array, ARRAY_BYTES, cudaMemcpyDeviceToHost);
    // What will be values in each element in h_array?
    // free GPU memory allocation and exit
    cudaFree(d_array);
    return 0;}
```

Demo code with timer utility to experiment with

```
#include <stdio.h>
#include "gputimer.h"
#define NUM_THREADS 1000000
#define ARRAY_SIZE  10
#define BLOCK_WIDTH 1000

int main(int argc,char **argv)
{
   int runs;
  GpuTimer timer;
   printf("%d total threads in %d blocks writing into %d array elements\n",
        NUM_THREADS, NUM_THREADS / BLOCK_WIDTH, ARRAY_SIZE);
    // declare and allocate host memory
   int h_array[ARRAY_SIZE];
   const int ARRAY_BYTES = ARRAY_SIZE * sizeof(int);

   // declare, allocate, and zero out GPU memory
   int * d_array;
   cudaMalloc((void **) &d_array, ARRAY_BYTES);
   cudaMemset((void *) d_array, 0, ARRAY_BYTES);
    // launch the kernel - comment out one of these
   timer.Start();
   for (runs=0;runs<100;runs++){
     // increment_naive<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
       increment_atomic<<<NUM_THREADS/BLOCK_WIDTH, BLOCK_WIDTH>>>(d_array);
    }
   timer.Stop();
   // copy back the array of sums from GPU and print
   cudaMemcpy(h_array, d_array, ARRAY_BYTES, cudaMemcpyDeviceToHost);
   print_array(h_array, ARRAY_SIZE);
   printf("For %d runs total time elapsed = %g ms with average per run =%g ms\n", runs,timer.Elapsed(),timer.Elapsed()/runs);
   // free GPU memory allocation and exit
   cudaFree(d_array);
   return 0;}
```

Atomic Add Demo

- **Refer to D2L**
  - Content ➔ Demo ➔ 6.Atomic
  - Comment out the “print_array” in the main
  - Run the increment_atomic kernel with the following configurations
    - 1 million threads, incrementing 10 elements
    - 1 million threads, incrementing 100 elements
    - 1 million threads, incrementing 500 elements
    - 1 million threads, incrementing 1000 elements
    - 1 million threads, incrementing 5000 elements
  - Does the execution time trend make sense?

Reading

- CUDA Programming Guide

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

  – Section 5.4.2: control flow and predicates
  – Section 5.4.3: synchronization
  – Appendix B.5: __threadfence() and variants
  – Appendix B.6: __syncthreads() and variants
  – Appendix B.14: atomic functions
  – Appendix B.18: warp voting

Next

- **Histogram**