GPU Programming

Rupesh Nasre.
http://www.cse.iitm.ac.in/~rupesh

IIT Madras

July 2017

Let's Sync-up!

- Print numbers 1 to 10 in sequence.
- Launch kernel with 3 threads.
- Each kernel prints threadIdx.x modulo 3.

```
__global__ void onetoten() {
   for (int ii = 0; ii < 10; ++ii)
     if (ii % 3 == threadIdx.x)
       printf("%d: %d\n", threadIdx.x, ii);
}
```

```
__global__ void onetoten() {
        __shared__ int n;
        n = 0;
        __syncthreads();
        while (n < 10) {
                if (n % 3 == threadIdx.x) {
                        printf("%d: %d\n", threadIdx.x, n);
                        ++n;
                }
                __syncthreads();
        }
}
```

Let's Sync-up!

- Print numbers 1 to 10 in sequence.
- Launch kernel with 3 threads.
- Each kernel prints threadIdx.x modulo 3.

```
__device__ volatile int n;

__global__ void onetoten() {
    n = 0;
    __syncthreads();
    while (n < 10) {
        if (n % 3 == threadIdx.x) {
            printf("%d: %d\n", threadIdx.x, n);
            ++n;
        }
    }
}
```

```
__global__ void onetoten() {
    volatile __shared__ int n;
    n = 0;
    __syncthreads();
    while (n < 10) {
        if (n % 3 == threadIdx.x) {
            printf("%d: %d\n", threadIdx.x, n);
            ++n;
        }
    }
}
```

Let's Sync-up!

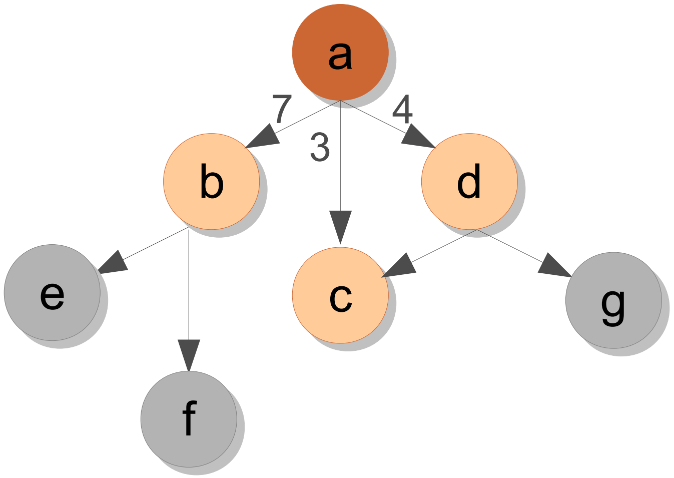
- Print numbers 1 to 10 in sequence.
- Launch kernel with 3 threads.
- Each kernel prints threadIdx.x modulo 3.

```
__global__ void onetoten() {
    __shared__ unsigned int n;
    n = 0;
    __syncthreads();

    while (n < 10) {
        int oldn = atomicInc(&n, 100);
        if (oldn % 3 == threadIdx.x) {
            printf("%d: %d\n", threadIdx.x, oldn);
        }
    }
}
```

Note that some of these codes are faulty.

Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {     // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            dist[n] = altdist;
}   }   }
```

What is the error in this code?

Data Race

- A datarace occurs if ***all*** of the following hold:
  1. Multiple threads
  2. Common memory location
  3. At least one write
  4. Concurrent execution
- Ways to remove datarace:
  1. Execute sequentially
  2. Privatization / Data replication
  3. Separating reads and writes by a barrier
  4. Mutual exclusion

Classwork

- Is there a datarace in this code?
- What does the code ensure?
- Can you ensure the same with barriers?
- Can you ensure the same with atomics?
- Generalize it for N threads.

T1

```
flag = 1;
while (flag)
      ;
S1;
```

T2

```
while (!flag)
      ;
S2;
flag = 0;
```

Synchronization

- Atomics
- Barriers
- Control + data flow
- ...

atomics

- Atomics are primitive operations whose effects are visible either none or fully (never partially).
- Need hardware support.
- Several variants: atomicCAS, atomicMin, atomicAdd, ...
- Work with both global and shared memory.

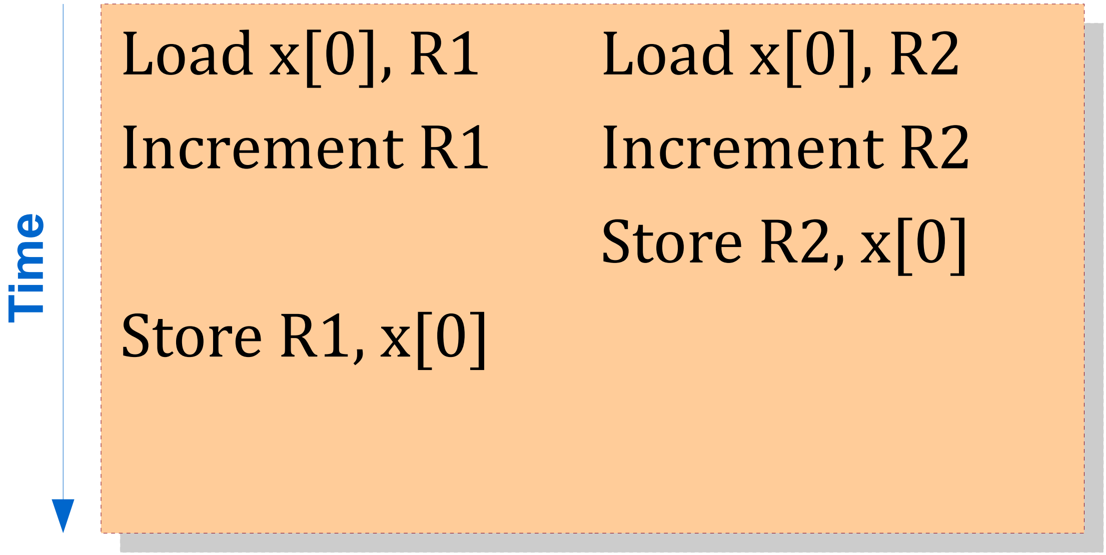
atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<1, 2>>>(x);
```

After dkernel completes, what is the value of x[0]?

++x[0] is equivalent to:

```
    Load x[0], R1
    Increment R1
    Store R1, x[0]
```



Load x[0], R1	Load x[0], R2
Increment R1	Increment R2
	Store R2, x[0]
Store R1, x[0]	

Final value stored in x[0] could be 1 (rather than 2).
What if x[0] is split into multiple instructions? What if there are more threads?

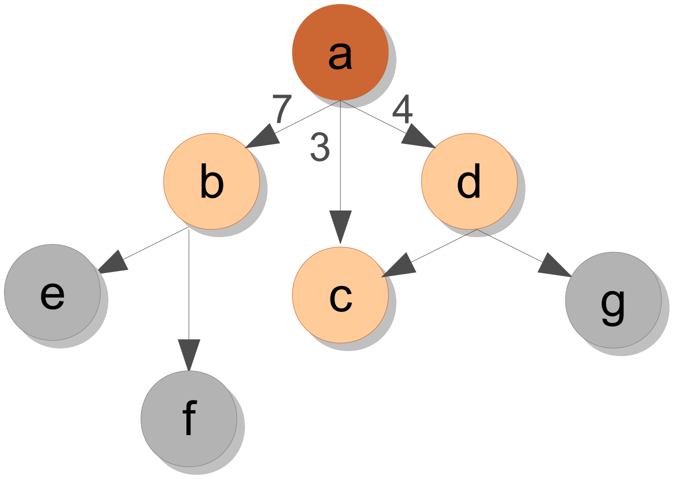
atomics

```
__global__ void dkernel(int *x) {
    ++x[0];
}
...
dkernel<<<1, 2>>>(x);
```

- Ensure all-or-none behavior.
  - e.g., atomicInc(&x[0], ...);
- **dkernel**<<<K1, K2>>> would ensure x[0] to be incremented by exactly K1*K2 – irrespective of the thread execution order.
  - When would this effect be visible?

Let's Compute the Shortest Paths

- You are given an input graph of India, and you want to compute the shortest path from Nagpur to every other city.
- Assume that you are given a GPU graph library and the associated routines.



```
__global__ void dsssp(Graph g, unsigned *dist) {
    unsigned id = ...
    for each n in g.allneighbors(id) {     // pseudo-code.
        unsigned altdist = dist[id] + weight(id, n);
        if (altdist < dist[n]) {
            ~~dist[n] = altdist;~~   atomicMin(&dist[n], altdist);
}   }   }
```

Classwork

1. Compute sum of all elements of an array.
2. Find the maximum element in an array.
3. Each thread adds elements to a worklist.
   - e.g., next set of nodes to be processed in SSSP.

AtomicCAS

- Syntax: oldval = atomicCAS(&var, x, y);

- Typical usecases:
  - *Locks* (critical section processing)
  - *Single*
  - Other atomic variants

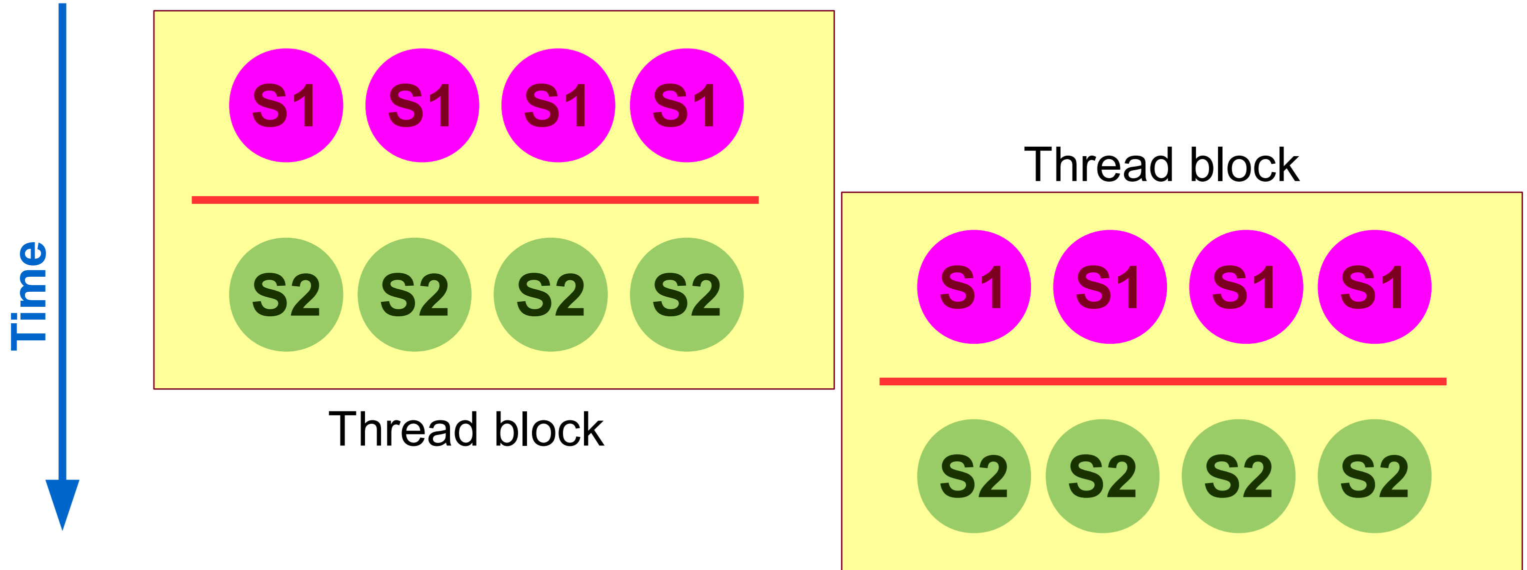
Classwork: Implement *single* with *atomicCAS*.

