Analyzing and Implementing GPU Hash Tables

Agnieszka Łupińska

UC Davis

alupinska@ucdavis.edu

Paper by: Muhammad A. Awad[1,*], Saman Ashkiani[2], Serban D. Porumbescu[1], Martín Farach-Colton[3], and John D. Owens[1]

[1]**UC Davis** [2]**NVIDIA** [3]**Rutgers University**

[*]**Now at AMD Research**

Agenda

- Summary of our paper
- Hash tables parameter space landscape
  - Evaluated probing schemes
- Our generic implementation
- Results and discussion

Summary of our paper

- Our principal hypothesis:

  “The main factor influencing the performance of insertion (or query) in a hash table is the number of probes each operation performs to complete an insertion (or query)”

- Our contributions:
  - Efficient hash table implementation that decouple probing schemes from hash table and GPU-specific details,
  - Analysis-driven recommendations for different hash table uses cases, and
  - Hardware-agnostic analysis of two probing schemes
- Our conclusions:
  - Bucketed hash tables where the bucket size matches the cache line size achieve the best performance
  - Cuckoo hash tables with a cache-line-sized bucket achieves the best overall performance
    - {1.43, 1.39 and 2.8} memory operations for {insertion, positive and negative queries) at 99% load factor.

Summary of our paper

- Our principal hypothesis:

  "The main factor influencing the performance of insertion (or query) in a hash table is the number of probes each operation performs to complete an insertion (or query)"

- Our contributions:
  - Efficient hash table implementation that decouples probing schemes from hash table and GPU-specific details,
  - Analysis-driven recommendations for different hash table uses cases, and
  - Hardware-agnostic analysis of two probing schemes
- Our conclusions:
  - Bucketed hash tables where the bucket size matches the cache line size achieve the best performance
  - Cuckoo hash tables with a cache-line-sized bucket achieves the best overall performance
    - {1.43, 1.39 and 2.8} memory operations for {insertion, positive and negative queries) at 99% load factor.

Summary of our paper

- Our principal hypothesis:

  "The main factor influencing the performance of insertion (or query) in a hash table is the number of probes each operation performs to complete an insertion (or query)"

- Our contributions:
  - Efficient hash table implementation that decouple probing schemes from hash table and GPU-specific details,
  - Analysis-driven recommendations for different hash table uses cases, and
  - Hardware-agnostic analysis of two probing schemes
- Our conclusions:
  - Bucketed hash tables where the bucket size matches the cache line size achieve the best performance
  - Cuckoo hash tables with a cache-line-sized bucket achieves the best overall performance
    - {1.43, 1.39 and 2.8} memory operations for {insertion, positive and negative queries) at 99% load factor.

Summary of our paper

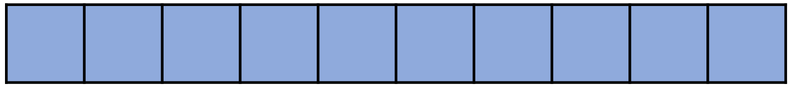
- Our principal hypothesis:

  "The main factor influencing the performance of insertion (or query) in a hash table is the number of probes each operation performs to complete an insertion (or query)"

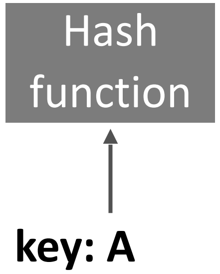
- Our contributions:
  - Efficient hash table implementation that decouples probing schemes from hash table and GPU-specific details,
  - Analysis-driven recommendations for different hash table uses cases, and
  - Hardware-agnostic analysis of two probing schemes
- Our conclusions:
  - Bucketed hash tables where the bucket size matches the cache line size achieve the best performance
  - Cuckoo hash tables with a cache-line-sized bucket achieves the best overall performance
    - {1.43, 1.39 and 2.8} memory operations for {insertion, positive and negative queries) at 99% load factor.

Performance metrics for hash tables

- Given a hash table that occupies space *m*:
  - Can we insert *n* keys?
    - i.e., achieve a load factor = n / m
  - What is the insertion rate?
  - What is the query rate?
    - e.g., all positive, all negative, both



$$h(k; a, b) = ((ak + b) \mod p) \mod L,$$



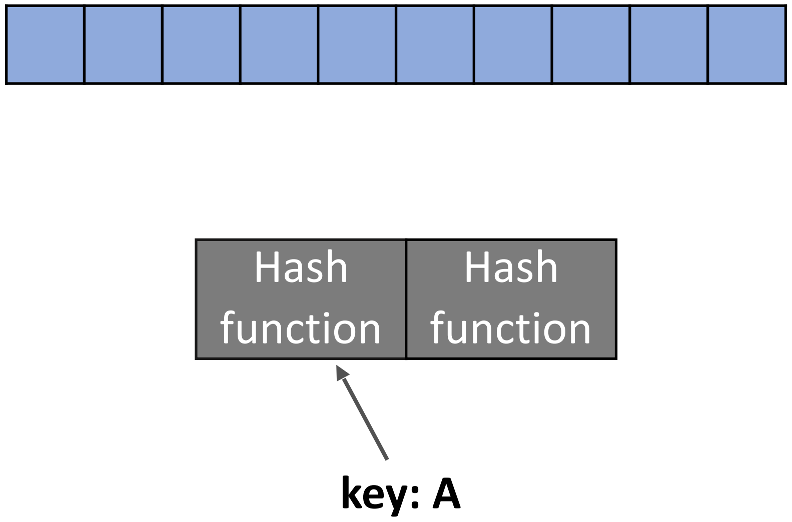
Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo and iceberg hashing**
- Bucket size
- Probe complexity
  - Number of hash functions
- Placement strategy
  - Balanced or not balanced

Probing Scheme

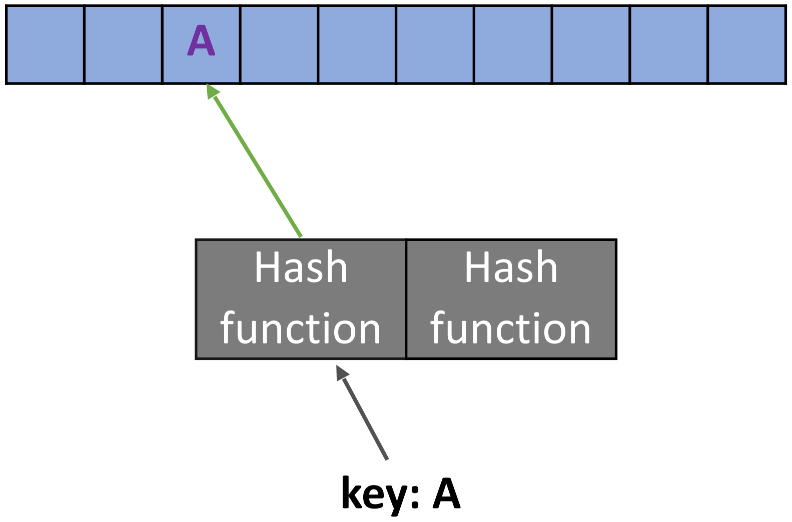
Cuckoo hashing

- Insert(A)



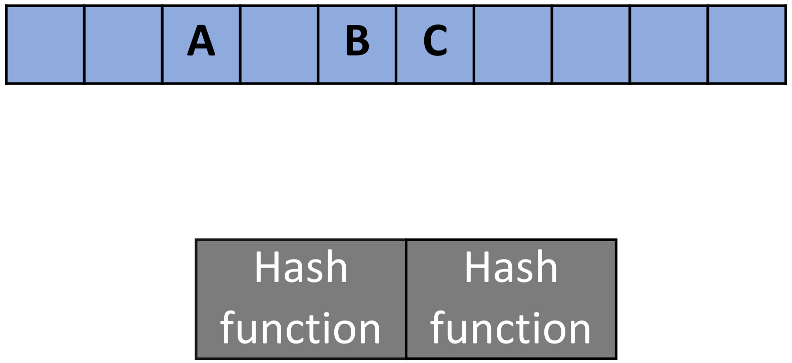
Cuckoo hashing

- Insert(A)
  - One probe (memory access)



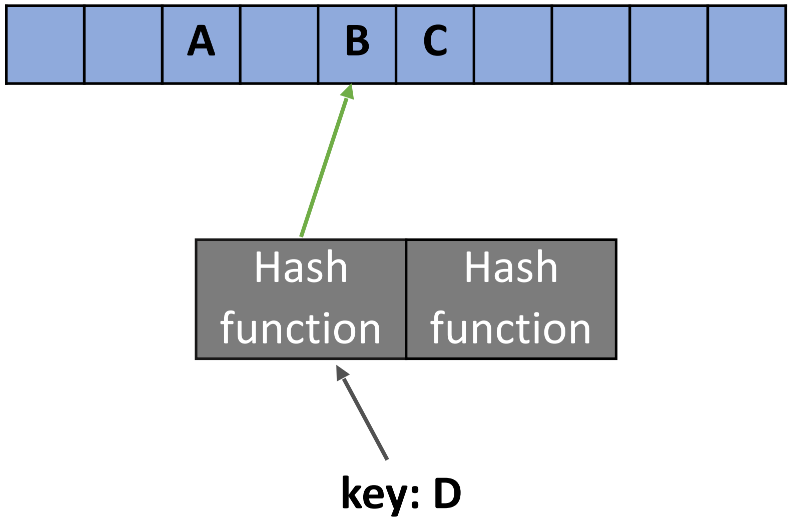
Cuckoo hashing

- Insert(A)
  - One probe (memory access)
- Insert(B)
- Insert(C)



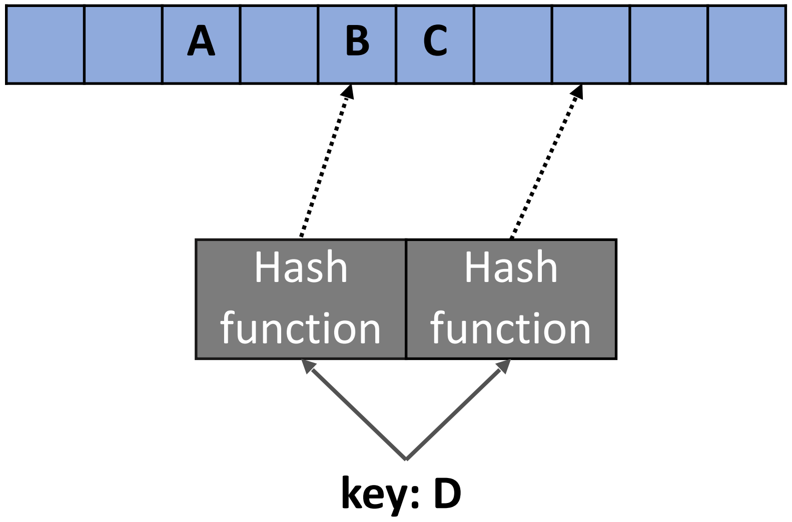
Cuckoo hashing

- Insert(A)
  - One probe (memory access)
- Insert(B)
- Insert(C)
- Insert(D)



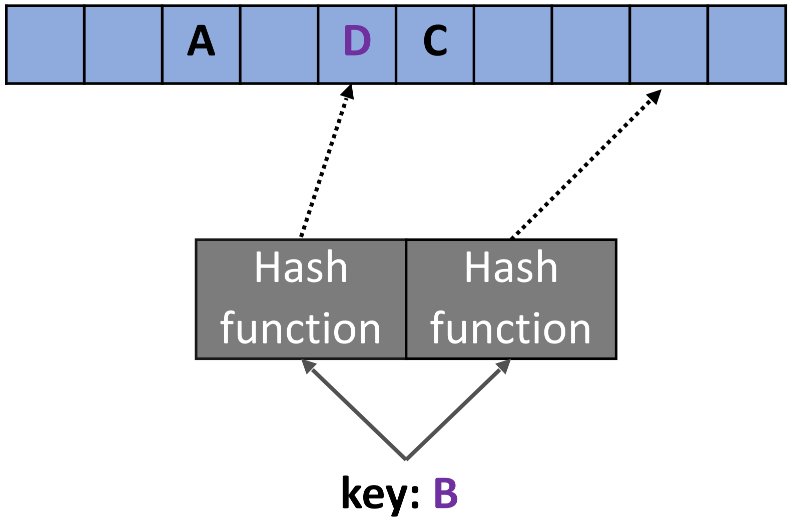
Cuckoo hashing

- Insert(A)
  - One probe (memory access)
- Insert(B)
- Insert(C)
- Insert(D)



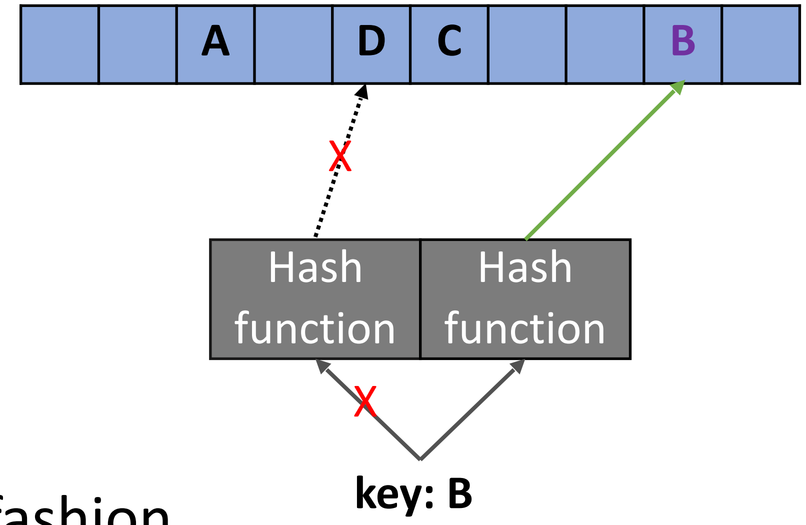
Cuckoo hashing

- Insert(A)
  - One probe (memory access)
- Insert(B)
- Insert(C)
- Insert(D)
  - Exchange **B** with **D**
  - Reinsert(B)



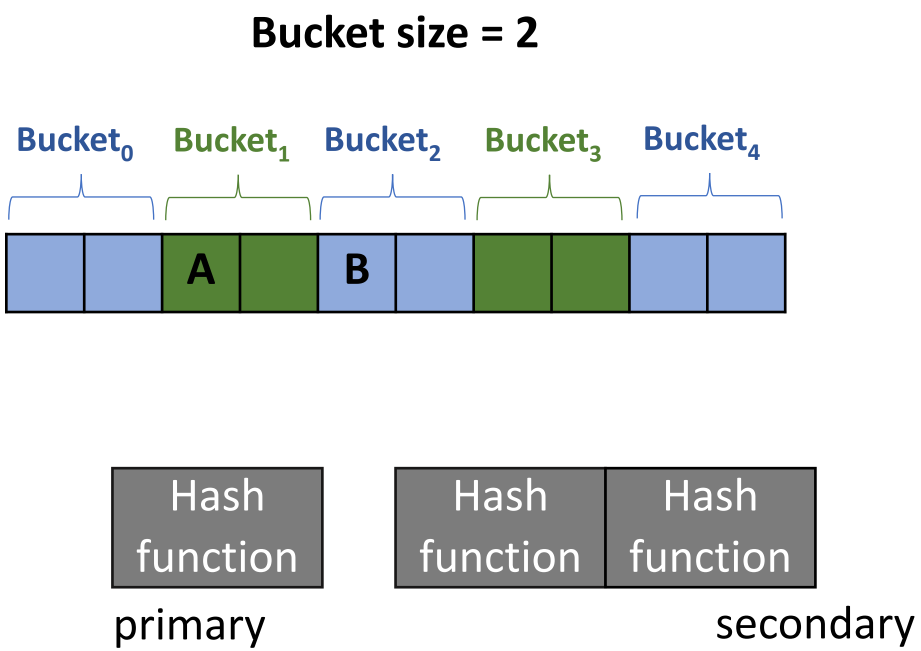
Cuckoo hashing

- Insert(A)
  - One probe (memory access)
- Insert(B)
- Insert(C)
- Insert(D)
  - Exchange **B** with **D**
  - Reinsert(B)
  - Use the two hash functions in a round-robin fashion



