GPU TECHNOLOGY CONFERENCE

Understanding and Using Atomic Memory Operations

Lars Nyland & Stephen Jones, NVIDIA

GTC 2013

What Is an Atomic Memory Operation?

- Uninterruptable read-modify-write memory operation
  - Requested by threads
  - Updates a value at a specific address
- Serializes contentious updates from multiple threads
- Enables co-ordination among >1 threads
- Limited to specific functions & data sizes

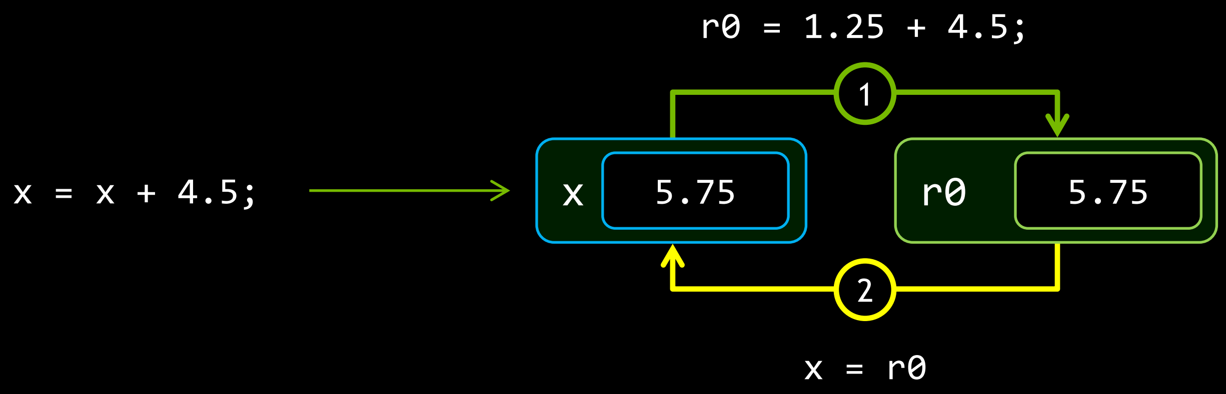
Precise Meaning of atomicAdd()

```
int atomicAdd(int *p, int v)
{
    int old;
    exclusive_single_thread
    {
        // atomically perform LD; ADD; ST ops
        old = *p; // Load from memory
        *p = old + v; // Store after adding v
    }
    return old;
}
```

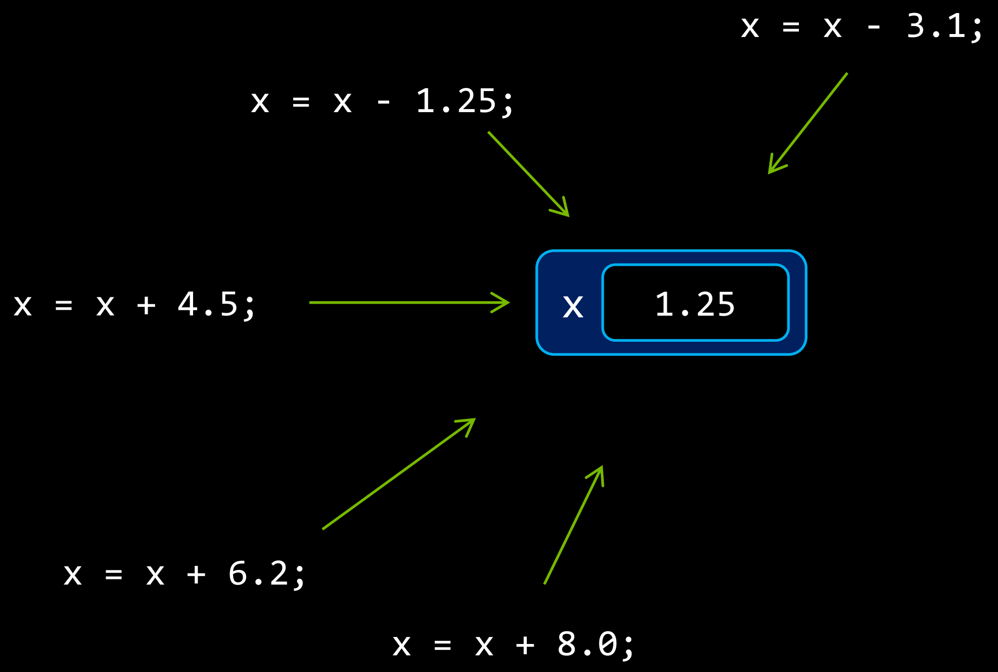
Simple Atomic Example

```
x = x + 4.5;
```