Barriers

- A barrier is a program point where all threads need to reach before any thread can proceed.
- End of kernel is an implicit barrier for all GPU threads (global barrier).
- There is no explicit global barrier supported in CUDA.
- Threads in a thread-block can synchronize using __syncthreads().
- How about barrier within warp-threads?

Barriers

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
    unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
    vector[id] = id;  S1
    __syncthreads();
    if (id < vectorsize - 1 && vector[id + 1] != id + 1)  S2
        printf("syncthreads does not work.\n");
}
```



Barriers

- __syncthreads() is not only about control synchronization, it also has data synchronization mechanism.
- It performs a memory fence operation.
  - A memory fence ensures that the writes from a thread are made visible to other threads.
  - __syncthreads() executes a fence for all the block-threads.
- There is a separate __threadfence_block() instruction also. Then, there is __threadfence().
- *[In general]* A fence does not ensure that other thread will read the updated value.
  - This can happen due to caching.
  - The other thread needs to use volatile data.
- *[In CUDA]* a fence applies to both read and write.

Classwork

- Write a CUDA kernel to find maximum over a set of elements, and then let thread 0 print the value in the same kernel.
- Each thread is given work[id] amount of work. Find average work per thread and if a thread's work is above average + K, push extra work to a worklist.
  - This is useful for load-balancing.
  - Also called work-donation.

Synchronization

- Atomics
- Barriers
- **Control + data flow**
- …

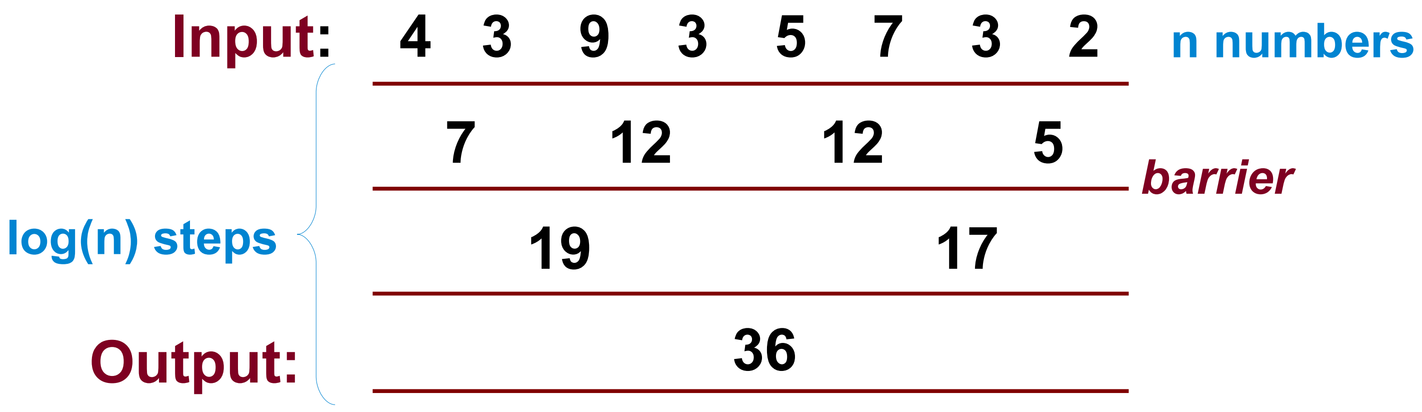
Initially, flag == false.

```

while (!flag) ;
S1;
```

```
S2;
flag = true;
```

Reductions

- What are reductions?
- Computation properties required.
- Complexity measures



Input: 4 3 9 3 5 7 3 2

n numbers

7 12 12 5

barrier

log(n) steps

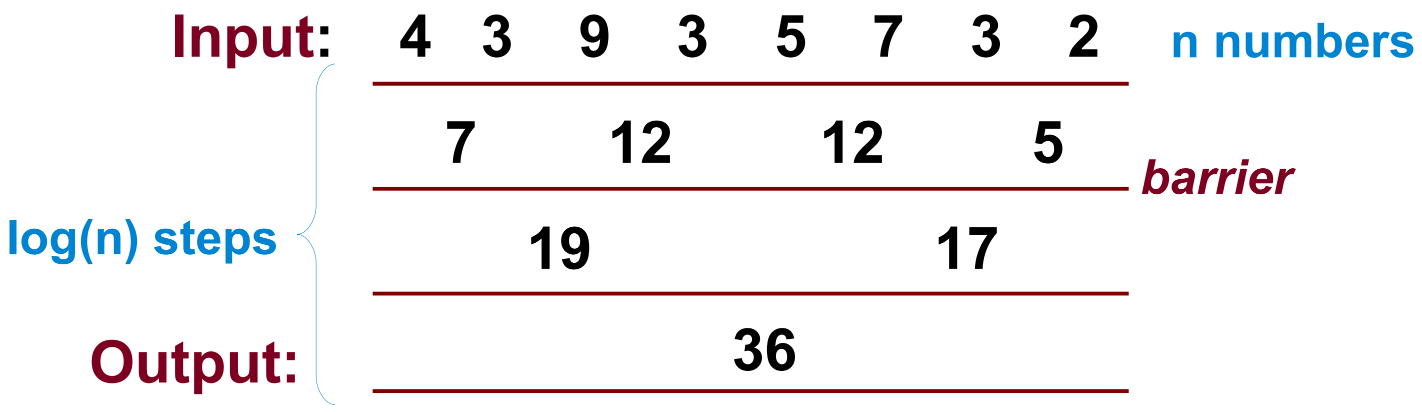
19 17

Output: 36

Classwork: Write the reduction code.

Reductions

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```



Input: 4 3 9 3 5 7 3 2 n numbers

7 12 12 5 *barrier*

log(n) steps

19 17

Output: 36

Reductions

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

- Write the reduction such that thread i sums a[i] and a[i + n/2].
- Assuming each a[i] is a character, find a concatenated string using reduction.
- String concatenation cannot be done using a[i] and a[i + n/2], but computing sum was possible; why?
- What other operations can be cast as reductions?

Prefix Sum

- Imagine threads wanting to push work-items to a central worklist.
- Each thread pushes different number of work-items.
- This can be computed using atomics or prefix sum (also called as *scan*).

```
Input:  4  3   9   3   5   7   3   2
Output: 4  7  16  19  24  31  34  36
OR
Output: 0  4   7  16  19  24  31  34
```

Classwork: Write the prefix-sum code.

Prefix Sum

```
for (int off = n/2; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

```
for (int off = n; off; off /= 2) {
        if (threadIdx.x < off) {
                a[threadIdx.x] += a[threadIdx.x + off];
        }
        __syncthreads();
}
```

```
Input:  4  3   9   3   5   7   3   2
Output: 4  7  16  19  24  31  33  35
OR
Output: 0  4   7  16  19  24  31  33
```

Prefix Sum

```
for (int off = n/2; off; off /= 2) {
    if (threadIdx.x < off) {
        a[threadIdx.x] += a[threadIdx.x + (n - off)];
    }
    __syncthreads();
}
```

```
for (int off = 0; off < n; off *= 2) {
    if (threadIdx.x > off) {
        a[threadIdx.x] += a[threadIdx.x - off];
    }
    __syncthreads();
}
```