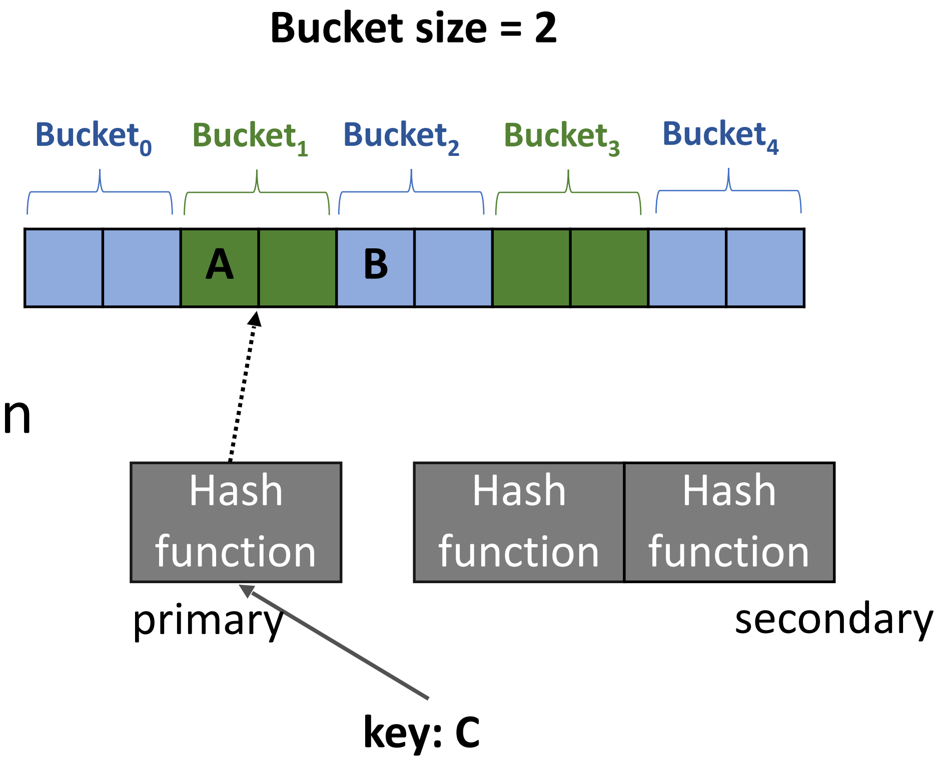
Iceberg hashing

- Uses at least three hash functions
  - On primary hash function
  - Two secondary hash functions



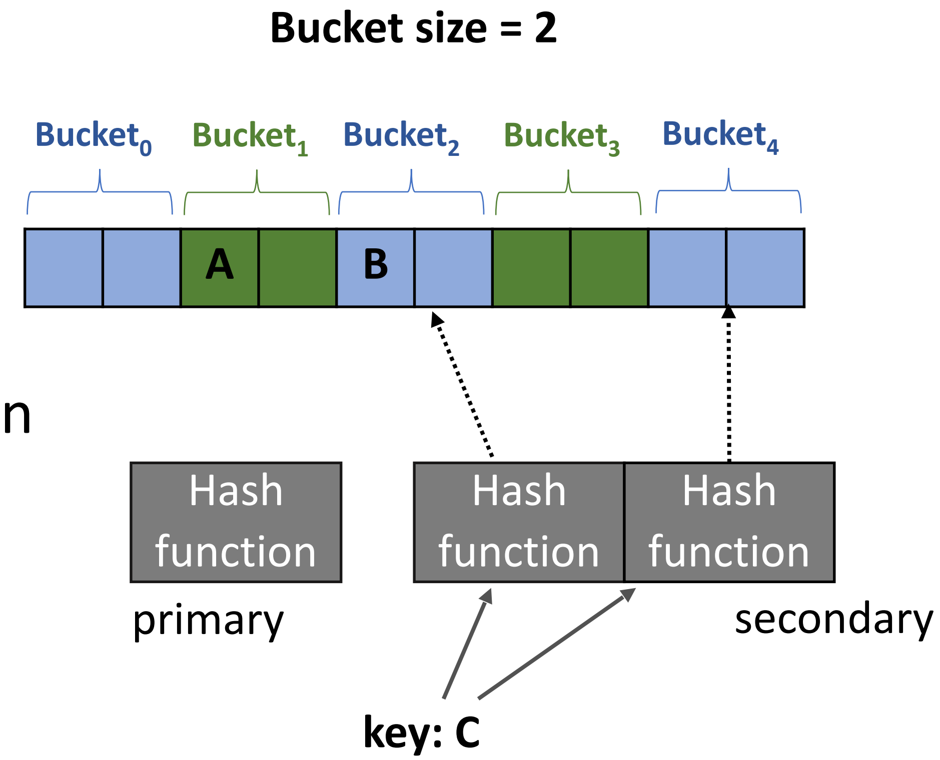
Iceberg hashing

- Uses at least three hash functions
  - On primary hash function
  - Two secondary hash functions
- Insertion:
  - Evaluate the load of the primary hash function
  - If the load is less than a threshold ***t***
    - Insert into the primary bucket



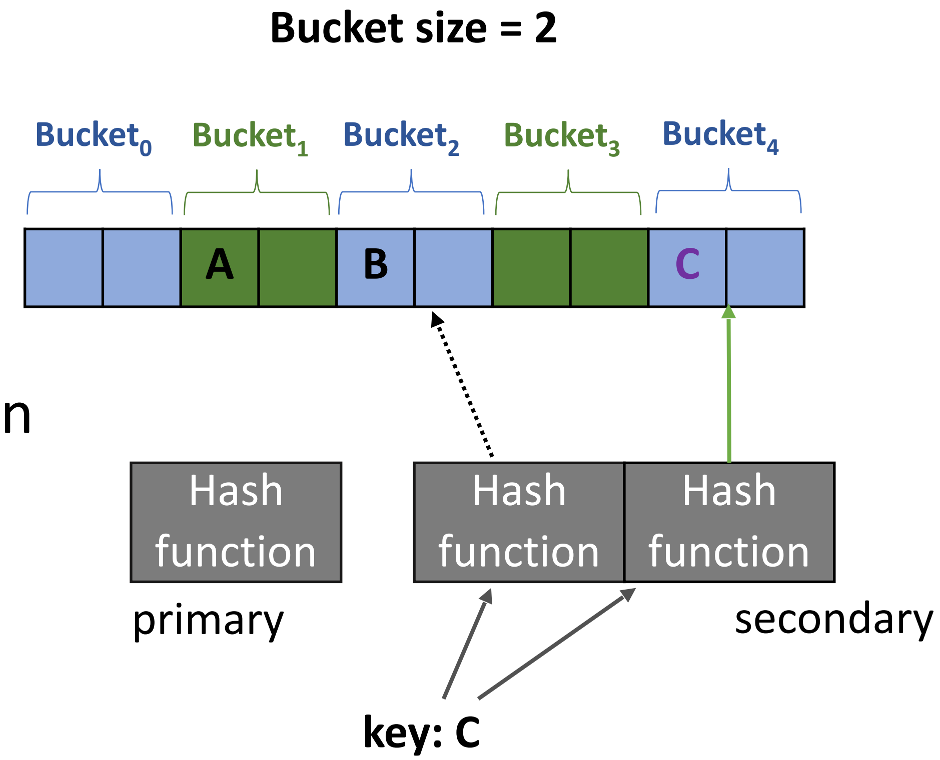
Iceberg hashing

- Uses at least three hash functions
  - On primary hash function
  - Two secondary hash functions
- Insertion:
  - Evaluate the load of the primary hash function
  - If the load is less than a threshold ***t***
    - Insert into the primary bucket
  - Otherwise,
    - Insert into the secondary buckets
      - E.g., using power of two



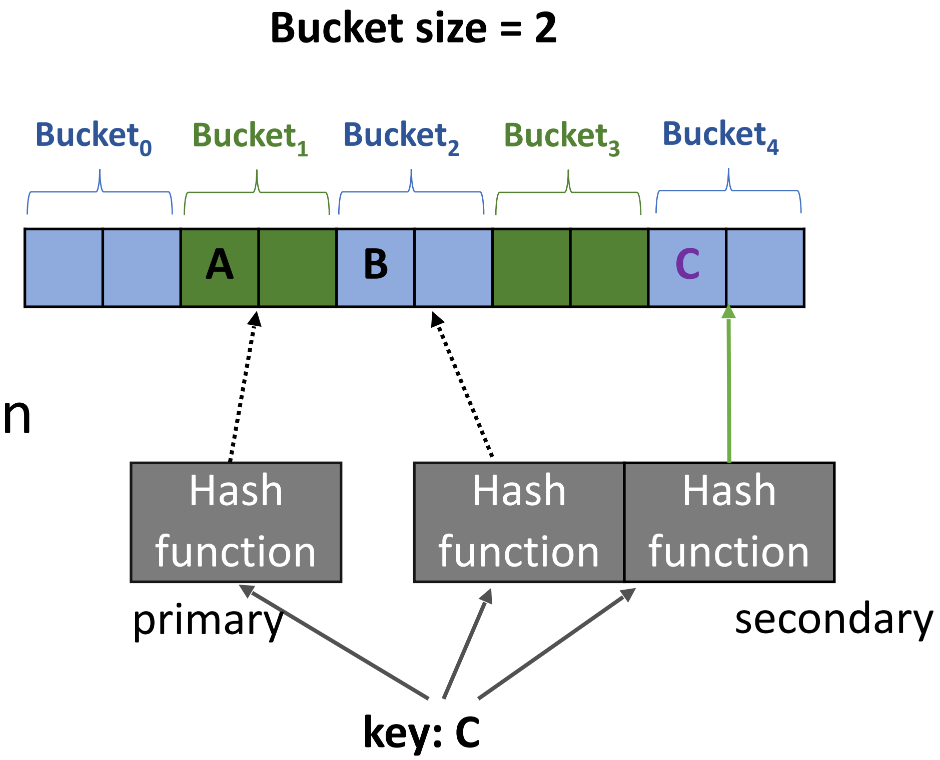
Iceberg hashing

- Uses at least three hash functions
  - On primary hash function
  - Two secondary hash functions
- Insertion:
  - Evaluate the load of the primary hash function
  - If the load is less than a threshold ***t***
    - Insert into the primary bucket
  - Otherwise,
    - Insert into the secondary buckets
      - E.g., using power of two



Iceberg hashing

- Uses at least three hash functions
  - On primary hash function
  - Two secondary hash functions
- Insertion:
  - Evaluate the load of the primary hash function
  - If the load is less than a threshold ***t***
    - Insert into the primary bucket
  - Otherwise,
    - Insert into the secondary buckets
      - E.g., using power of two
- Requires between **one** and **three** probes



Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size
- Probe complexity
  - Number of hash functions
- Placement strategy
  - Balanced or not balanced

Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size

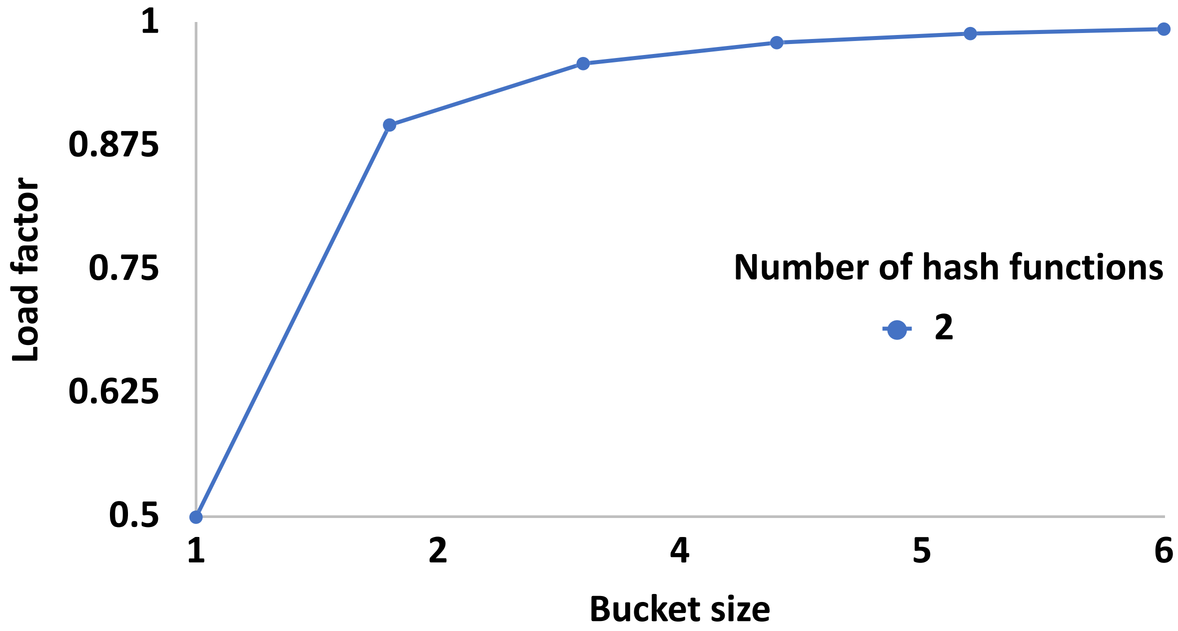


Figure from: **Stefan Walzer,** *Random hypergraphs for hashing-based data structures*. Ph.D. dissertation, TU Ilmenau.

Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size
- Probe complexity
  - Number of hash functions

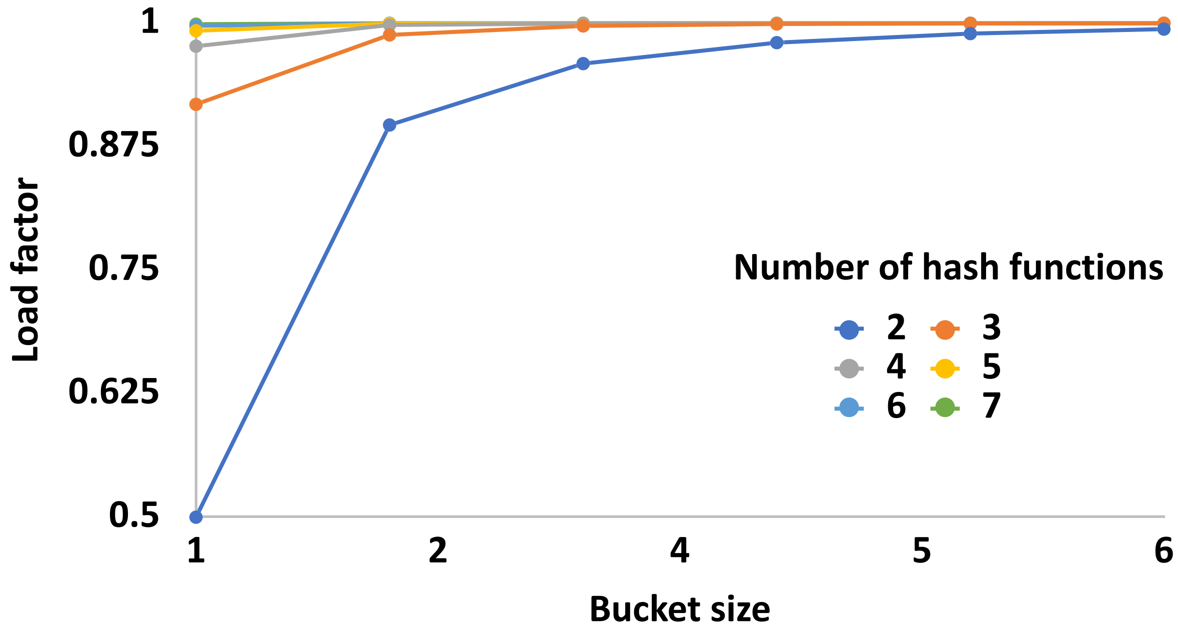


Figure from: **Stefan Walzer,** *Random hypergraphs for hashing-based data structures*. Ph.D. dissertation, TU Ilmenau.

Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size
- Probe complexity
  - Number of hash functions
- Placement strategy
  - Balanced or not balanced

Parameter space landscape

- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size
- Probe complexity
  - Number of hash functions
- Placement strategy
  - Balanced or not balanced

Implementation

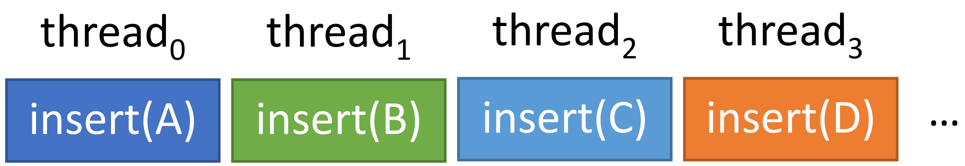
Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {




}
```



Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

}
```

Tile with a size of 4

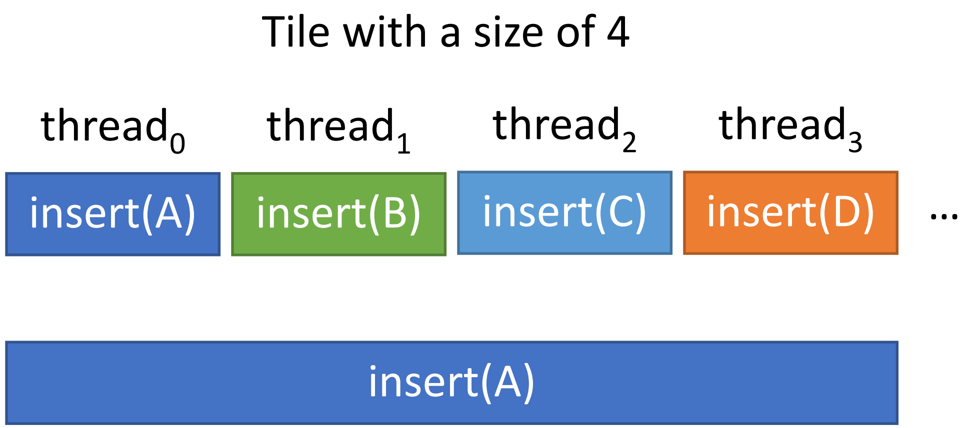
$thread_0$	$thread_1$	$thread_2$	$thread_3$	
insert(A)	insert(B)	insert(C)	insert(D)	…

Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // Perform the insertions
    while (uint32_t work_queue = tile.ballot(to_insert)) {
        auto cur_lane = __ffs(work_queue) - 1;
        auto cur_pair = tile.shfl(pair, cur_lane);
        auto cur_result = insert(tile, cur_pair, table);
        if (tile.thread_rank() == cur_lane) {
            to_insert = false;
            success = cur_result;
        }
    }
    return success;
}
```



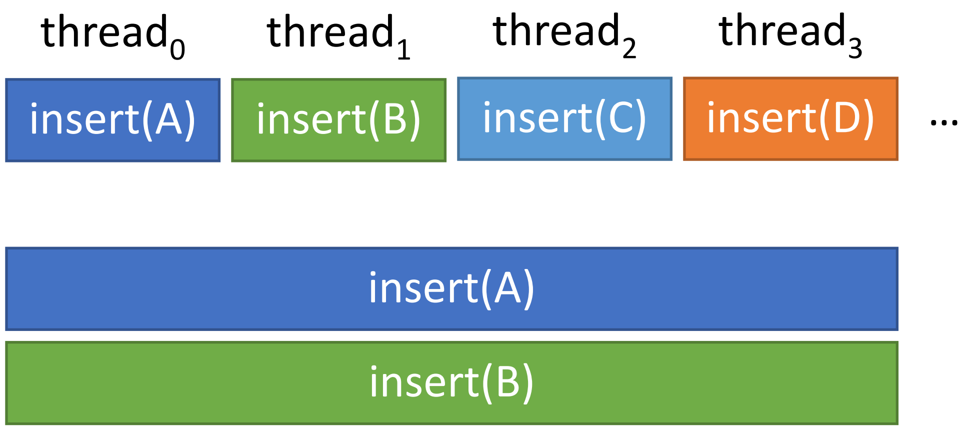
Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // Perform the insertions
    while (uint32_t work_queue = tile.ballot(to_insert)) {
        auto cur_lane = __ffs(work_queue) - 1;
        auto cur_pair = tile.shfl(pair, cur_lane);
        auto cur_result = insert(tile, cur_pair, table);
        if (tile.thread_rank() == cur_lane) {
            to_insert = false;
            success = cur_result;
        }
    }
    return success;
}
```

Tile with a size of 4

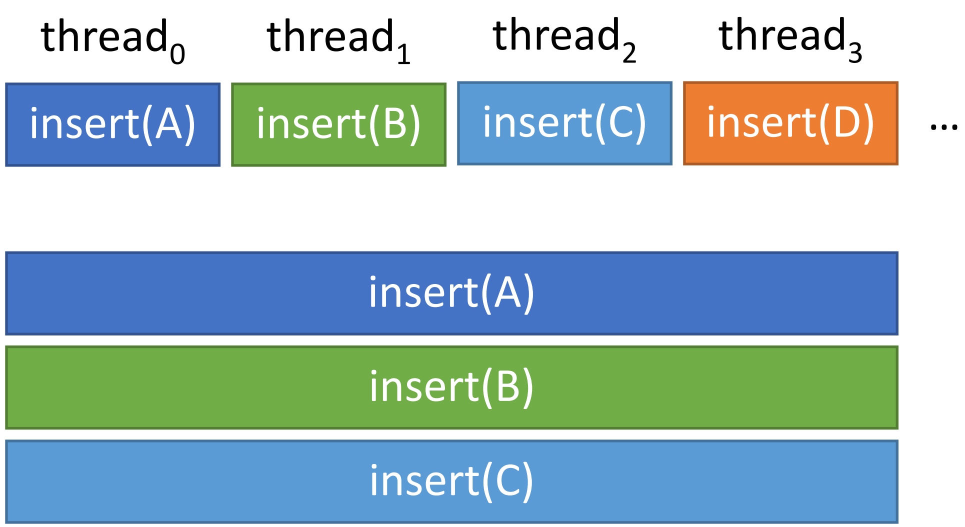


Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // Perform the insertions
    while (uint32_t work_queue = tile.ballot(to_insert)) {
        auto cur_lane = __ffs(work_queue) - 1;
        auto cur_pair = tile.shfl(pair, cur_lane);
        auto cur_result = insert(tile, cur_pair, table);
        if (tile.thread_rank() == cur_lane) {
            to_insert = false;
            success = cur_result;
        }
    }
    return success;
}
```



Tile with a size of 4

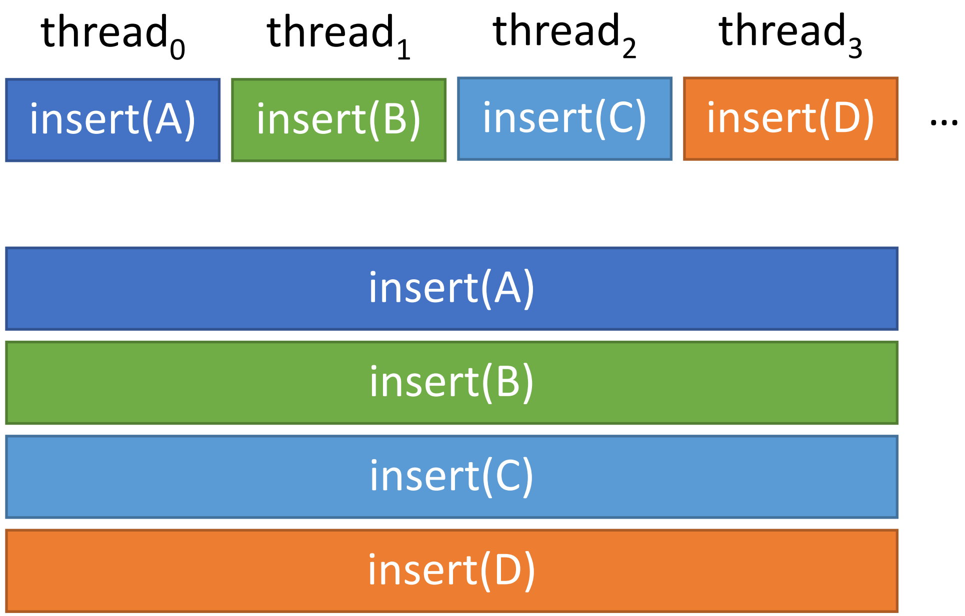
Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // Perform the insertions
    while (uint32_t work_queue = tile.ballot(to_insert)) {
        auto cur_lane = __ffs(work_queue) - 1;
        auto cur_pair = tile.shfl(pair, cur_lane);
        auto cur_result = insert(tile, cur_pair, table);
        if (tile.thread_rank() == cur_lane) {
            to_insert = false;
            success = cur_result;
        }
    }
    return success;
}
```

Tile with a size of 4



Tile-wide cooperative insertion

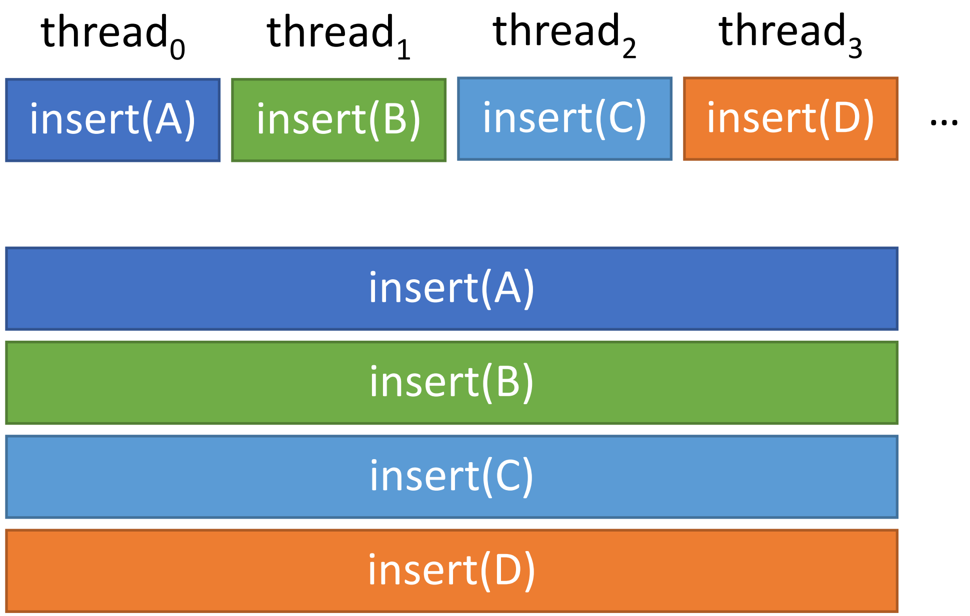
- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing

Tile size (i.e., bucket size) is a parameter in our implementation.

```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // Perform the insertions
    while (uint32_t work_queue = tile.ballot(to_insert)) {
        auto cur_lane = __ffs(work_queue) - 1;
        auto cur_pair = tile.shfl(pair, cur_lane);
        auto cur_result = insert(tile, cur_pair, table);
        if (tile.thread_rank() == cur_lane) {
            to_insert = false;
            success = cur_result;
        }
    }
    return success;
}
```

Tile with a size of 4



Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing
  - Achieves coalesced memory access
    - Using a CUDA C++ abstraction for tile-wide bucket storage.

```
template <typename pair_type, typename tile_type>
struct bucket {
  void load(...);
  int compute_load(...);
  int find_key_location(...);
  pair_type::second_type get_value_from_lane(...);
  bool weak_cas_at_location(...);
  bool strong_cas_at_location(...);
  pair_type exch_at_location(...);

 private:
  pair_type lane_pair_;
  tile_type tile_;
};
```

Tile-wide cooperative insertion

- Our efficient implementation:
  - Avoids branch divergence
    - Using tile-cooperative processing
  - Achieves coalesced memory access
    - Using a CUDA C++ abstraction for tile-wide bucket storage.

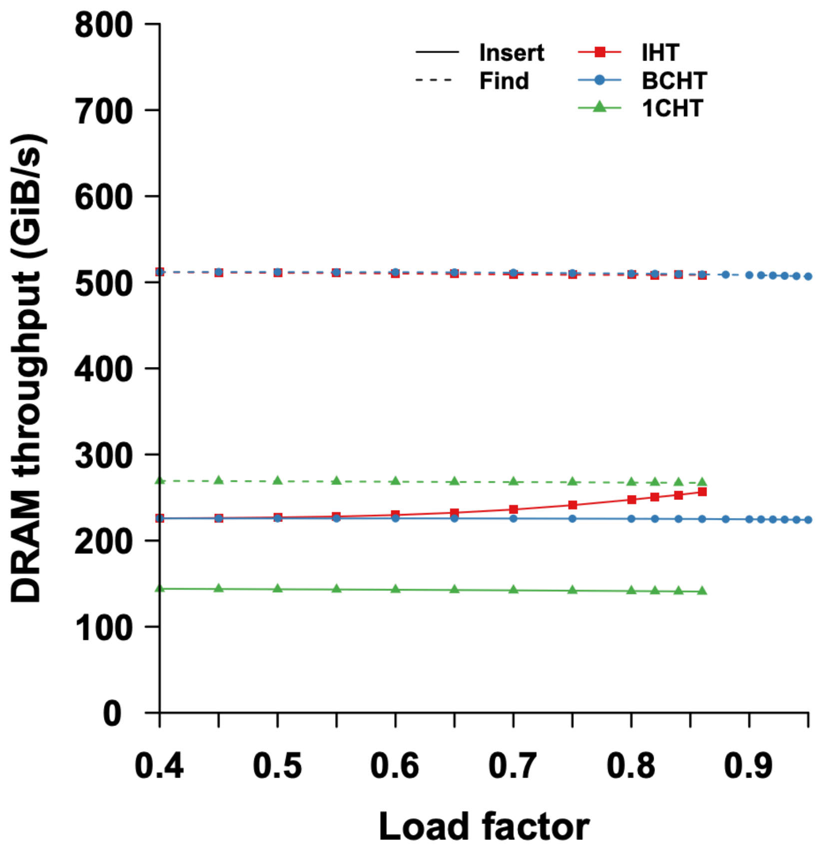
Probing schemes are implemented on top of the bucket abstraction.

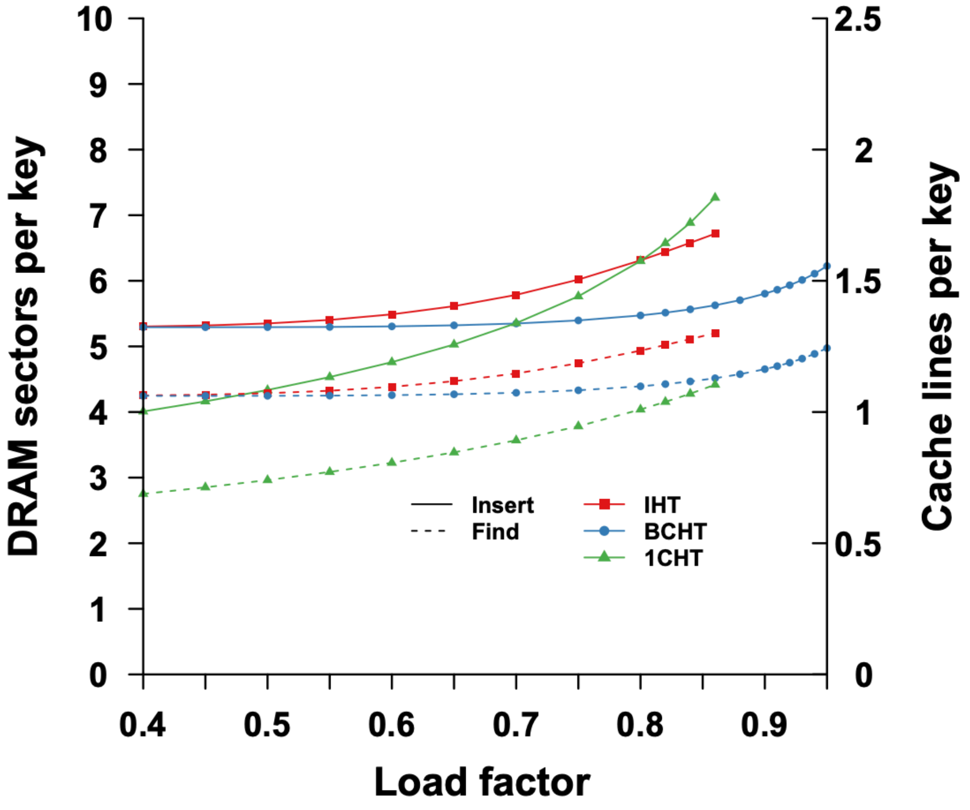
```
template <typename pair_type, typename tile_type>
struct bucket {
  void load(...);
  int compute_load(...);
  int find_key_location(...);
  pair_type::second_type get_value_from_lane(...);
  bool weak_cas_at_location(...);
  bool strong_cas_at_location(...);
  pair_type exch_at_location(...);

 private:
  pair_type lane_pair_;
  tile_type tile_;
};
```

Results

How efficient is our implementation?