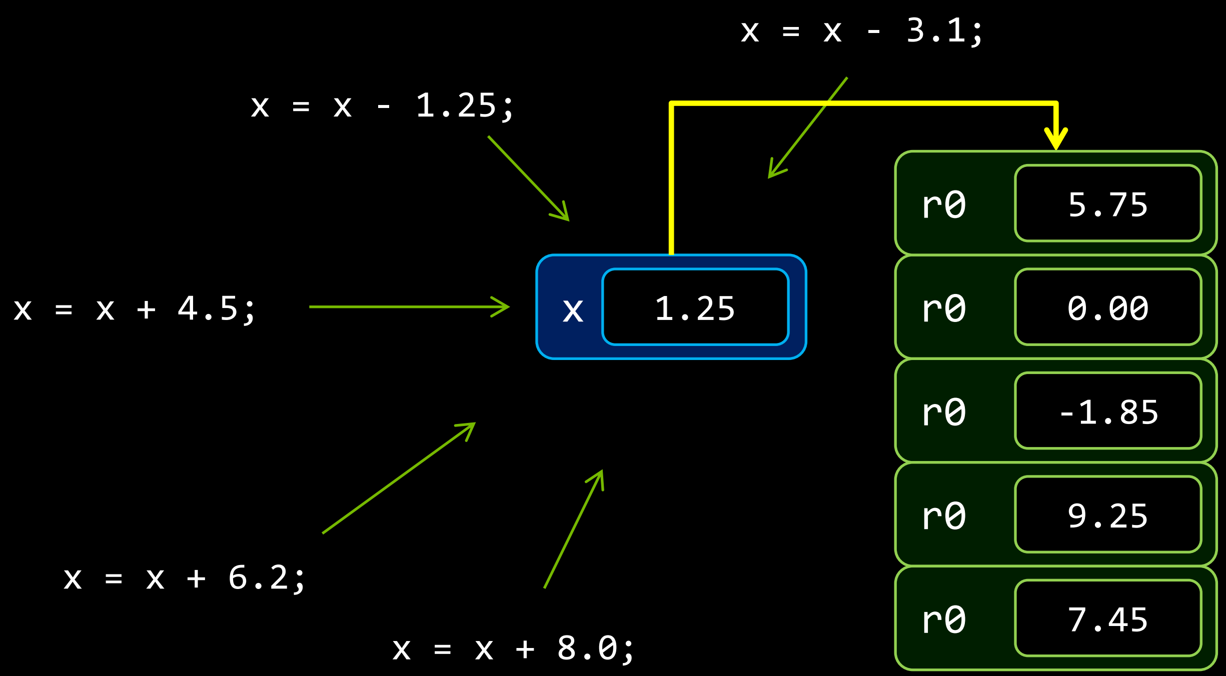
Simple Atomic Example



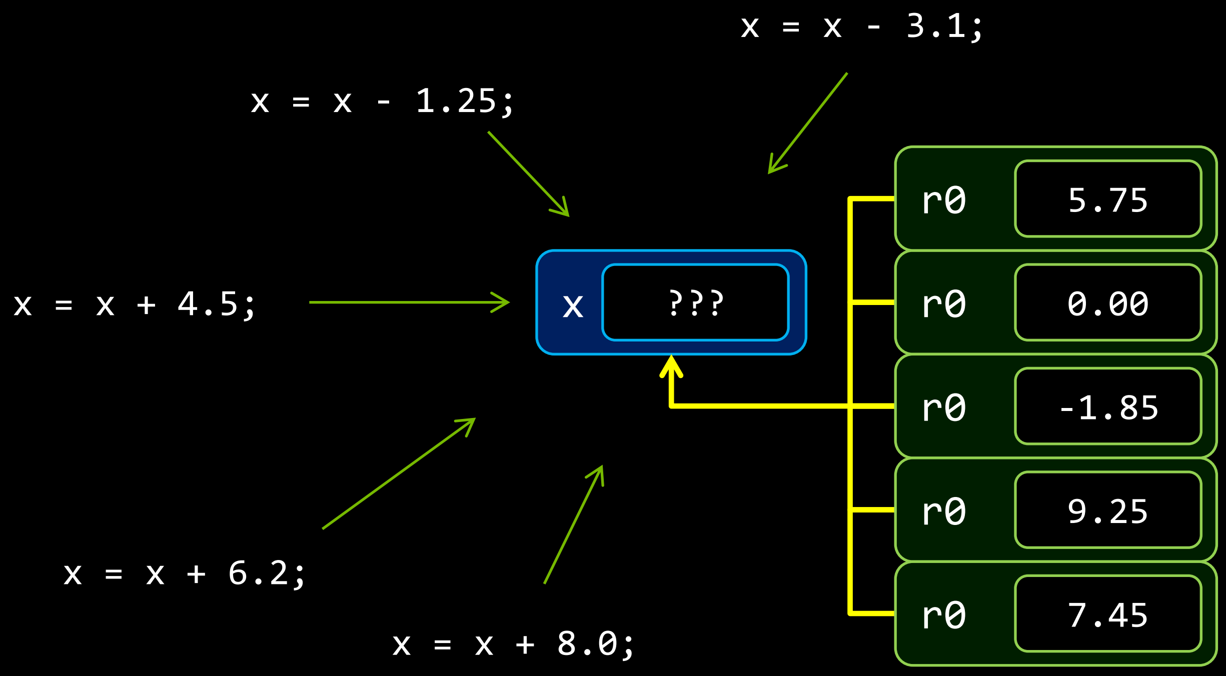
Simple Atomic Example



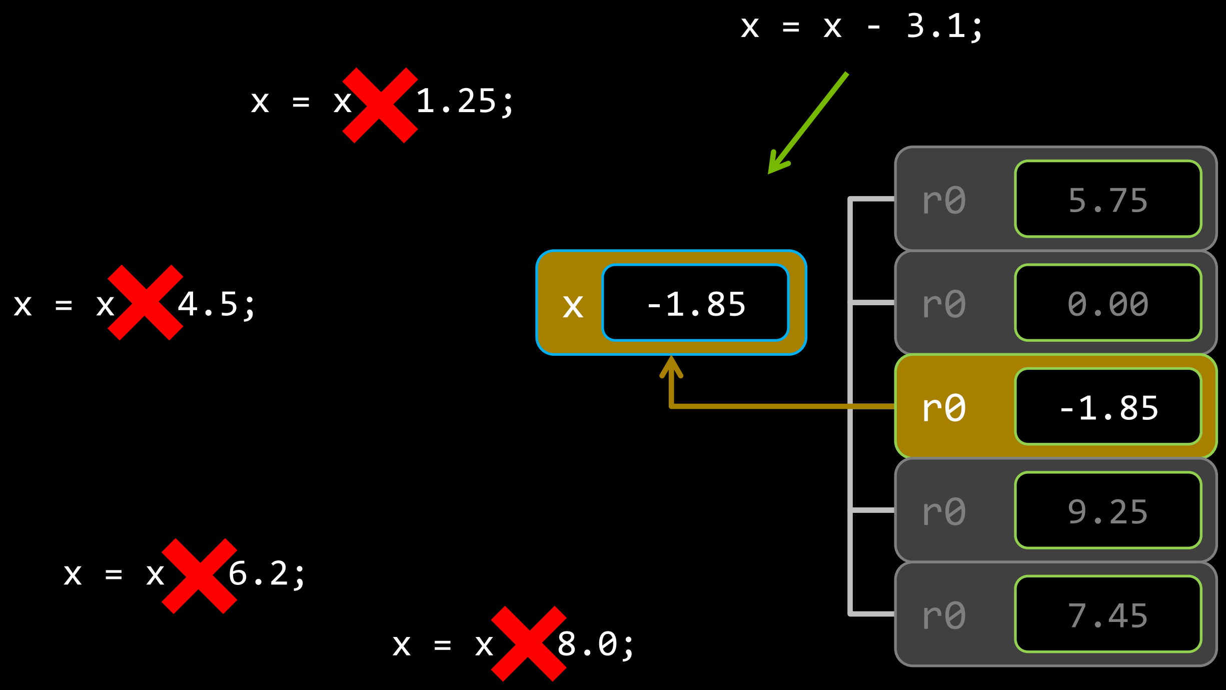
Simple Atomic Example



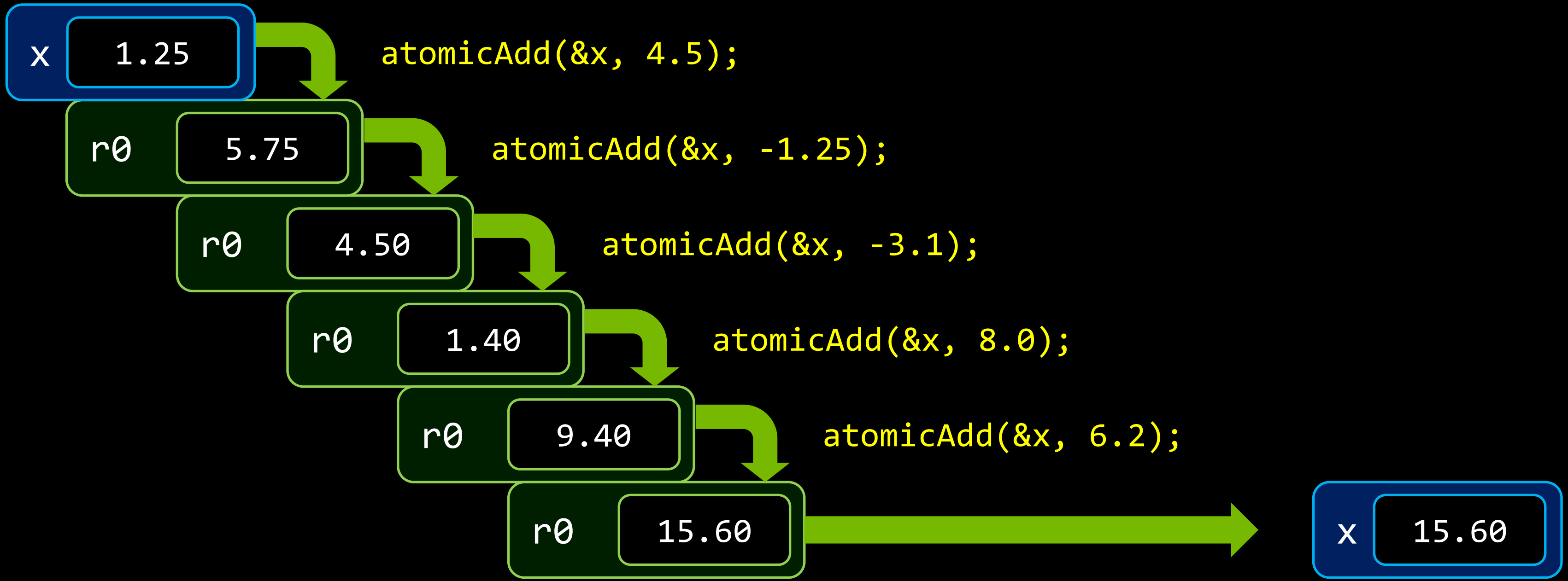
Simple Atomic Example



Simple Atomic Example



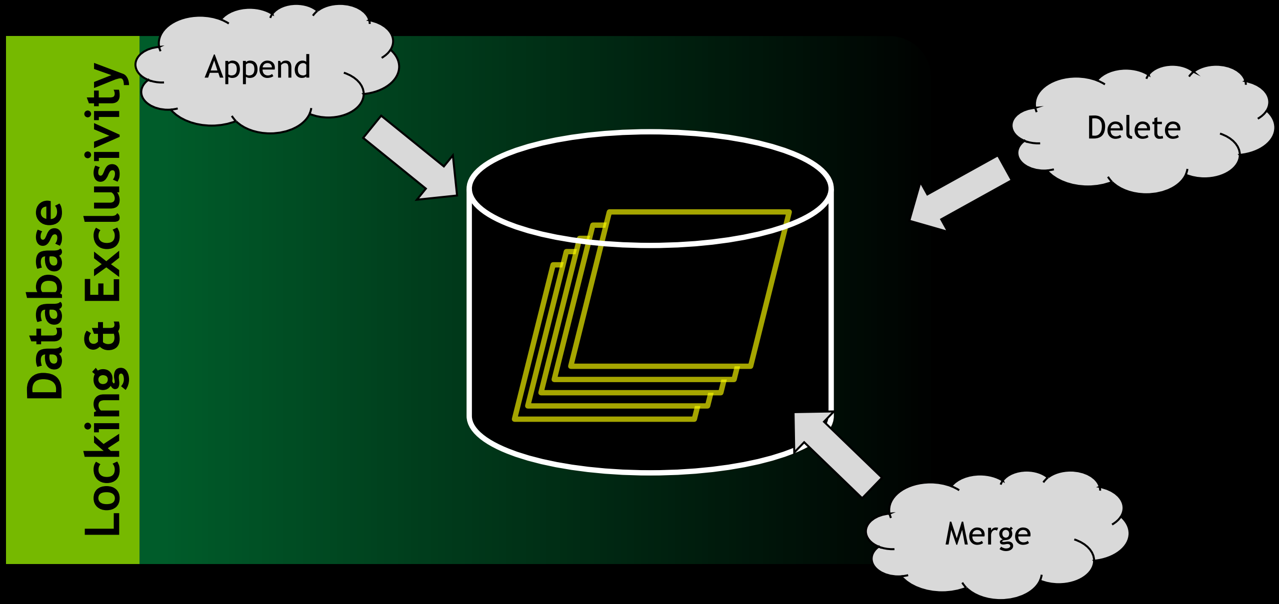
Simple Atomic Example



Why Use Atomics?

Common problem: races on read-modify-write of shared data

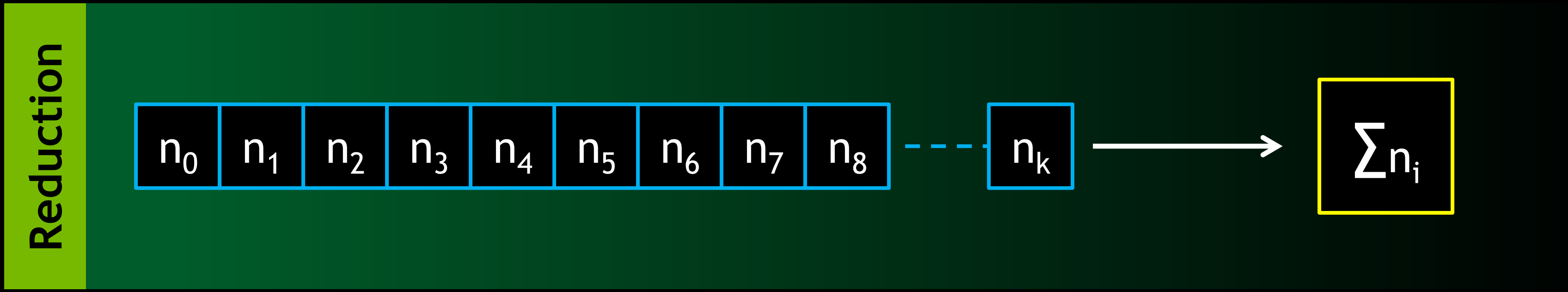
— Transactions & Data Access Control



Why Use Atomics?

Common problem: races on read-modify-write of shared data

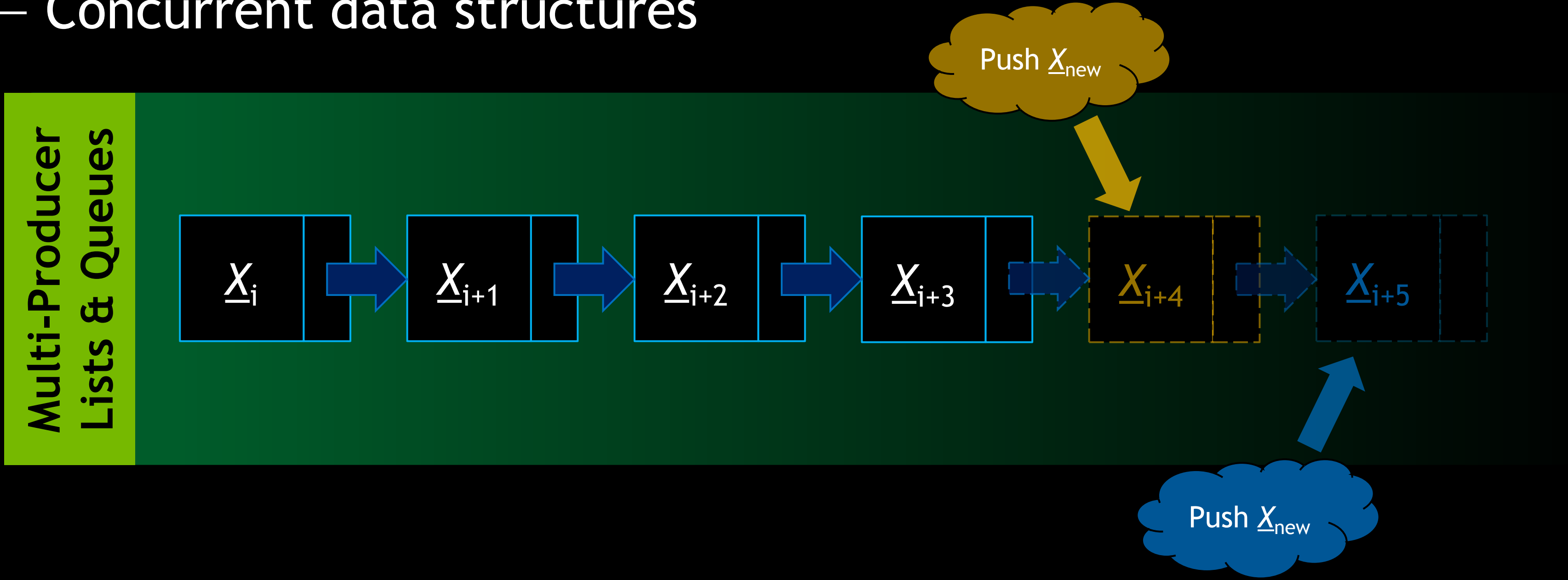
- Transactions & Data Access Control
- Data aggregation & enumeration



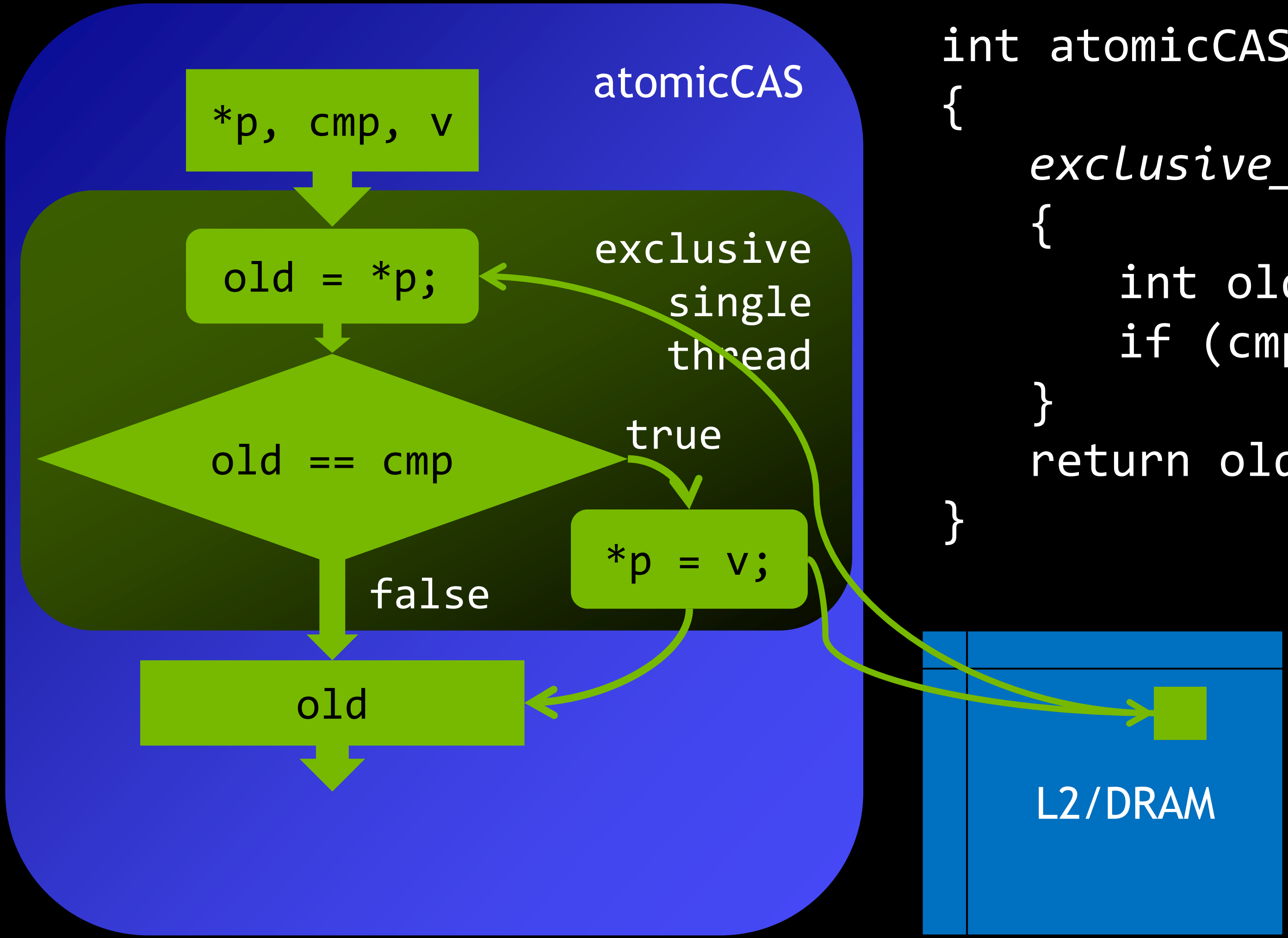
Why Use Atomics?

Common problem: races on read-modify-write of shared data

- Transactions & Data Access Control
- Data aggregation & enumeration
- Concurrent data structures

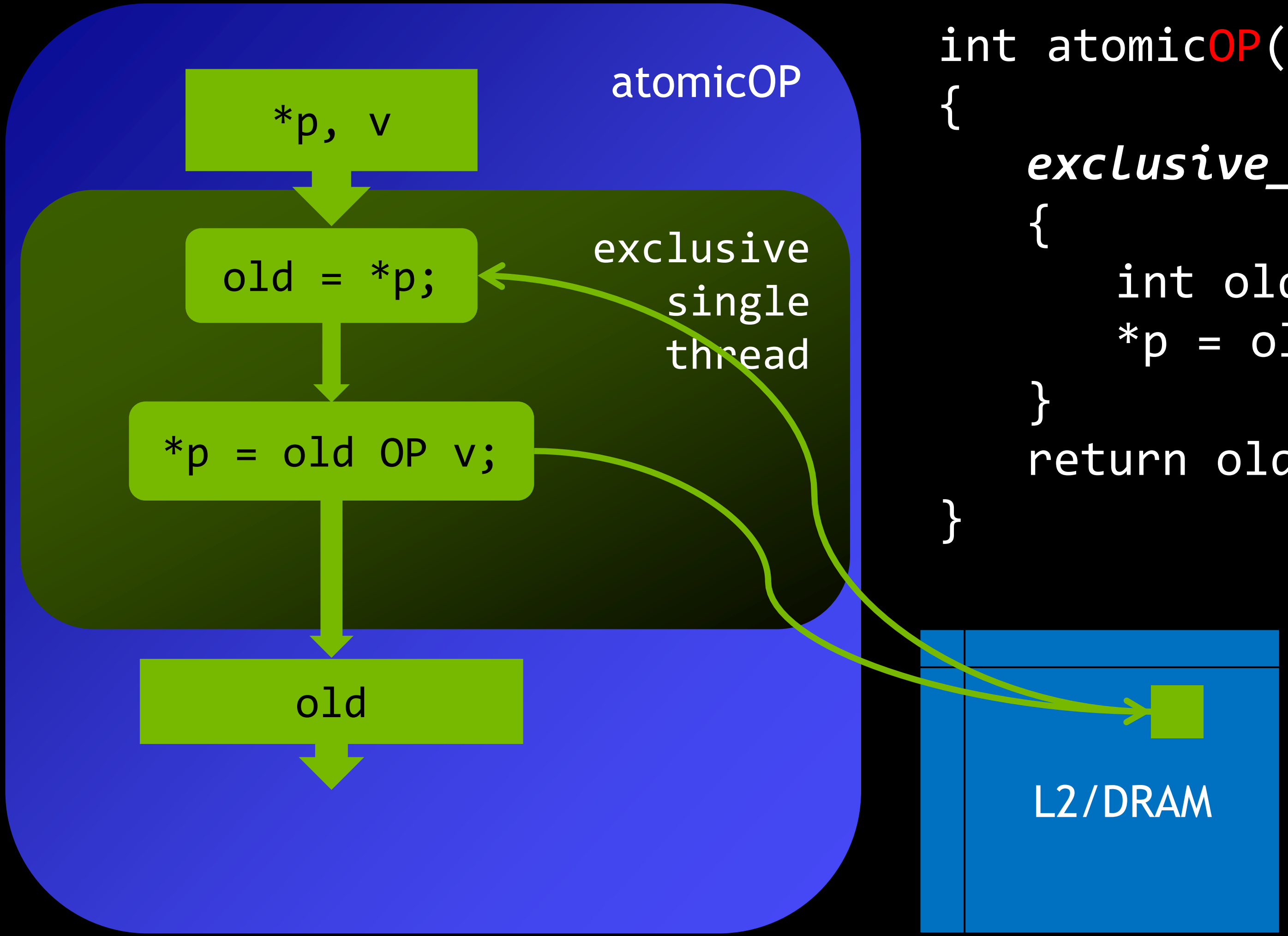


Compare-and-Swap




```
int atomicCAS(int *p, int cmp, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        if (cmp == old) *p = v;
    }
    return old;
}
```