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

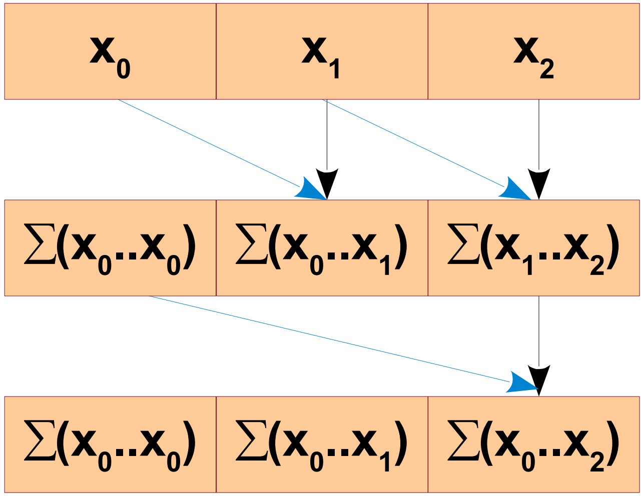
Prefix Sum

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$	$\Sigma(x_0..x_7)$

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

Prefix Sum



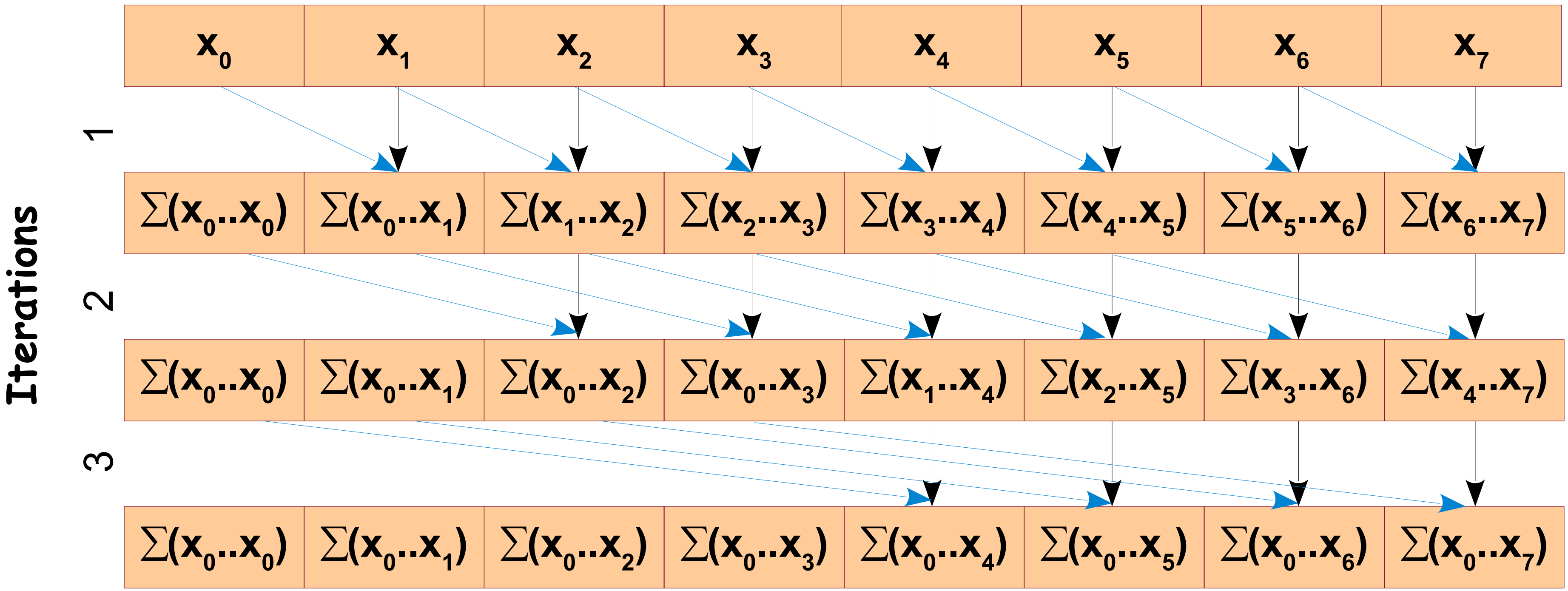
x_0 x_1 x_2

$\sum(x_0..x_0)$ $\sum(x_0..x_1)$ $\sum(x_1..x_2)$

$\sum(x_0..x_0)$ $\sum(x_0..x_1)$ $\sum(x_0..x_2)$

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

Prefix Sum



x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_1..x_2)$	$\Sigma(x_2..x_3)$	$\Sigma(x_3..x_4)$	$\Sigma(x_4..x_5)$	$\Sigma(x_5..x_6)$	$\Sigma(x_6..x_7)$
$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_1..x_4)$	$\Sigma(x_2..x_5)$	$\Sigma(x_3..x_6)$	$\Sigma(x_4..x_7)$
$\Sigma(x_0..x_0)$	$\Sigma(x_0..x_1)$	$\Sigma(x_0..x_2)$	$\Sigma(x_0..x_3)$	$\Sigma(x_0..x_4)$	$\Sigma(x_0..x_5)$	$\Sigma(x_0..x_6)$	$\Sigma(x_0..x_7)$

Input: 4 3 9 3 5 7 3 2
Output: 4 7 16 19 24 31 33 35
OR
Output: 0 4 7 16 19 24 31 33

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
        if (threadIdx.x > off) {
                a[threadIdx.x] += a[threadIdx.x - off];
        }
        __syncthreads();
}
```

Datarace

```
for (int off = 0; off < n; off *= 2) {
        if (threadIdx.x > off) {
                tmp = a[threadIdx.x – off];
                __syncthreads();
                a[threadIdx.x] += tmp;
        }
        __syncthreads();
}
```

Separating R and W in time

Prefix Sum

```
for (int off = 1; off < n; off *= 2) {
        if (threadIdx.x >= off) {
                tmp = a[threadIdx.x - off];
        }
        __syncthreads();

        if (threadIdx.x >= off) {
                a[threadIdx.x] += tmp;
        }
        __syncthreads();
}
```

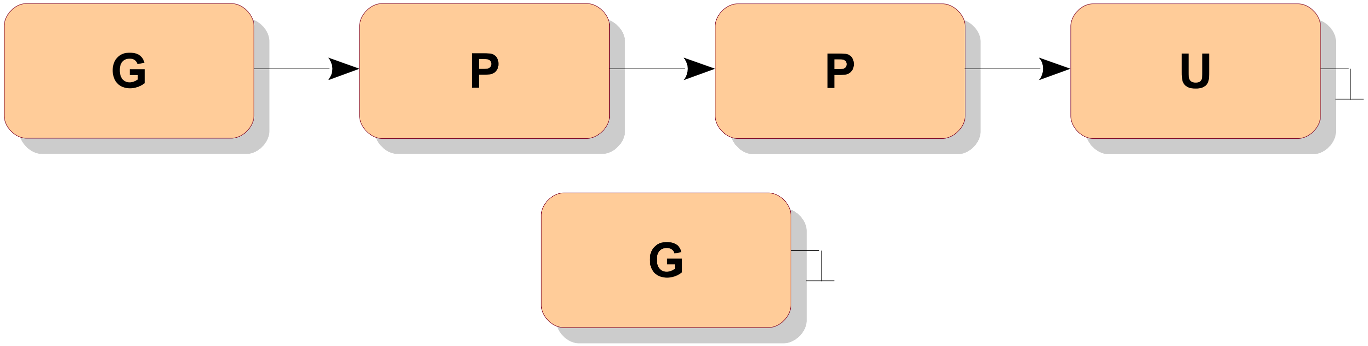
Homework: Can one of the barriers be avoided?

Application of Prefix Sum

- Assuming that you have the prefix sum kernel, insert elements into the worklist.
  - Each thread inserts $\mathrm{nelem[tid]}$ many elements.
  - The order of elements is not important.
  - You are forbidden to use atomics.
- Computing cumulative sum
  - Histogramming
  - Area under the curve

Concurrent Data Structures

- Array
  - atomics for index update
  - prefix sum for coarse insertion
- Singly linked list
  - insertion
  - deletion [marking, actual removal]



Concurrent Data Structures

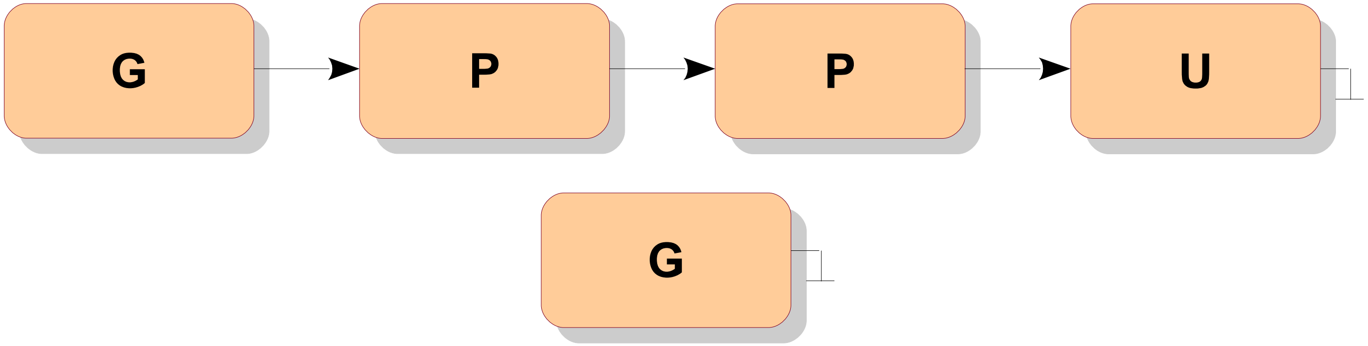
Type definition

```
struct node {
      char item;
      struct node *next;
};
```

Sequential insert

```
G->next = P2;
P1->next = G;
```

How to execute the two instructions atomically?

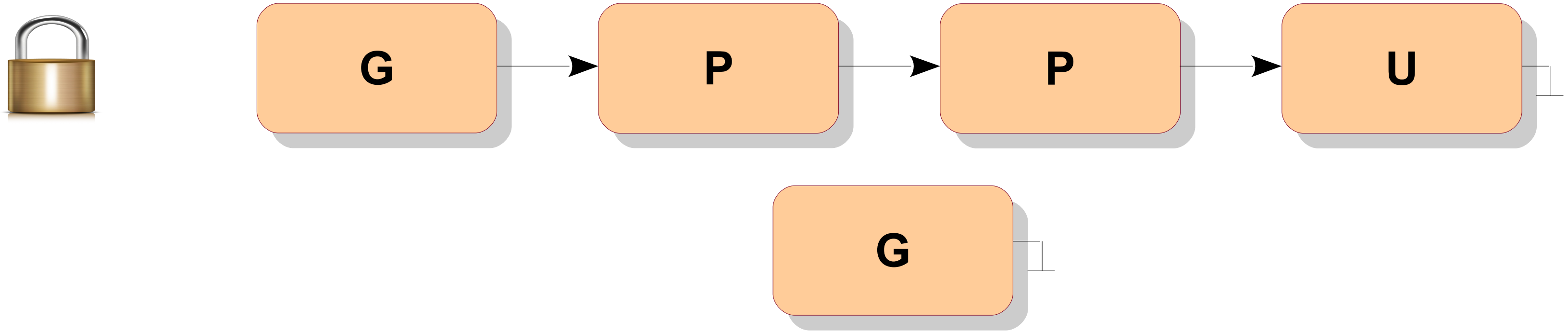


Concurrent Linked List

Solution 1: Keep a lock with the list.

- Coarse-grained synchronization
- Low concurrency / sequential access
- Easy to implement
- Easy to argue about correctness

Classwork: Implement lock() and unlock().



lock() and unlock()

```
void lock(List &list) {
        while (list.sema == 1)
                ;
        list.sema = 1;
}
void unlock(List &list) {
        list.sema = 0;
}
```

time

T1	T2	T3
sema = 1		
-- CS --		
sema = 0		
	sema == 1	sema == 1
	sema = 1	sema = 1
	-- CS --	-- CS --

What is the problem here? How to fix it?

lock() and unlock()

