# A GPU Multiversion B-Tree

Muhammad A. Awad*, Serban D. Porumbescu, and John D. Owens

*Currently at AMD Research

AMD

UC DAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Outline

1. Fully concurrent GPU data structure example
2. Challenges with GPU data structures (and how we address them)
3. Results
4. Summary and future research directions

# Data analytics on GPUs

blazingSQL

HEAVY.AI

ARESDB

RAPIDS

VOLTRON DATA

# Can we write code like this?

```
data_structure_0 edges;
data_structure_1 graph;

std::for_each(std::execution::par,  // in parallel
                std::begin(edges),  // from the first element
                std::end(edges),    // to the last element
                [&](const auto e) { // perform this lambda function
                    graph.insert(e);
                    auto neighbors = graph.get_neighbors(e.first);
               });
```

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Can we write code like this on the GPU?

```
data_structure_0 edges;
data_structure_1 graph;

std::for_each(std::execution::par,  // in parallel
                std::begin(edges),  // from the first element
                std::end(edges),    // to the last element
                [&](const auto e) { // perform this lambda function
                    graph.insert(e);
                    auto neighbors = graph.get_neighbors(e.first);
               });
```

# Can we write code like this on the GPU?

```
data_structure_0 edges;
data_structure_1 graph;

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),  // from the first element
              std::end(edges),    // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
});
```

**Fully concurrent GPU data structure**

Support for **concurrent** read-only and update operations

My dissertation topic

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),  // from the first element
              std::end(edges),    // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
                  for(n in neighbors) printf("%i, ", n);
              });
```

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

Thread$_0$ Thread$_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
                std::begin(edges),  // from the first element
                std::end(edges),    // to the last element
                [&](const auto e) { // perform this lambda function
                    graph.insert(e);
                    auto neighbors = graph.get_neighbors(e.first);
                    for(n in neighbors) printf("%i, ", n);
                });
```

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

Thread$_0$ Thread$_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),  // from the first element
              std::end(edges),    // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
                  for(n in neighbors) printf("%i, ", n);
              });
```

Execution order

Thread$_0$ ← Success

Thread$_1$ ← Success

Thread$_0$ ← (a)

Thread$_1$ ← (a, b)

# What will be the output on the GPU?

Thread$_0$ Thread$_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),  // from the first element
              std::end(edges),    // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
                  for(n in neighbors) printf("%i, ", n);
              });
```

Execution order

Thread$_0$ ← Success

Thread$_1$ ← Success

Thread$_0$ ← (a, b)

Thread$_1$ ← (b)

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

$Thread_0$ $Thread_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
                std::begin(edges),  // from the first element
                std::end(edges),    // to the last element
                [&](const auto e) { // perform this lambda function
                    graph.insert(e);
                    auto neighbors = graph.get_neighbors(e.first);
                    for(n in neighbors) printf("%i, ", n);
                });
```

Execution order

$Thread_0$ ← Success

$Thread_1$ ← Success

$Thread_0$ ← (a)

$Thread_1$ ← (b)

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

Thread$_0$ Thread$_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),    // from the first element
              std::end(edges),      // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
                  for(n in neighbors) printf("%i, ", n);
              });
```

Execution order

Thread$_0$ ← Success

Thread$_1$ ← Success

Thread$_0$ ← (a)

Thread$_1$ ← (b)

# What will be the output on the GPU?

$\text{Thread}_0$ $\text{Thread}_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),  // from the first element
              std::end(edges),    // to the last element
              [&](const auto e) { // perform this lambda function
                  graph.insert(e);
                  auto neighbors = graph.get_neighbors(e.first);
                  for(n in neighbors) printf("%i, ", n);
              });
```

Execution order

$\text{Thread}_0$ ← Success

$\text{Thread}_1$ ← Success

$\text{Thread}_0$ ← (a)

$\text{Thread}_1$ ← (b)

**Our paper addresses the semantics of concurrent operations on GPUs using *snapshots***

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# What will be the output on the GPU?

Thread$_0$ Thread$_1$

```
data_structure_0 edges{{0,'a'}, {0,'b'}};
data_structure_1 graph{};