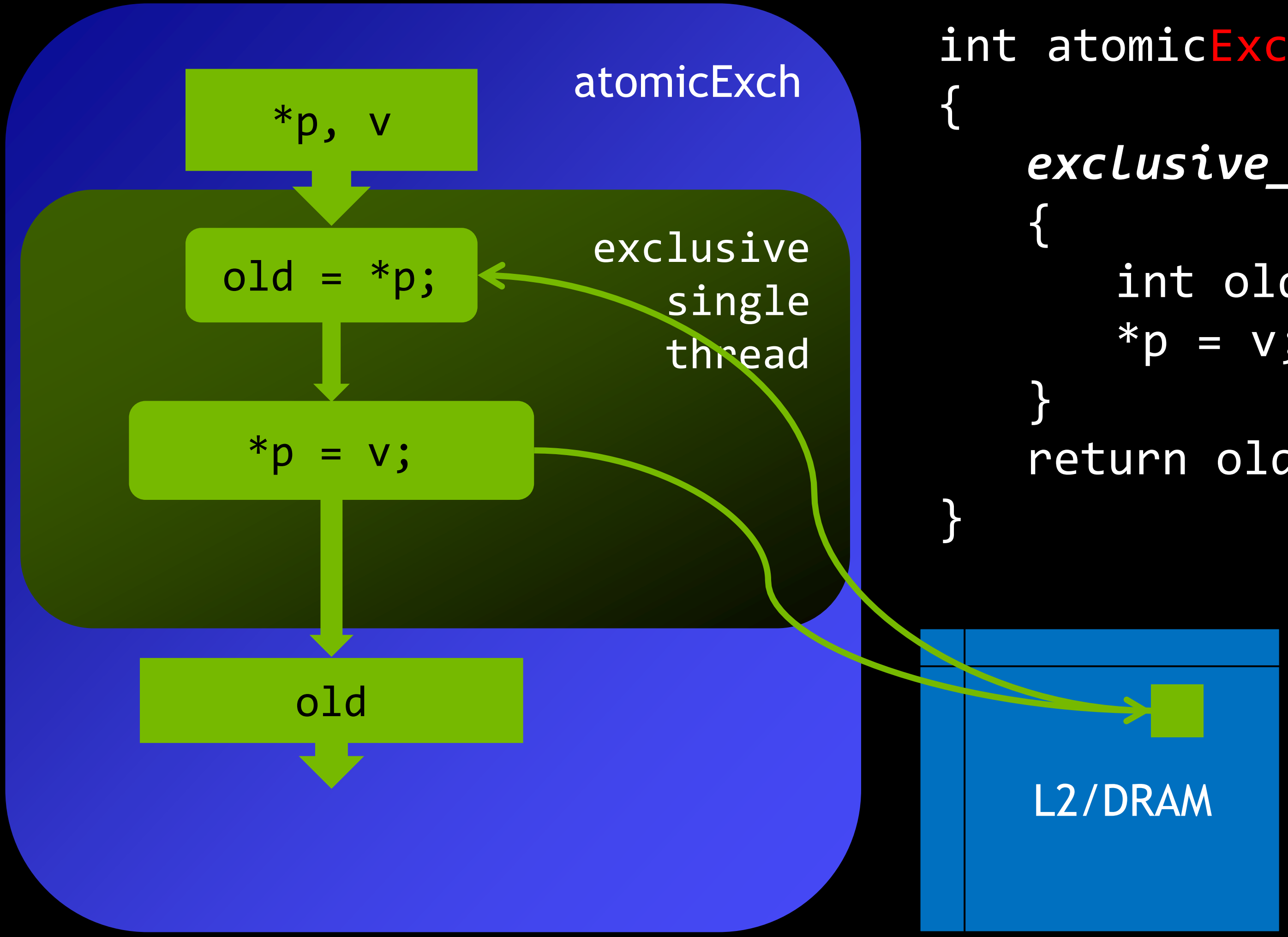
Arithmetic/Logical Atomic Operations




```
int atomicOP(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = old OP v;
    }
    return old;
}
```

Binary Ops:
Add, Min, Max
And, Or, Xor

Overwriting Atomic Operations




```
int atomicExch(int *p, int v)
{
    exclusive_single_thread
    {
        int old = *p;
        *p = v;
    }
    return old;
}
```

Programming Styles using Coordination

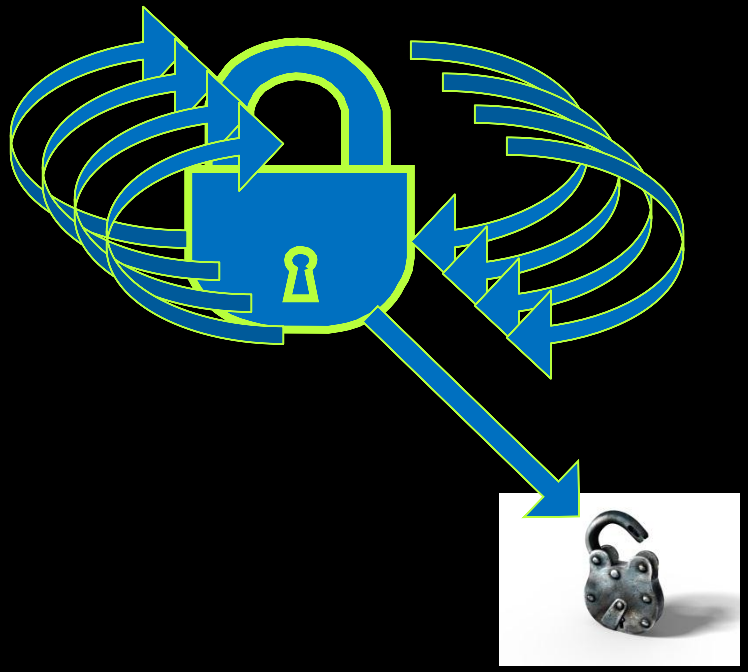
Locking

Lock-free

Wait-free

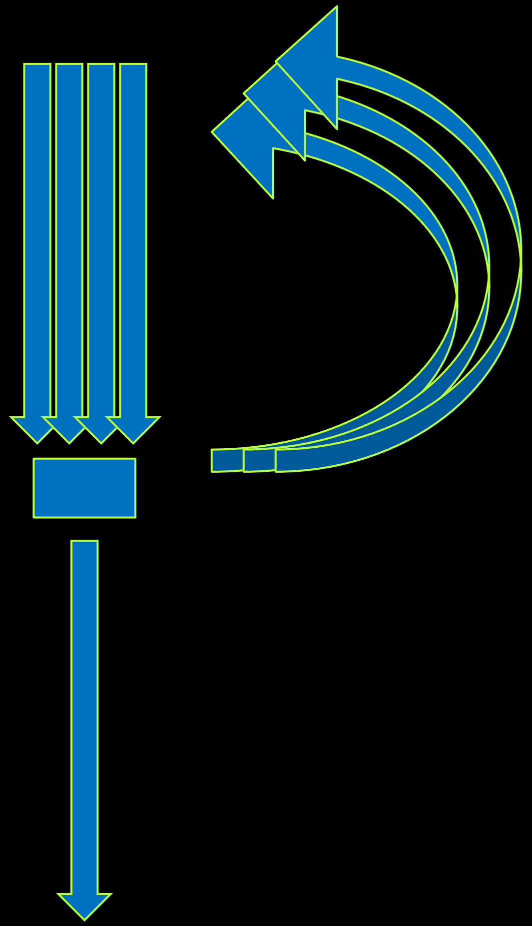
Locking Style of Programming

- All threads try to get the lock
- One does
  - Does its work
  - Releases the lock



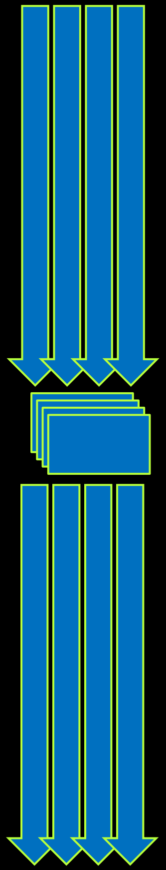
Lock-Free Style of Programming

- At least one thread always makes progress
- Try to write their result
  - On failure, repeat
- Usually atomicCAS
  - atomicExch, atomicAdd also used

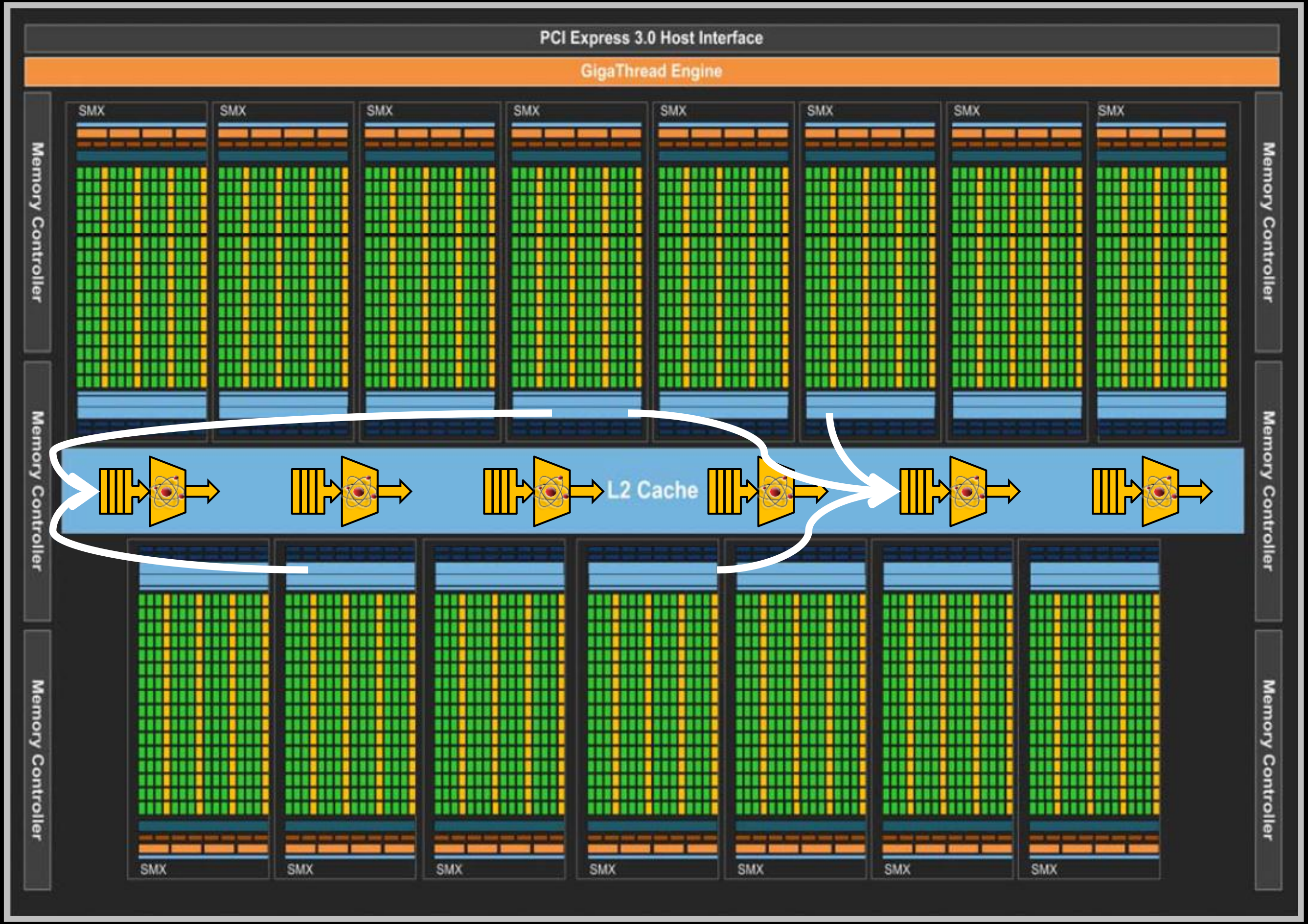


Wait-free Style of Programming

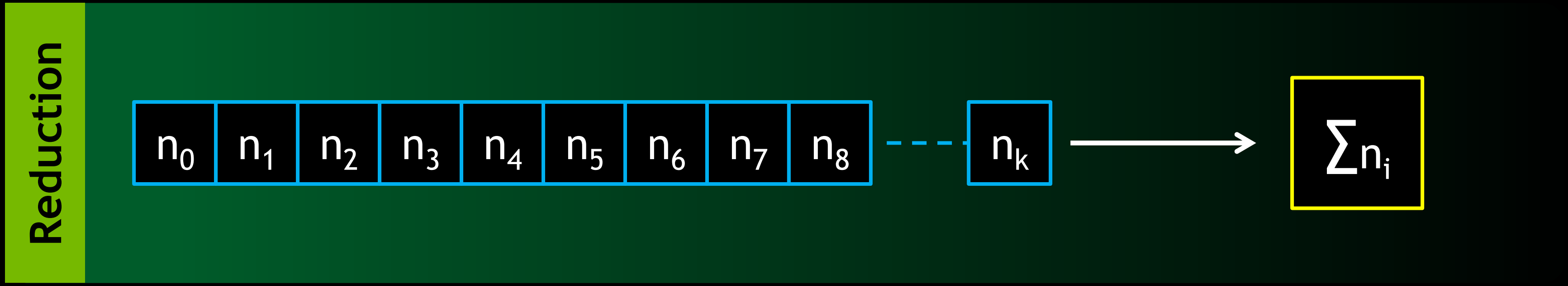
- All threads make progress
- Each updates memory atomically
- No thread blocked by other threads