```
void lock(List &list) {
      atomicCAS(&list.sema, 0, 1);
}
void unlock(List &list) {
      atomicCAS(&list.sema, 1, 0);
}
```

```
void lock(List &list) {
      do {
            old = atomicCAS(&list.sema, 0, 1);
      } while (old == 1);
}
void unlock(List &list) {
      list.sema = 0;
}
```

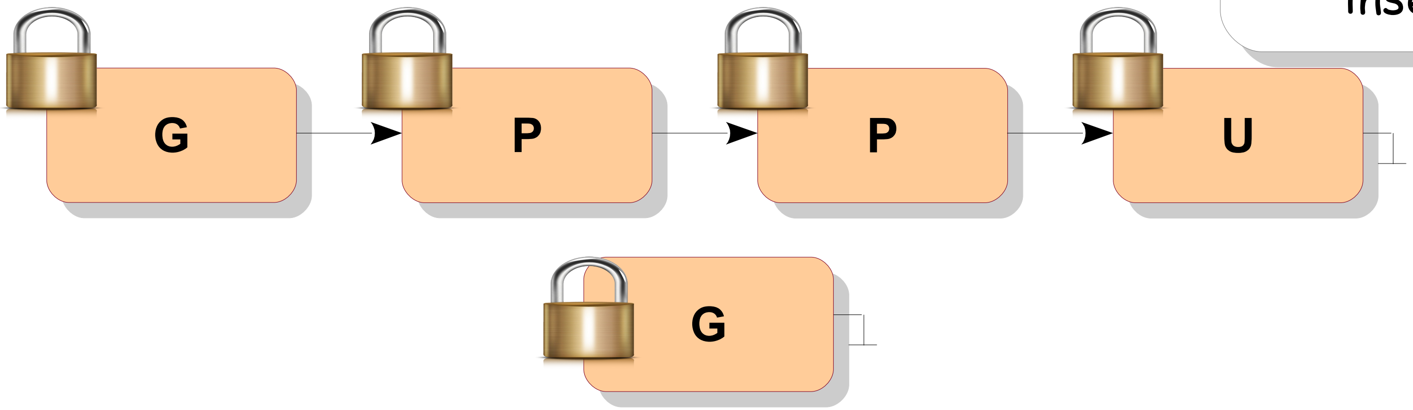
Concurrent Linked List

Solution 2: Keep a lock with each node.

- Fine-grained synchronization
- Better concurrency
- Moderately difficult to implement, need to finalize the supported operations
- Difficult to argue about correctness when multiple nodes are involved

Classwork: Implement insert().

Classwork: Check if two concurrent inserts work.

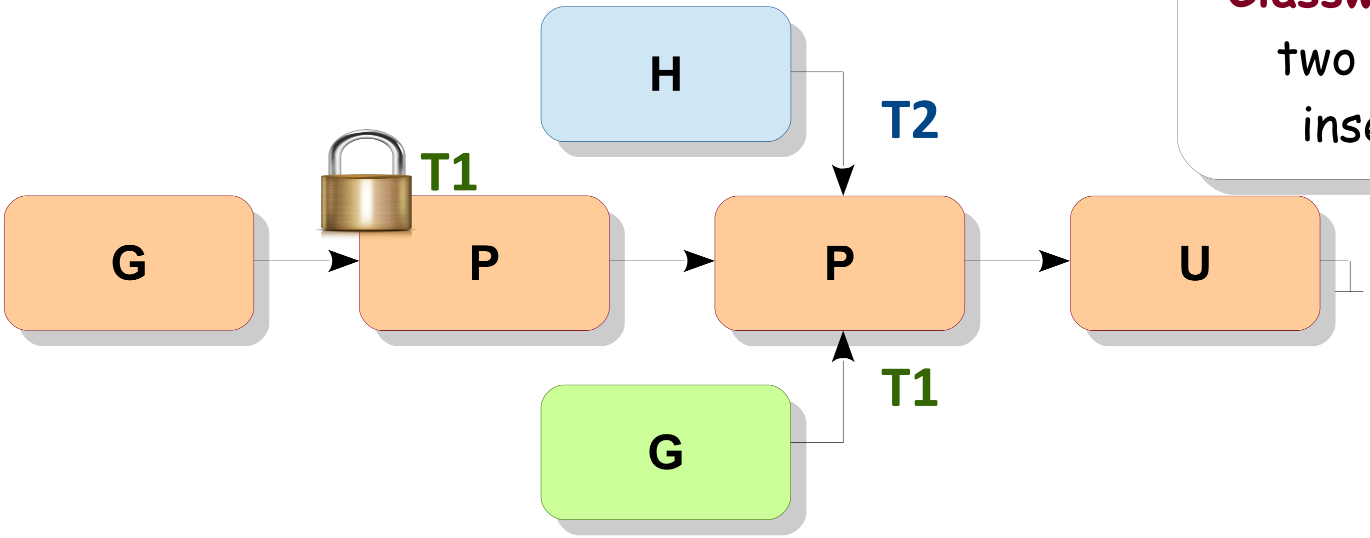


Concurrent Linked List

```
void insert(Node &prev, Node &naya) {
        naya.next = prev.next;
        prev.lock();
        prev.next = naya;
        prev.unlock();
}
```

Classwork: Implement insert().

Classwork: Check if two concurrent inserts work.

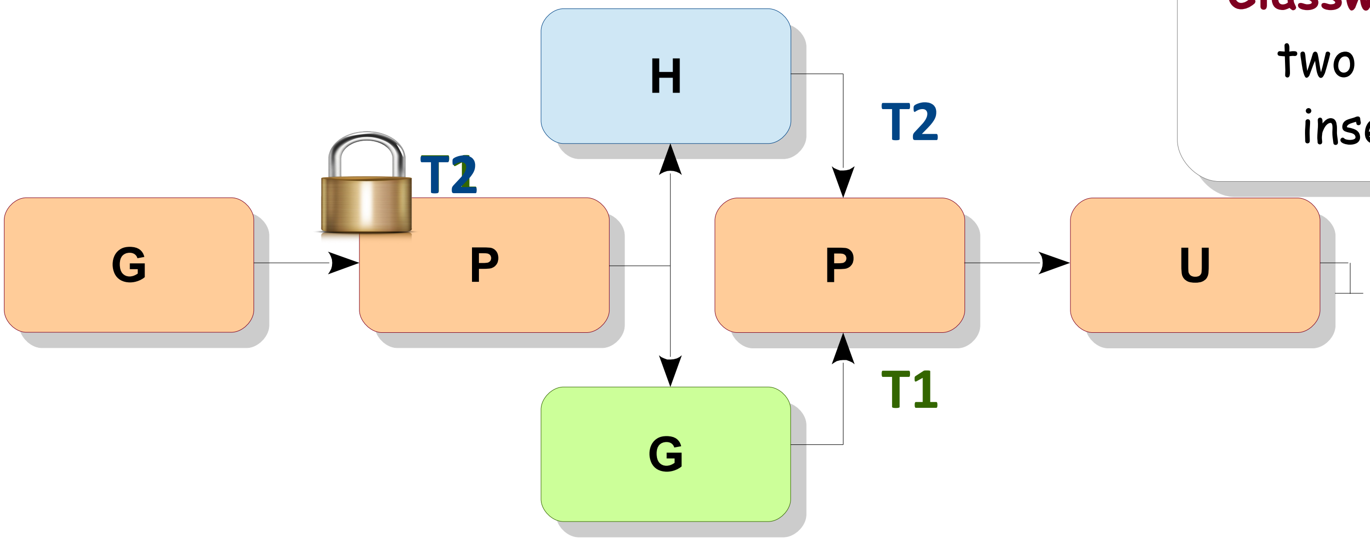


Concurrent Linked List

```
void insert(Node &prev, Node &naya) {
      naya.next = prev.next;
      prev.lock();
      prev.next = naya;
      prev.unlock();
}
```

Classwork: Implement insert().

Classwork: Check if two concurrent inserts work.

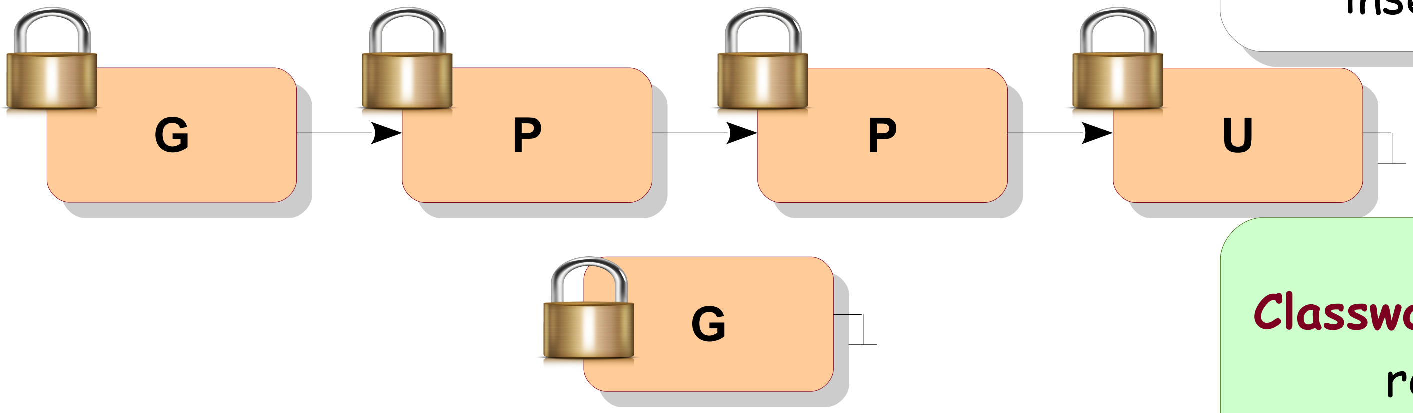


Concurrent Linked List