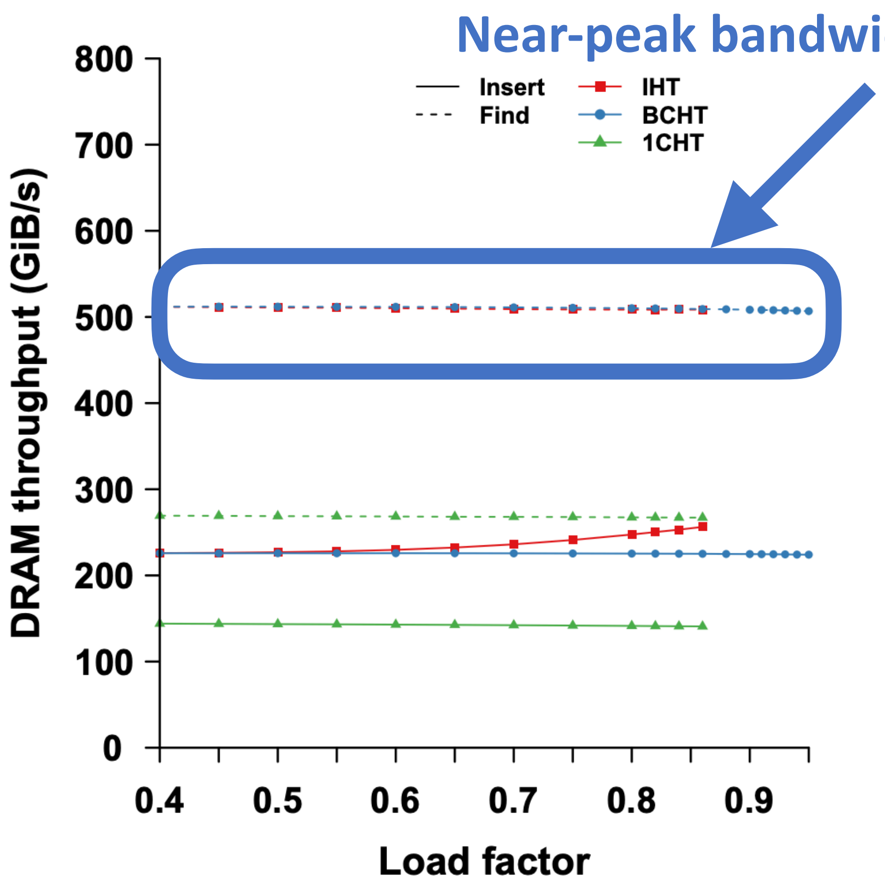


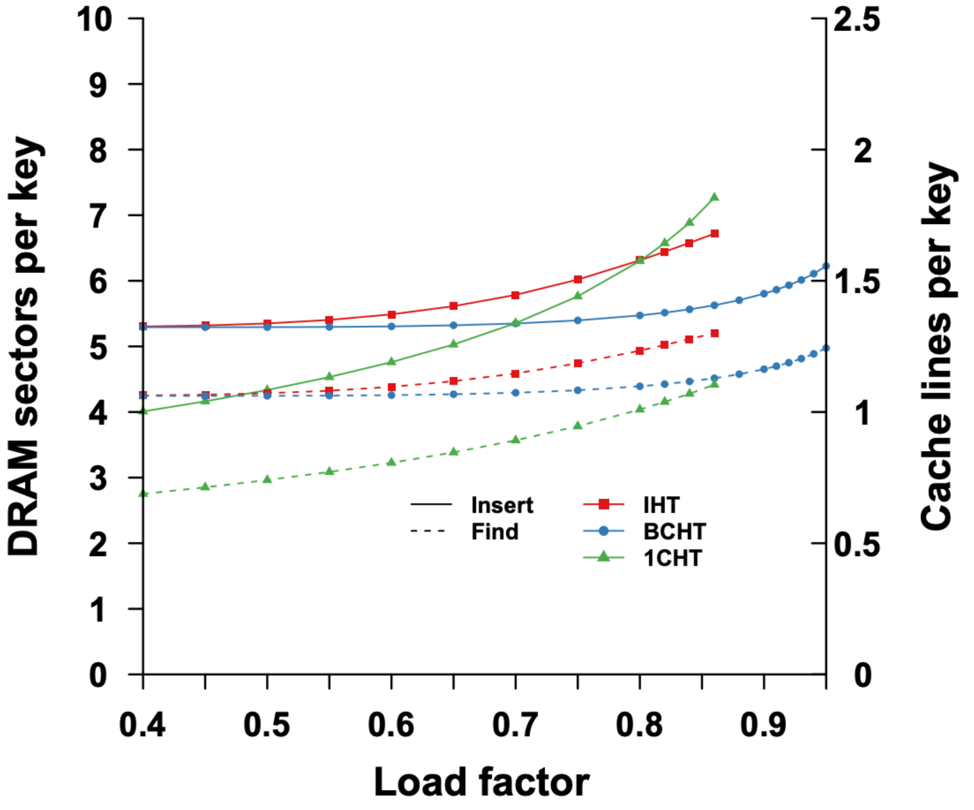


IHT → *Iceberg HT, b = 16*
1CHT → *Cuckoo HT, b = 1*
BCHT → *Cuckoo HT, b = 16*

Sector is 32 bytes (i.e., a cache line = 4 sectors)
GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

How efficient is our implementation?

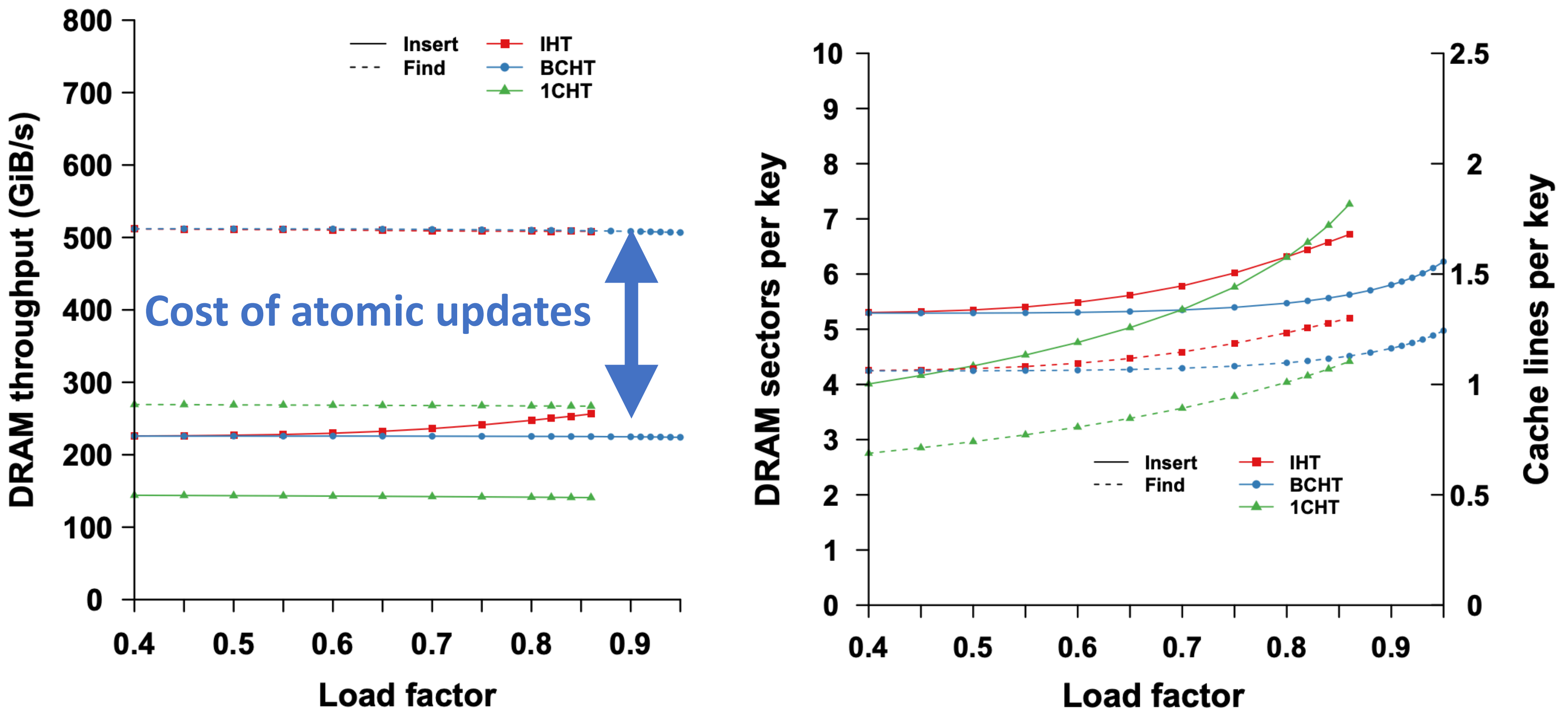




IHT → *Iceberg HT, b = 16*
1CHT → *Cuckoo HT, b = 1*
BCHT → *Cuckoo HT, b = 16*

Sector is 32 bytes (i.e., a cache line = 4 sectors)
GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

How efficient is our implementation?



IHT → *Iceberg HT, b = 16*

1CHT → *Cuckoo HT, b = 1*

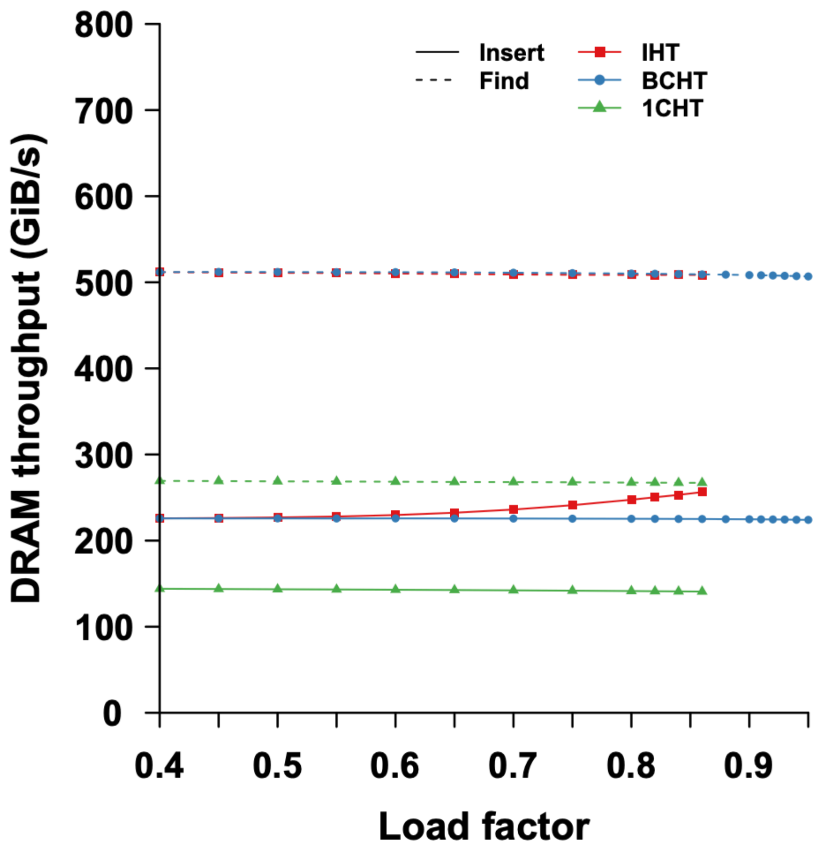
BCHT → *Cuckoo HT, b = 16*

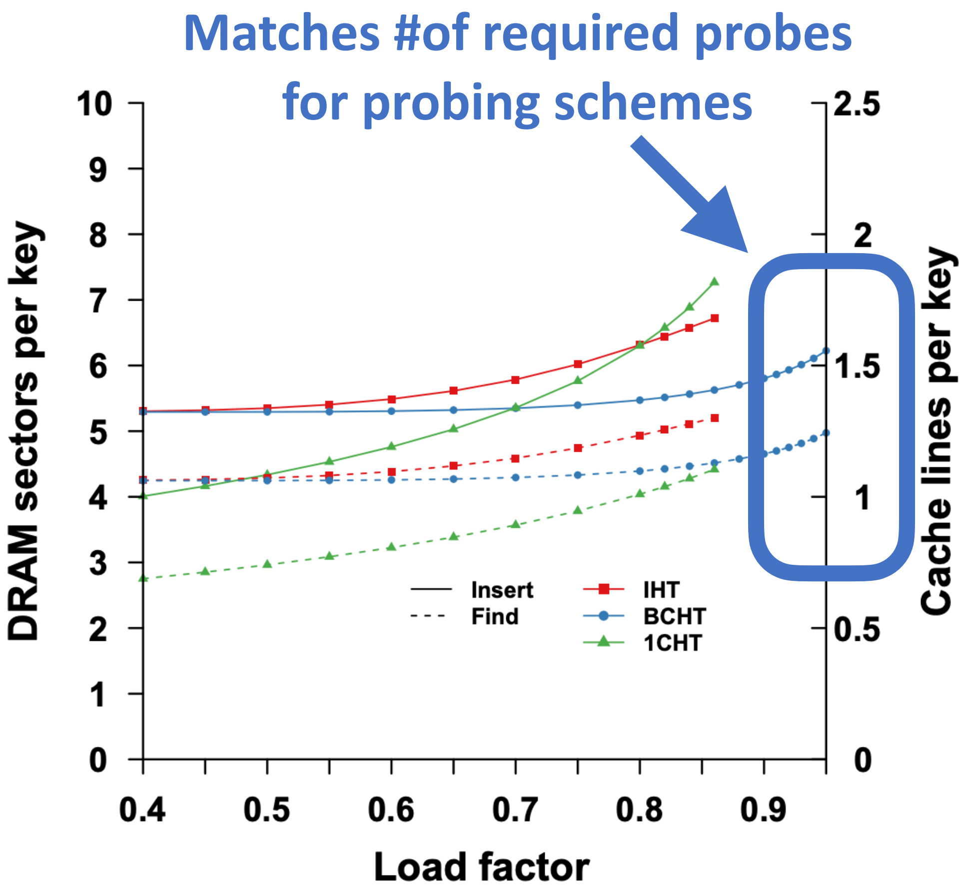
Sector is 32 bytes (i.e., a cache line = 4 sectors)

GPU is TITAN V, peak bandwidth 652.8 GB/s

4 bytes keys (uniform random), 4 bytes values

How efficient is our implementation?