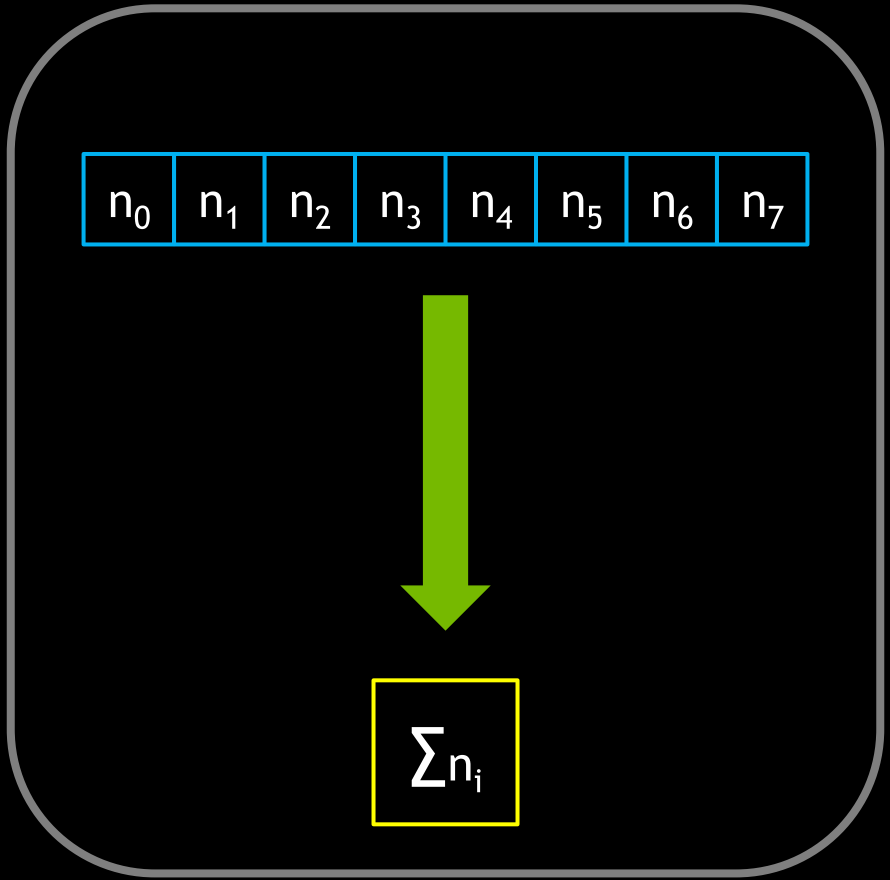
Hardware Managed Memory Update



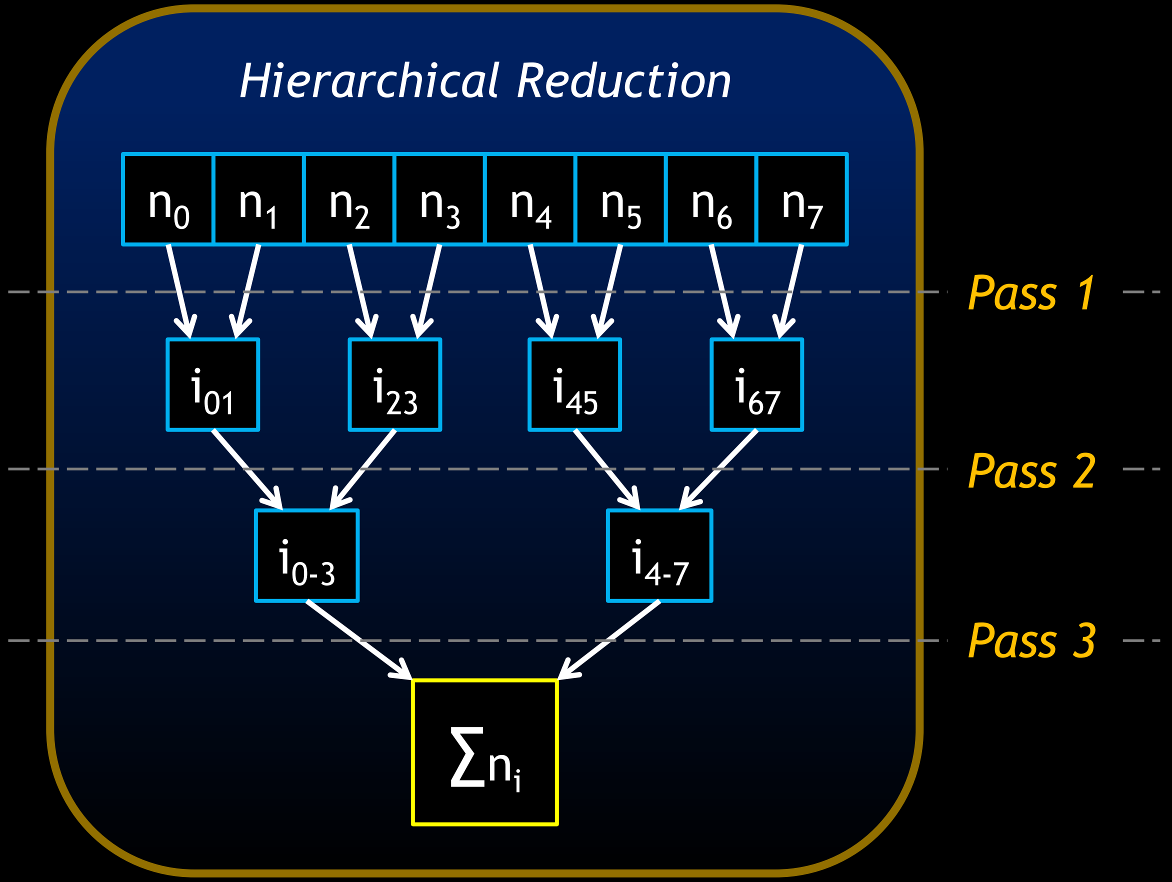
Atomic Arithmetical Operations



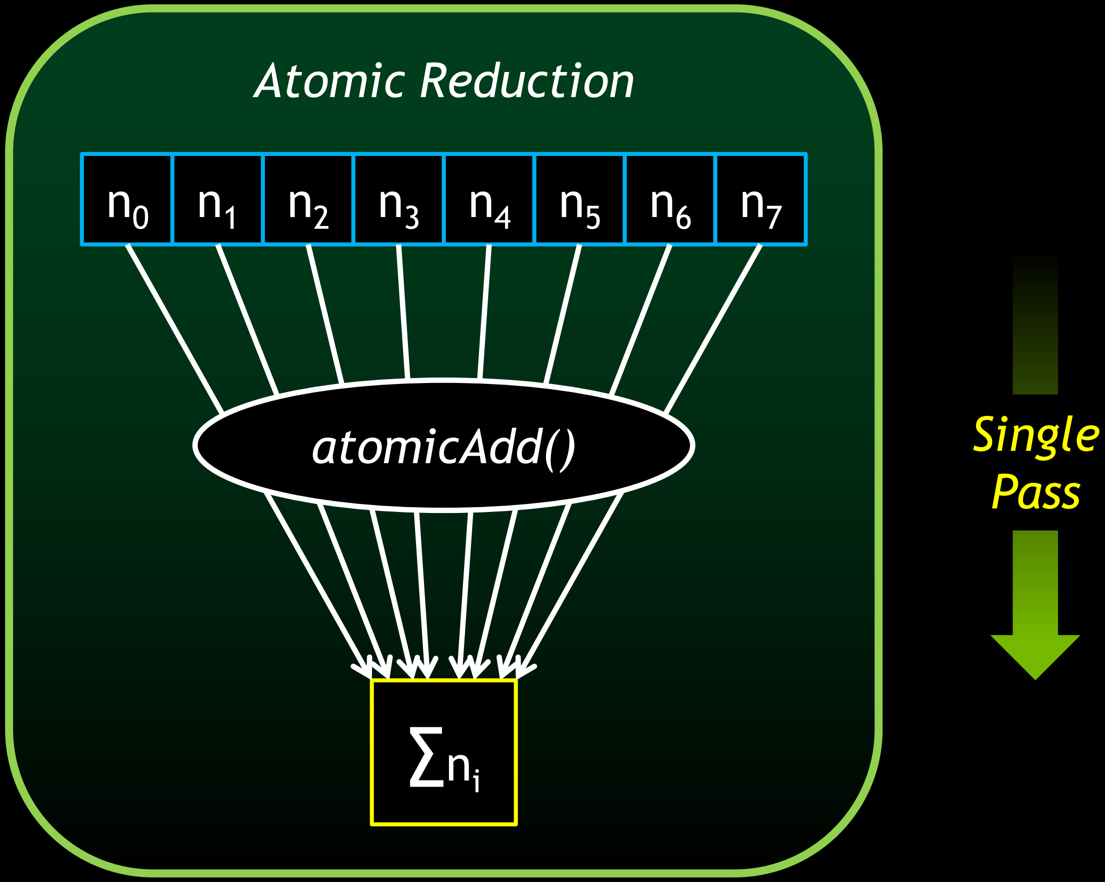
Atomic Arithmetical Operations



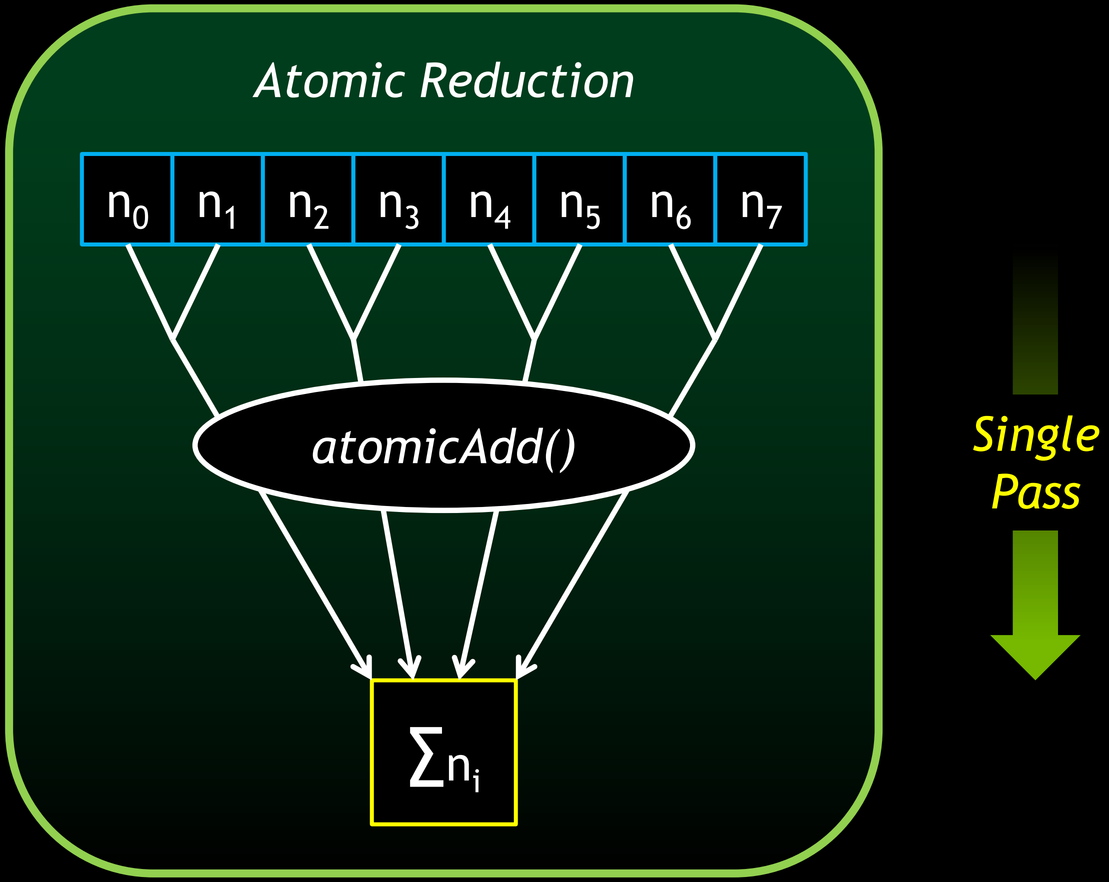
Atomic Arithmetical Operations



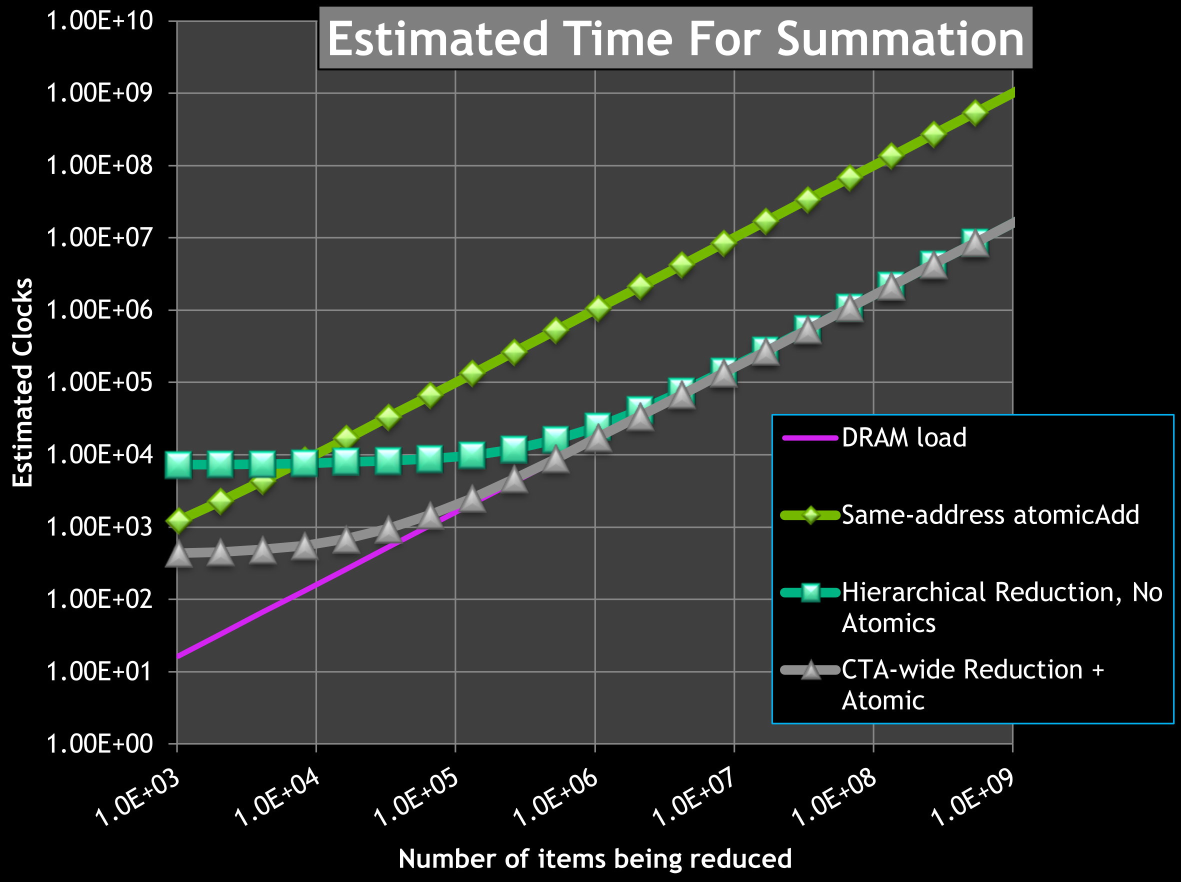
Atomic Arithmetical Operations

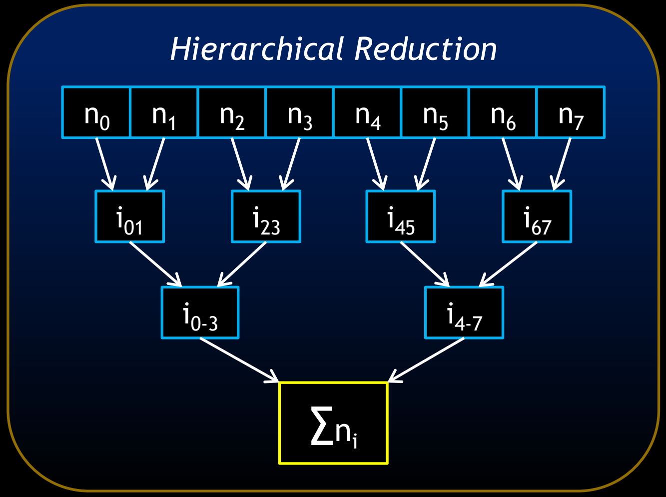


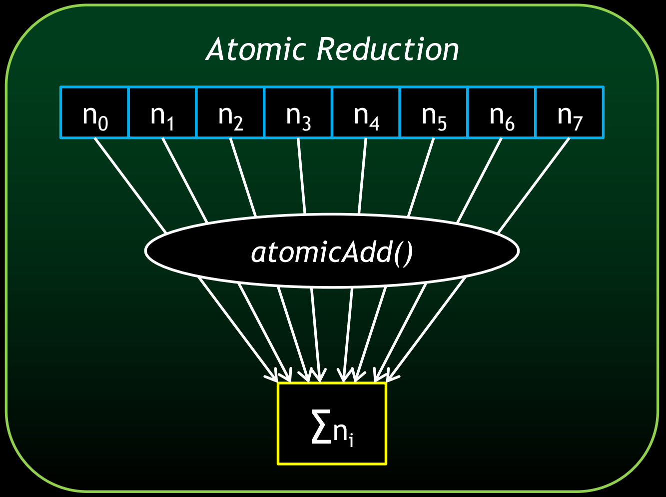
Atomic Arithmetical Operations



Atomic Arithmetical Operations

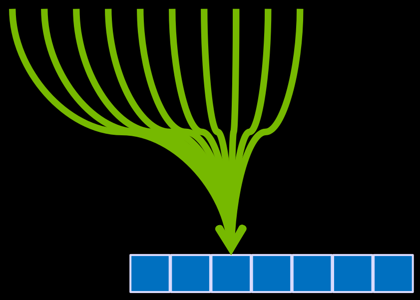






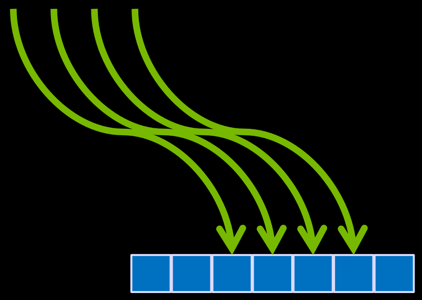
Atomic Access Patterns

Same Address



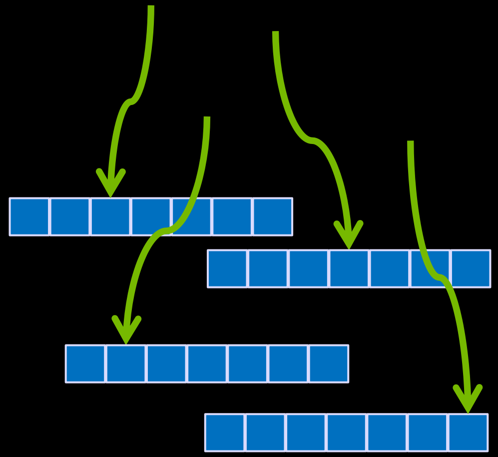
- 1 per clock

Same Cache Line



- Adjacent addresses
- Same issuing warp
- 8 per SM per clock

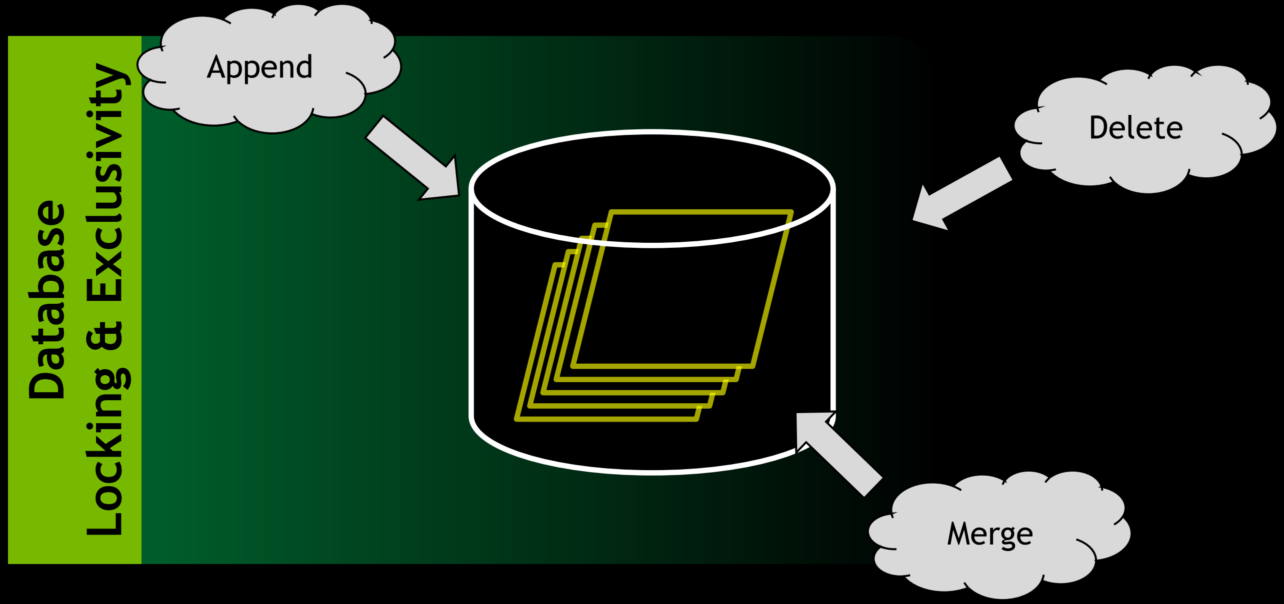
Scattered



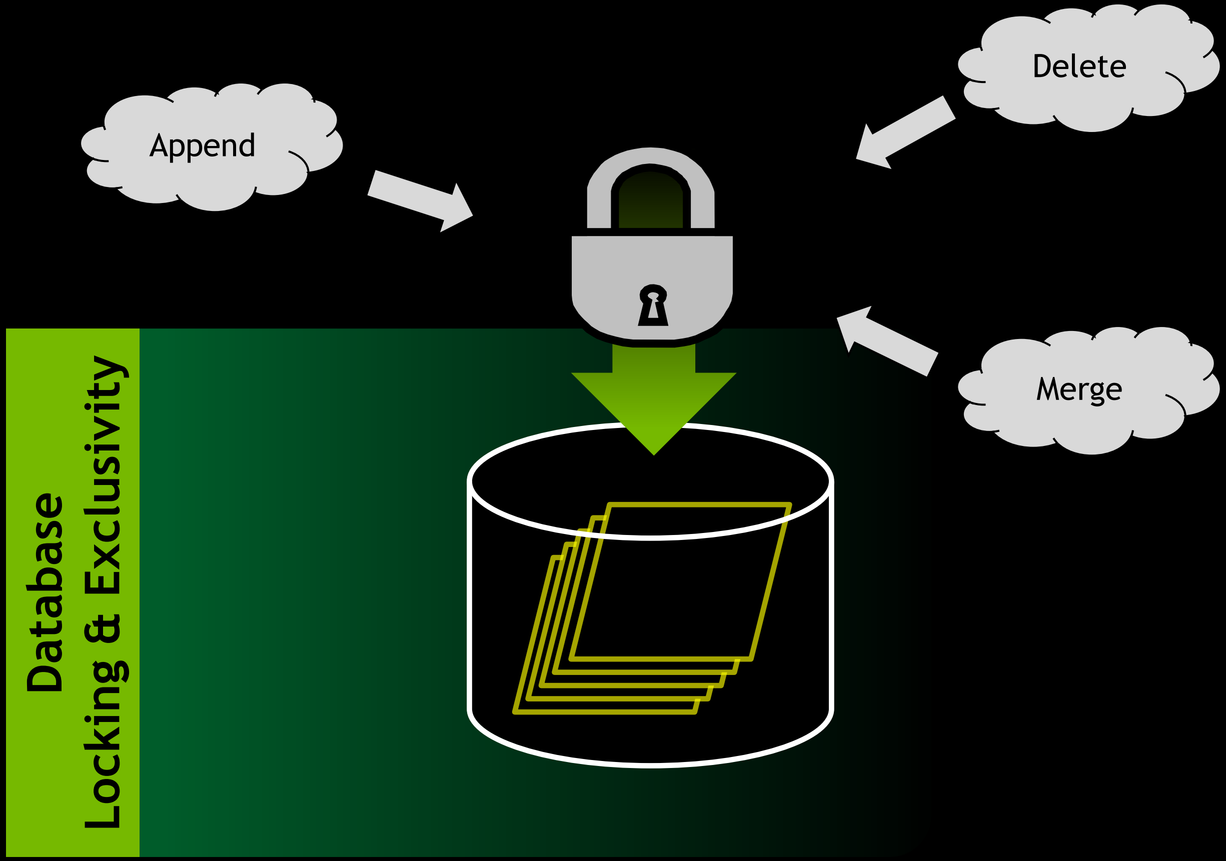
- Issued per cache-line
- 1 per SM per clock

Locks & Access Control

Locking guarantees exclusive access to data



Locks & Access Control



Locks & Access Control

Multi-threaded arithmetic

- Double precision addition
- Simple code is unsafe

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{


    double old = *data;
    *data = old + val;


    return old;
}
```

Locks & Access Control

Multi-threaded arithmetic

- Double precision addition
- Simple code is unsafe
- Add locks to protect critical section