std::for_each(std::execution::par,  // in parallel
              std::begin(edges),    // from the first element
              std::end(edges),      // to the last element
              [&](const auto e) {   // perform this lambda function
                  graph.insert(e);
                  auto timestamp = graph.take_snapshot();
                  auto neighbors = graph.get_neighbors(e.first, timestamp);
                  for(n in neighbors) printf("%i, ", n);
              });
```

Execution order

Thread$_0$ ← Success

Thread$_1$ ← Success

Thread$_0$ ← (a,b)

Thread$_1$ ← (b)

😊

**Our paper addresses the semantics of concurrent operations on GPUs using *snapshots***

UC DAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

# Challenges with GPU data structures

1. Memory access patterns during traversal
    - Data structure layout in memory

# Challenges with GPU data structures

1. Memory access patterns during traversal
   - Data structure layout in memory
2. Contention during updates
   - Atomics and locks

# Challenges with GPU data structures

1. Memory access patterns during traversal
    - Data structure layout in memory
2. Contention during updates
    - Atomics and locks
3. Memory allocation and reclamation
    - When can we free a pointer?

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

1. Memory access patterns during traversal
   - Data structure layout in memory
2. Contention during updates
   - Atomics and locks
3. Memory allocation and reclamation
   - When can we free a pointer?
4. Abstractions
   - Defining APIs for GPU data structures

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

1. Memory access patterns during traversal
    - Data structure layout in memory
2. Contention during updates
    - Atomics and locks
3. Memory allocation and reclamation
    - When can we free a pointer?
4. Abstractions
    - Defining APIs for GPU data structures
5. Semantics
    - Meaningful results of concurrent operations

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

1. Memory access patterns during traversal
   - Data structure layout in memory
2. Contention during updates
   - Atomics and locks
3. Memory allocation and reclamation
   - When can we free a pointer?
4. Abstractions
   - Defining APIs for GPU data structures
5. Semantics
   - Meaningful results of concurrent operations

# Challenges with GPU data structures

1. Data structure traversal
    - Fundamental operation in updates and queries

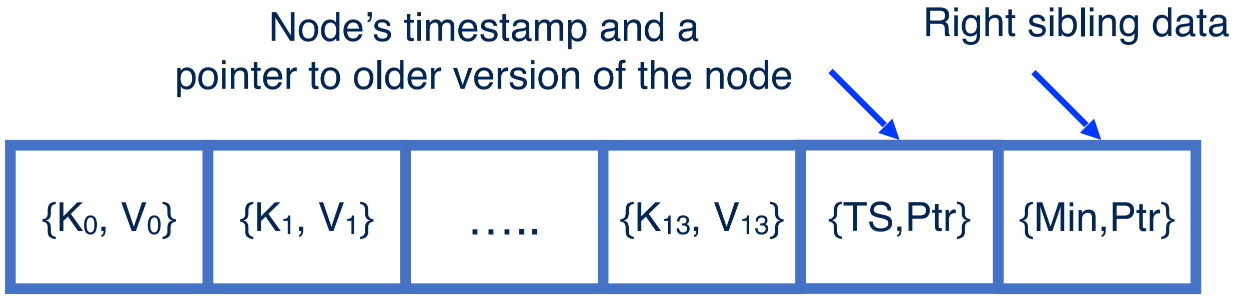
# Challenges with GPU data structures

1. Data structure traversal
    - Fundamental operation in updates and queries
        - Memory access patterns
        - Branch divergence

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

1. Data structure traversal
   - Fundamental operation in updates and queries
     - Memory access patterns
     - Branch divergence




**Multiversion B-Tree node**

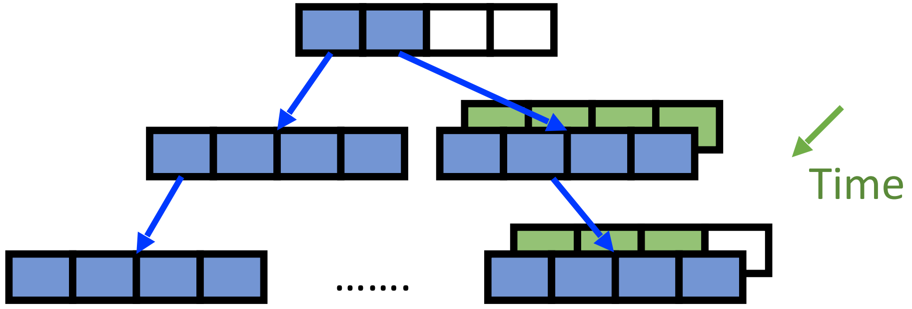
Node size = 128 bytes or B = 14
(assuming 32-bit key-value (or pivot-pointer) pairs)

**Our solution: use cache-line-sized nodes and WCWS**

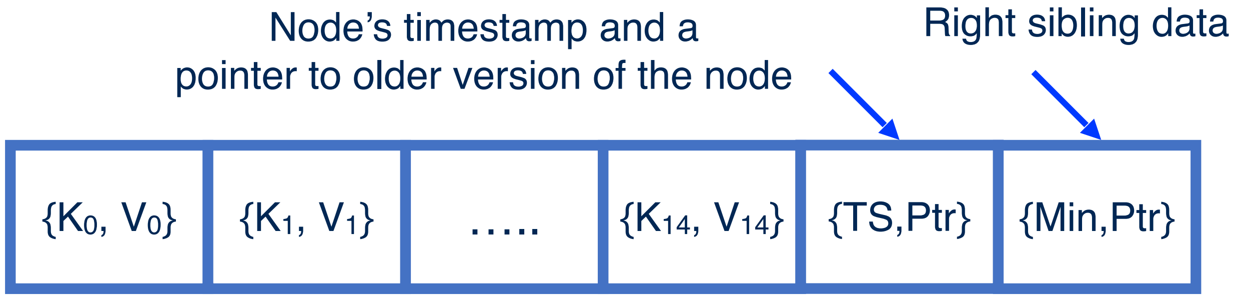
UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

1. Data structure traversal
   - Fundamental operation in updates and queries
     - Memory access patterns
     - Branch divergence




**Multiversion B-Tree**




**Multiversion B-Tree node**

Node size = 128 bytes or B = 14
(assuming 32-bit key-value (or pivot-pointer) pairs)

**Our solution: use cache-line-sized nodes and WCWS**

# Challenges with GPU data structures

2. Contention during updates

# Challenges with GPU data structures

2. Contention during updates
    - Relaxing the data structure invariants

# Challenges with GPU data structures

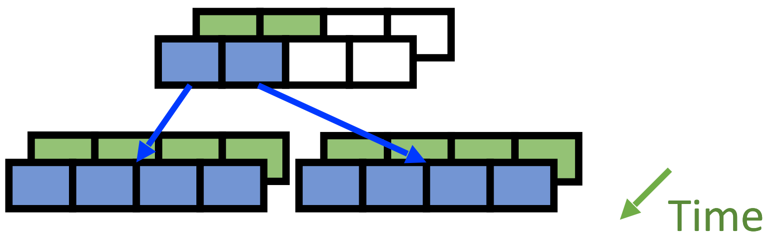
2. Contention during updates
    - Relaxing the data structure invariants
    - Restarts (aborts)
        - Instead of spin locks

# Challenges with GPU data structures

2. Contention during updates
   - Relaxing the data structure invariants
   - Restarts (aborts)
     - Instead of spin locks
   - Proactively perform operations (e.g., splitting)

# Challenges with GPU data structures

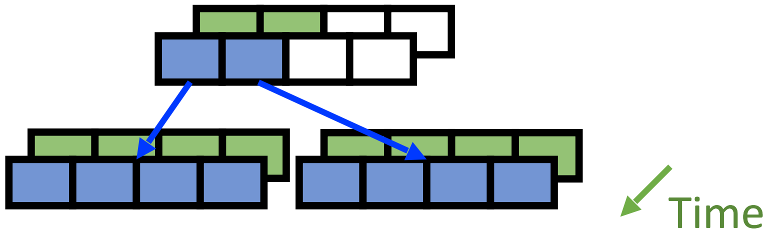
2. Contention during updates
    - Relaxing the data structure invariants



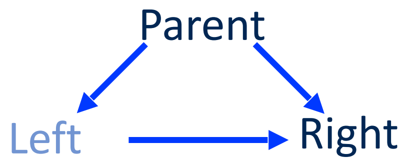

Multiversion B-Tree

# Challenges with GPU data structures

2. Contention during updates
    - Relaxing the data structure invariants

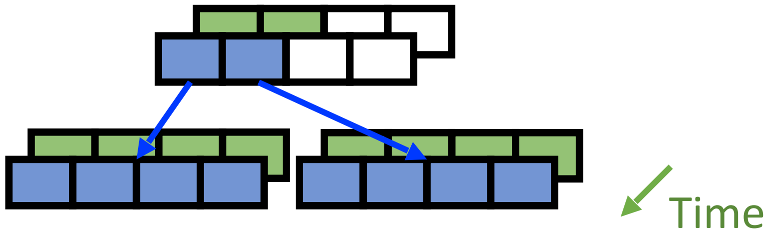



Multiversion B-Tree

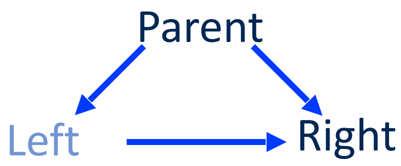


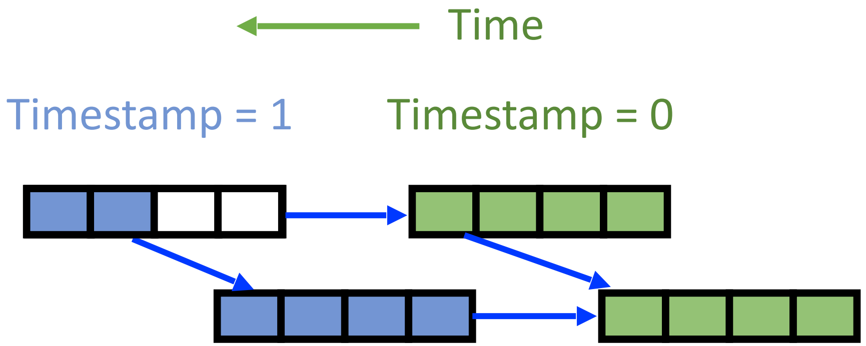
# Challenges with GPU data structures

2. Contention during updates
   - Relaxing the data structure invariants




Multiversion B-Tree



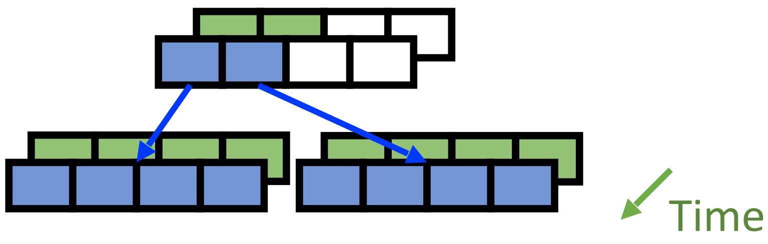




UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

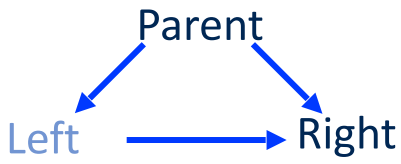
# Challenges with GPU data structures

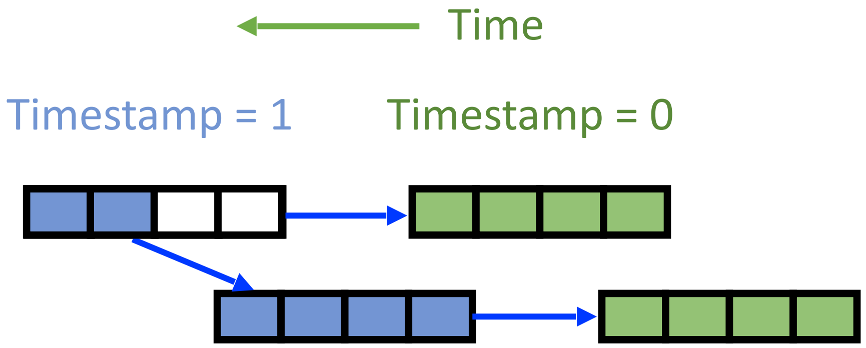
## 2. Contention during updates

- Relaxing the data structure invariants




Multiversion B-Tree




**Our solution: use single entry-point version lists**

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

2. Contention during updates
   - Relaxing the data structure invariants
   - Restarts (aborts)
     - Instead of spin locks
   - Proactively perform operations (e.g., splitting)

**Our solution: use single entry-point version lists, use restarts and proactive splitting**

# Challenges with GPU data structures

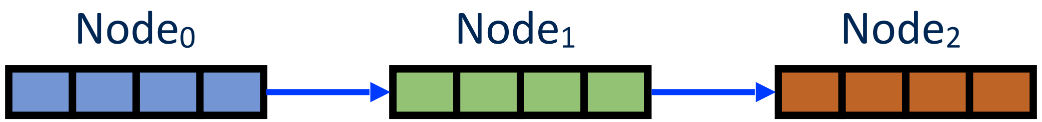
3. Memory reclamation

# Challenges with GPU data structures

3. Memory reclamation
    - When can we free a pointer?

# Challenges with GPU data structures

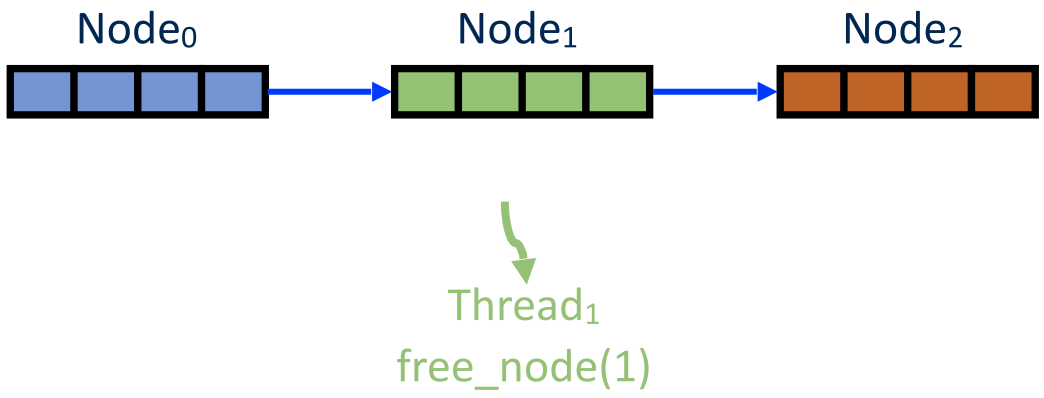
3. Memory reclamation
    - When can we free a pointer?




UC DAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

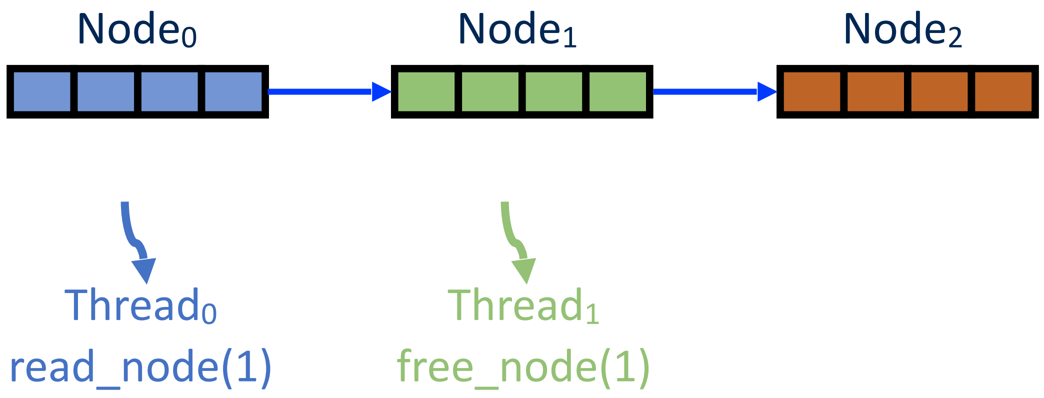
3. Memory reclamation
   - When can we free a pointer?