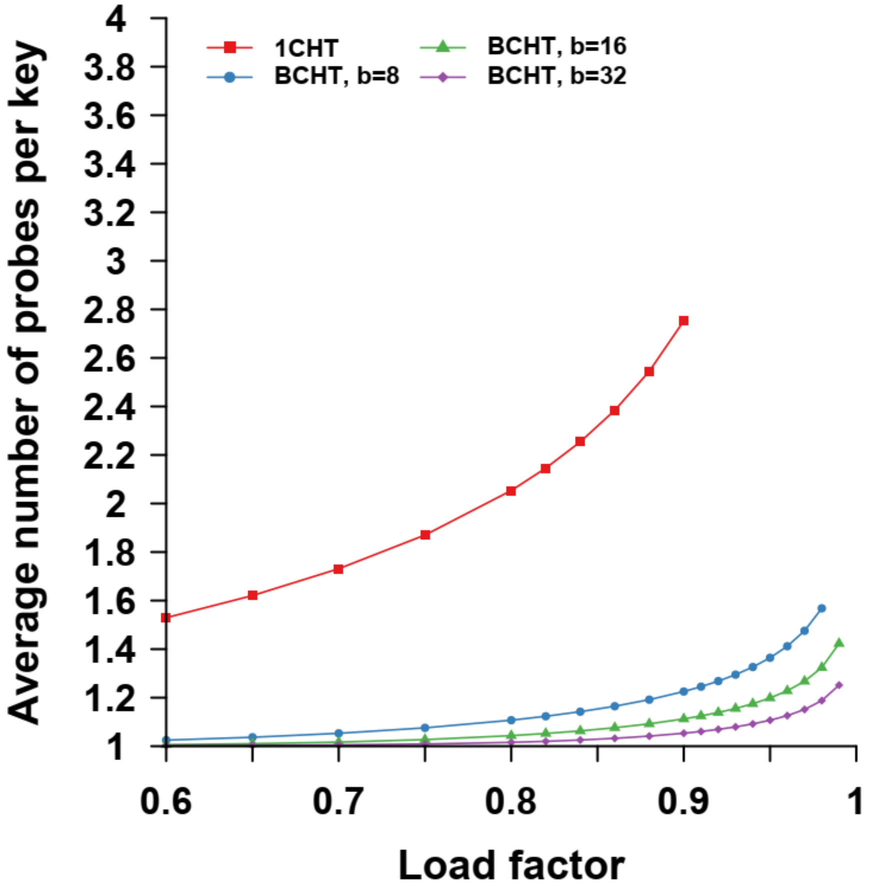


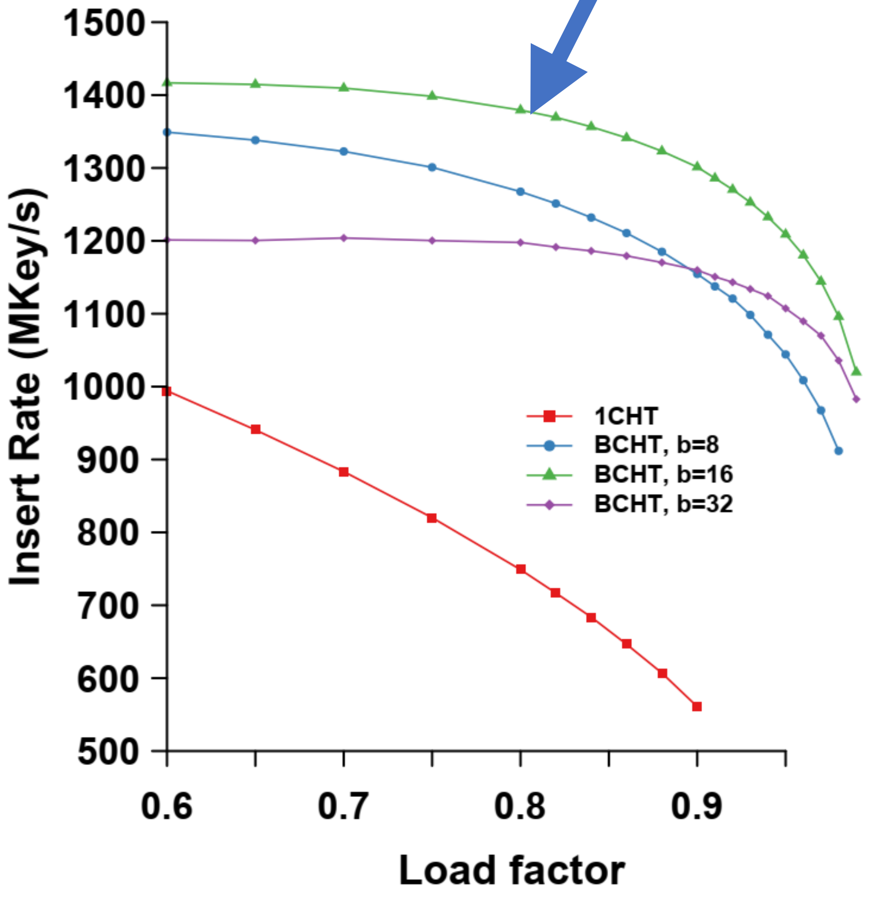
IHT → *Iceberg HT, b = 16*
1CHT → *Cuckoo HT, b = 1*
BCHT → *Cuckoo HT, b = 16*

Sector is 32 bytes (i.e., a cache line = 4 sectors)
GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

BCHT insertion performance

b=16 is the optimal bucket size on TITAN V GPU



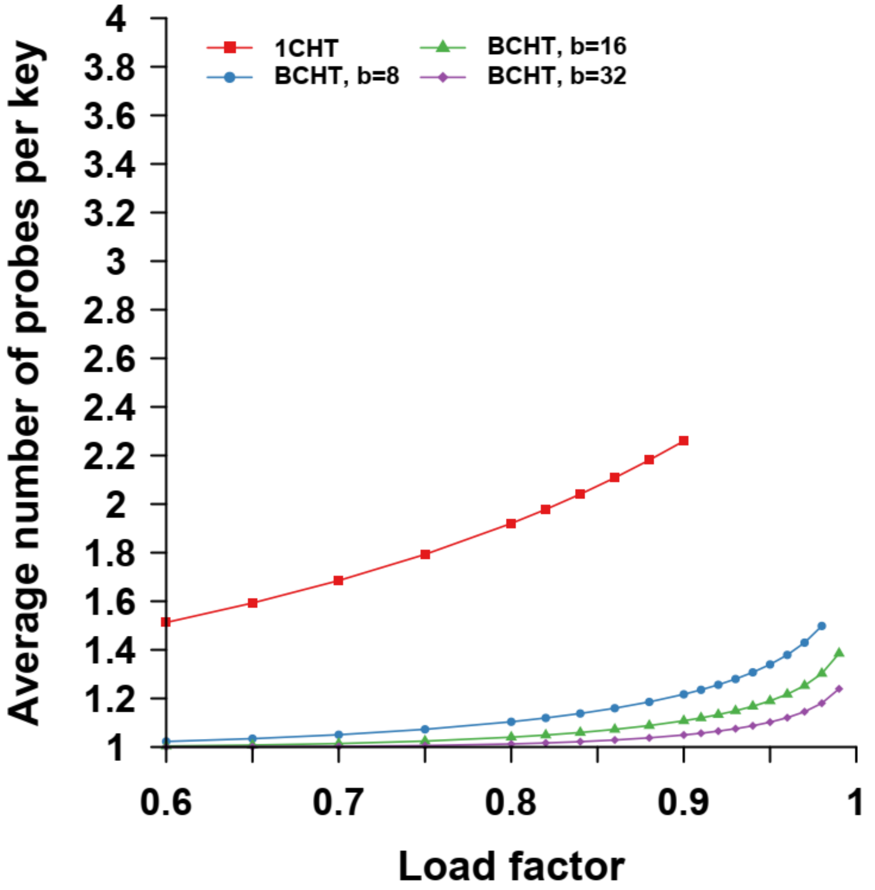


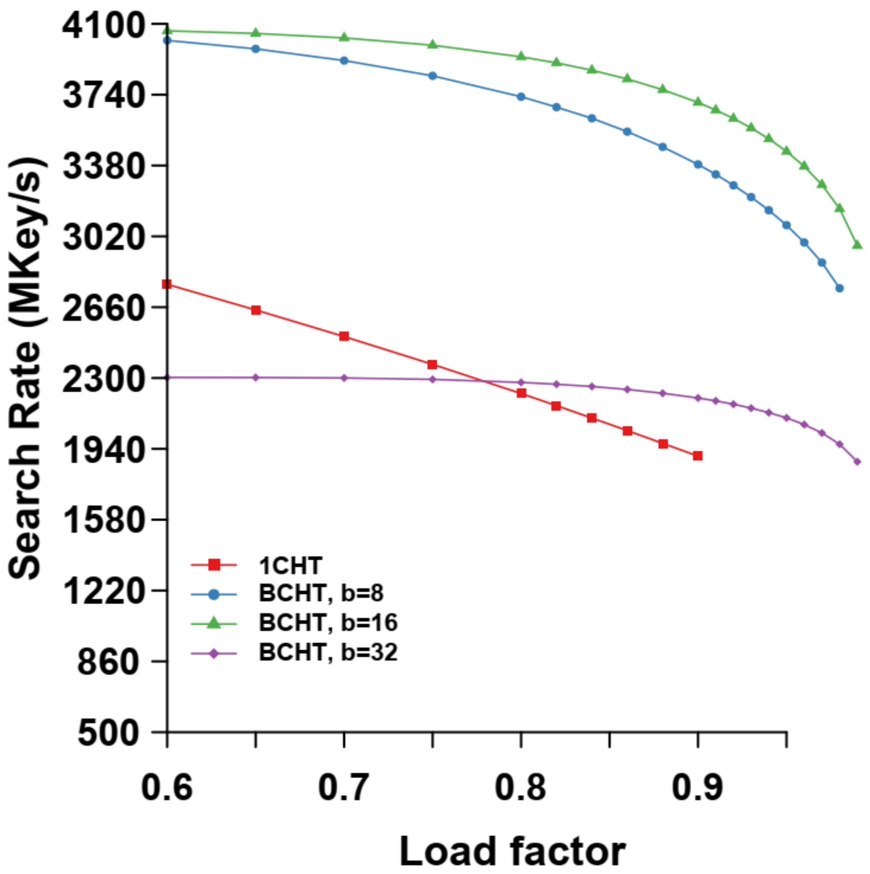
1CHT → *Cuckoo HT, b = 1*

BCHT → *Cuckoo HT, b = 8, 16, 32*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

BCHT find performance (positive queries)



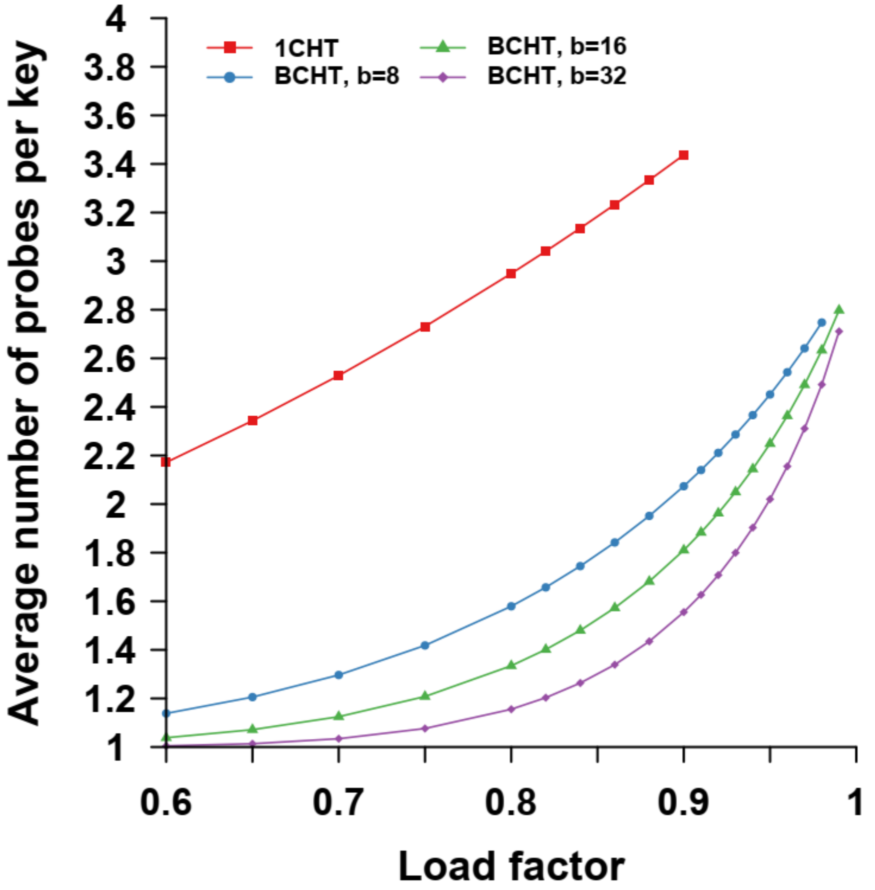


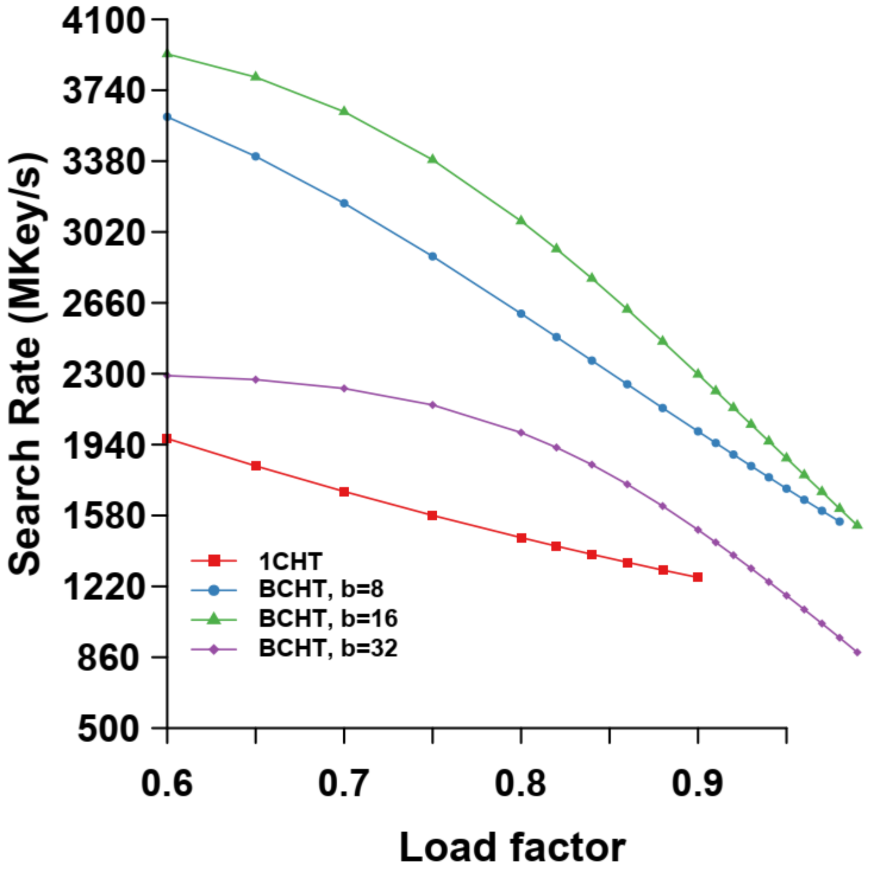
1CHT → *Cuckoo HT, b = 1*

BCHT → *Cuckoo HT, b = 8, 16, 32*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

BCHT find performance (negative queries)