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(try_lock() == false)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    unlock();

    return old;
}
```

Locks & Access Control

```
int locked = 0;
bool try_lock()
{
    if(locked == 0) {
        locked = 1;
        return true;
    }
    return false;
}
```

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(try_lock() == false)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    unlock();

    return old;
}
```

Locks & Access Control

```
int locked = 0;
bool try_lock()
{
    int prev = atomicExch(&locked, 1);
    if(prev == 0)
        return true;

    return false;
}
```

```
// Add “val” to “*data”. Return old value.
double atomicAdd(double *data, double val)
{
    while(try_lock() == false)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    unlock();

    return old;
}
```

`int atomicExch(int *data, int new)`

Atomically set (*data = new), and return the previous value

Locks & Access Control

Lock-based double precision atomicAdd()

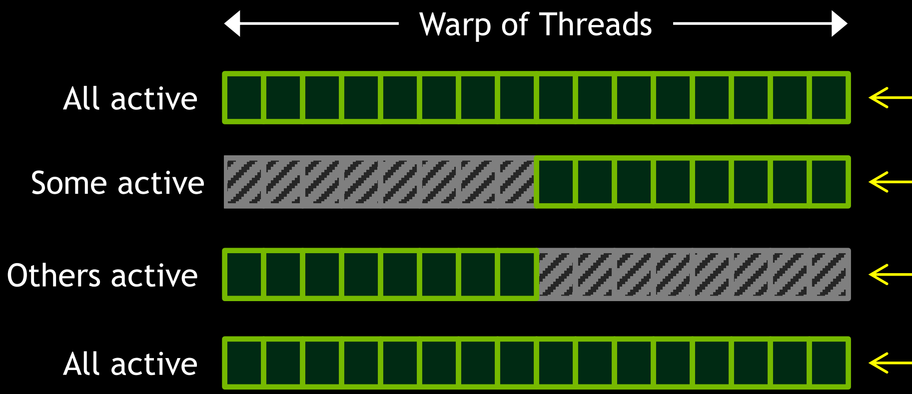
- But there's a problem…
- Don't use this code!

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;    // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

Locks & Warp Divergence

A CUDA *warp*:

- A group of threads (32 on current GPUs) scheduled in lock-step
- All threads execute the same line of code
- Any thread not participating is idle

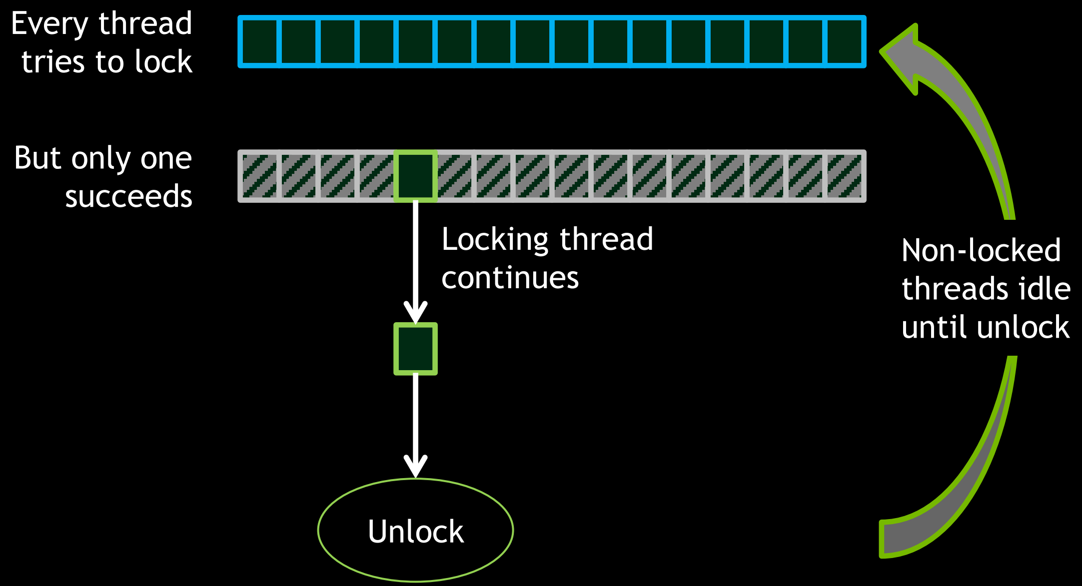