# Challenges with GPU data structures

## 3. Memory reclamation

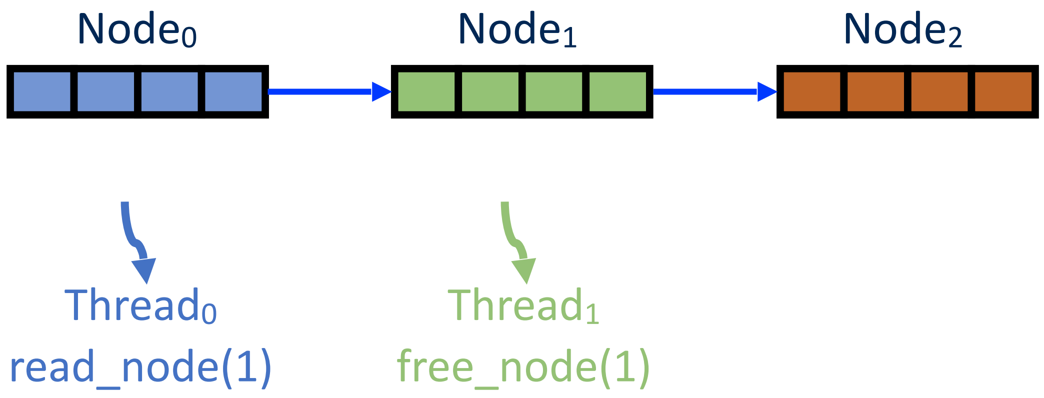
- When can we free a pointer?



# Challenges with GPU data structures

## 3. Memory reclamation

- When can we free a pointer?




Data structure

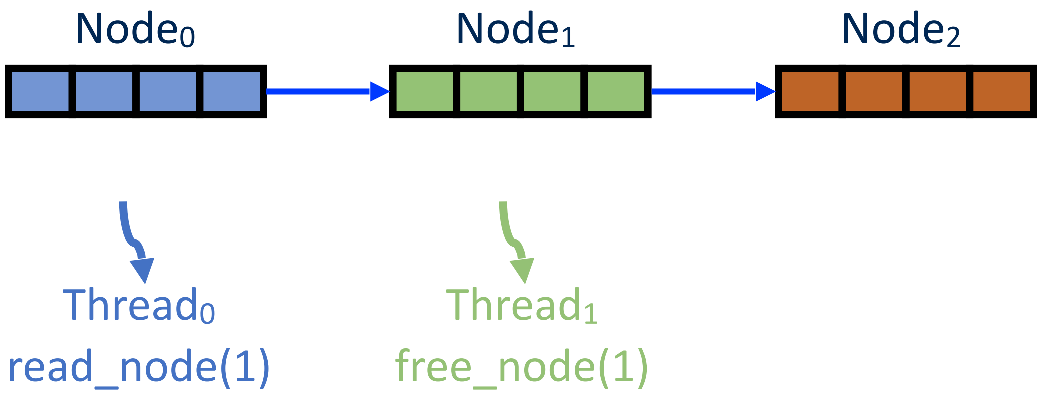
Memory allocator

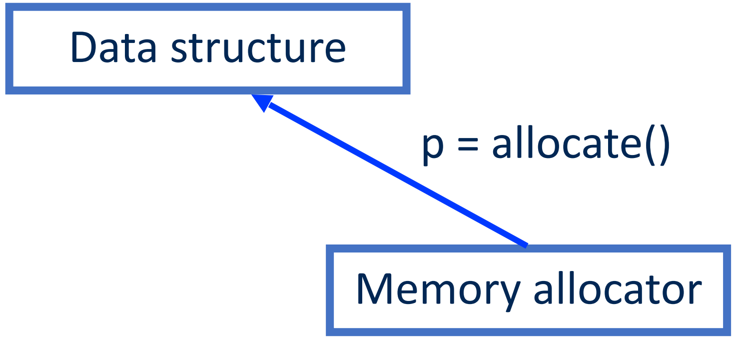
Lifecycle of a pointer (p)

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Memory reclamation
   - When can we free a pointer?



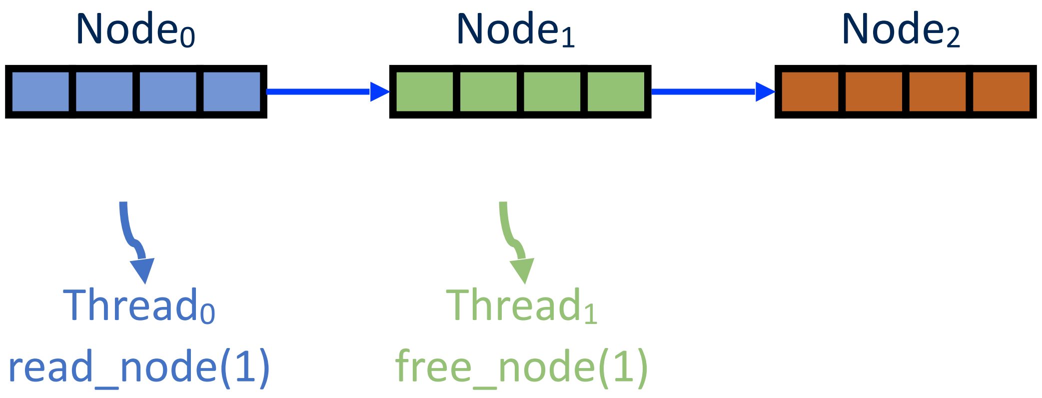



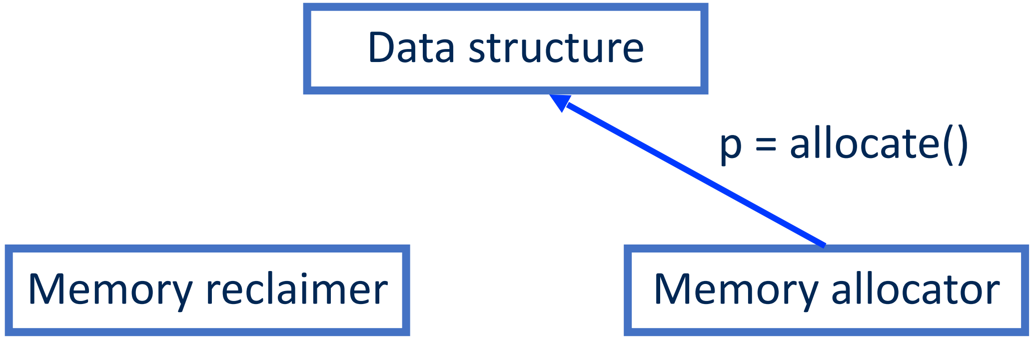

Lifecycle of a pointer (p)

# Challenges with GPU data structures

## 3. Memory reclamation

- When can we free a pointer?



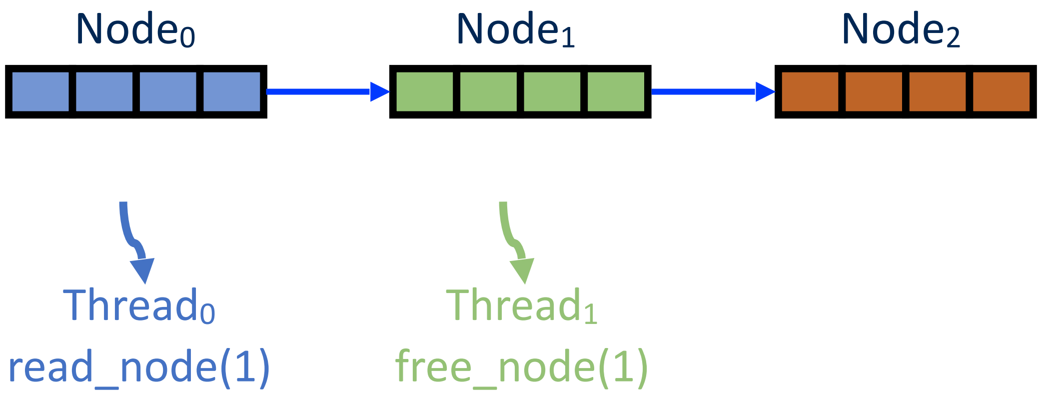

Lifecycle of a pointer (p)

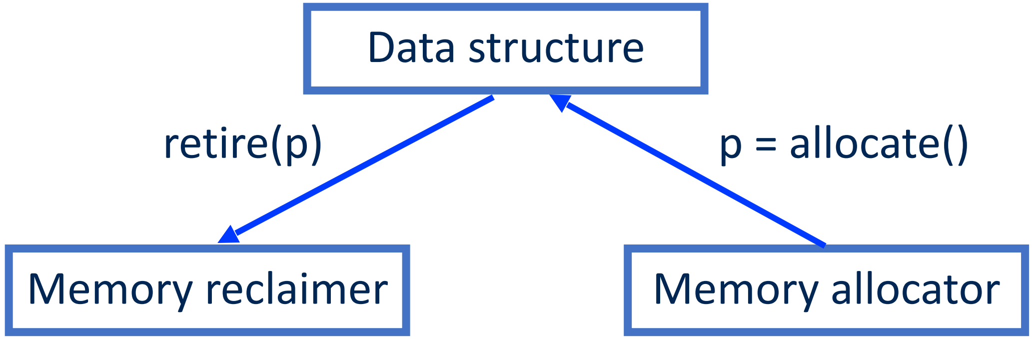
UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

## 3. Memory reclamation

- When can we free a pointer?



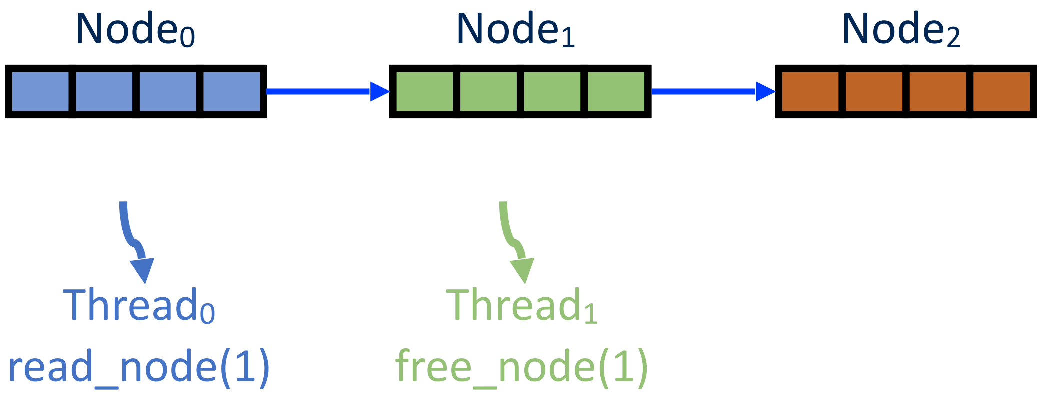



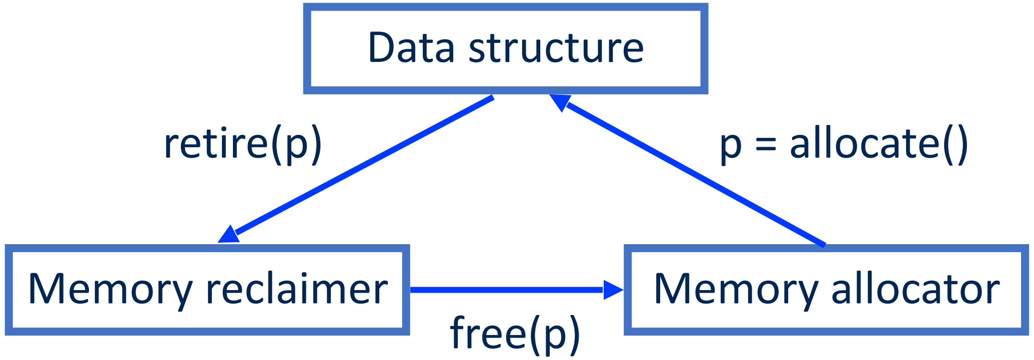

Lifecycle of a pointer (p)

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Memory reclamation
    - When can we free a pointer?




Lifecycle of a pointer (p)

UC DAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Safe memory reclamation

# Challenges with GPU data structures

3. Safe memory reclamation
    - Epoch-based reclamation (EBR)
        - Divide up the execution into epochs

# Challenges with GPU data structures

3. Safe memory reclamation
    - Epoch-based reclamation (EBR)
        - Divide up the execution into epochs
        - Maintain ***per-process*** retired pointers in per-epoch limbo bags

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Safe memory reclamation
   - Epoch-based reclamation (EBR)
     - Divide up the execution into epochs
     - Maintain ***per-process*** retired pointers in per-epoch limbo bags
     - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Safe memory reclamation
   - Epoch-based reclamation (EBR)
     - Divide up the execution into epochs
     - Maintain ***per-process*** retired pointers in per-epoch limbo bags
     - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Safe memory reclamation
   - Epoch-based reclamation (EBR)
     - Divide up the execution into epochs
     - Maintain ***per-process*** retired pointers in per-epoch limbo bags
     - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

Limbo bags

$\text{Process}_0$ $\text{Process}_1$ $\text{Process}_2$

# Challenges with GPU data structures

3. Safe memory reclamation
   - Epoch-based reclamation (EBR)
     - Divide up the execution into epochs
     - Maintain ***per-process*** retired pointers in per-epoch limbo bags
     - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

Limbo bags

| $\text{Process}_0$ | $\text{Process}_1$ | $\text{Process}_2$ |
|---|---|---|
| | | |

Observed Epoch number

| | | |
|---|---|---|
| 4 | 5 | 5 |

# Challenges with GPU data structures

3. Safe memory reclamation
   - Epoch-based reclamation (EBR)
     - Divide up the execution into epochs
     - Maintain ***per-process*** retired pointers in per-epoch limbo bags
     - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

Limbo bags


Observed Epoch number


What is a suitable GPU granularity for a process?

UC DAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

3. Safe memory reclamation

- Epoch-based reclamation (EBR)
  - Divide up the execution into epochs
  - Maintain ***per-process*** retired pointers in per-epoch limbo bags
  - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

Limbo bags

Process$_0$ Process$_1$ Process$_2$

Observed Epoch number

4 5 5

What is a suitable GPU granularity for a process?

**CUDA Block**

# Challenges with GPU data structures

3. Safe memory reclamation
- Epoch-based reclamation (EBR)
  - Divide up the execution into epochs
  - Maintain ***per-process*** retired pointers in per-epoch limbo bags
  - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*

Epoch number 5

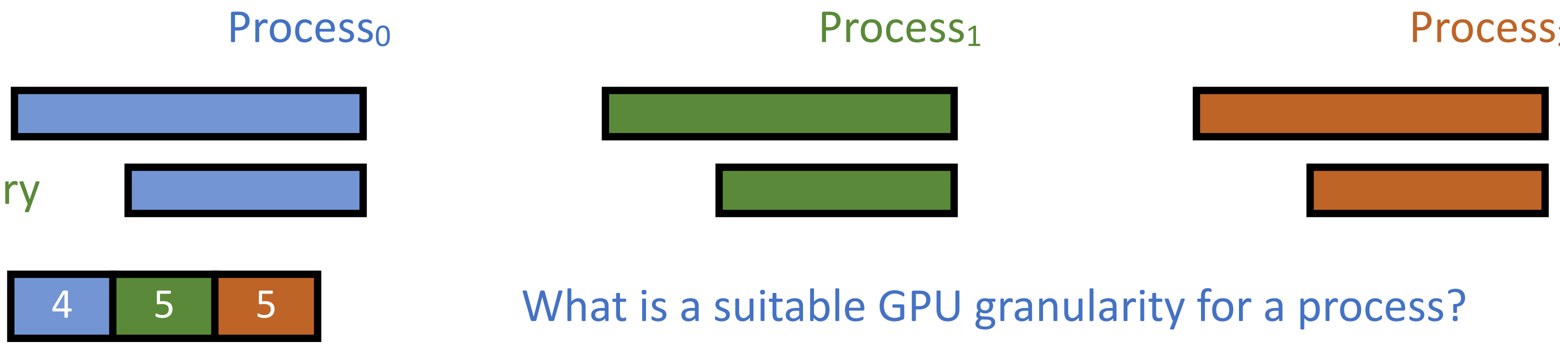
Limbo bags

DRAM

Shared memory

Observed

Epoch number




What is a suitable GPU granularity for a process?

**CUDA Block**

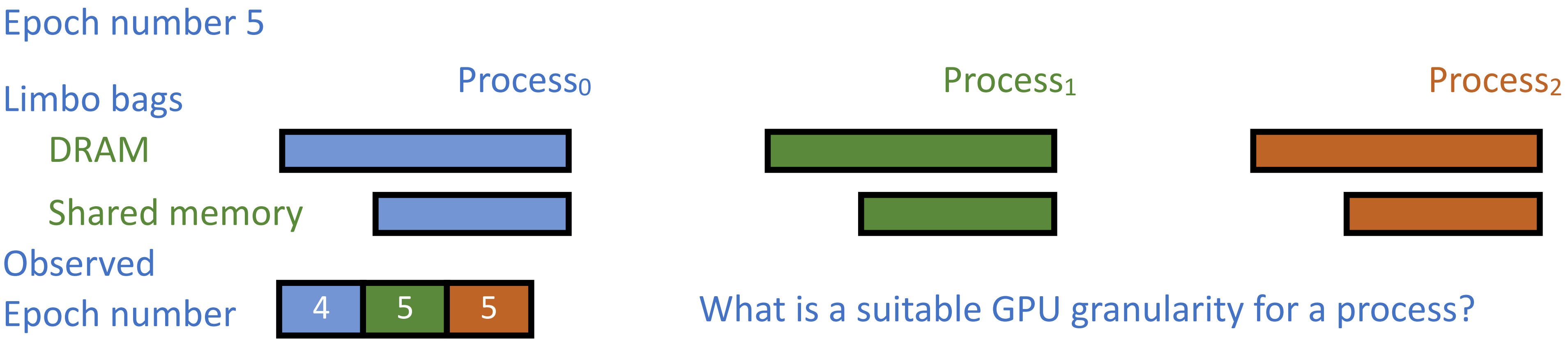
**Scan 80x16 states (80 SMs with 16 resident blocks)**

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

**Our solution: block-wide EBR**

3. Safe memory reclamation
- Epoch-based reclamation (EBR)
  - Divide up the execution into epochs
  - Maintain ***per-process*** retired pointers in per-epoch limbo bags
  - When **all *processes*** are at epoch e, advance the epoch and free all pointers at limbo bags of epoch *e - 2*




What is a suitable GPU granularity for a process?

**CUDA Block**

**Scan 80x16 states (80 SMs with 16 resident blocks)**

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

5. Semantics

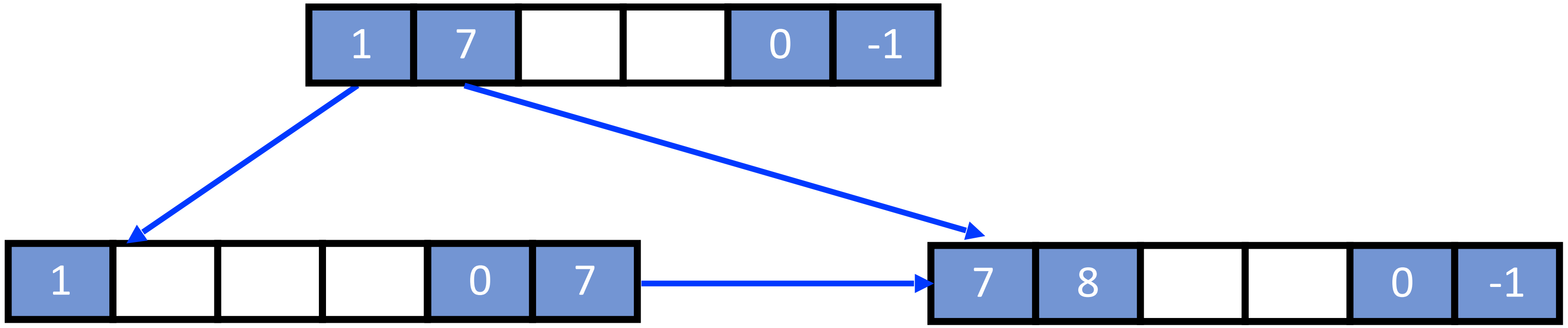
# Challenges with GPU data structures

5. Semantics
    - Meaningful results of concurrent operations
        - We achieve linearizability using snapshots

# Challenges with GPU data structures

5. Semantics
    - Meaningful results of concurrent operations
        - We achieve linearizability using snapshots

Global timestamp = 0




-1 = nullptr

UC DAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

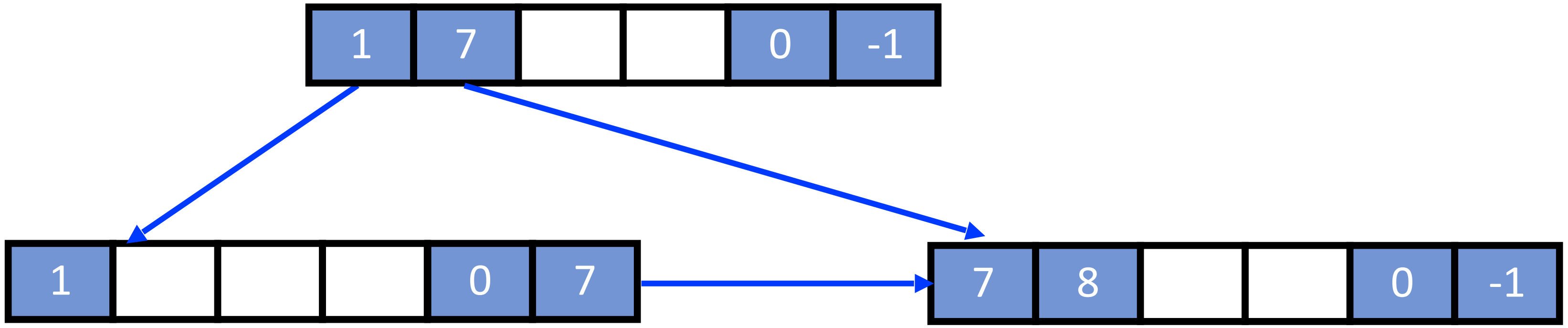
## 5. Semantics

- Meaningful results of concurrent operations
  - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 0




-1 = nullptr

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

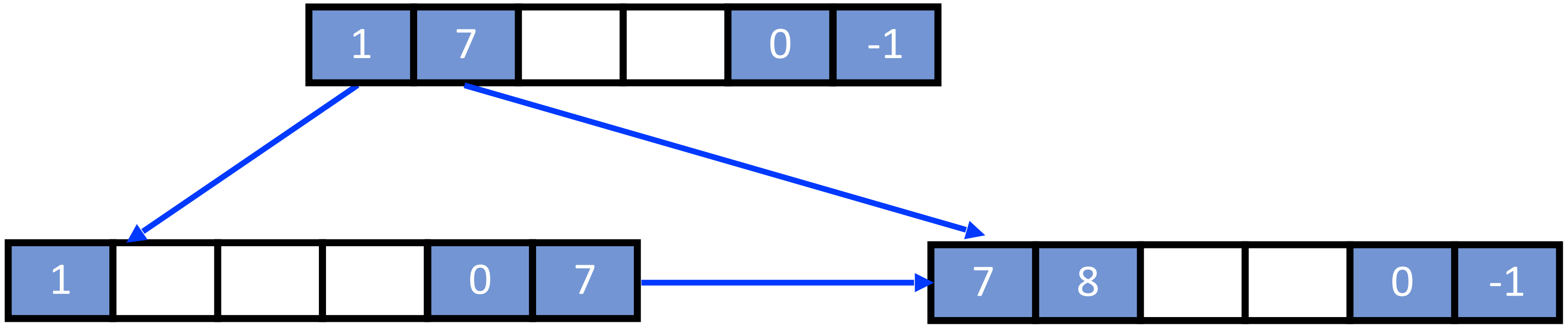
# Challenges with GPU data structures

5. Semantics
    - Meaningful results of concurrent operations
        - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




-1 = nullptr

# Challenges with GPU data structures

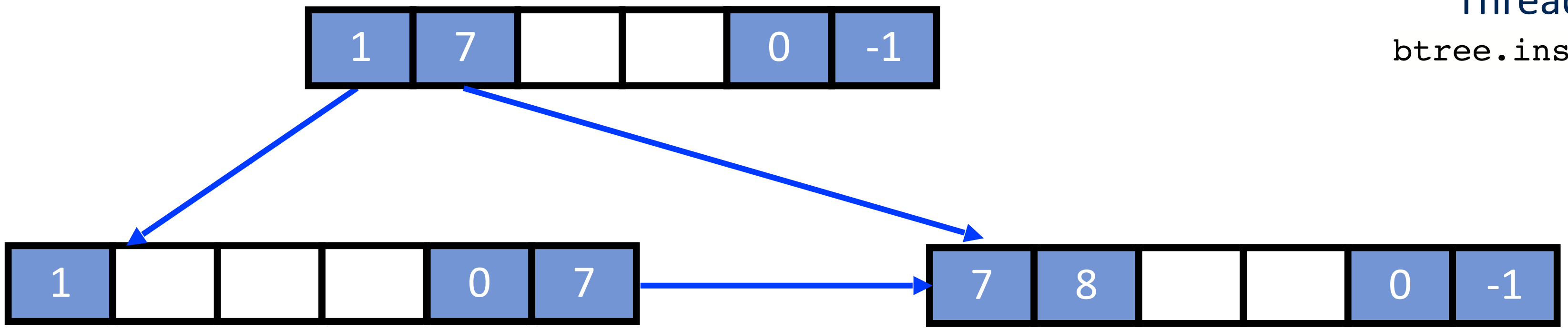
## 5. Semantics

- Meaningful results of concurrent operations
  - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




Thread performing insertion