```
void insert(Node &prev, Node &naya) {
        prev.lock();
        naya.next = prev.next;
        prev.next = naya;
        prev.unlock();
}
```

Classwork: Implement insert().

Classwork: Check if two concurrent inserts work.



Classwork: Now allow remove().

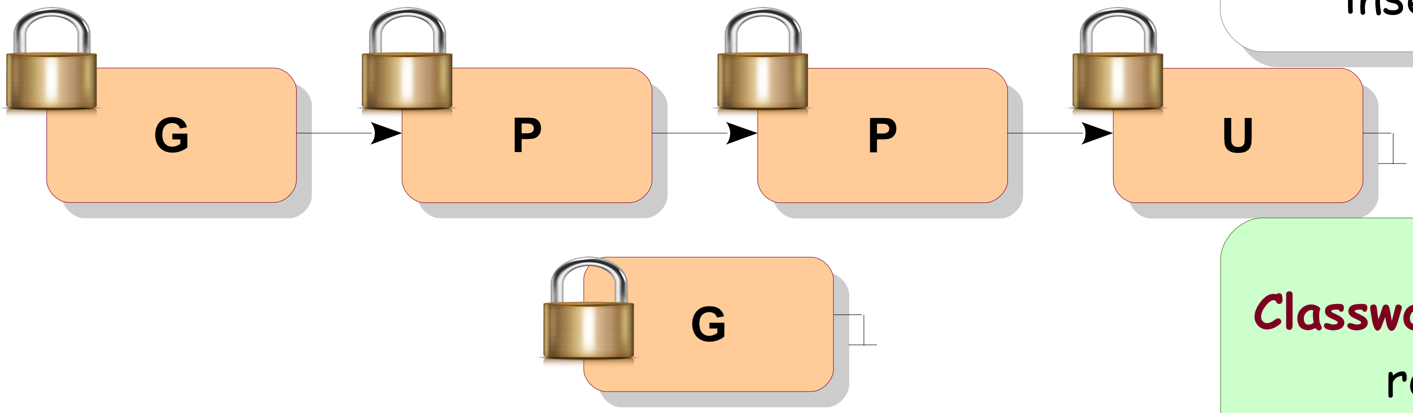
```
void insert(Node &prev, Node &naya) {
        prev.lock();
        naya.next = prev.next;
        prev.next = naya;
        prev.unlock();
}
void remove(Node &prev, Node &tbr) {
        prev.lock();
        tbr.lock();
        prev.next = tbr.next;
        // process tbr.
        tbr.unlock();
        prev.unlock();
}
```

If the order of locks is reversed?

If the order of unlocks is reversed?

Classwork: Implement insert().

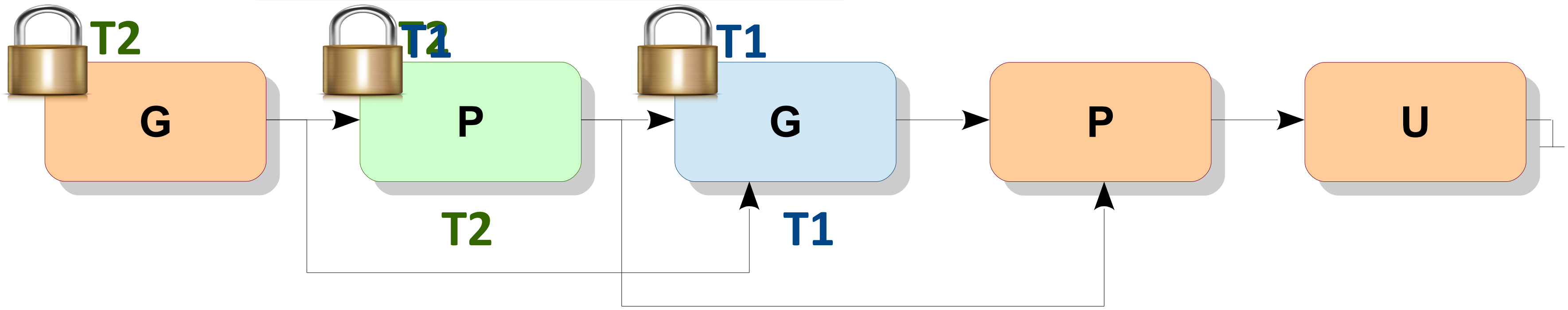
Classwork: Check if two concurrent inserts work.



Classwork: Now allow remove().

```
void insert(Node &prev, Node &naya) {
        prev.lock();
        naya.next = prev.next;
        prev.next = naya;
        prev.unlock();
}
void remove(Node &prev, Node &tbr) {
        tbr.lock();
        prev.lock();
        prev.next = tbr.next;
        // process tbr.
        tbr.unlock();
        prev.unlock();
}
```

If the order of locks is reversed?

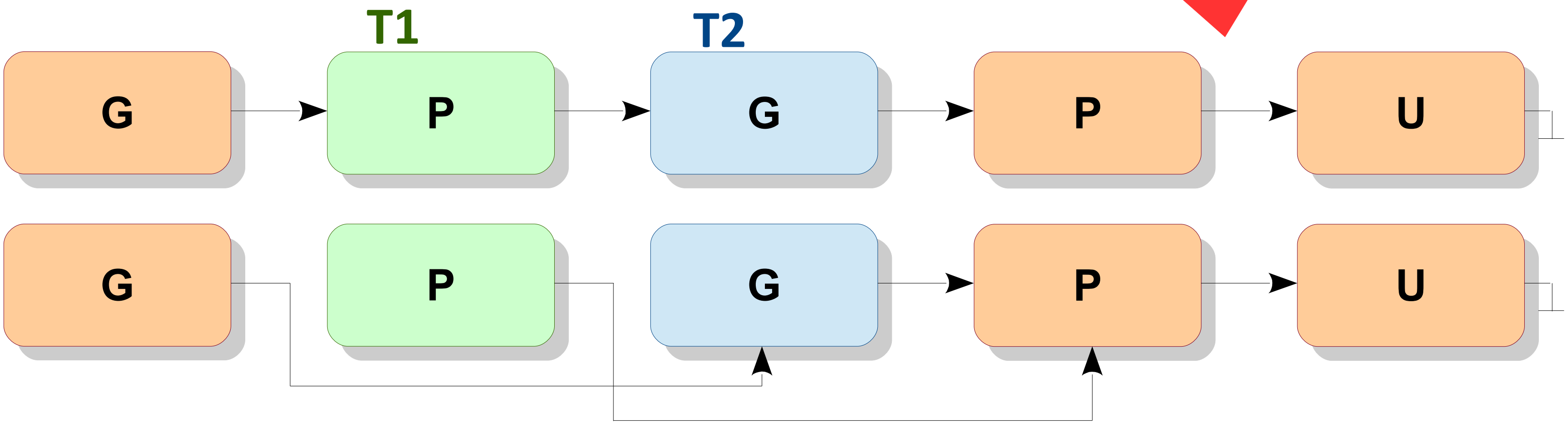


```
void insert(Node &prev, Node &naya) {
        prev.lock();
        naya.next = prev.next;
        prev.next = naya;
        prev.unlock();
}
void remove(Node &prev, Node &tbr) {
        prev.lock();
        tbr.lock();
        prev.next = tbr.next;
        // process tbr.
        tbr.unlock();
        prev.unlock();
}
```

Isn't a similar issue possible with our current implementation?

Where is the problem?

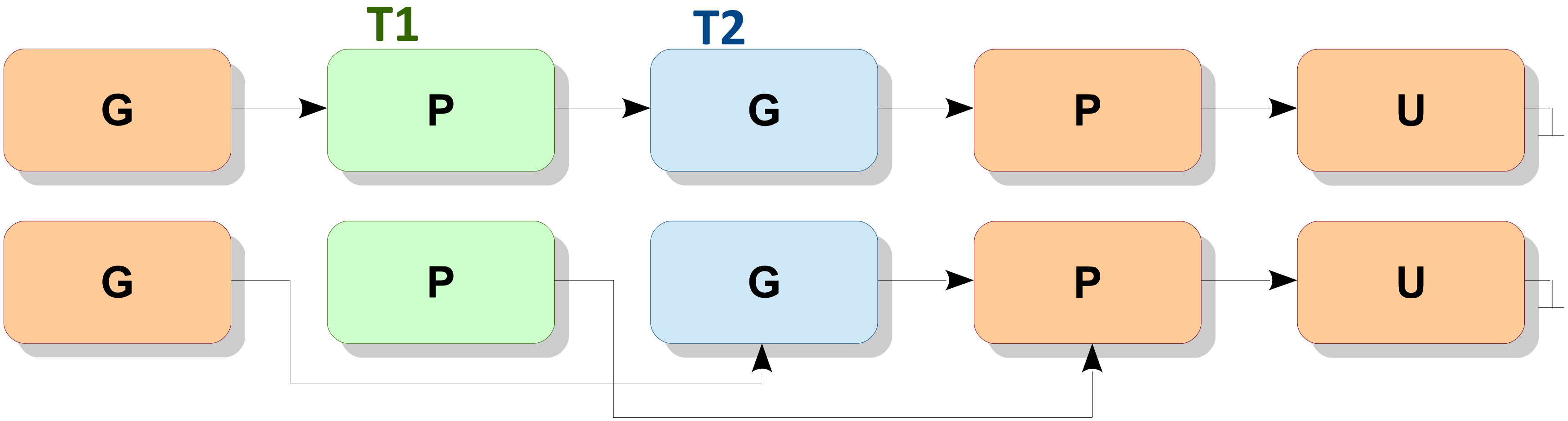
Expected GPU, received GGPU.