```
__device__ void example(bool condition)
{
    if(condition)

        run_this_first();
    else
        then_run_this();

    converged_again();
}
```

Locks & Warp Divergence

What does this mean for locks?

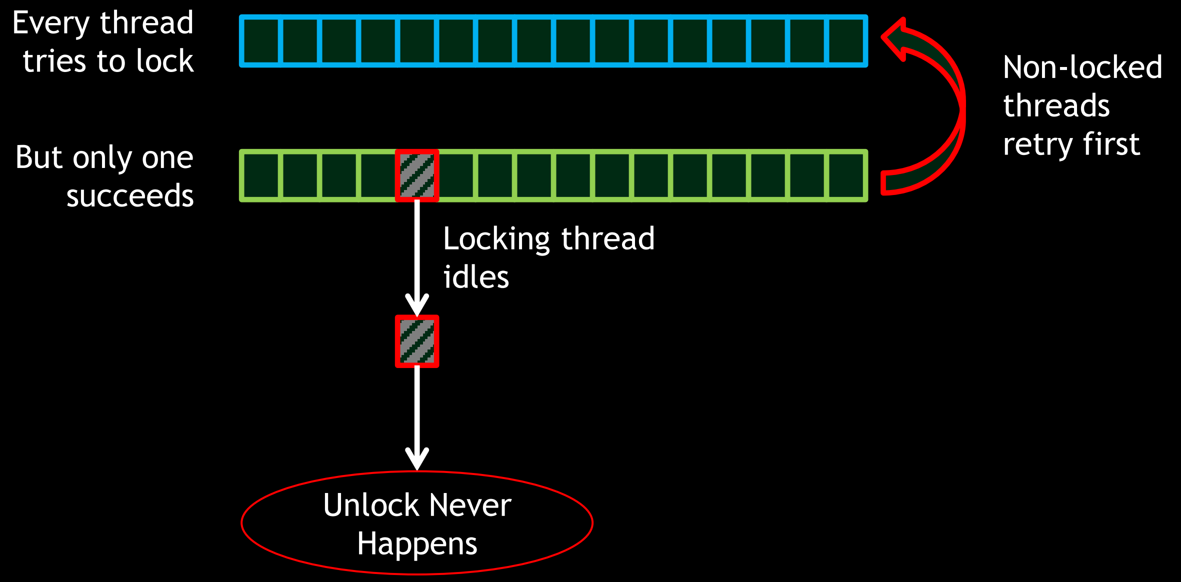
- Only one thread in the warp will lock
- We're okay so long as that's the thread which continues



Locks & Warp Divergence

What does this mean for locks?

- BUT: If the wrong thread idles, we deadlock
- No way to predict which threads idle



Locks & Warp Divergence

Working around divergence deadlock

1. Don’t use locks between threads in a warp
2. Elect one thread to take the lock, then iterate
3. Use a lock-free algorithm…

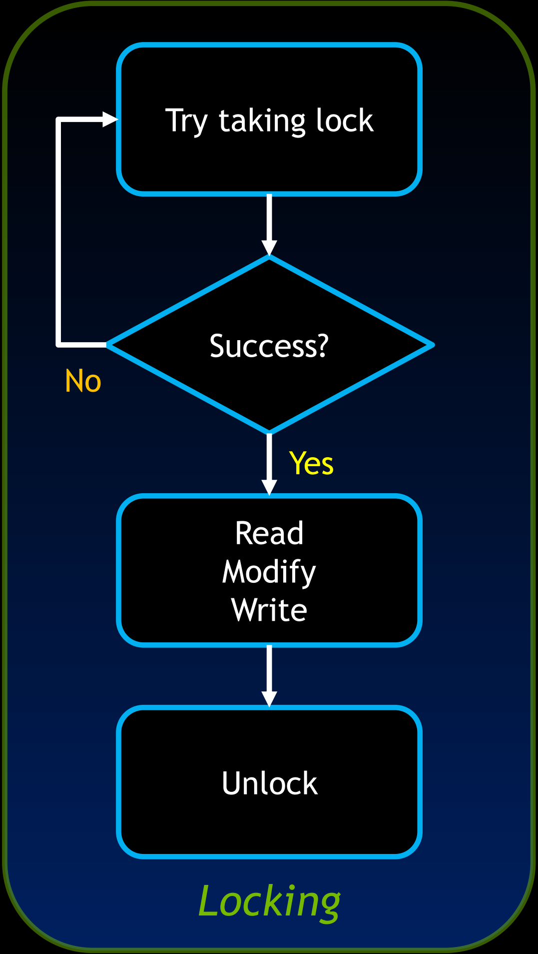
Lock Free Algorithms: Better Than Locks

Use atomic compare-and-swap to combine read, modify, write

- Under contention, exactly one thread is guaranteed to succeed
- High throughput - less work in critical section
- Only applies if transaction is a single operation

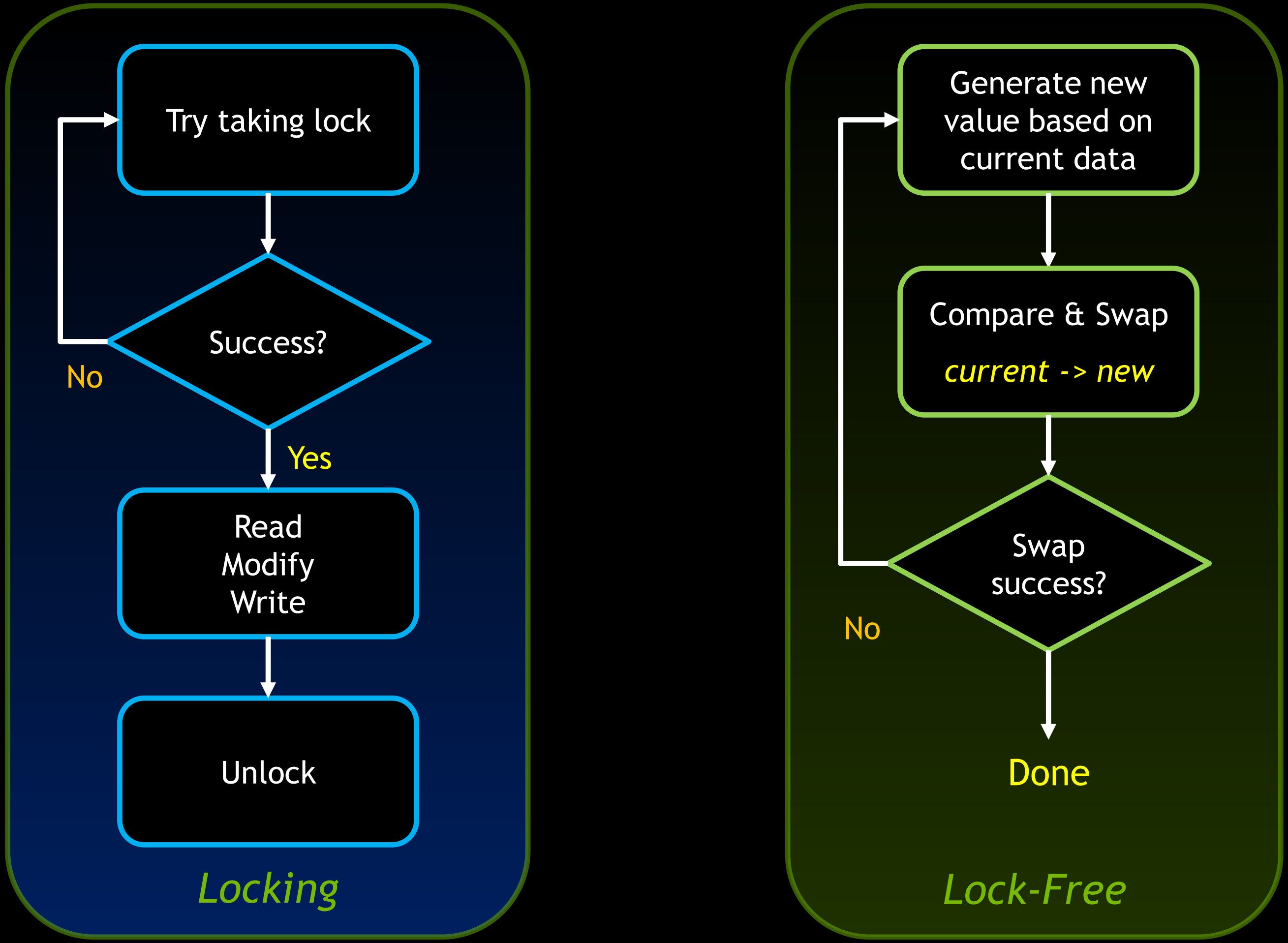
```
uint64 atomicCAS(uint64 *data, uint64 oldval, uint64 newval);
    If "*data" is equal to "oldval", replace it with "newval"
    Always returns original value of "*data"
```

Lock-Free Data Updates

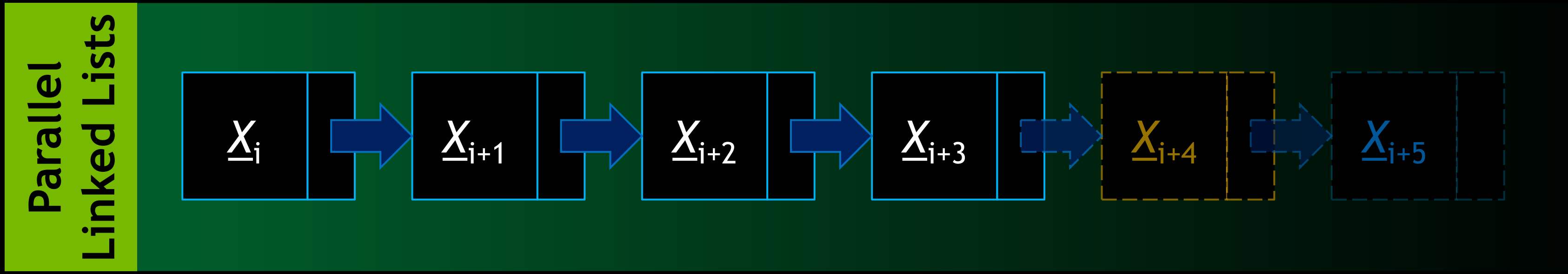



```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    while(atomicExch(&locked, 1) != 0)
        ;   // Retry lock

    double old = *data;
    *data = old + val;
    locked = 0;