```
btree.insert({5});
```

-1 = nullptr

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

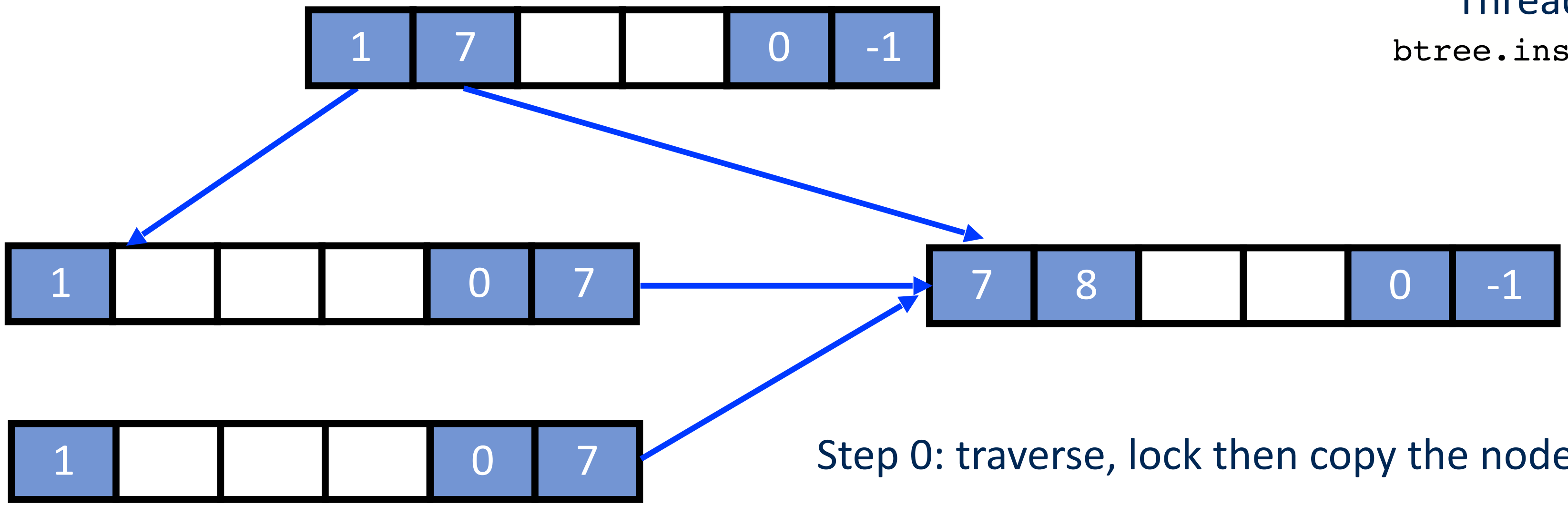
# Challenges with GPU data structures

5. Semantics
   - Meaningful results of concurrent operations
     - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




Thread performing insertion