```
void remove(Node &prev) {
        prev.lock();
        tbr = prev.next;
        tbr.lock();
        prev.next = tbr.next;
        // process tbr.
        tbr.unlock();
        prev.unlock();
}
```

This would solve the problem of tbr, but what about remove(P) and remove(P) executing concurrently?

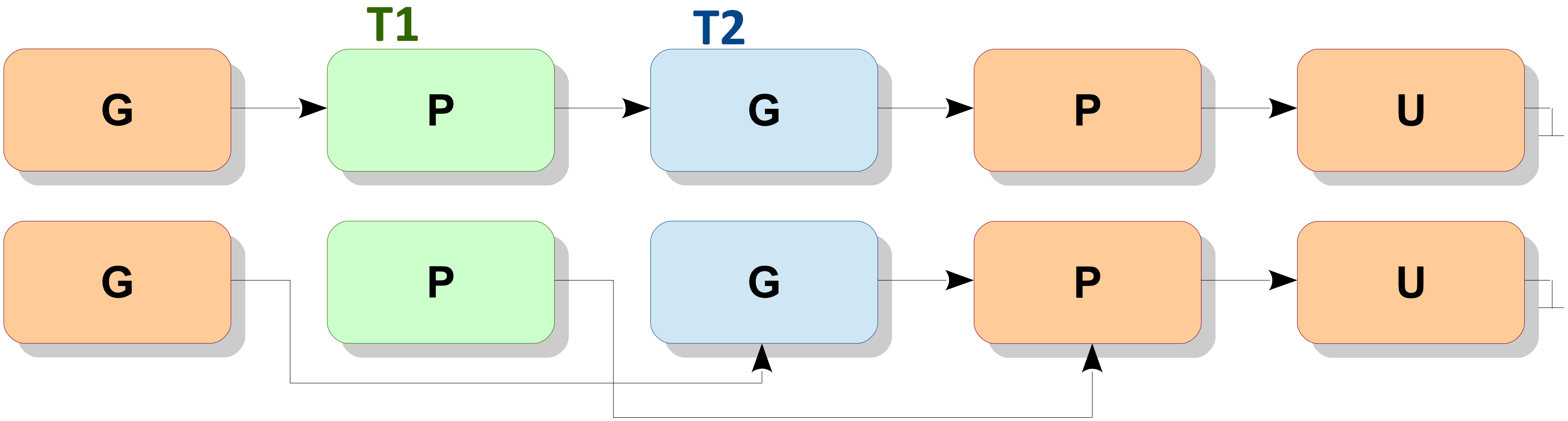
This requires us to check for a node's validity.



```
int remove(Node &prev) {
    if (prev.valid == 0) return -1;
    prev.lock();
    tbr = prev.next;
    tbr.lock();
    prev.next = tbr.next;
    // process tbr.
    tbr.valid = 0;
    tbr.unlock();
    prev.unlock();
}
```

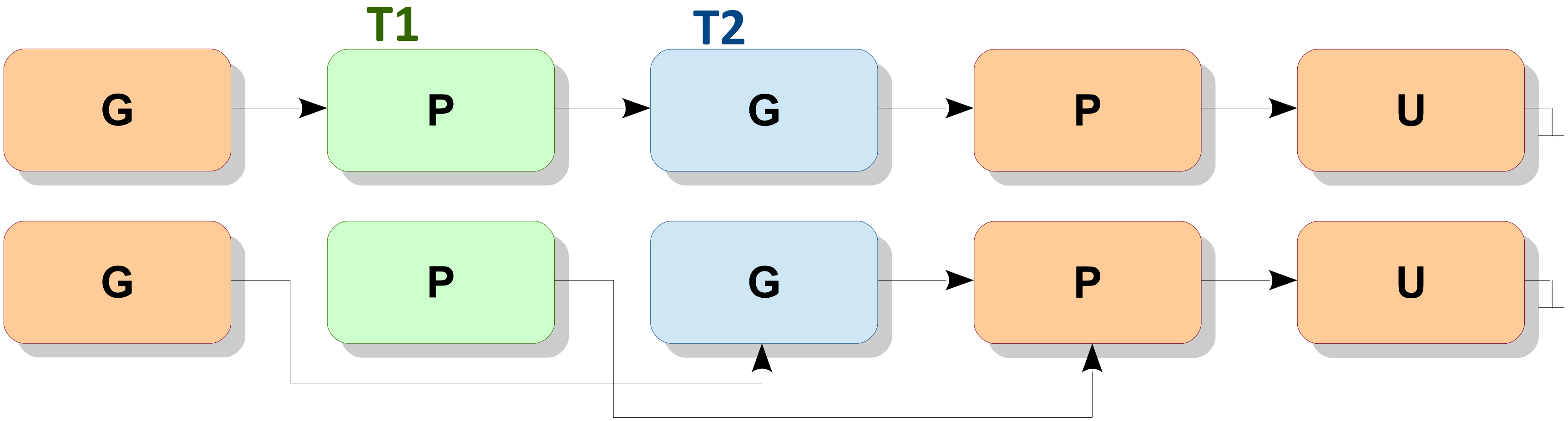
Checking for validity and locking needs to be *atomic*!

- Memory is not reclaimed!
- Only insert and remove!
- No traversal yet!
- Direct pointer is given!
- Still so many complications!



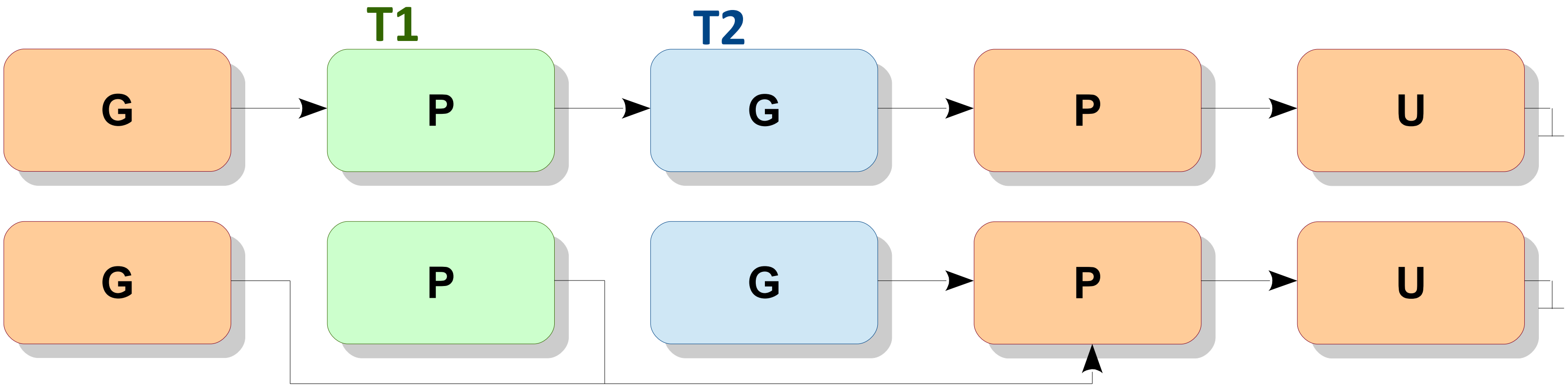
```
int remove(Node &prev) {
    prev.lock();
    if (prev.valid == 0) return -1;
    tbr = prev.next;
    tbr.lock();
    if (tbr.valid == 0) return -2;
    prev.next = tbr.next;
    // process tbr.
    tbr.valid = 0;
    tbr.unlock();
    prev.unlock();
}
```

Does not unlock on error.



```
int remove(Node &prev) {
        prev.lock();
        if (prev.valid) {
                tbr = prev.next;
                tbr.lock();
                if (tbr.valid) {
                        prev.next = tbr.next;
                        // process tbr.
                        tbr.valid = 0;
                }
                tbr.unlock();
        }
        prev.unlock();
}
```

Homework: Find out issues with this code.

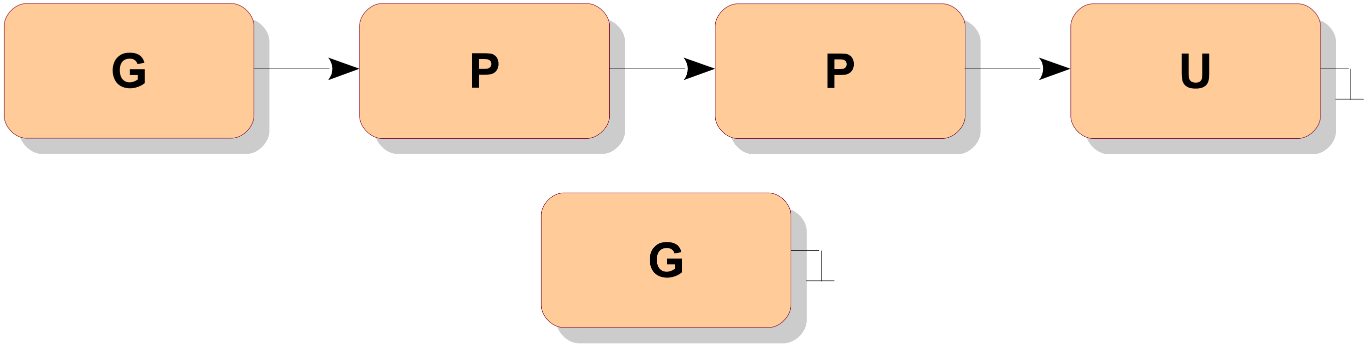


Concurrent Linked List

Solution 3: Use atomics to insert.

- Possible only in a few cases
- Difficult to implement with multiple inserts
- Difficult to prove correctness

Classwork: Implement insert().

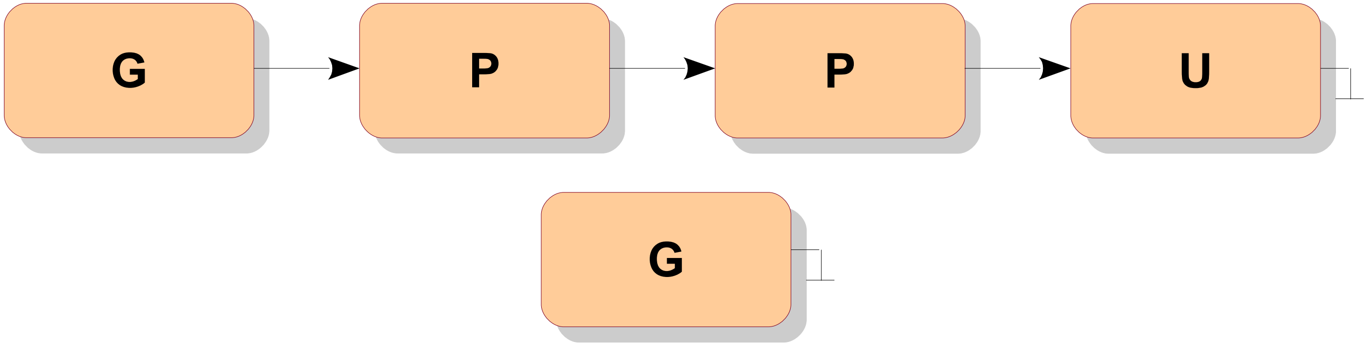


```
void insert(Node &prev, Node &naya) {
        do {
                naya.next = prev.next;
                old = atomicCAS(&prev.next, naya.next, &naya);
        } while (old != naya.next);
}
```

Dare to support remove?