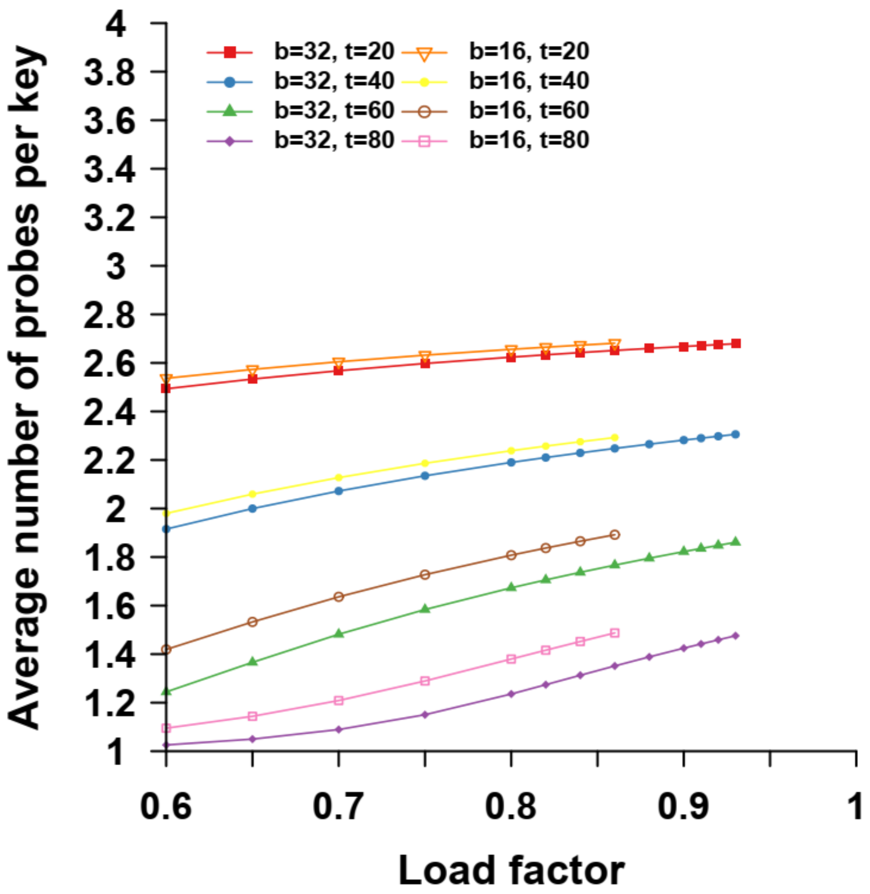


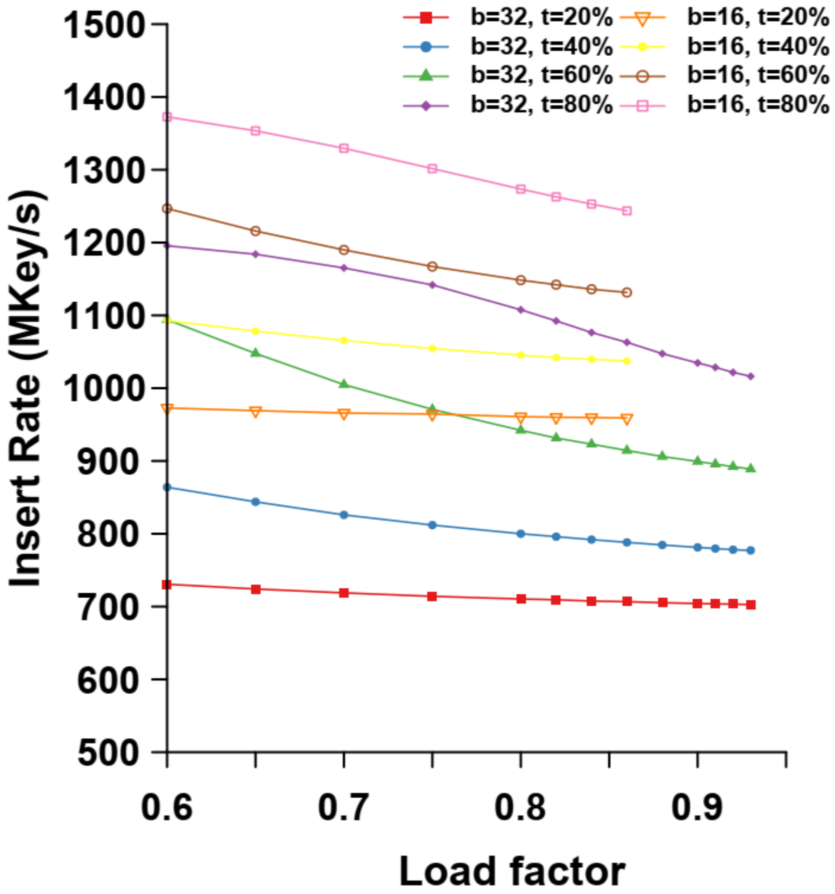
1CHT → *Cuckoo HT, b = 1*

BCHT → *Cuckoo HT, b = 8, 16, 32*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

IHT insertion performance

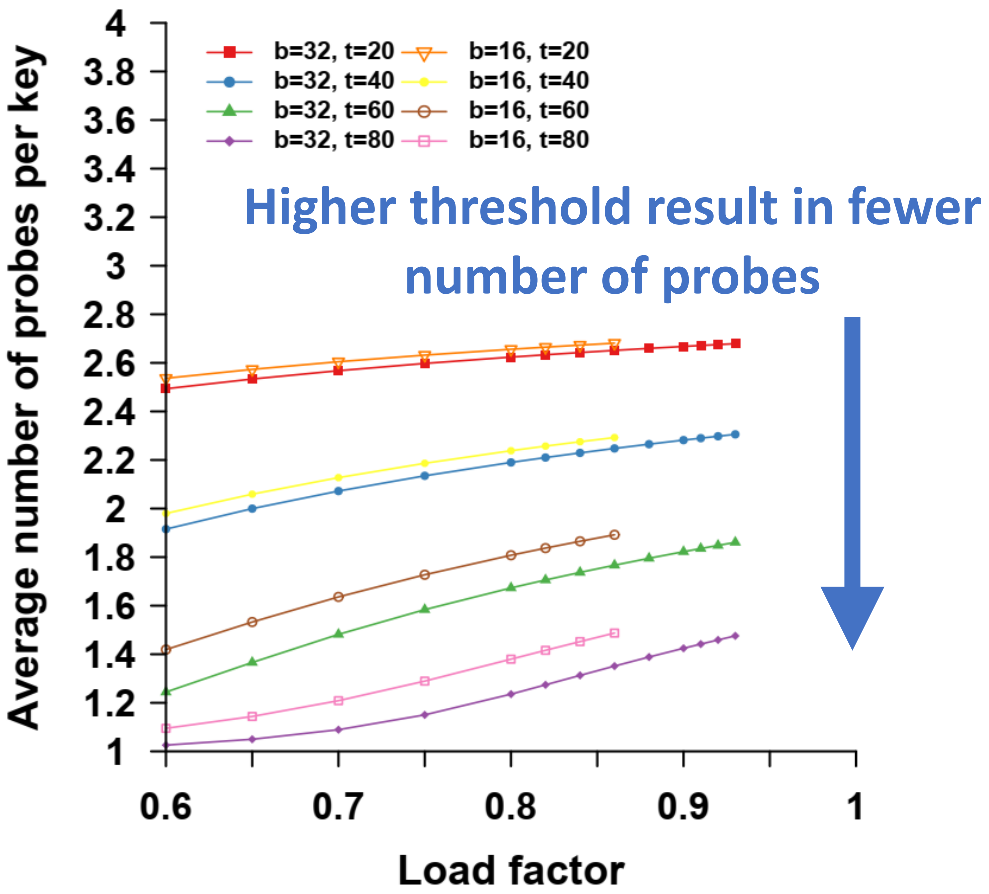


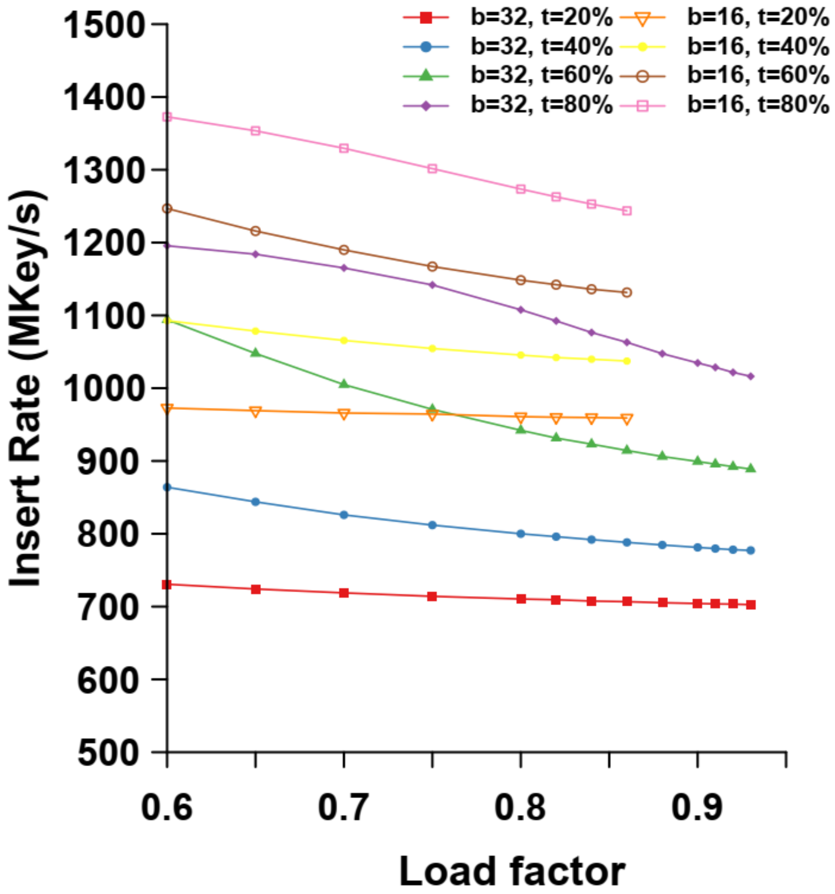


IHT → *Iceberg HT, b = 16, 32, threshold t*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

IHT insertion performance

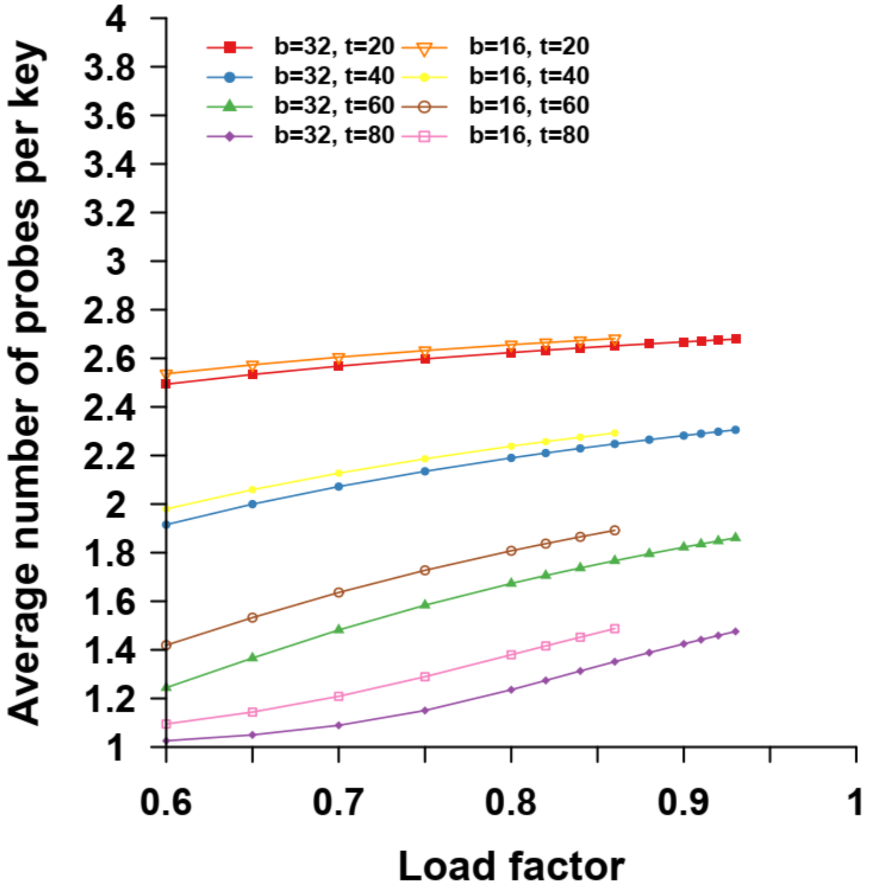


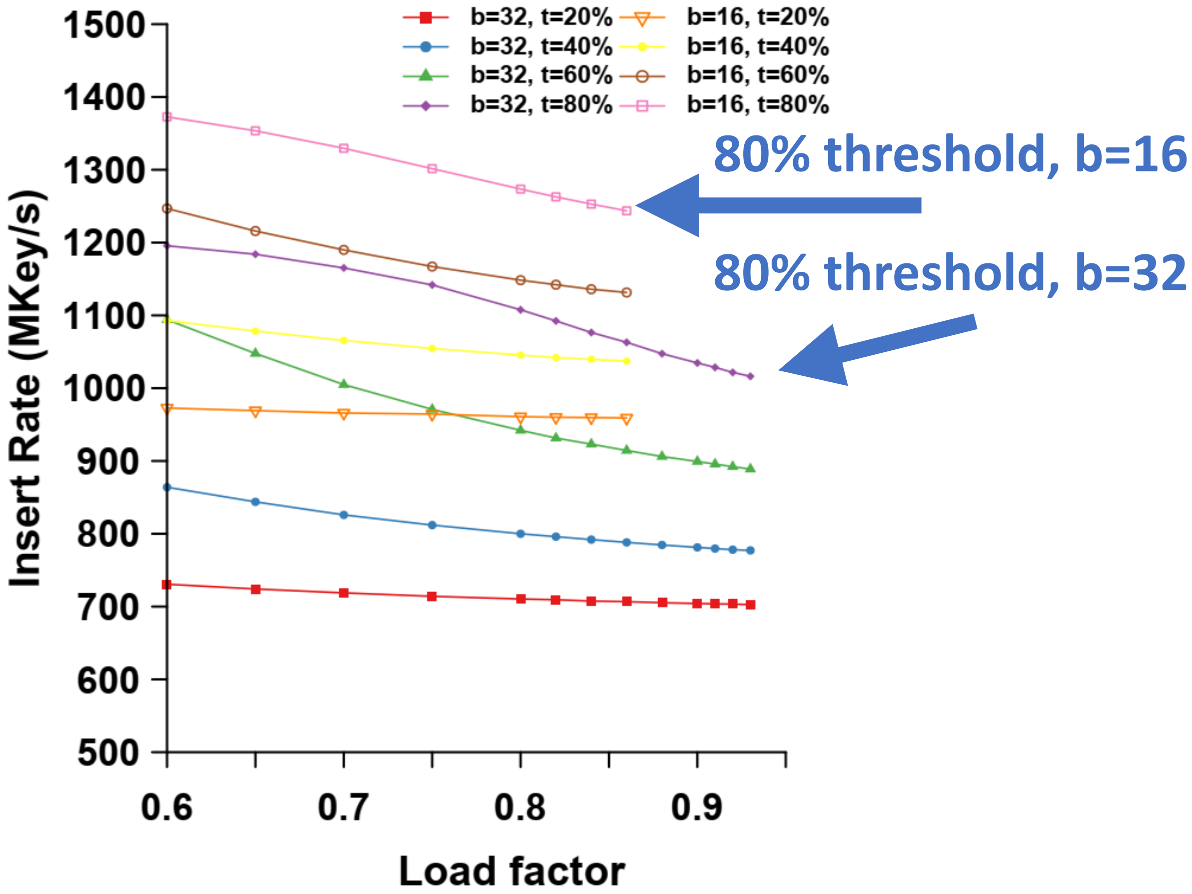


IHT → *Iceberg HT, b = 16, 32, threshold t*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