```
btree.insert({5});
```

-1 = nullptr

Step 0: traverse, lock then copy the node

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

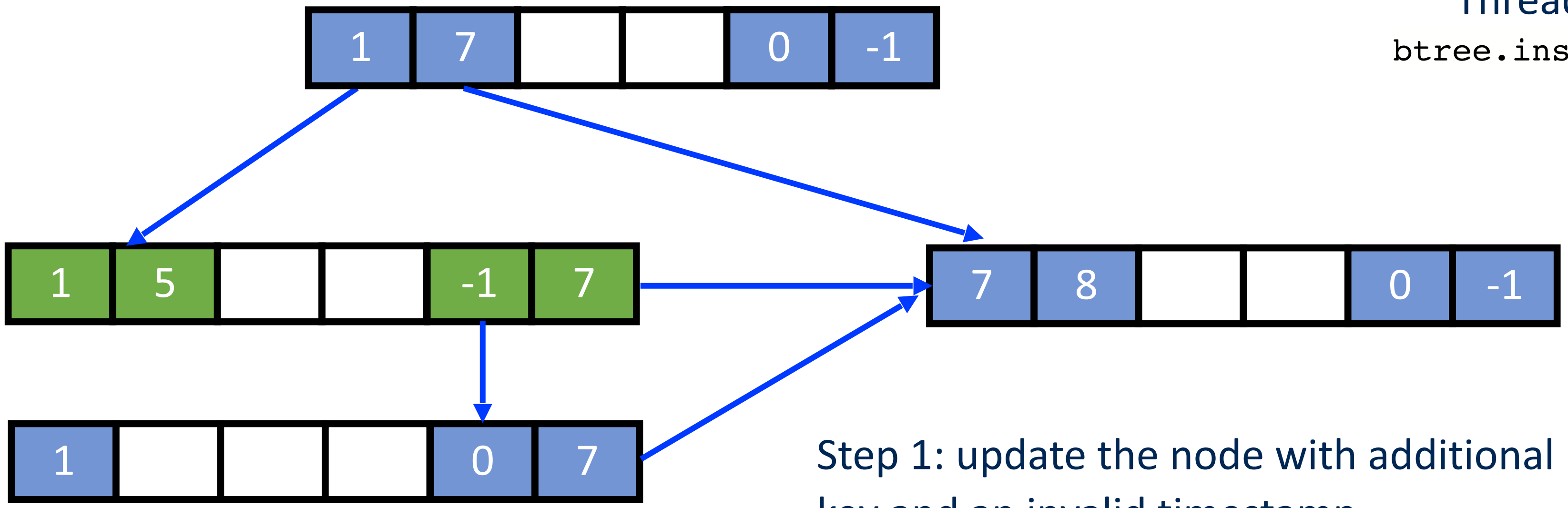
## 5. Semantics

- Meaningful results of concurrent operations
  - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




Thread performing insertion

```
btree.insert({5});
```

-1 = nullptr

Step 1: update the node with additional key and an invalid timestamp
(Node is still locked)

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Challenges with GPU data structures

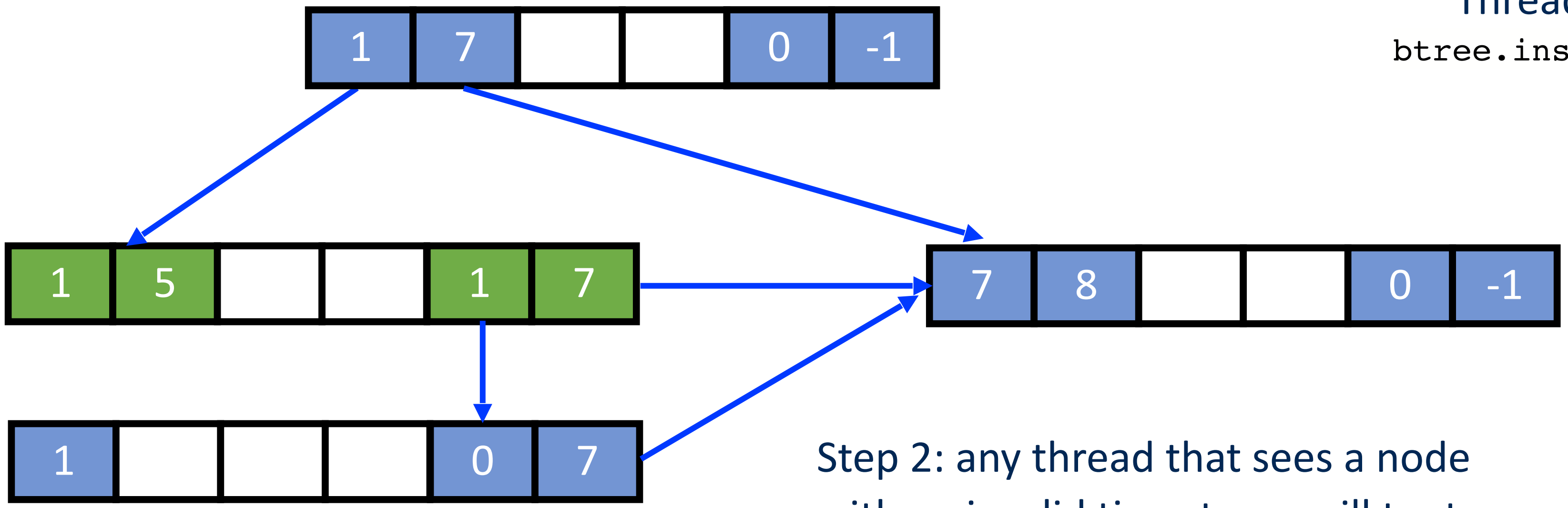
## 5. Semantics

- Meaningful results of concurrent operations
  - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




Thread performing insertion