```
void remove(Node &prev) {
        do {
                tbr = prev.next;
                old = atomicCAS(&prev.next, tbr, tbr.next);
        } while (old != tbr);
}
```

Mostly works.
Problem with multiple concurrent remove(P).



CPU-GPU Synchronization

- While GPU is busy doing work, CPU may perform useful work.
- If CPU-GPU collaborate, they require synchronization.

Classwork: Implement
a functionality to print sequence 0..10.
CPU prints even numbers,
GPU prints odd.

CPU-GPU Synchronization

```
#include <cuda.h>
#include <stdio.h>

__global__ void printk(int *counter) {
        ++*counter;
        printf("\t%d\n", *counter);
}
int main() {
        int hcounter = 0, *counter;

        cudaMalloc(&counter, sizeof(int));

        do {
                printf("%d\n", hcounter);
                cudaMemcpy(counter, &hcounter, sizeof(int), cudaMemcpyHostToDevice);
                printk <<<1, 1>>>(counter);
                cudaMemcpy(&hcounter, counter, sizeof(int), cudaMemcpyDeviceToHost);
        } while (++hcounter < 10);

        return 0;
}
```

Pinned Memory

- Typically, memories are pageable (swappable).
- CUDA allows to make host memory pinned.
- CUDA allows direct access to pinned host memory from device.
- cudaHostAlloc(&pointer, size, 0);

Classwork: Implement
the same functionality to print sequence 0..10.
CPU prints even numbers,
GPU prints odd.

Pinned Memory

```
#include <cuda.h>
#include <stdio.h>

__global__ void printk(int *counter) {
        ++*counter;
        printf("\t%d\n", *counter);
}
int main() {
        int *counter;

        cudaHostAlloc(&counter, sizeof(int), 0);

        do {
                printf("%d\n", *counter);
                printk <<<1, 1>>>(counter);
                cudaDeviceSynchronize();
                ++*counter;
        } while (*counter < 10);

        cudaFreeHost(counter);
        return 0;
}
```

No cudaMempcy!

Classwork: Can we avoid repeated kernel calls?

Persistent Kernels

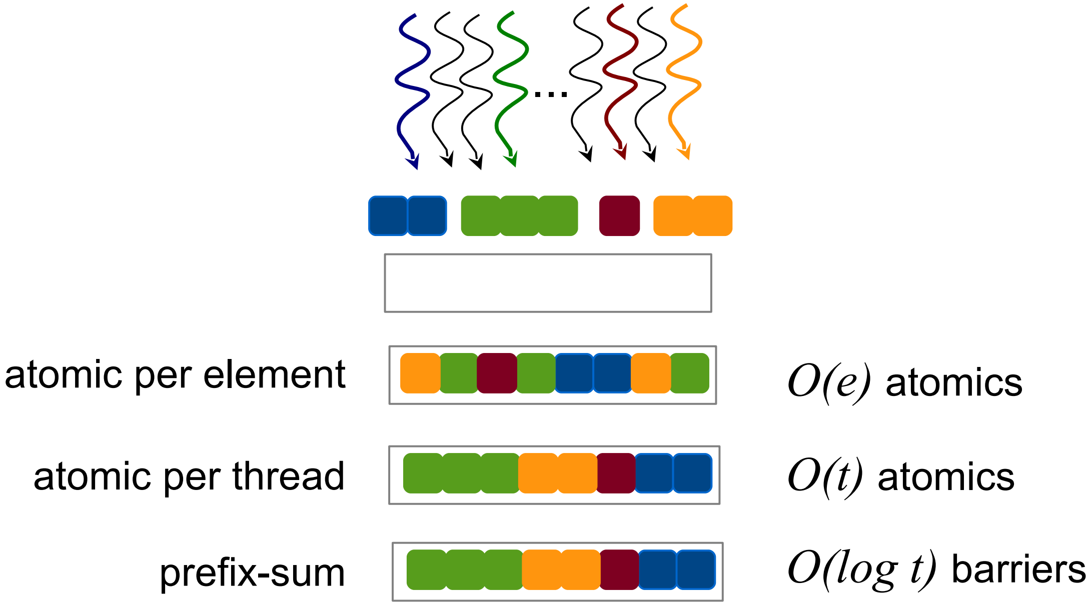
```
__global__ void printk(int *counter) {
        do {
                while (*counter % 2)  ;
                ++*counter;
                printf("\t%d\n", *counter);
        } while (*counter < 10);
}
int main() {
        int *counter;

        cudaHostAlloc(&counter, sizeof(int), 0);
        printk <<<1, 1>>>(counter);

        do {
                printf("%d\n", *counter);
                while (*counter % 2 == 0)  ;
                ++*counter;
        } while (*counter < 10);