IHT insertion performance

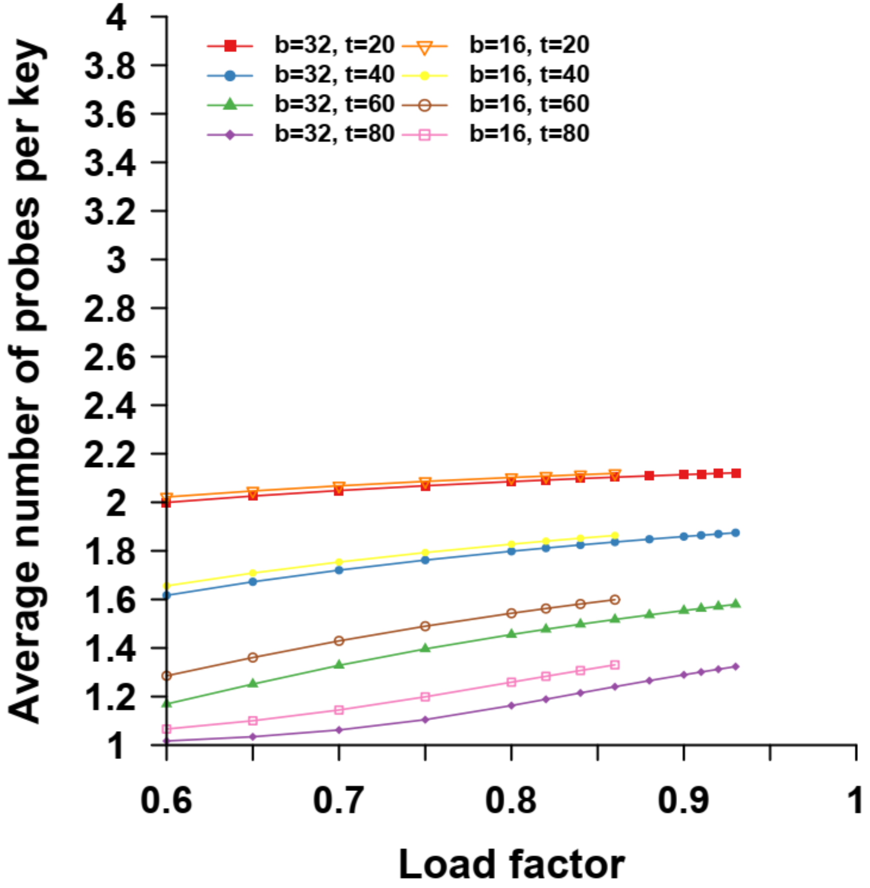


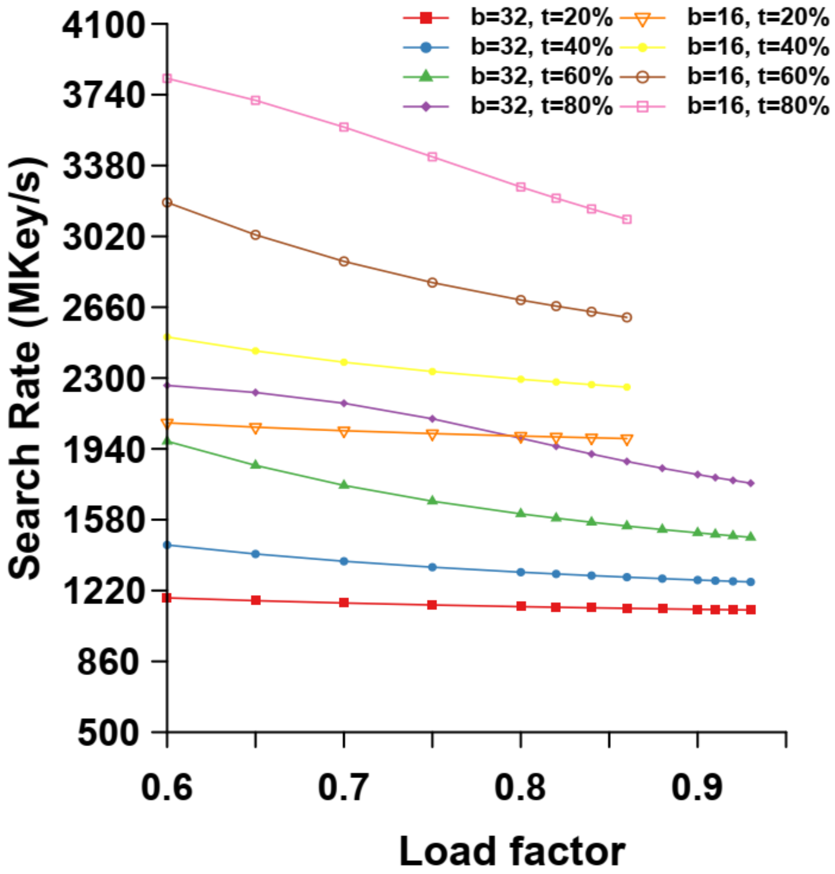


IHT → *Iceberg HT, b = 16, 32, threshold t*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

IHT find performance (positive queries)

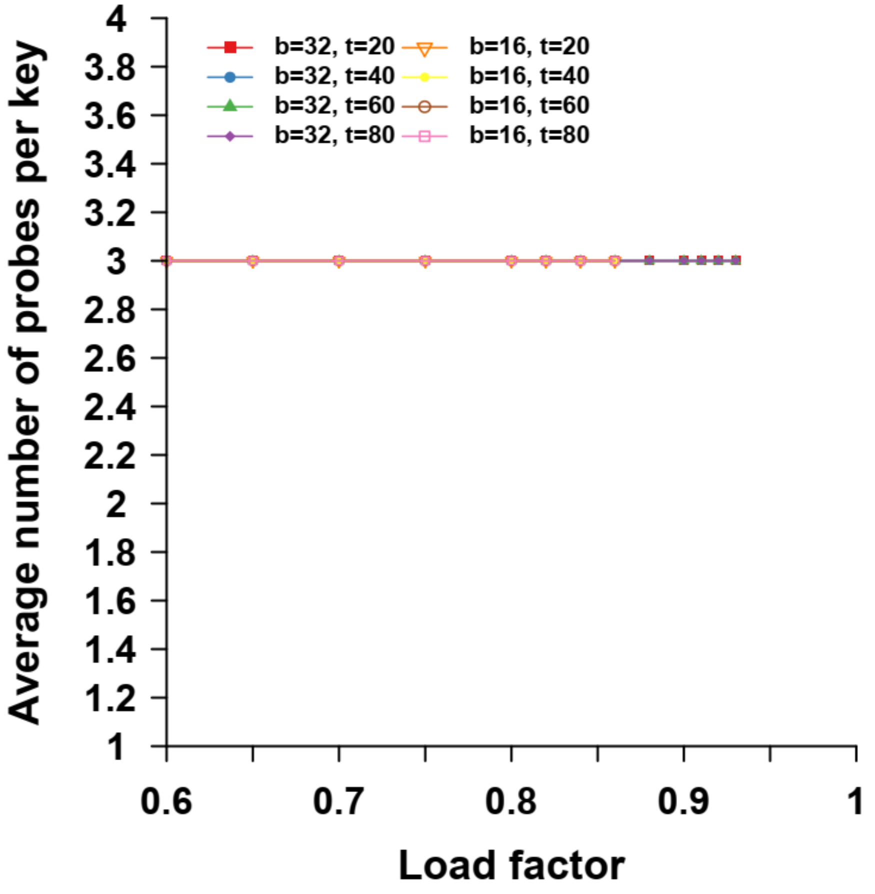


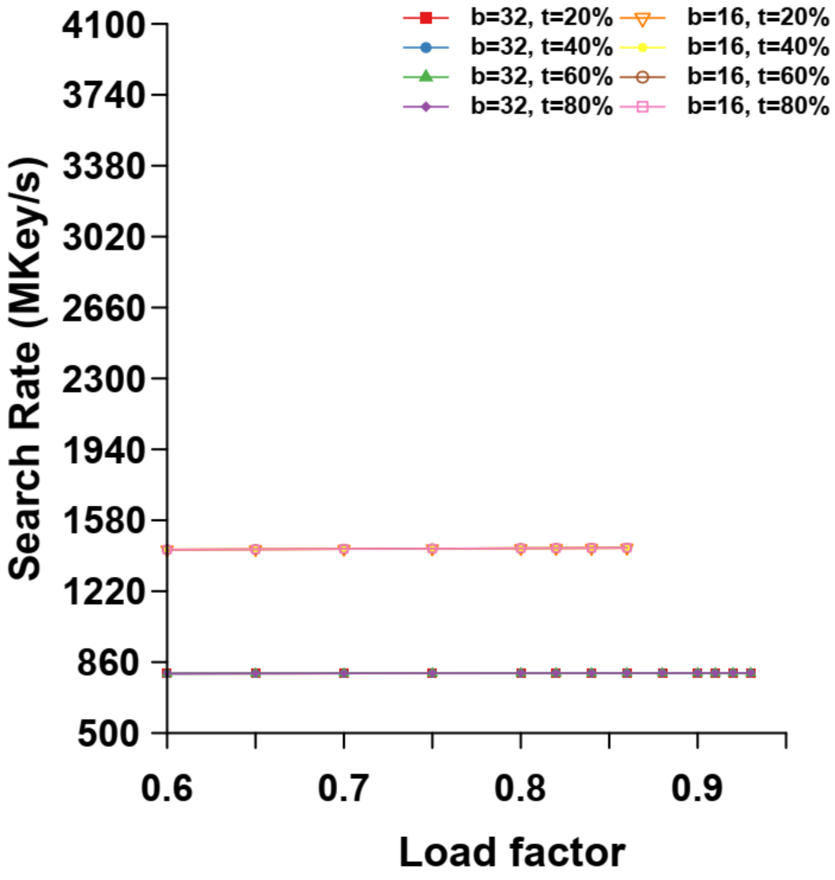


IHT → *Iceberg HT, b = 16, 32, threshold t*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

IHT find performance (negative queries)

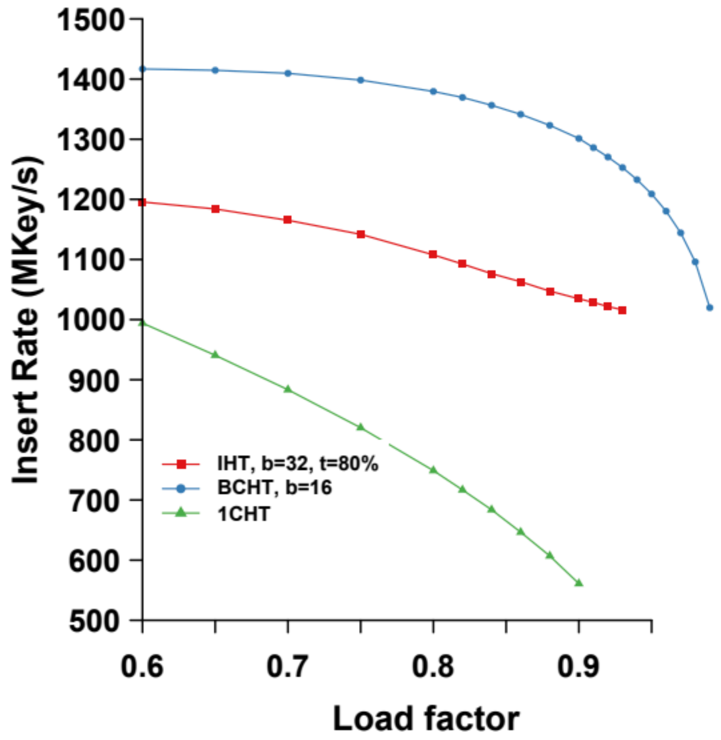


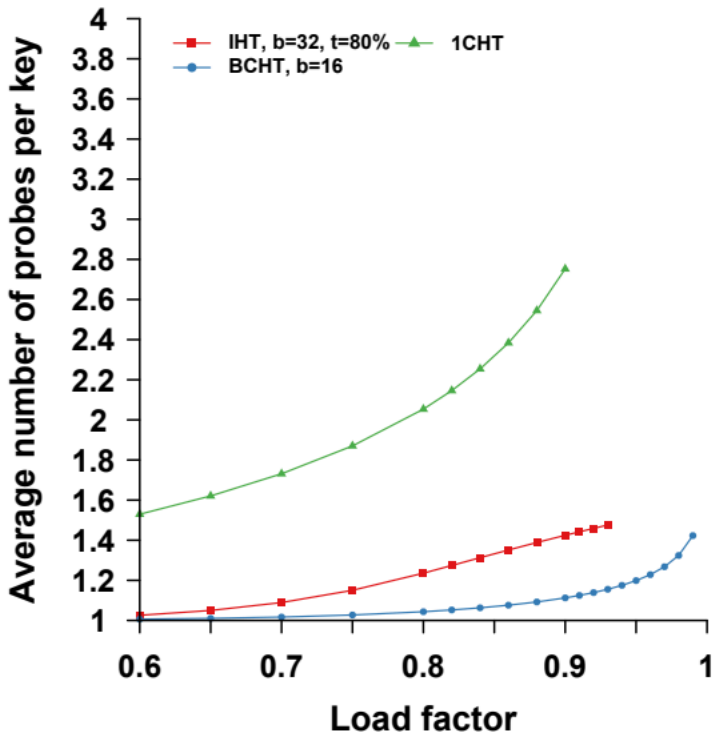


IHT → *Iceberg HT, b = 16, 32, threshold t*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

Query and build rates (across implementations)





IHT → *Iceberg HT, b = 32*

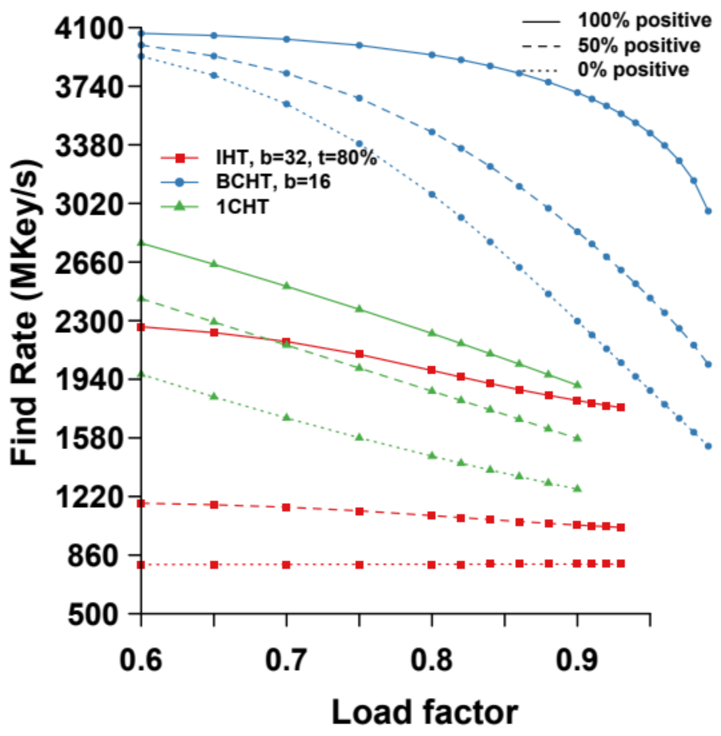
1CHT → *Cuckoo HT, b = 1*

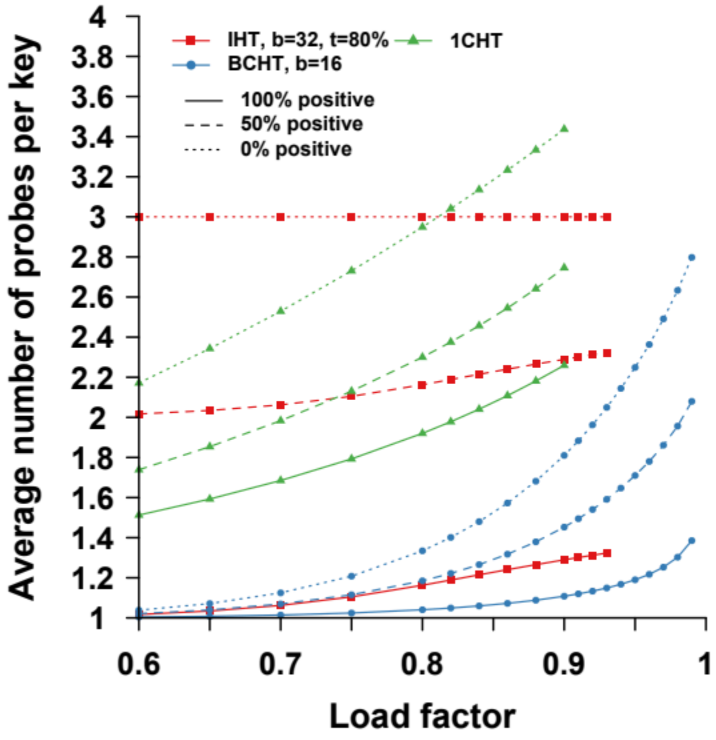
BCHT → *Cuckoo HT, b = 16*

"best" from each family

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

Query and build rates (across implementations)





IHT → *Iceberg HT, b = 32*

1CHT → *Cuckoo HT, b = 1*

BCHT → *Cuckoo HT, b = 16*

"best" from each family

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

Summary and future work

- It's all about the number of probes
- Bucketed techniques are suitable for the GPU
  - Optimal bucket size is 128 bytes
  - Larger buckets: higher load factors
- Increasing the number of hash functions
  - Higher load factors
  - Lower negative query rates and higher insertion rates
- We have choices
  - What does the workload require?