```
btree.insert({5});
```

-1 = nullptr

Step 2: any thread that sees a node with an invalid timestamp will try to set the timestamp with the most recent one

# Challenges with GPU data structures

## 5. Semantics

- Meaningful results of concurrent operations
  - We achieve linearizability using snapshots

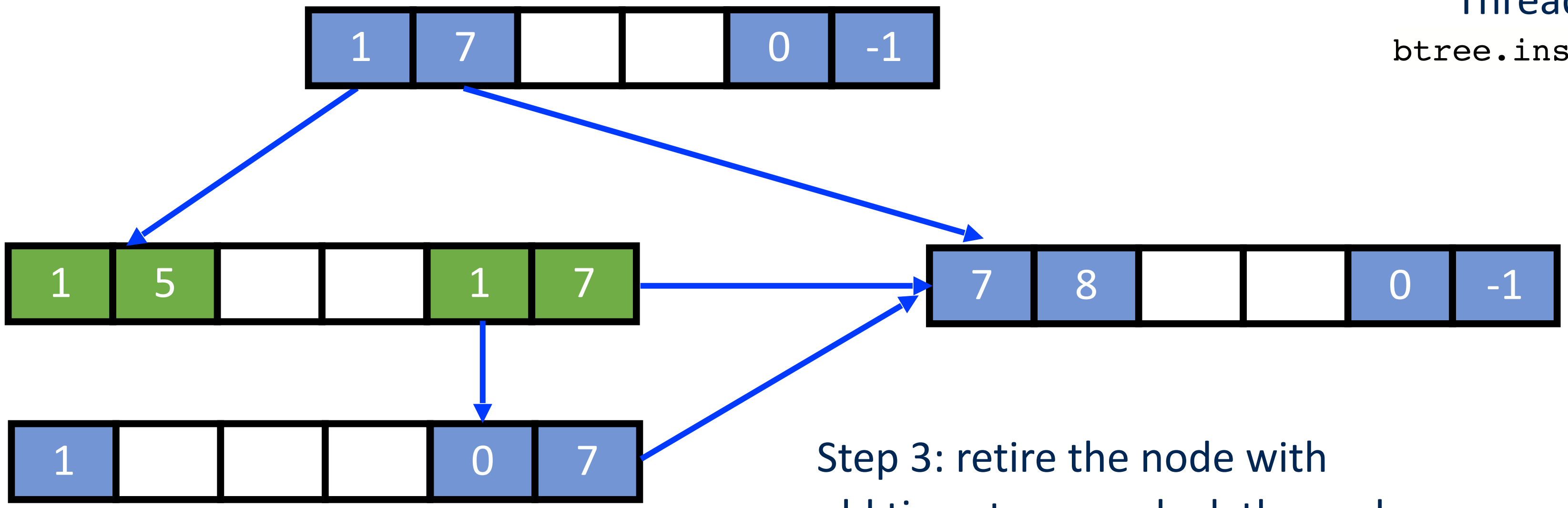
Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1

Thread performing insertion

```
btree.insert({5});
```




-1 = nullptr

Step 3: retire the node with
old timestamp, unlock the node

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

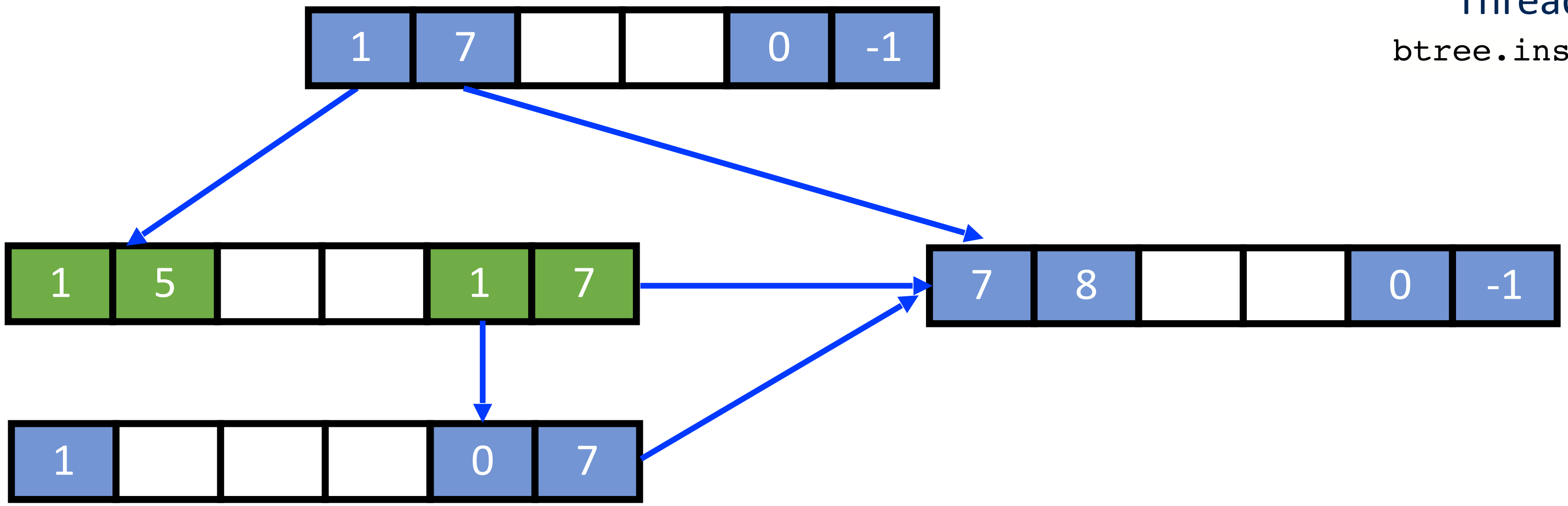
# Challenges with GPU data structures

5. Semantics
   - Meaningful results of concurrent operations
     - We achieve linearizability using snapshots

Thread performing query

```
auto timestamp = btree.take_snapshot();
```

Global timestamp = 1




Thread performing insertion