    return old;
}
```

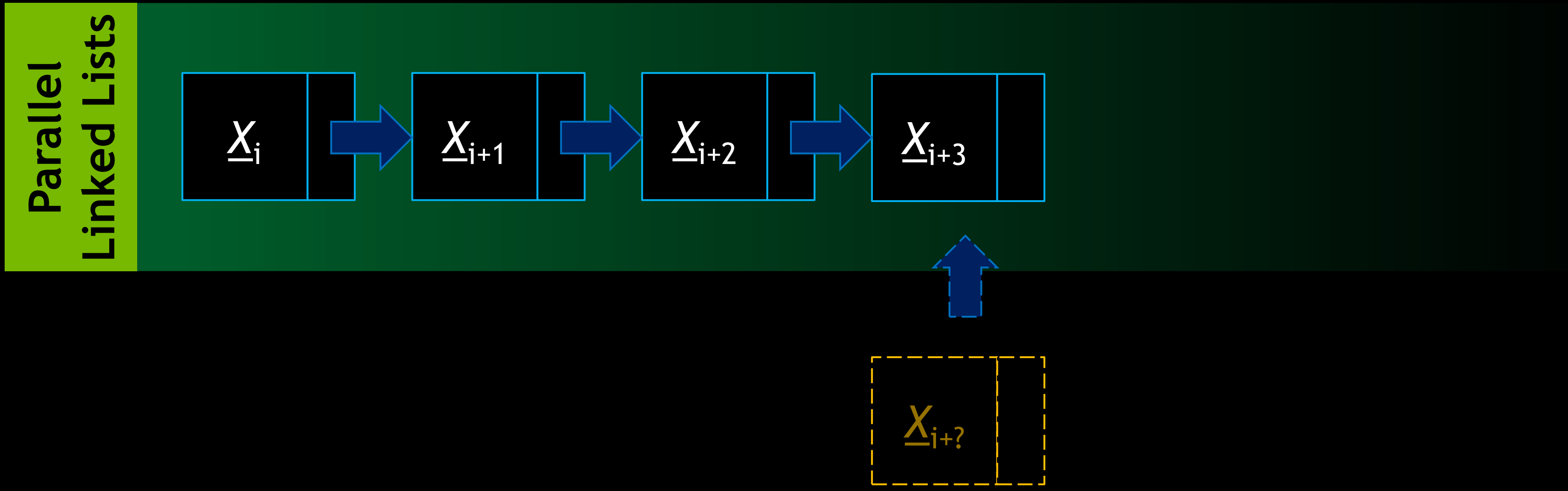
Lock-Free Data Updates



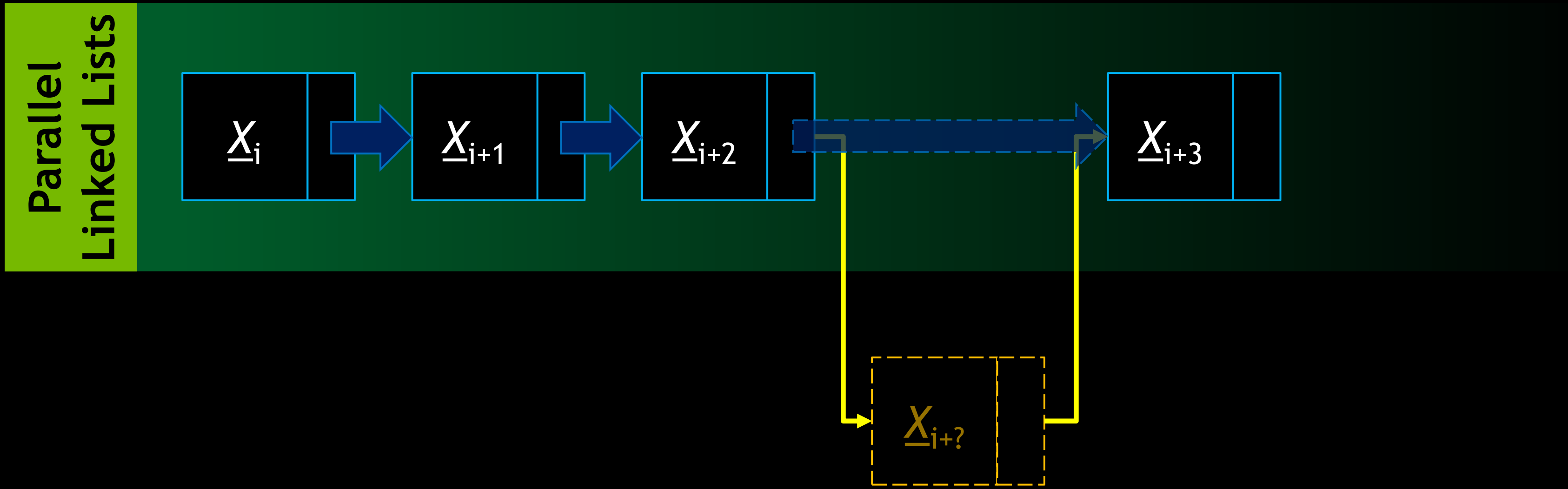
Lock-Free Parallel Data Structures



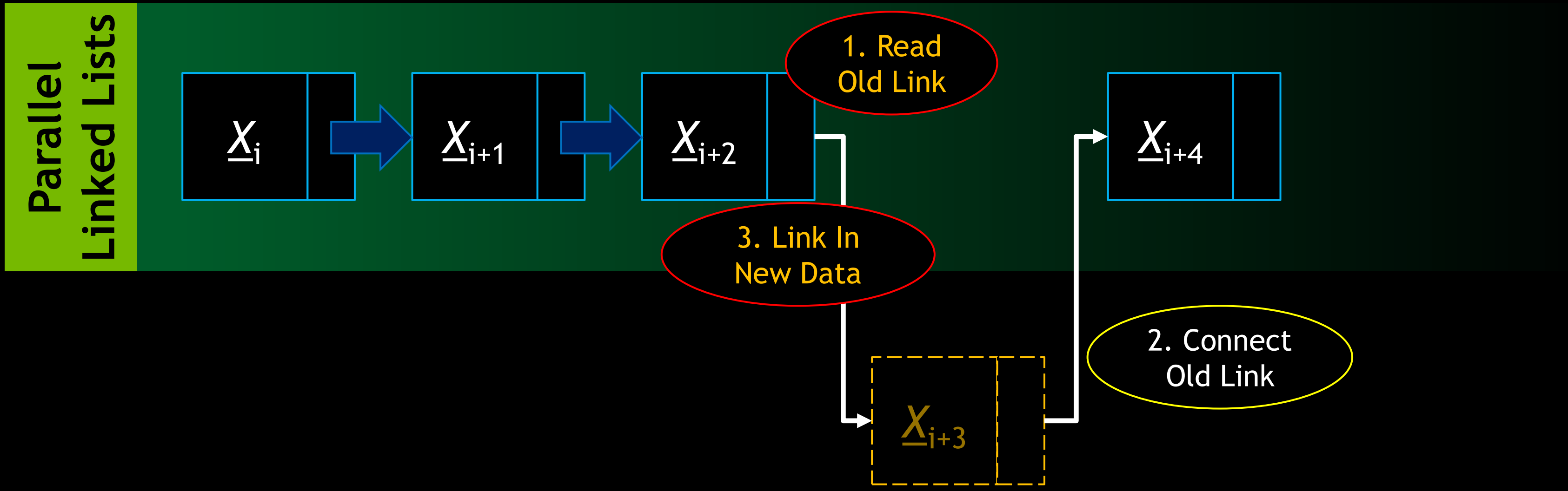
Lock-Free Parallel Data Structures



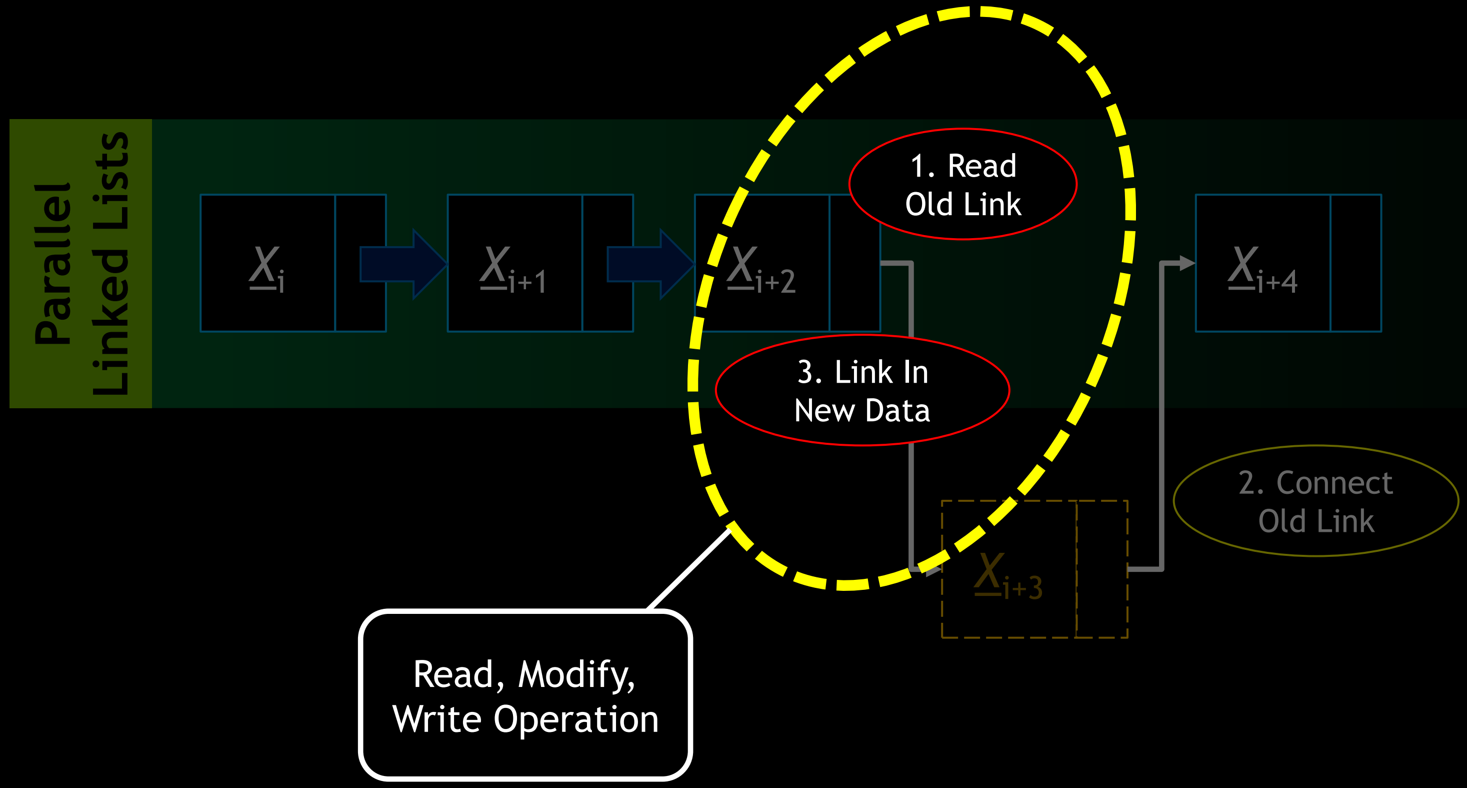
Lock-Free Parallel Data Structures



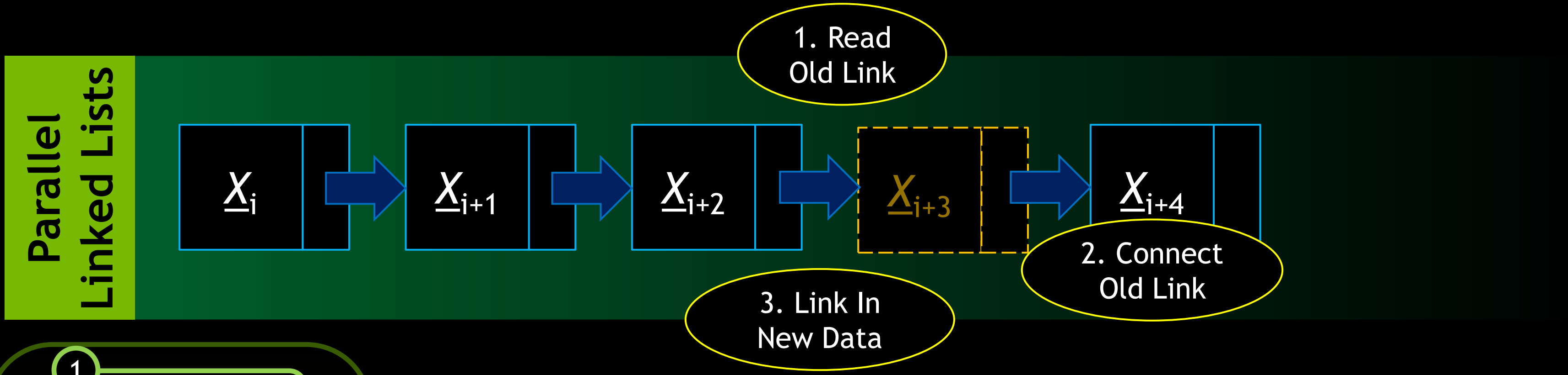
Lock-Free Parallel Data Structures

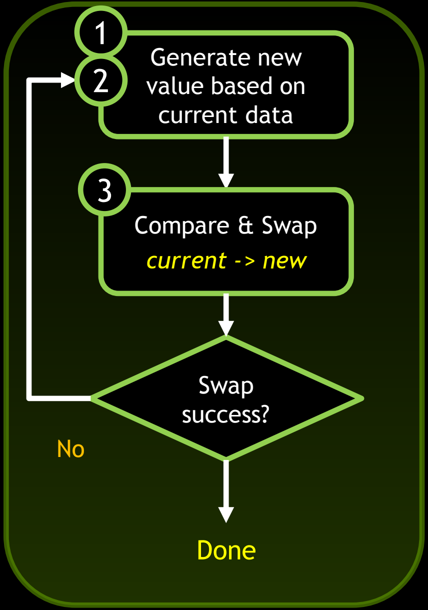


Lock-Free Parallel Data Structures



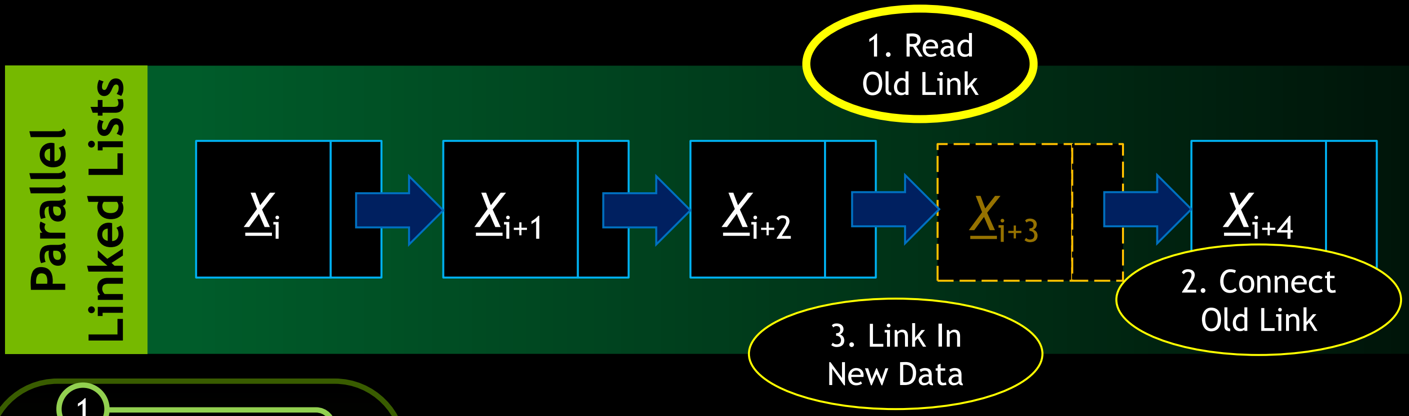
Lock-Free Parallel Data Structures

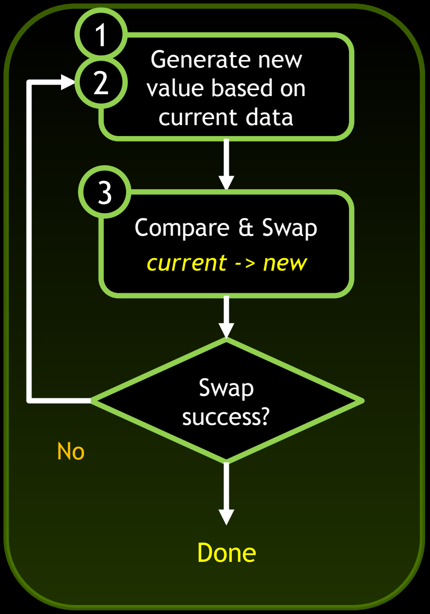





```
// Insert node “mine” after node “prev”
void insert(ListNode mine, ListNode prev)
{
    ListNode old, link = prev->next;
    do {
        old = link;
        mine->next = old;
        link = atomicCAS(&prev->next, link, mine);
    } while(link != old);
}
```

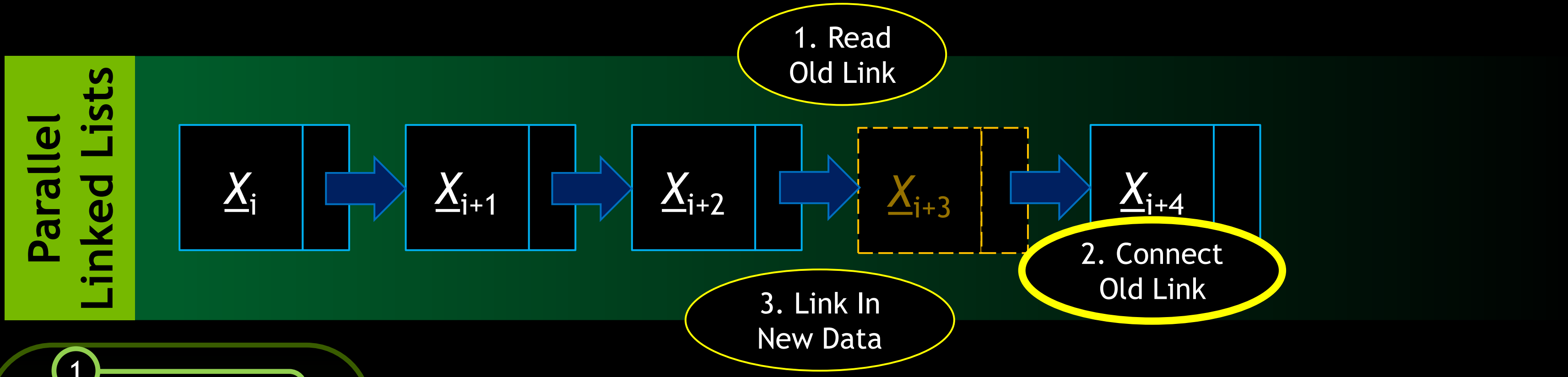
Lock-Free Parallel Data Structures

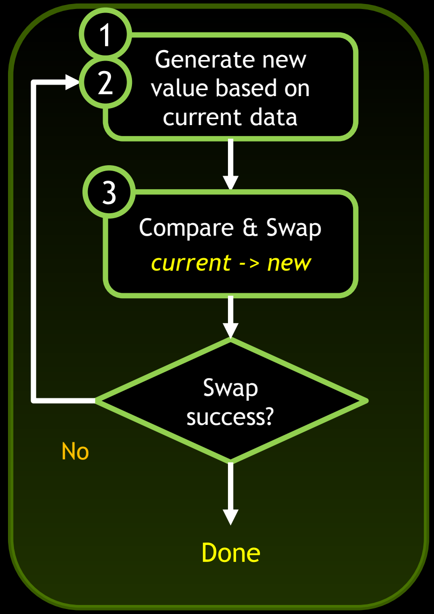





```
// Insert node “mine” after node “prev”
void insert(ListNode mine, ListNode prev)
{
    ListNode old, link = prev->next;
    do {
        old = link;
        mine->next = old;
        link = atomicCAS(&prev->next, link, mine);
    } while(link != old);
}
```

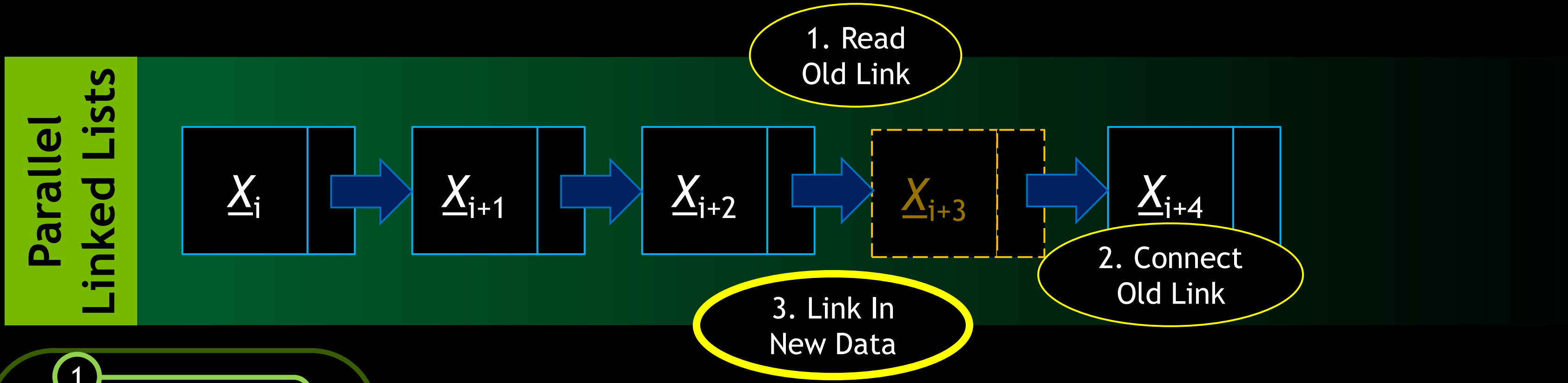
Lock-Free Parallel Data Structures

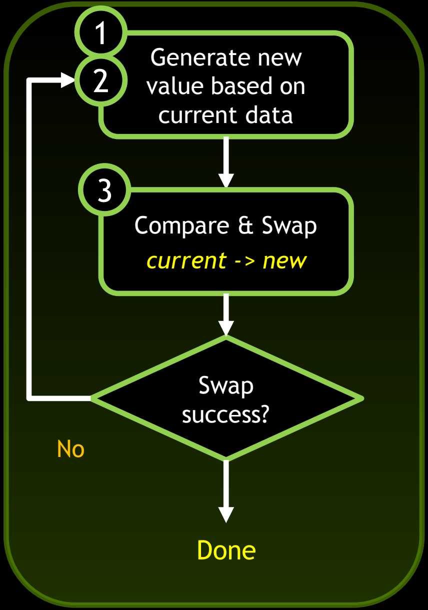