        cudaFreeHost(counter);
        return 0;
}
```

Extra

Barrier-based Synchronization

- → **Disjoint accesses**
- Overlapping accesses
- Benign overlaps

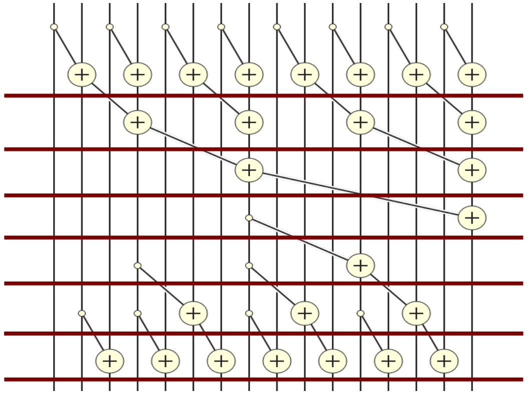
Consider threads pushing elements into a worklist



atomic per element — $O(e)$ atomics

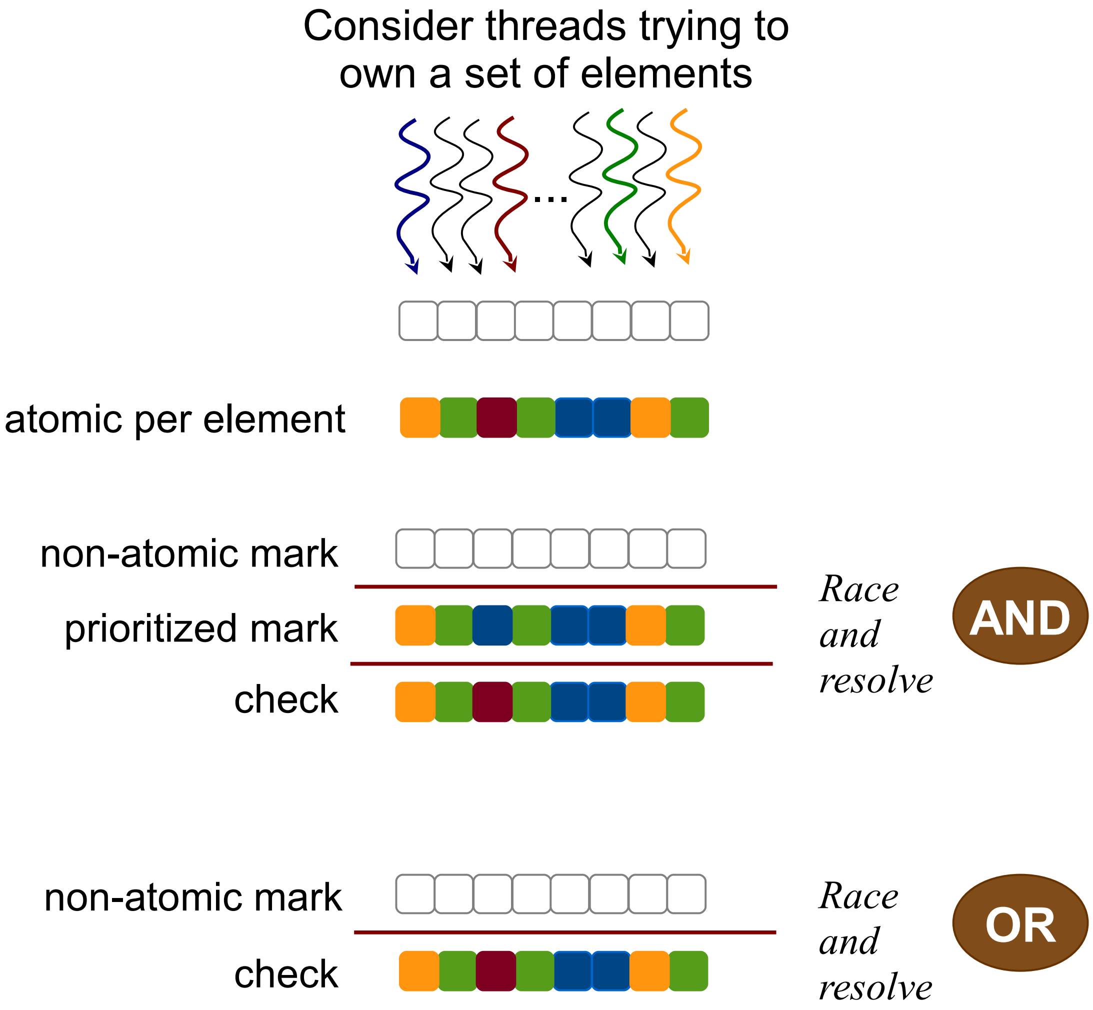
atomic per thread — $O(t)$ atomics

prefix-sum — $O(\log t)$ barriers



Barrier-based Synchronization

- Disjoint accesses
- ➔ **Overlapping accesses**
- Benign overlaps



Consider threads trying to
own a set of elements

atomic per element

e.g., for owning cavities in
Delaunay mesh refinement

non-atomic mark

prioritized mark

check

Race and resolve

AND

e.g., for inserting unique
elements into a worklist

non-atomic mark

check

Race and resolve

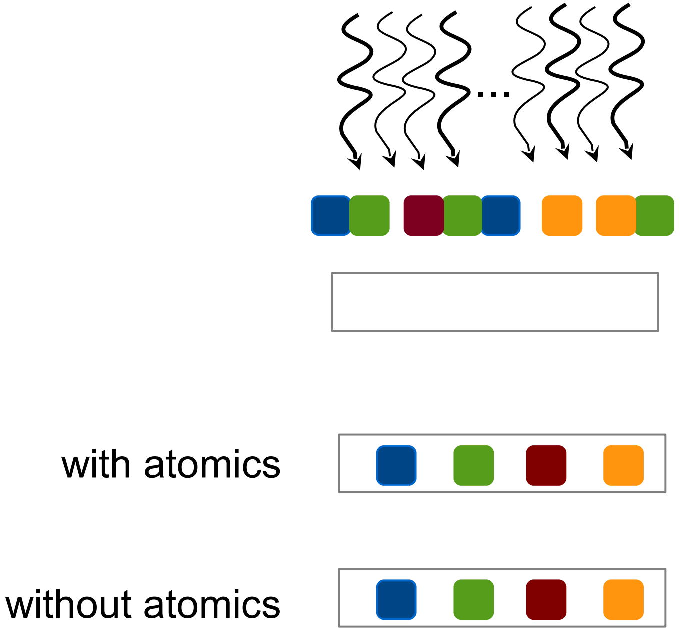
OR

Barrier-based Synchronization

- Disjoint accesses
- Overlapping accesses

➔ **Benign overlaps**

Consider threads updating shared variables to the same value



with atomics

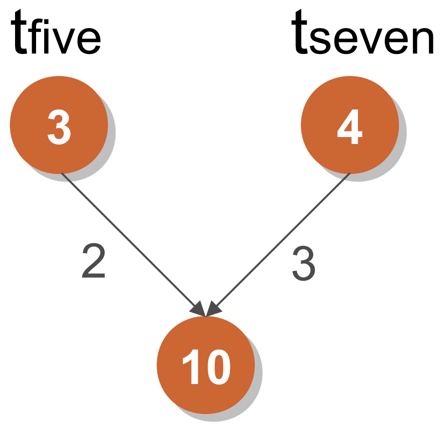
without atomics

e.g., level-by-level breadth-first search

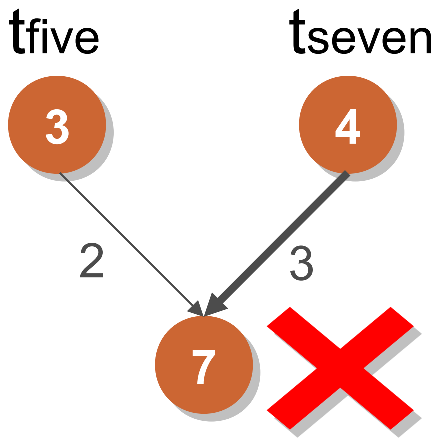
Exploiting Algebraic Properties

- **Monotonicity**
- Idempotency
- Associativity

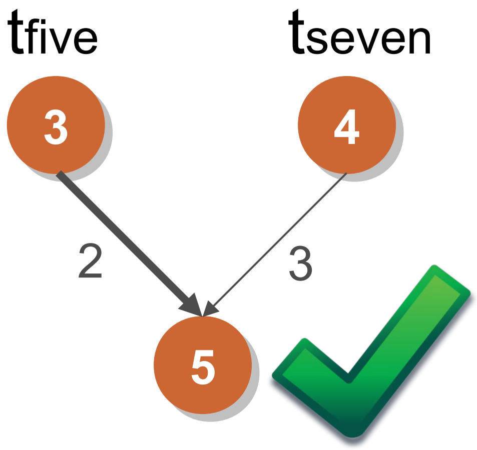
Consider threads updating distances in shortest paths computation



Atomic-free update



Lost-update problem

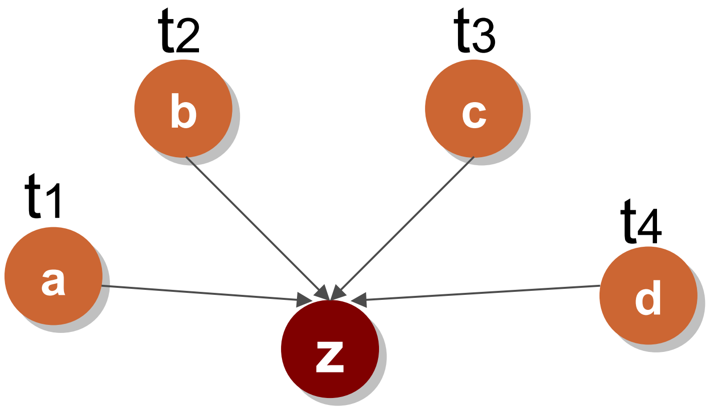


Correction by topology-driven processing, exploiting monotonicity

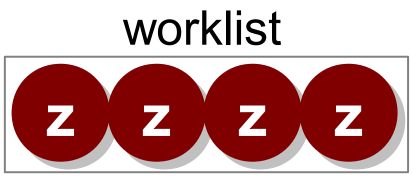
Exploiting Algebraic Properties

- Monotonicity
- ➔ **Idempotency**
- Associativity

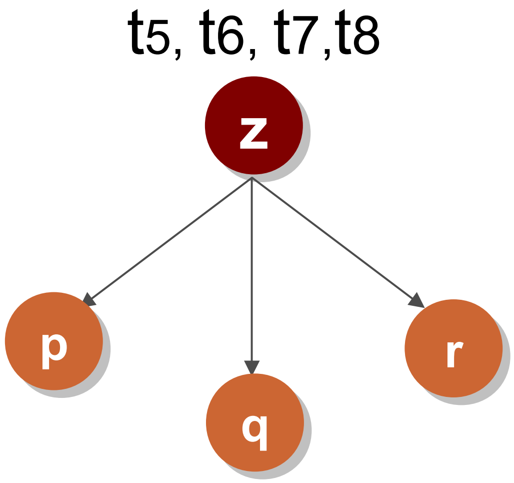
Consider threads updating distances in shortest paths computation



Update by multiple threads



Multiple instances of a node in the worklist

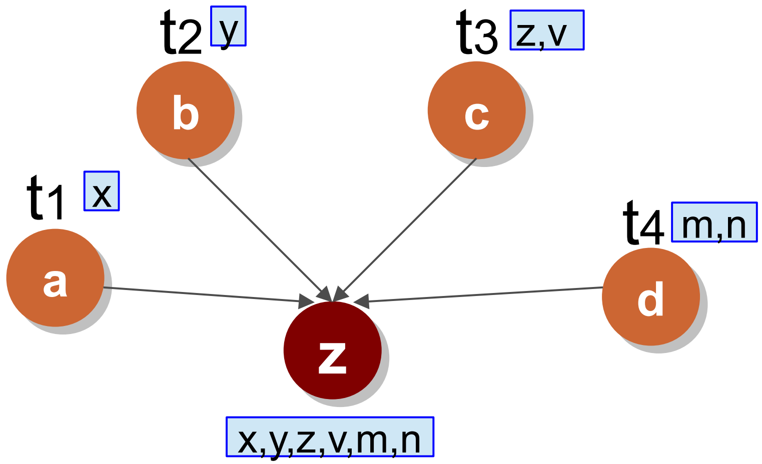


Same node processed by multiple threads

Exploiting Algebraic Properties

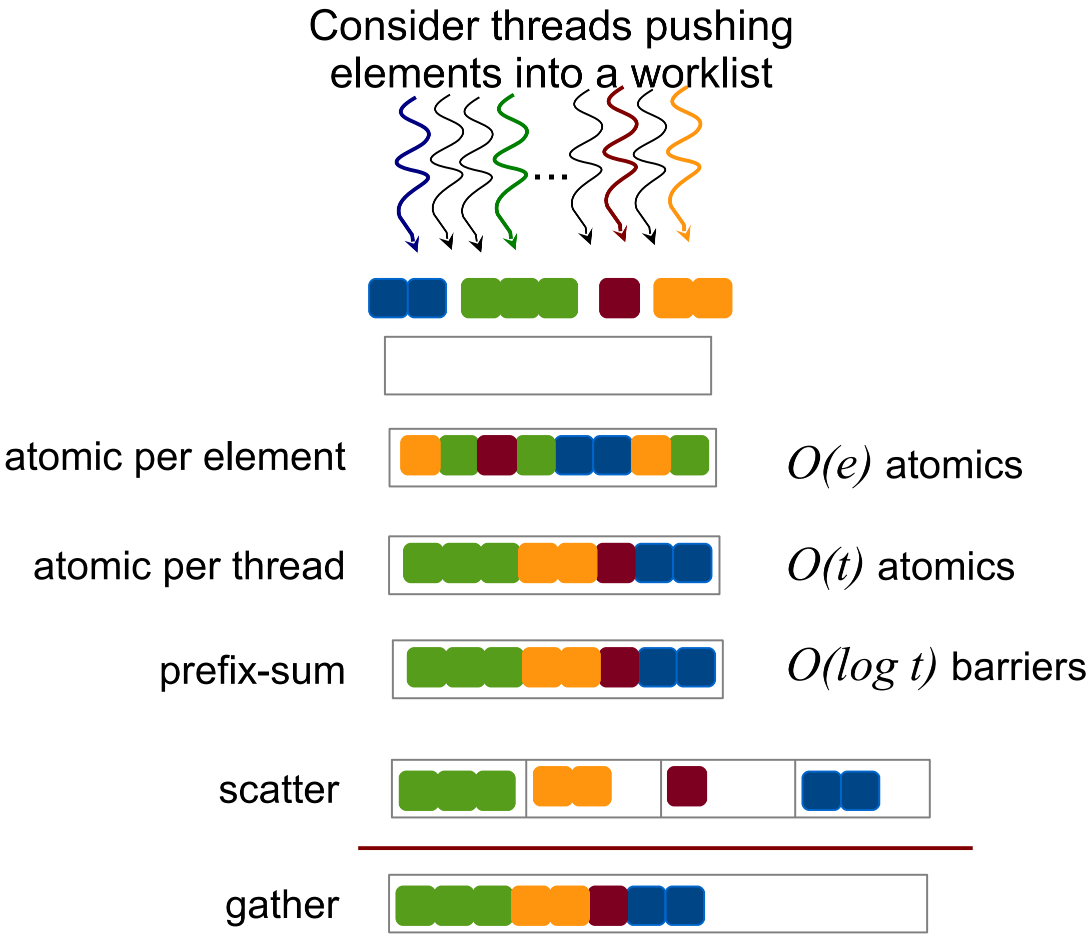
- Monotonicity
- Idempotency
- ➔ **Associativity**

Consider threads pushing information to a node



Associativity helps push information using prefix-sum

Scatter-Gather



Consider threads pushing elements into a worklist

atomic per element — $O(e)$ atomics

atomic per thread — $O(t)$ atomics

prefix-sum — $O(\log t)$ barriers

scatter

gather