```
btree.insert({5});
```

-1 = nullptr

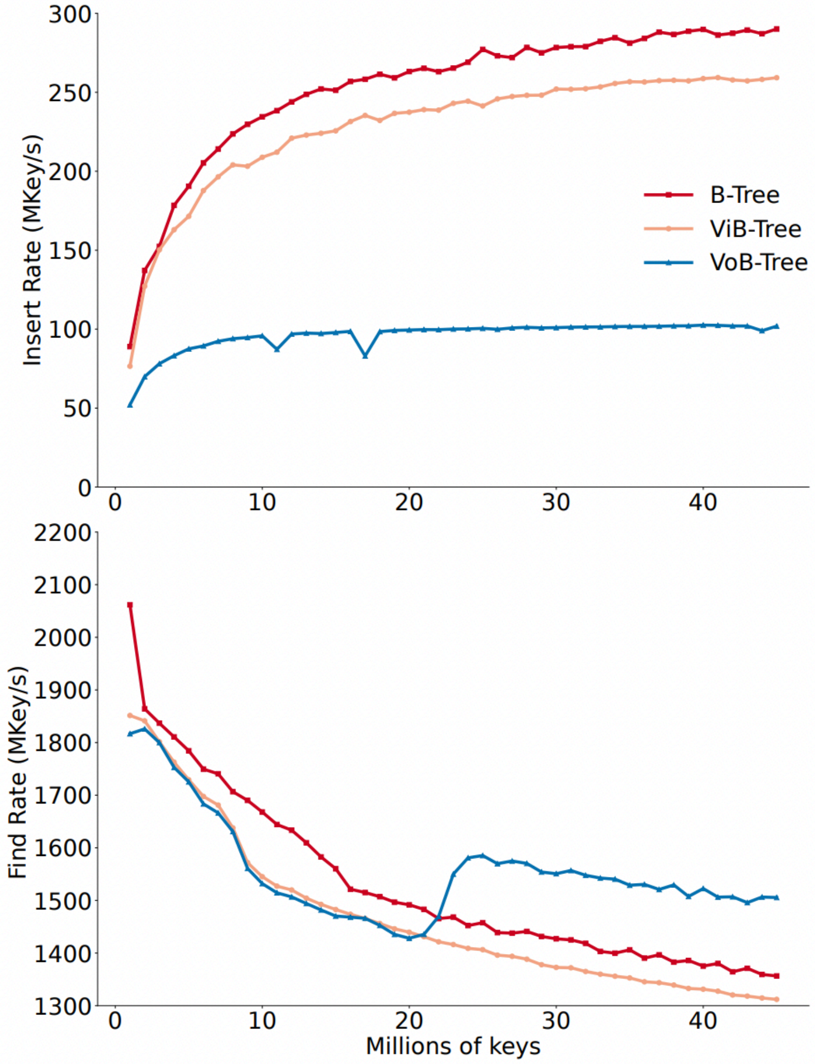
Snapshot algorithm based on:
*Wei et al., Constant-Time Snapshots with Applications to Concurrent Data Structures*

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates



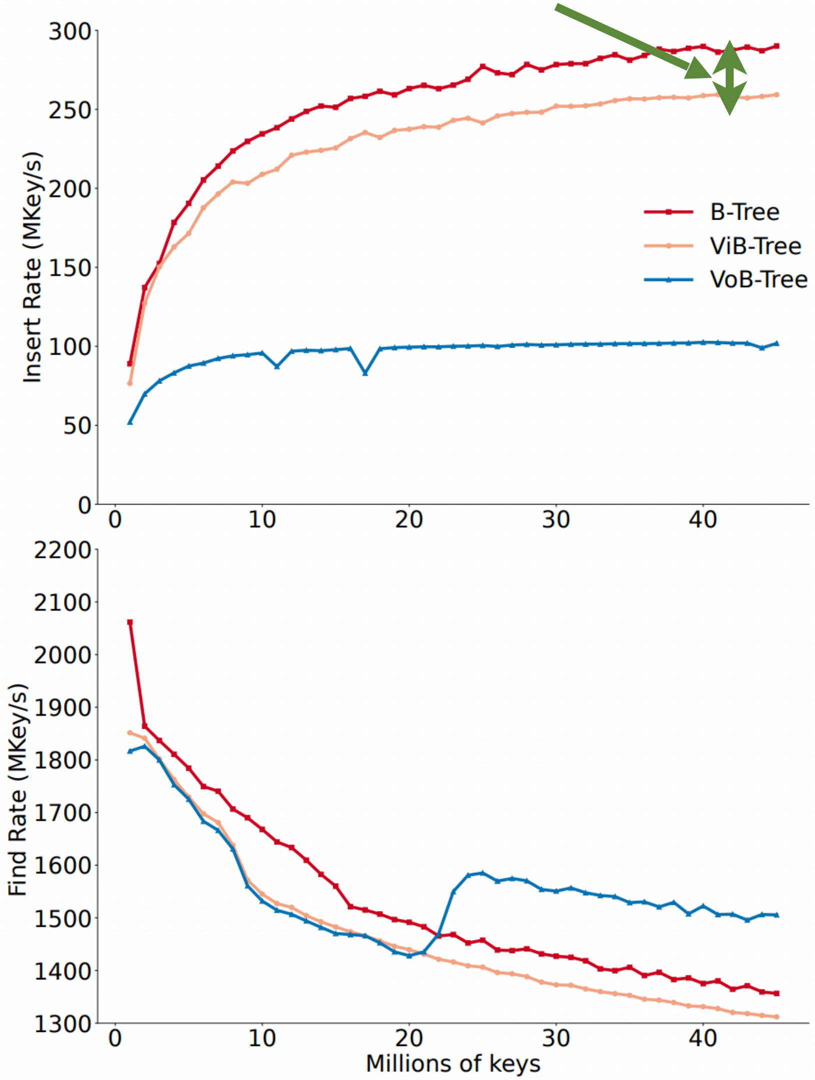

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates

Low overhead when all updates are in-place

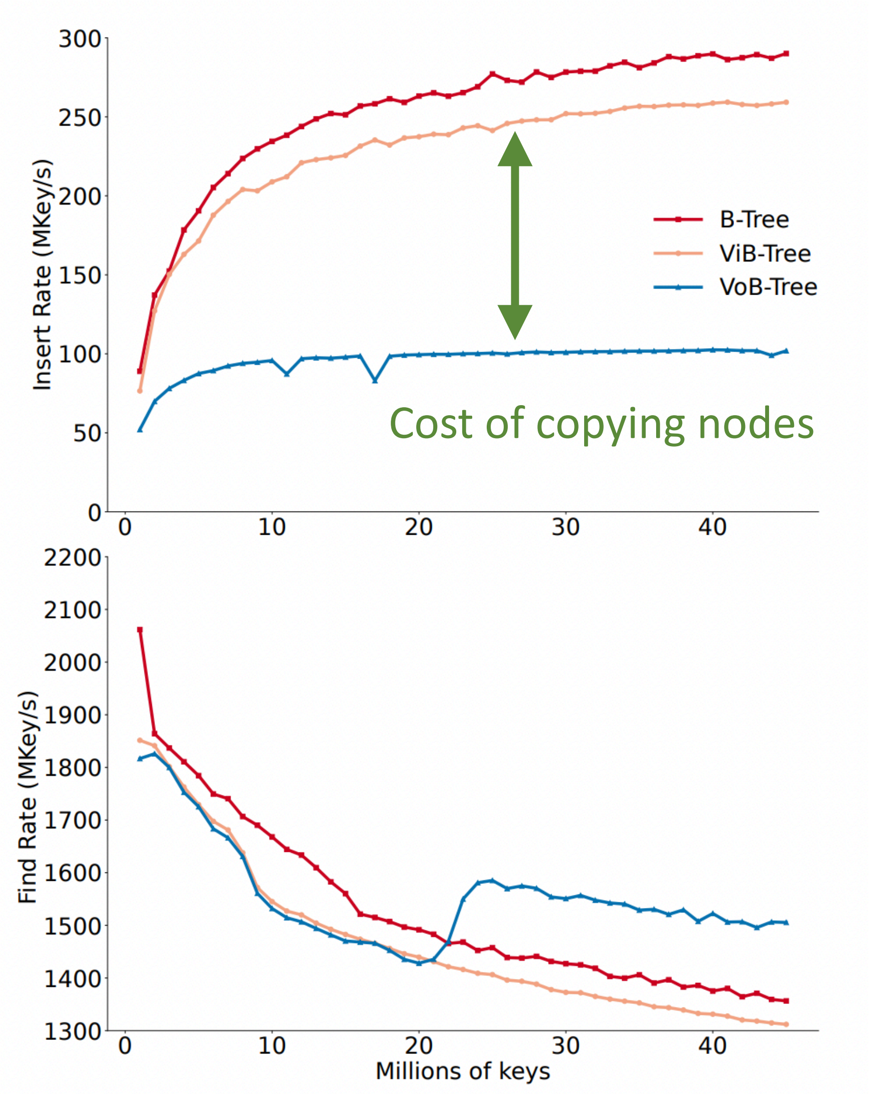



Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates

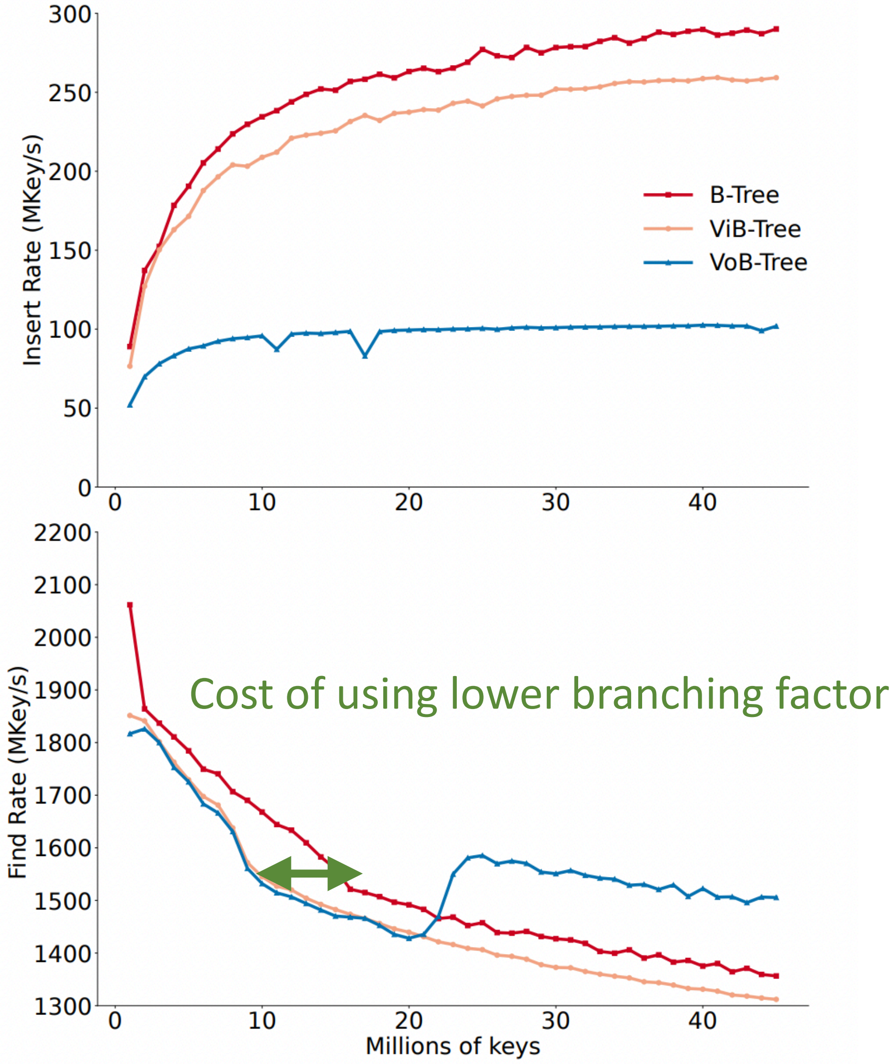



Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates

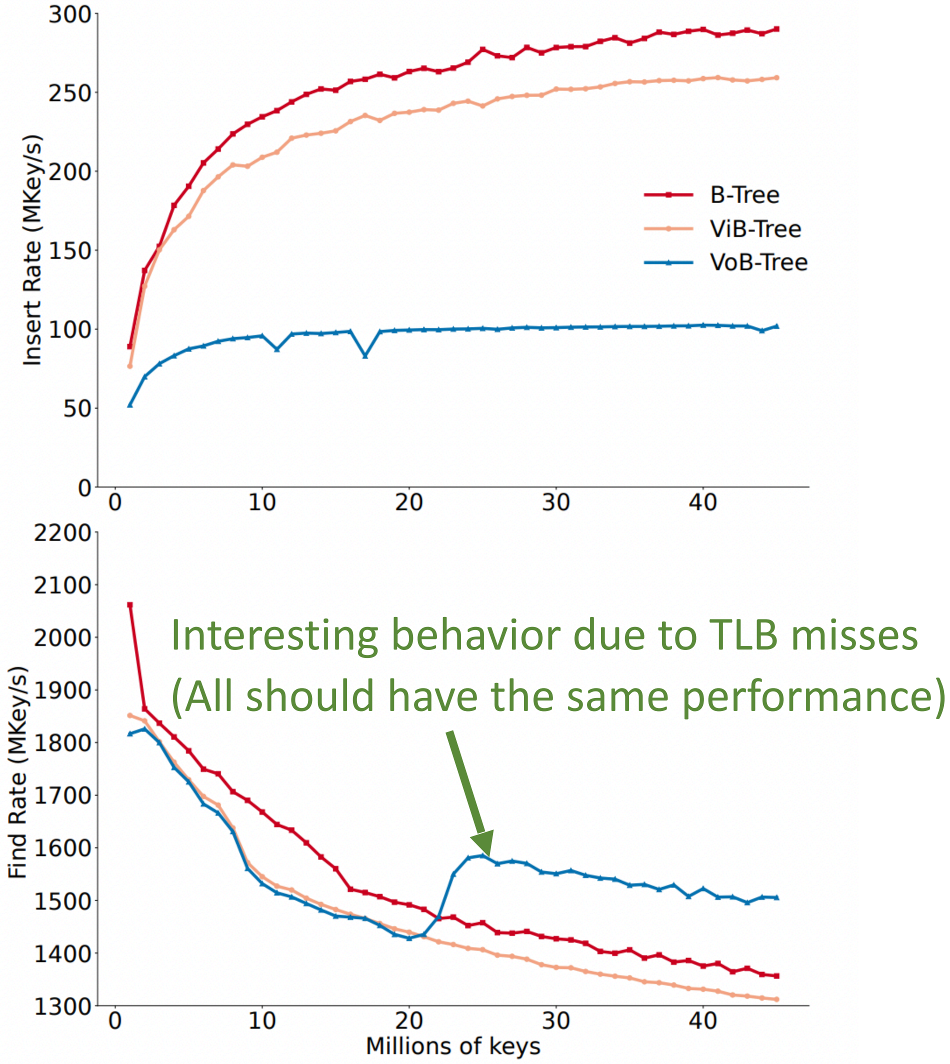



UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates

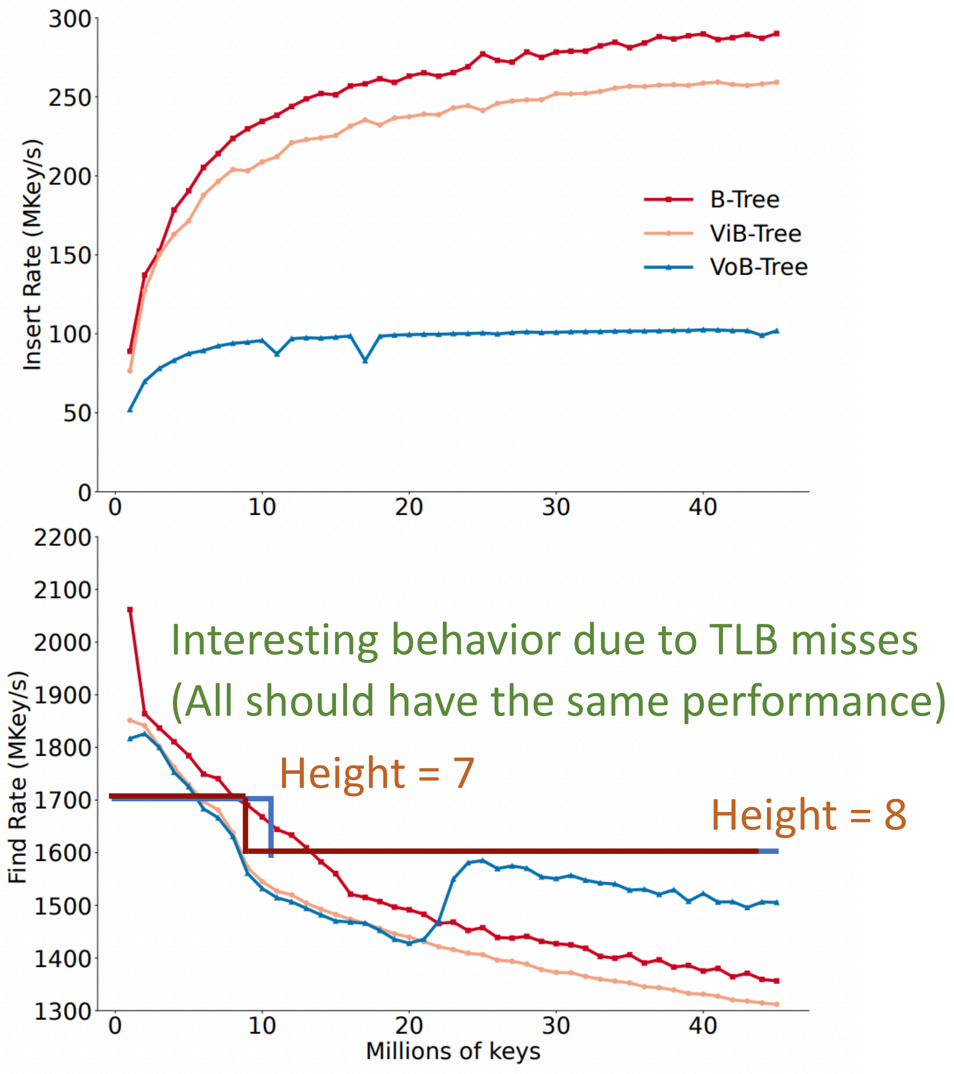



Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results: Versioning overhead

- Insertion and find rates
  - B-Tree
    - Baseline that is not linearizable
  - ViB-Tree
    - Multiversion B-Tree
      - Only perform in-place updates
  - VoB-Tree
    - Multiversion B-Tree
      - Only perform out-of-place updates

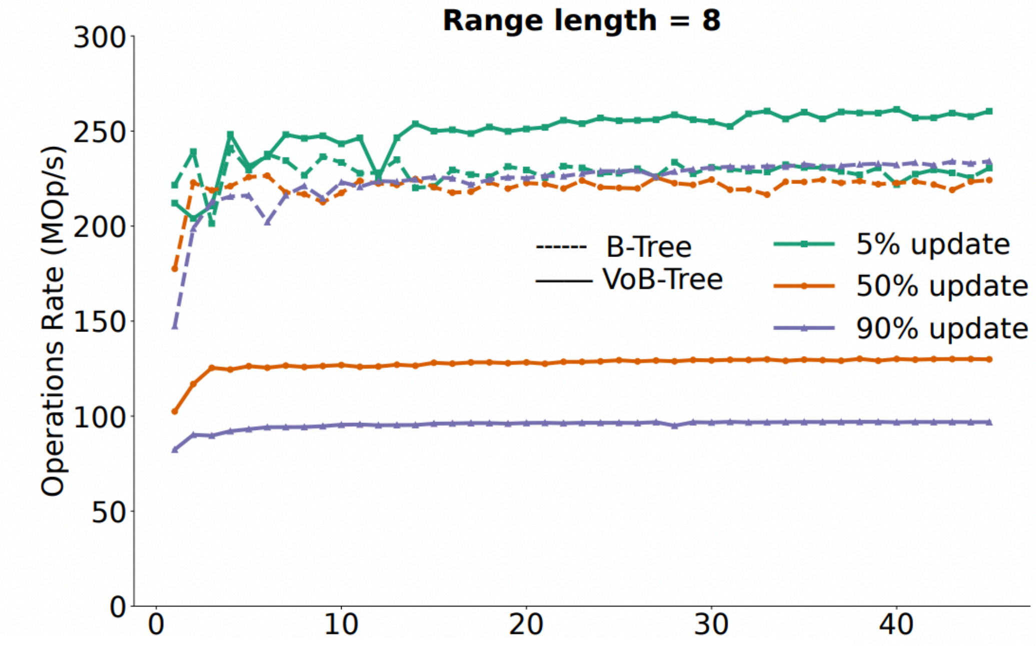


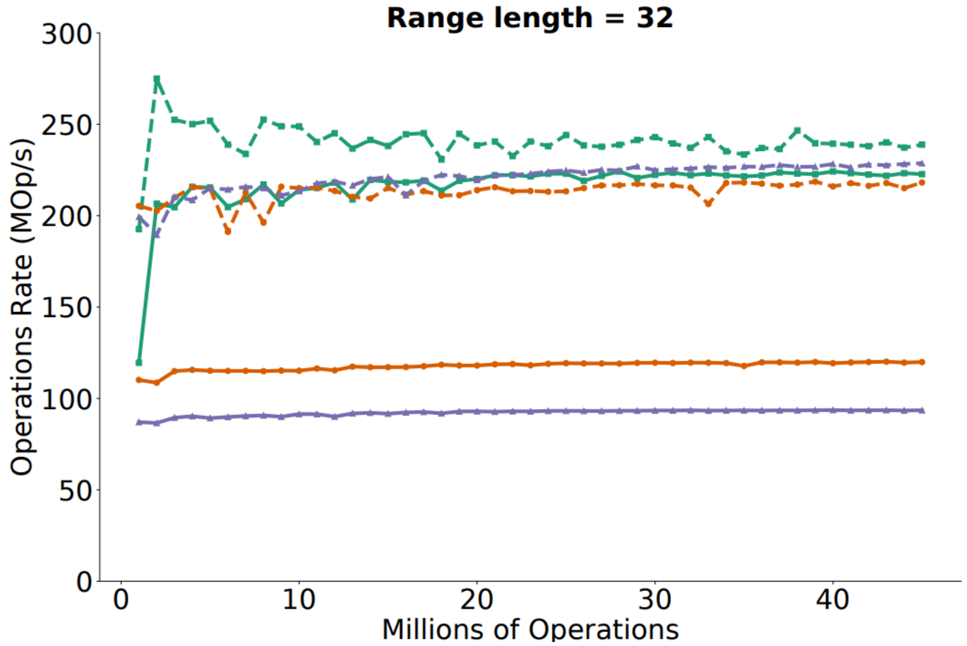

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results: Linearizable multipoint queries

- Concurrent insertion and range query operations



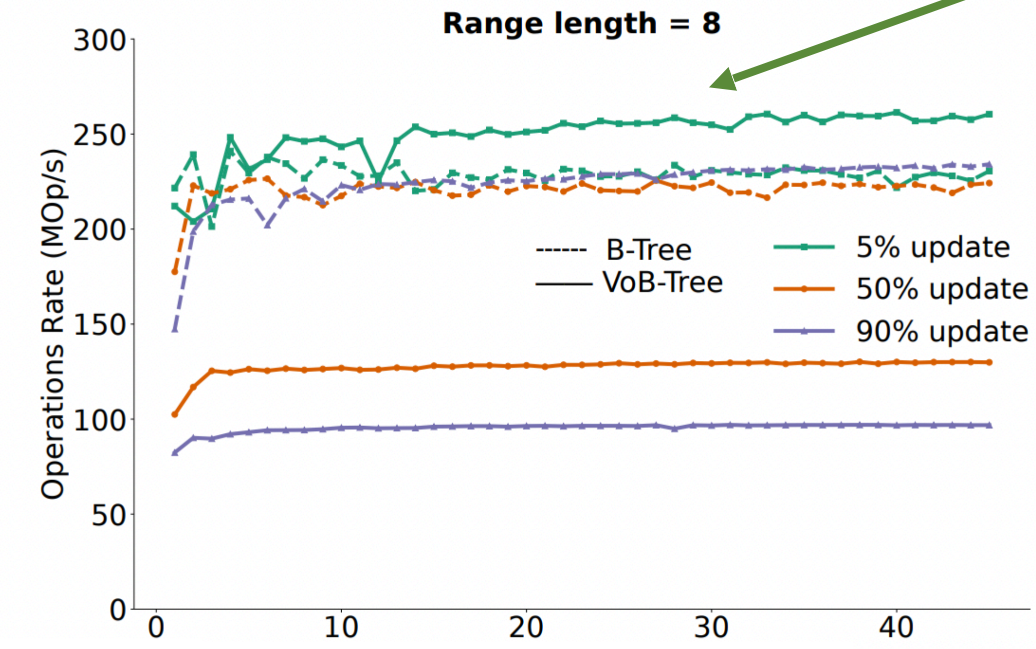

**Initial tree size of 40M.**

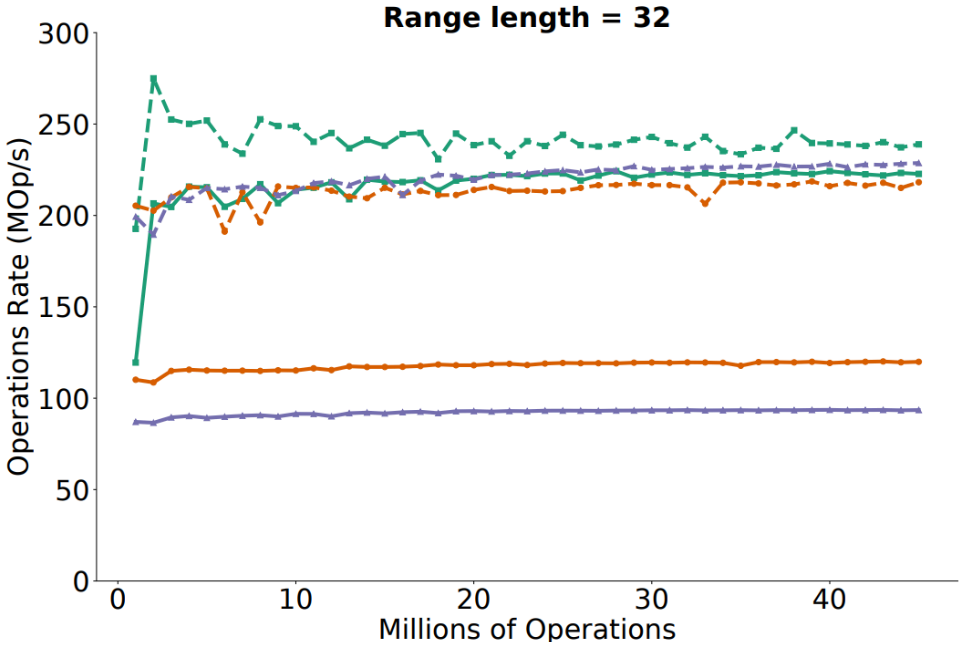
Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

# Results: Linearizable multipoint queries

- Concurrent insertion and range query operations

Interestingly, our VoB-Tree is faster than the baseline



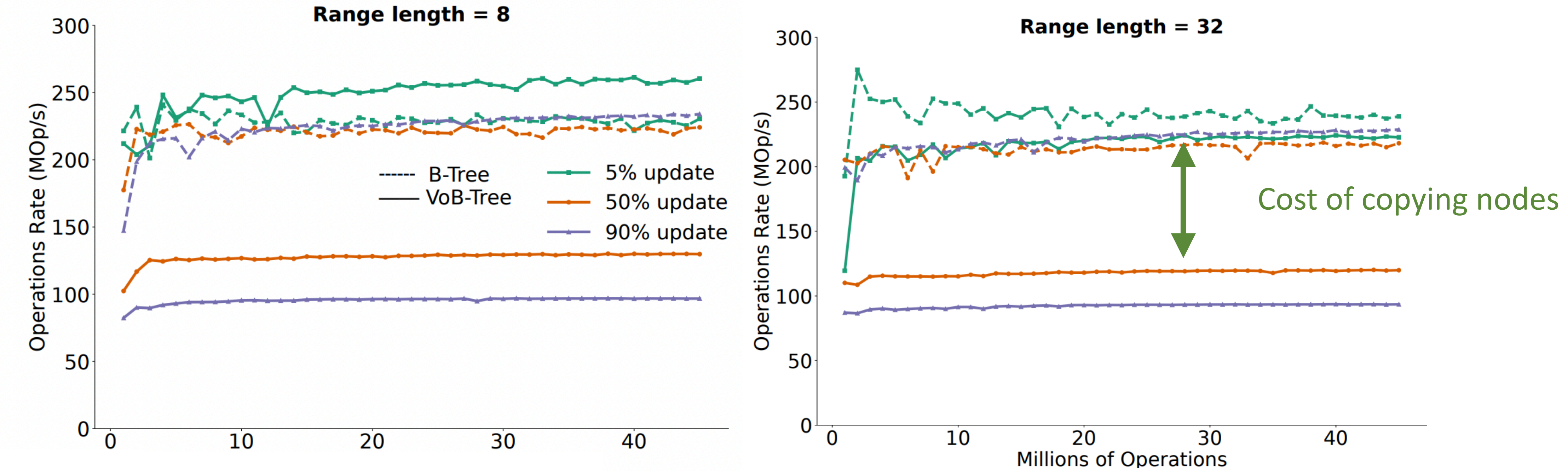

**Initial tree size of 40M.**

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

# Results: Linearizable multipoint queries

- Concurrent insertion and range query operations



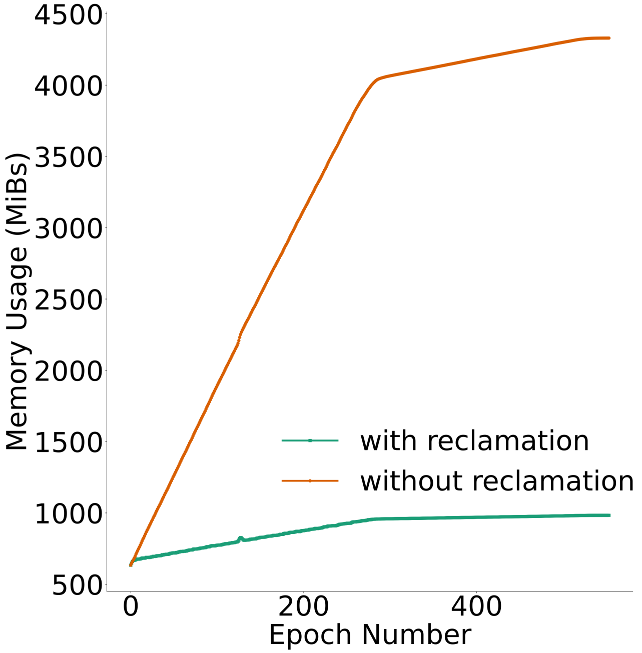

**Initial tree size of 40M.**

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results

- Safe memory reclamation




Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

Block-wide EBR performance

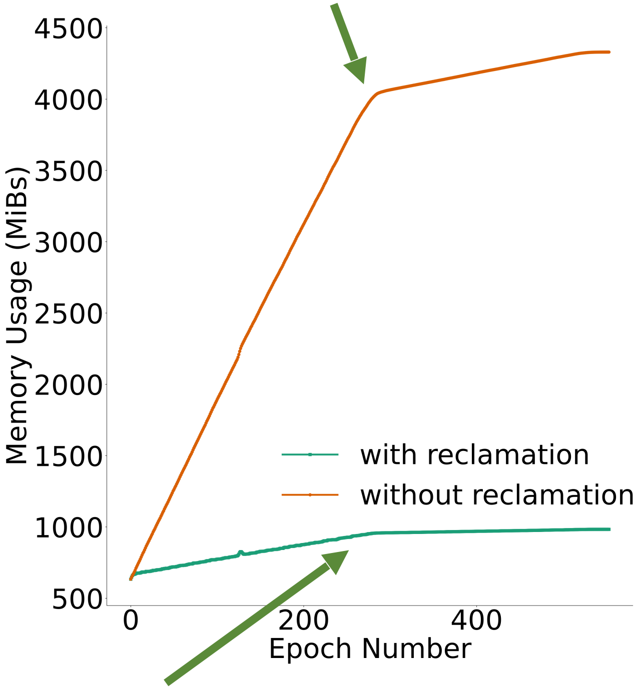
45 million insertion and range query operations
50% update ratio, and average range length of 16

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Results

- Safe memory reclamation

Blocks performing insertion start to exit




Block-wide EBR performance

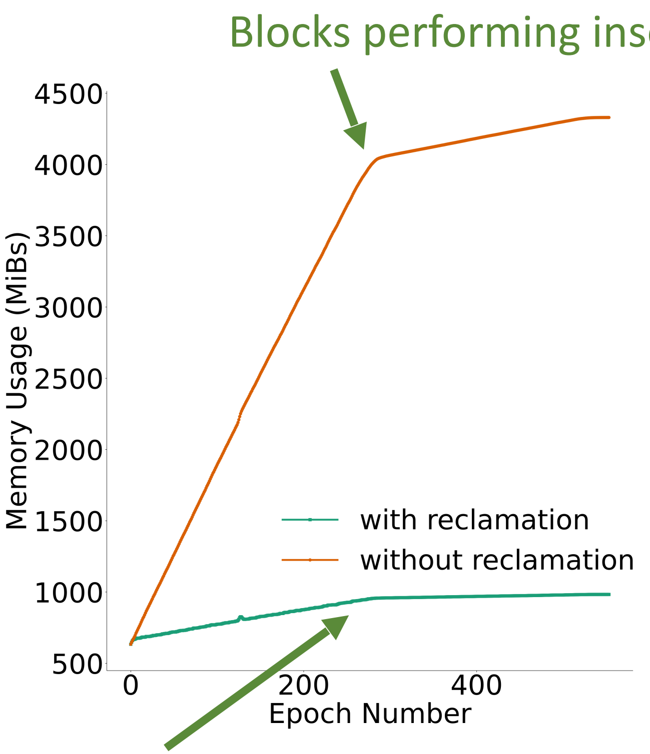
45 million insertion and range query operations
50% update ratio, and average range length of 16

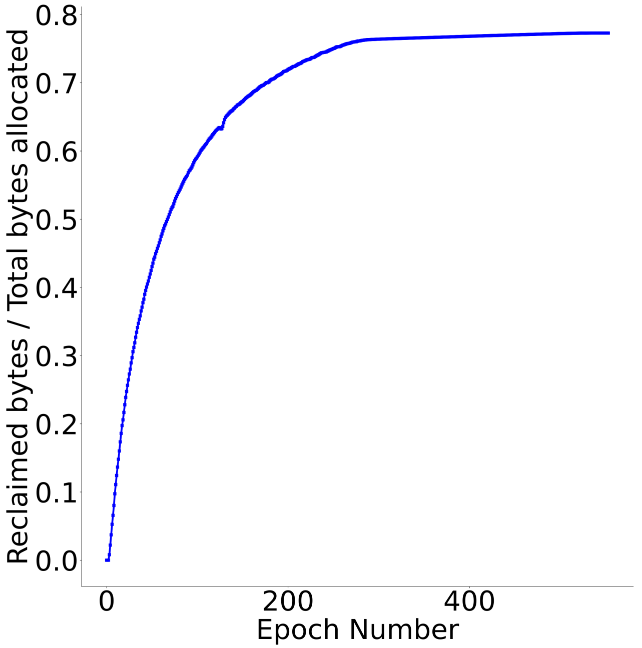
Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results

- Safe memory reclamation




Block-wide EBR performance

45 million insertion and range query operations
50% update ratio, and average range length of 16

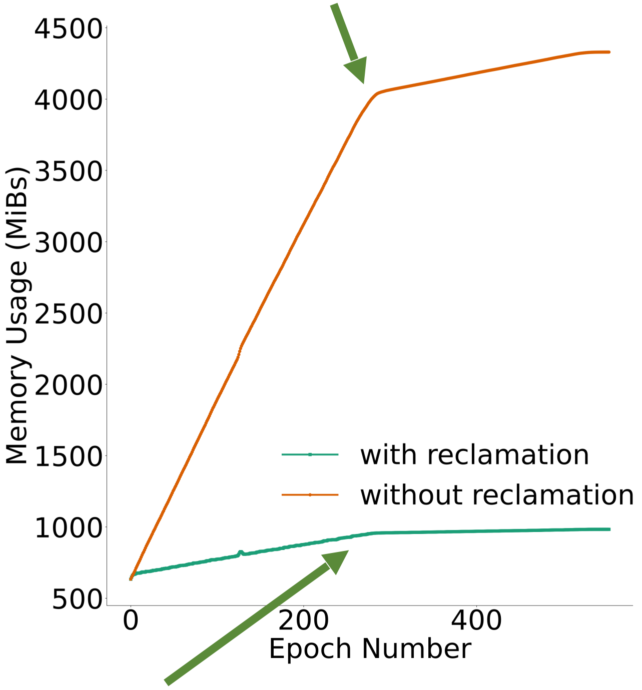
Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

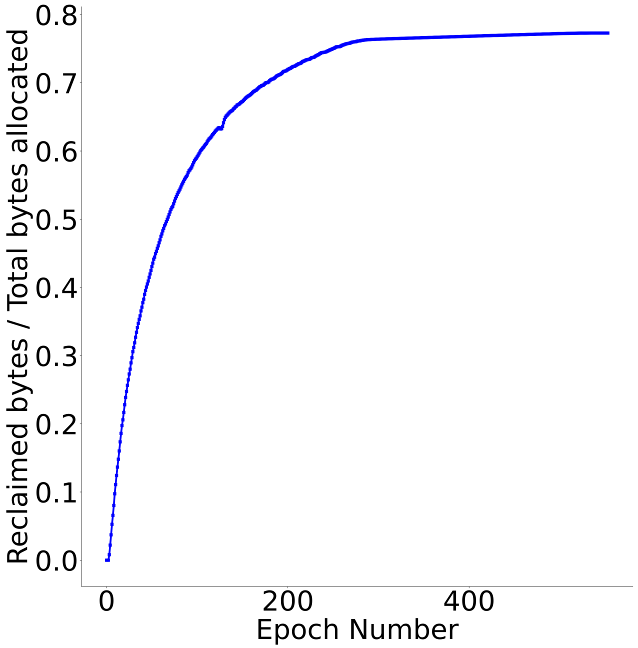
UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results

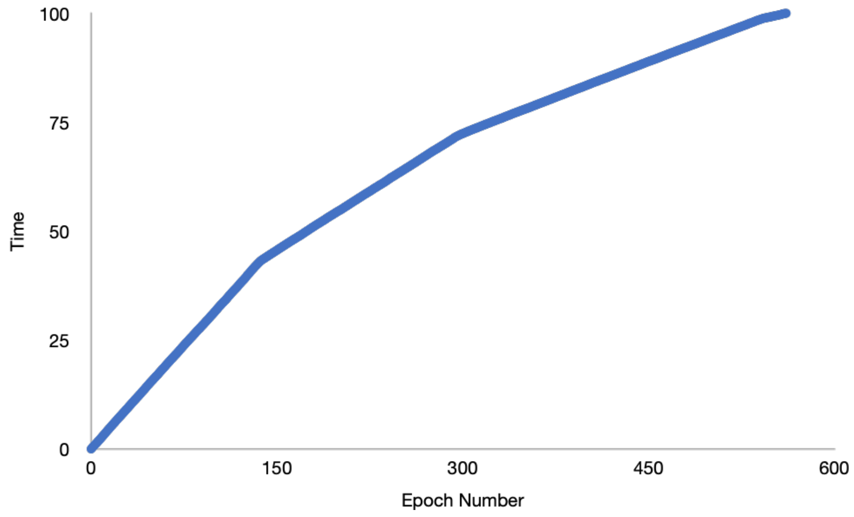
- Safe memory reclamation

Blocks performing insertion start to exit




Epoch is not time




Block-wide EBR performance

45 million insertion and range query operations