```
// Insert node “mine” after node “prev”
void insert(ListNode mine, ListNode prev)
{
    ListNode old, link = prev->next;   ①
    do {
        old = link;
        mine->next = old;   ②
        link = atomicCAS(&prev->next, link, mine);
    } while(link != old);
}
```

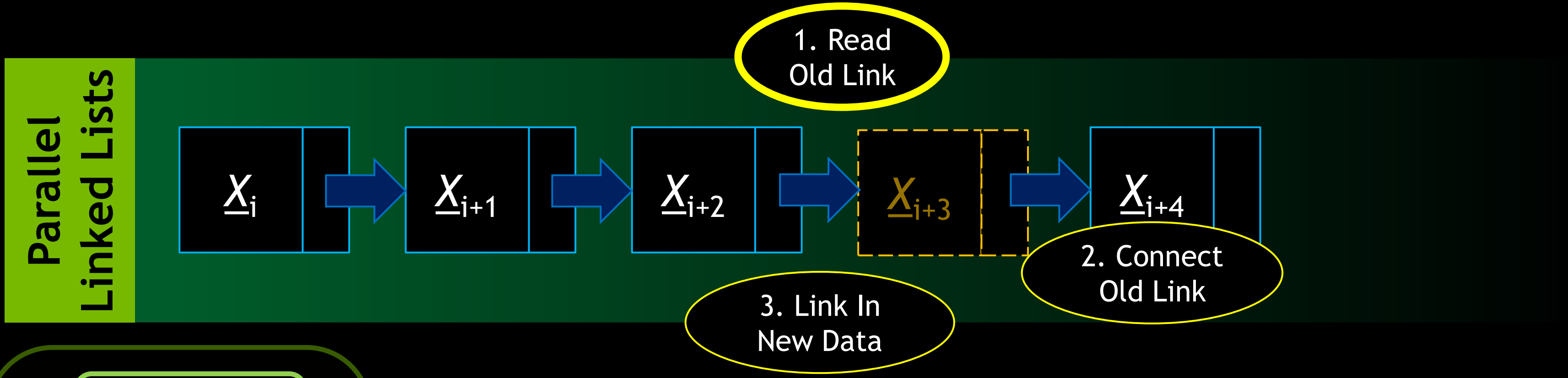
Lock-Free Parallel Data Structures

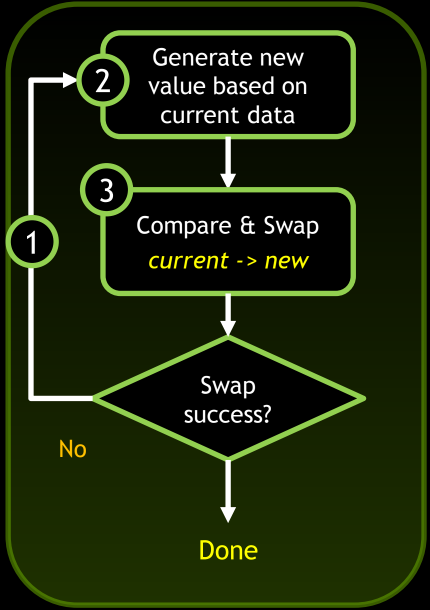





```
// Insert node "mine" after node "prev"
void insert(ListNode mine, ListNode prev)
{
    ListNode old, link = prev->next;      1
    do {
        old = link;
        mine->next = old;                 2
        link = atomicCAS(&prev->next, link, mine);   3
    } while(link != old);
}
```

Lock-Free Parallel Data Structures

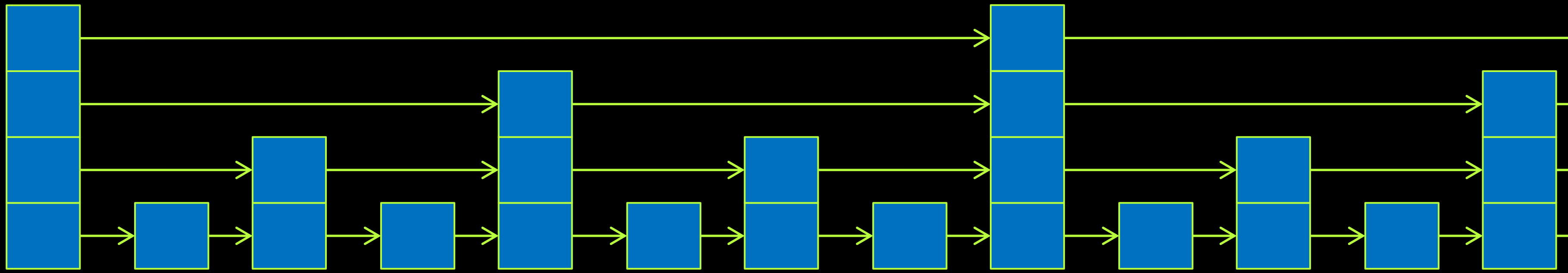





```
// Insert node "mine" after node "prev"
void insert(ListNode mine, ListNode prev)
{
    ListNode old, link = prev->next;
    do {
        old = link;
        mine->next = old;
        link = atomicCAS(&prev->next, link, mine);
    } while(link != old);
}
```

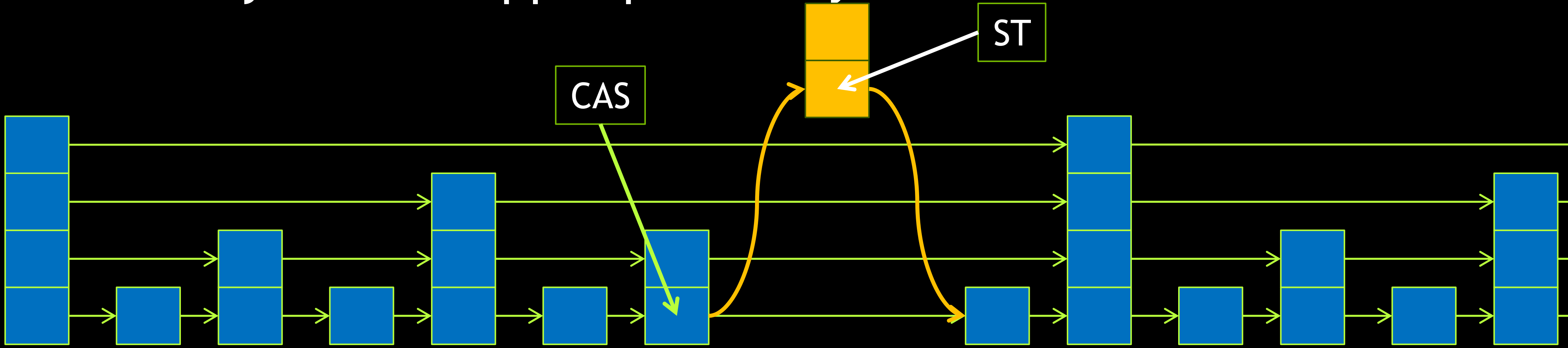
Worked Example: Skiplists & Sorting (LN)

- Skiplists – hierarchical linked lists, ordered
  - O(log n) lookup, insertion, deletion
  - Self-balancing with high probability
  - Concurrent operations well-defined, relies on atomic-CAS
- Sorting strategy
  - Use *p* threads to concurrently insert *n* items into a single skiplist



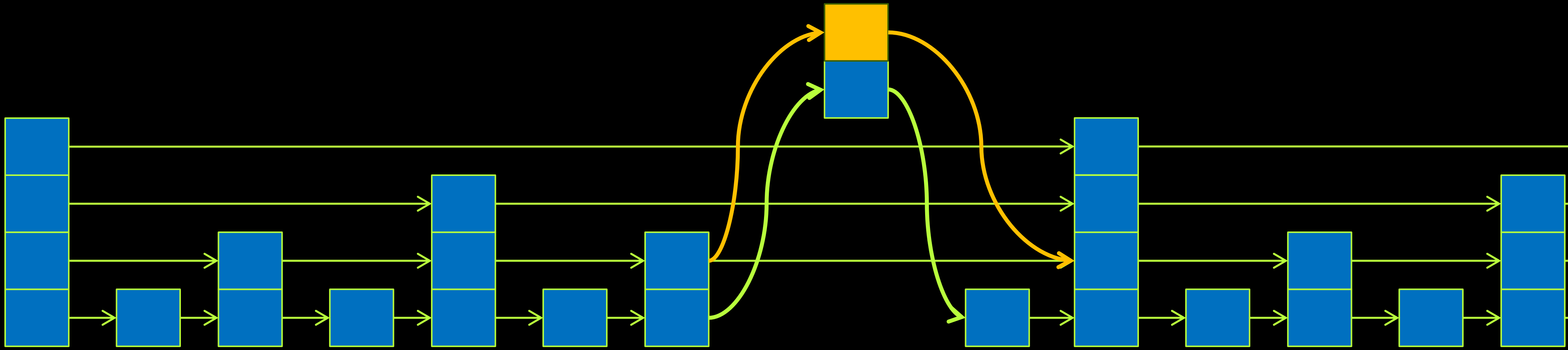
Skiplist insertion - bottom level

- Set *next* on new node, using ordinary STore
- Swing *prev* from existing node to new node with CAS
  - As long as it still points to the same node…
- Skiplist stays legal at all times
- Nobody can see upper pointers yet



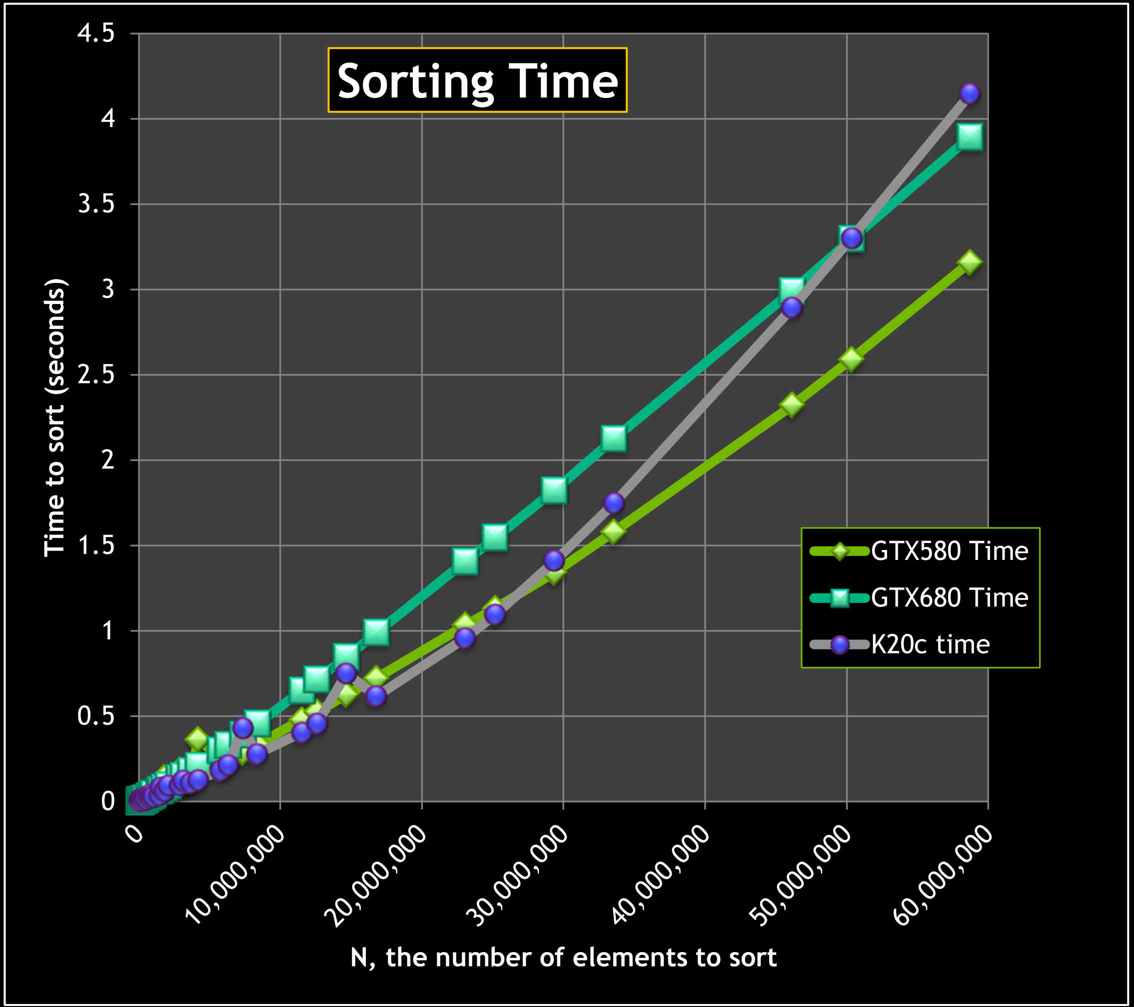
Skiplist insertion - upper levels

- Move up one level; repeat (find, point, swing)
- Lots could have changed
  - But as long as the pointers are the same when you try to point to the new node (with CAS), then all is well



Skiplist Sorting Observations

- Collisions high at first
  - but skiplist doubles in length every *iteration*
- Collisions diminish rapidly as N >> p
- Performance dominated by loads, not atomics
  - O(n log n) loads
  - O(n) atomics
- Insertion sort = $O(n^2)$ ops



Conclusions

- Atomics allow the creation of much more sophisticated algorithms that have higher performance
- GPU has parallel hardware to execute atomics
- AtomicCAS can be used to mimic any coordination primitive
- Atomics force serialization
  - don’t ask for serialization when you don’t need it
  - or, perform concurrent reductions when possible

Thankyou!

Extra Slides

Safe Ways to Lock - none are pretty

Serialise per-warp

```
__global__ void useLock()
{
    int tid = threadIdx.x % warpSize;

    // Perform warp operation by
    // one thread only
    if(tid == 0)
        lock();

    for(int i=0; i<warpSize; i++) {
        if(tid == i)
            do_stuff();
    }

    if(tid == 0)
        unlock();
}
```

Lock per-thread

```
__global__ void useLock()
{
    int done = 0;
    while(!done)
    {
        // Returns "true" for only
        // one active thread in warp
        if(elect_one_thread()) {
            lock();
            do_stuff();
            unlock();
            done = 1;
        }
    }
}
```

Both of these require knowledge of warp execution

Lock-Free Data Updates

```
// Add "val" to "*data". Return old value.
double atomicAdd(double *data, double val)
{
    double old, newval, curr = *data;
    do {
        // Generate new value from current data
        old = curr;
        newval = curr + val;

        // Attempt to swap old <-> new.
        curr = atomicCAS(data, old, newval);

    // Repeat if value has changed in the meantime.
    } while(curr != old);

    return ret;
}
```