GPU Teaching Kit

Accelerated Computing

# Module 7.2 – Parallel Computation Patterns (Histogram)

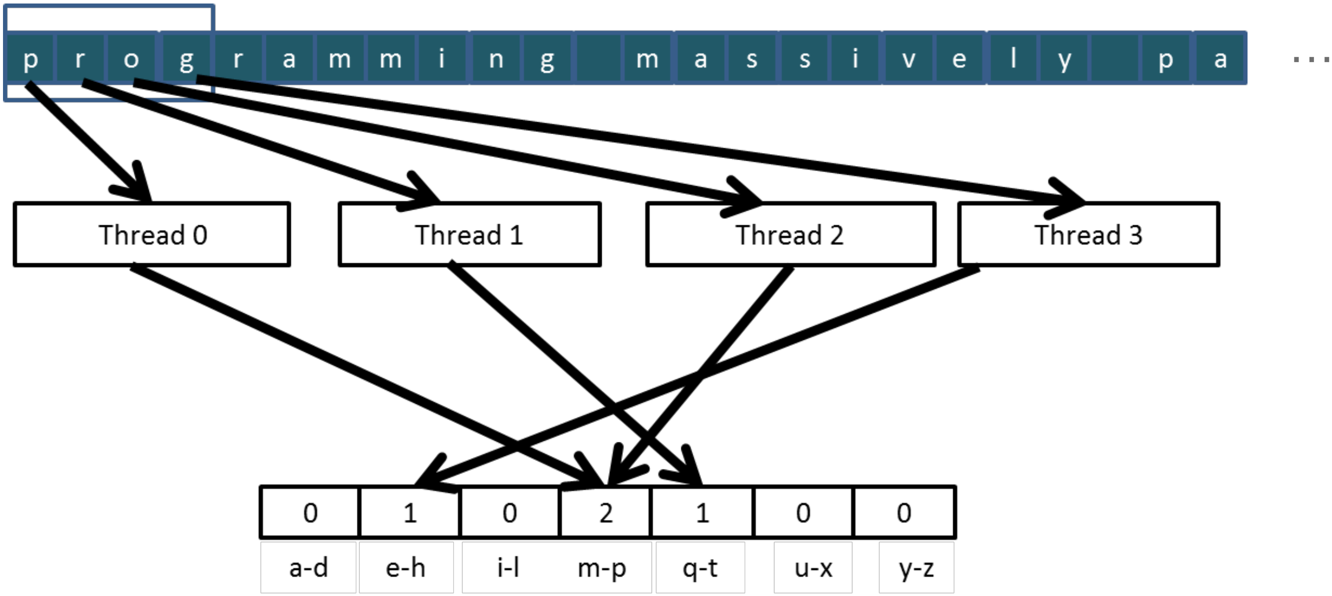
## Introduction to Data Races

# Objective

- To understand data races in parallel computing
  - Data races can occur when performing read-modify-write operations
  - Data races can cause errors that are hard to reproduce
  - Atomic operations are designed to eliminate such data races

# Read-modify-write in the Text Histogram Example

– For coalescing and better memory access performance



# 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

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

thread2: 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**.

# Timing Scenario #1

| 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 |

- Thread 1 Old = 0
- Thread 2 Old = 1
- Mem[x] = 2 after the sequence

# Timing Scenario #2

| 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 | |

– Thread 1 Old = 1
– Thread 2 Old = 0
– Mem[x] = 2 after the sequence

# Timing Scenario #3

| 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 |

– Thread 1 Old = 0
– Thread 2 Old = 0
– Mem[x] = 1 after the sequence

# Timing Scenario #4

| 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 | |

- Thread 1 Old = 0
- Thread 2 Old = 0
- Mem[x] = 1 after the sequence

# Purpose of Atomic Operations – To Ensure Good Outcomes

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

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

Or

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

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

GPU Teaching Kit

Accelerated Computing