50% update ratio, and average range length of 16

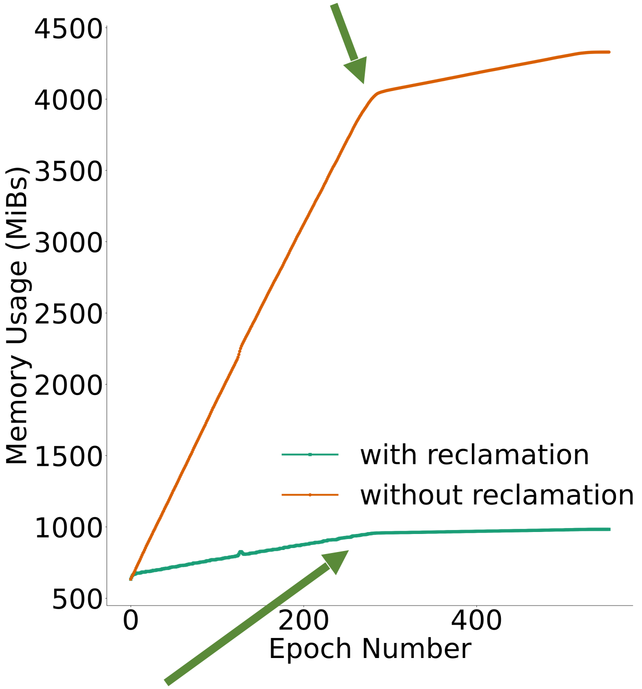
Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

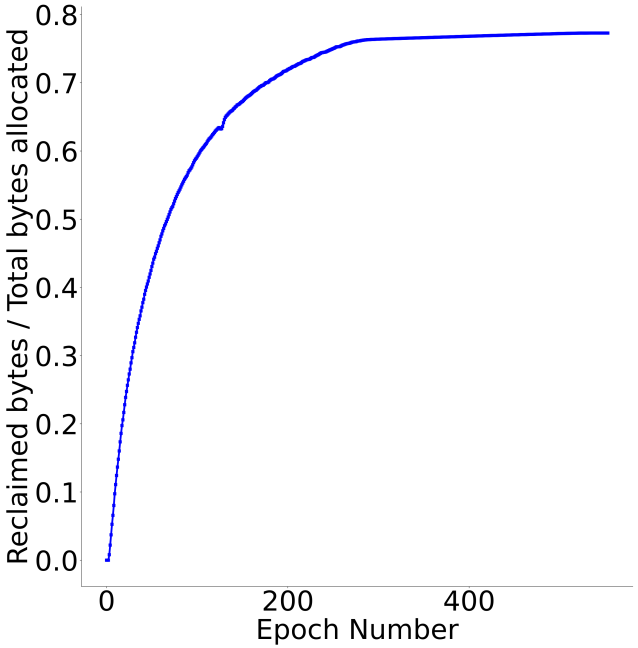
UCDAVIS ELECTRICAL AND COMPUTER ENGINEERING

# Results

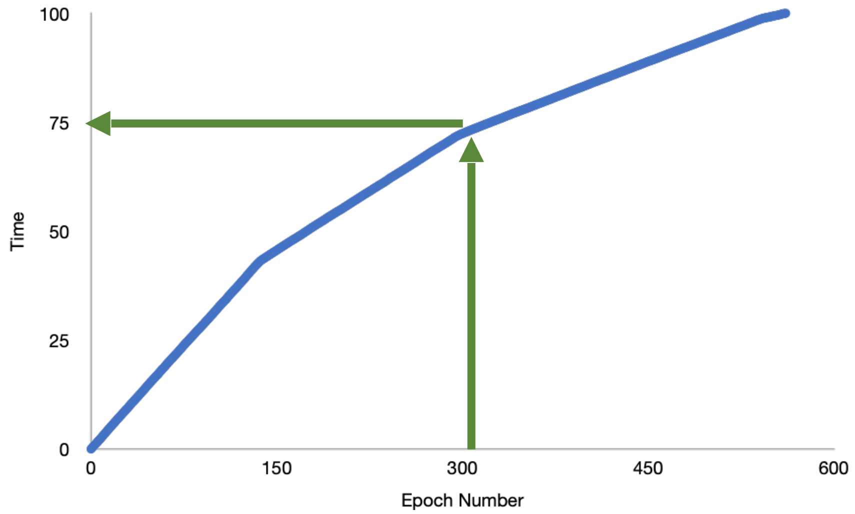
- Safe memory reclamation

Blocks performing insertion start to exit




Epoch is not time




Block-wide EBR performance

45 million insertion and range query operations
50% update ratio, and average range length of 16

Using Tesla V100 PCIe (Volta architecture) GPU with 32 GB DRAM

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# **Our solutions** to the Multiversion GPU B-Tree challenges

1. Memory access patterns during traversal
   - Data structure layout in memory
     - ***Use cache-line-sized nodes and cooperative processing***
2. Contention during updates
   - Atomics and locks
     - ***Relax data structure invariants, avoid spin locks, and use restarts***
3. Memory allocation and reclamation
   - When can we free a pointer?
     - ***Use block-wide epoch-based reclamation***
4. Abstractions
   - Defining APIs for GPU data structures
     - ***Define APIs at the lowest efficient level (e.g., tile)***
5. Semantics
   - Meaningful results of concurrent operations
     - ***Achieve linearizability using (scoped) snapshots***

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Summary and future research directions

**Summary**

- Our data structure supports snapshots with **minimal overhead** in point queries (1.04× slower) and insertions (1.11× slower) compared to baseline
- **Similar** performance for read-heavy workloads and **2.39× slower** for write-heavy workloads

**Future research directions**

- Wait-free data structure
  - **Helping** algorithms
- Multiversion GPU graph data structure
  - **Composed** from per-vertex multiversion B-Tree
- Exploration of various safe memory reclamation techniques

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Thank you!

## Other GPU data structures work:

"Better GPU Hash Tables", *APOCS 2023*.
https://github.com/owensgroup/BGHT
"Engineering a High-Performance GPU B-Tree", *PPoPP 2019.*
https://github.com/owensgroup/GpuBTree
"Dynamic Graphs on the GPU", *IPDPS 2020.*
https://github.com/gunrock/gunrock
"A GPU Multiversion B-Tree", *PACT 2022.*
https://github.com/owensgroup/MVGpuBTree
"Fully Concurrent GPU Data Structures", *Ph.D. dissertation, UC Davis, 2022.*
https://escholarship.org/uc/item/5kc834wm

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Summary and future research directions

## Summary

- Our data structure supports snapshots with **minimal overhead** in point queries (1.04× slower) and insertions (1.11× slower) compared to baseline
- **Similar** performance for read-heavy workloads and **2.39× slower** for write-heavy workloads

## Future research directions

- Wait-free data structure
  - **Helping** algorithms
- Multiversion GPU graph data structure
  - **Composed** from per-vertex multiversion B-Tree
- Exploration of various safe memory reclamation techniques

UCDAVIS
ELECTRICAL AND COMPUTER ENGINEERING

# Thank you!

UC DAVIS
ELECTRICAL AND COMPUTER ENGINEERING