Method	Load factor	Insertion Probes	Query Probes	Stability
1CHT	0.88	≈ 2.8	up to 4	no
BCHT	0.98	≈ 1.8	up to 3	no
IHT	0.92	1 or 3	up to 3	yes

Properties for different hash tables

	Load factor	Insertion	Query
Insertion	BCHT, $b = 16$	—	—
Query	BCHT, $b = 16$	BCHT, $b = 16$	—
Stability	IHT, $b = 32$	IHT, $b = 16$	IHT, $b = 16$

Recommendations

Summary and future work

- It’s all about the number of probes
- Bucketed techniques are suitable for the GPU
  - Optimal bucket size is 128 bytes
  - Larger buckets: higher load factors
- Increasing the number of hash functions
  - Higher load factors
  - Lower negative query rates and higher insertion rates
- We have choices
  - What does the workload require?

- Iceberg hashing:
  - High load factors: different secondary probing scheme
  - Improve negative queries performance: use a quotient filter

Thank you!

Our implementation is on GitHub.

- Supports custom/32/64-bit keys and values

"Analyzing and Implementing GPU Hash Tables", *APOCS 2023*.
https://github.com/owensgroup/BGHT

Our other GPU data structures work:

"A GPU Multiversion B-Tree", *PACT 2022.*
https://github.com/owensgroup/MVGpuBTree
"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/tree/dynamic-graph
"Fully Concurrent GPU Data Structures", *Ph.D. dissertation, UC Davis, 2022.*
https://escholarship.org/uc/item/5kc834wm

Acknowledgments

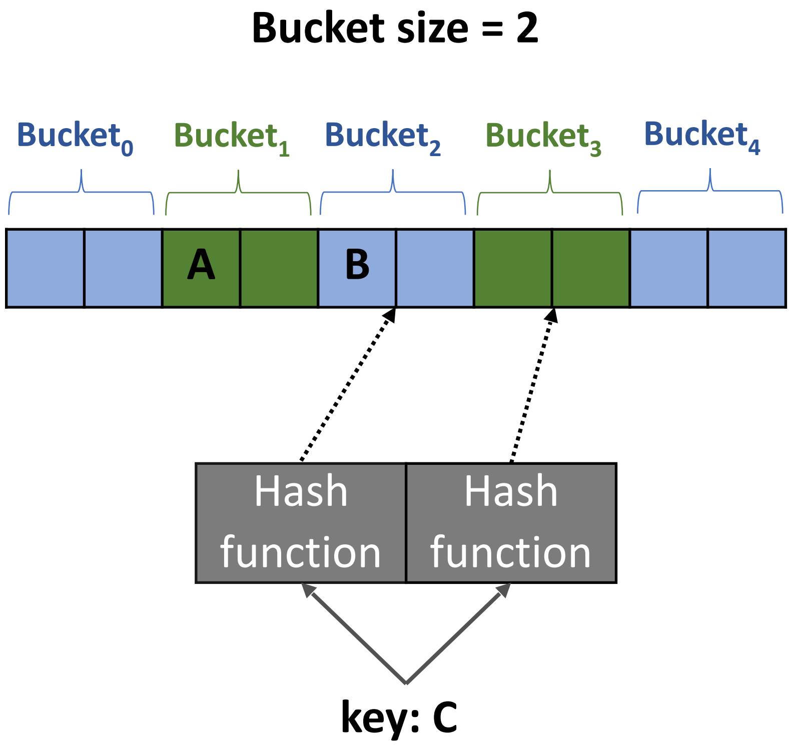
- Martin Dietzfelbinger, *TU Ilmenau*
- Lars Nyland, *NVIDIA*
- Alex Conway, *VMWare Research*

Summary and future work

- It’s all about the number of probes
- Bucketed techniques are suitable for the GPU
  - Optimal bucket size is 128 bytes
  - Larger buckets: higher load factors
- Increasing the number of hash functions
  - Higher load factors
  - Lower negative query rates and higher insertion rates
- We have choices
  - What does the workload require?
- Iceberg hashing:
  - High load factors: different secondary probing scheme
  - Improve negative queries performance: use a quotient filter

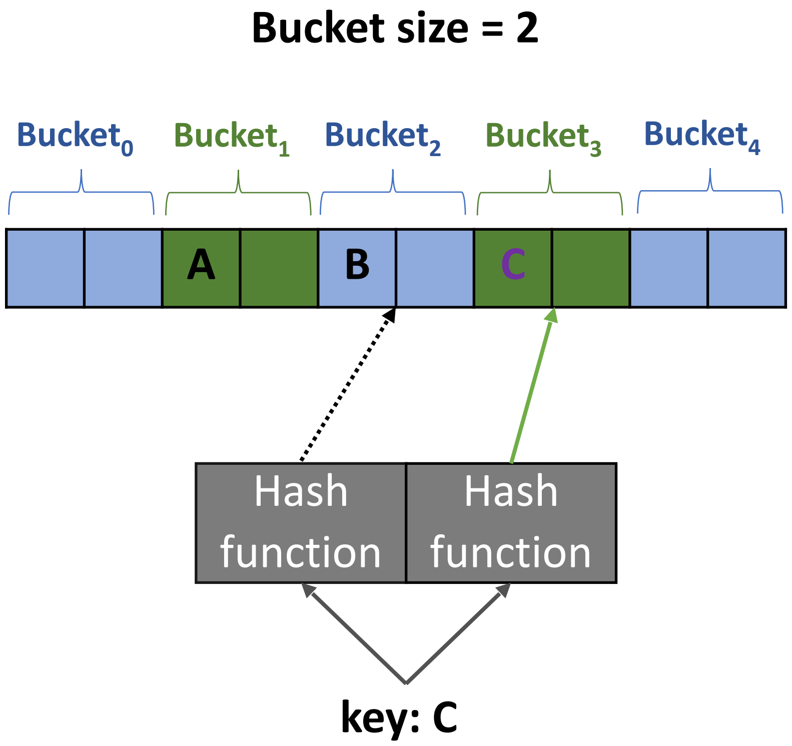
Power of two (or more) choices

- Given two hash functions (or more)
  - Evaluate the ***load*** of the two buckets
  - Insert into the least loaded bucket
    - E.g., if load(bucket_3) < load(bucket_2)
    - Then, we insert in bucket_3



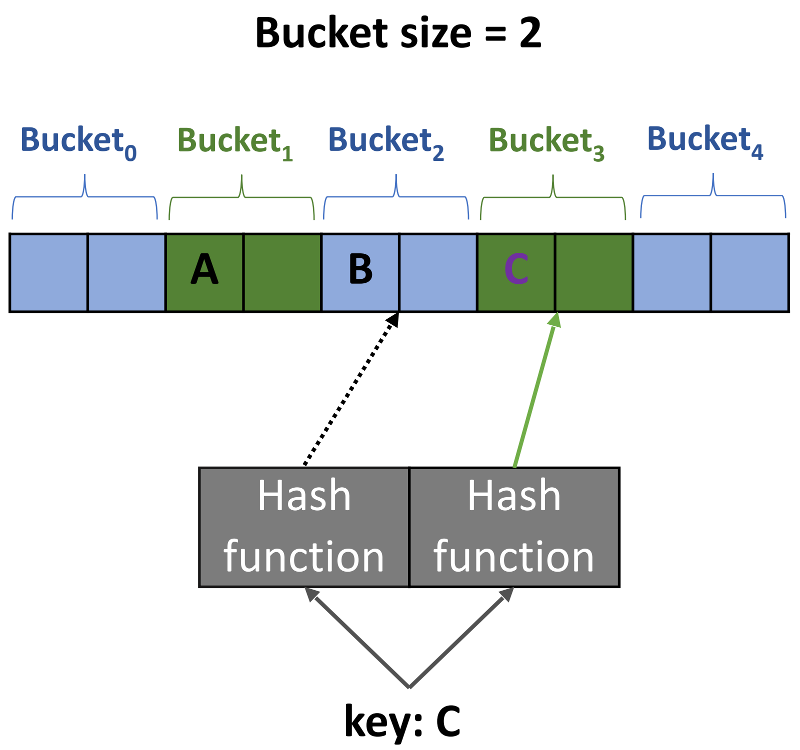
Power of two (or more) choices

- Given two hash functions (or more)
  - Evaluate the ***load*** of the two buckets
  - Insert into the least loaded bucket
    - E.g., if load($bucket_3$) < load($bucket_2$)
    - Then, we insert in $bucket_3$

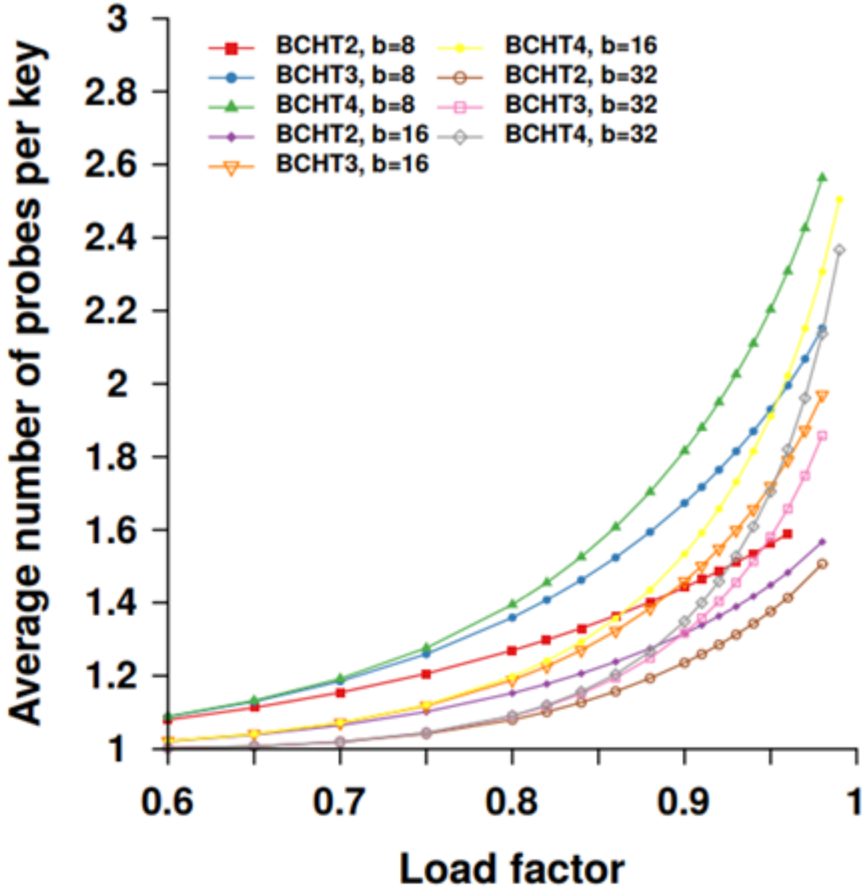


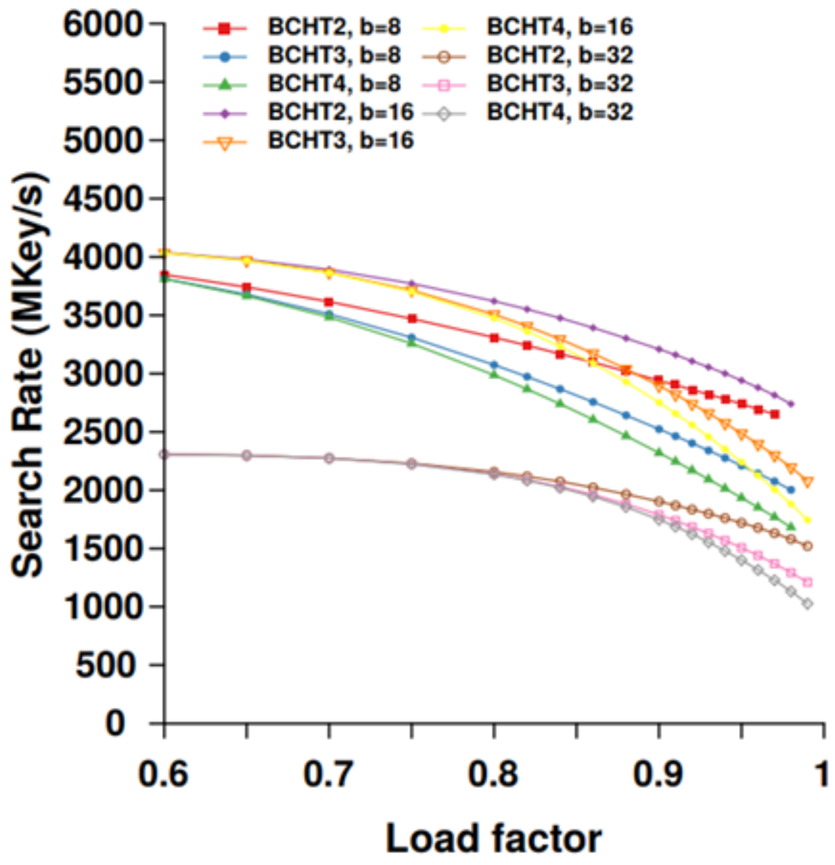
Power of two (or more) choices

- Given two hash functions (or more)
  - Evaluate the ***load*** of the two buckets
  - Insert into the least loaded bucket
    - E.g., if load($bucket_3$) < load($bucket_2$)
    - Then, we insert in $bucket_3$
- Achieves high load factors
- Can be combined with other schemes
- Requires at least ***two*** probes



It's all about the number of probes

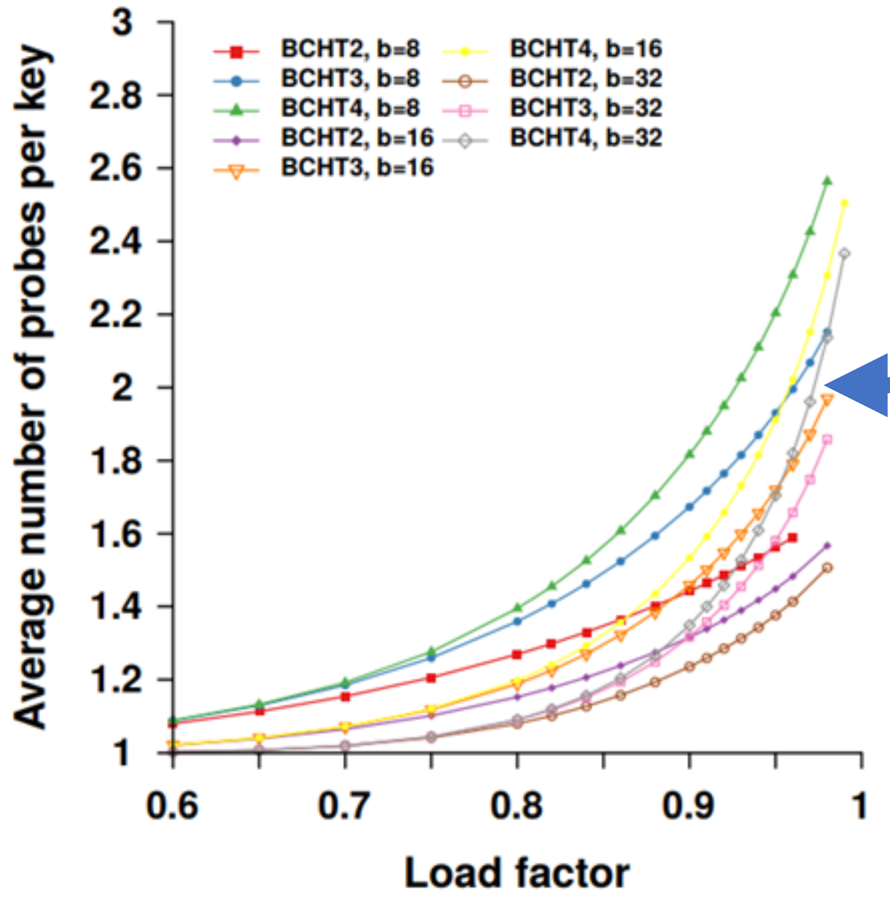


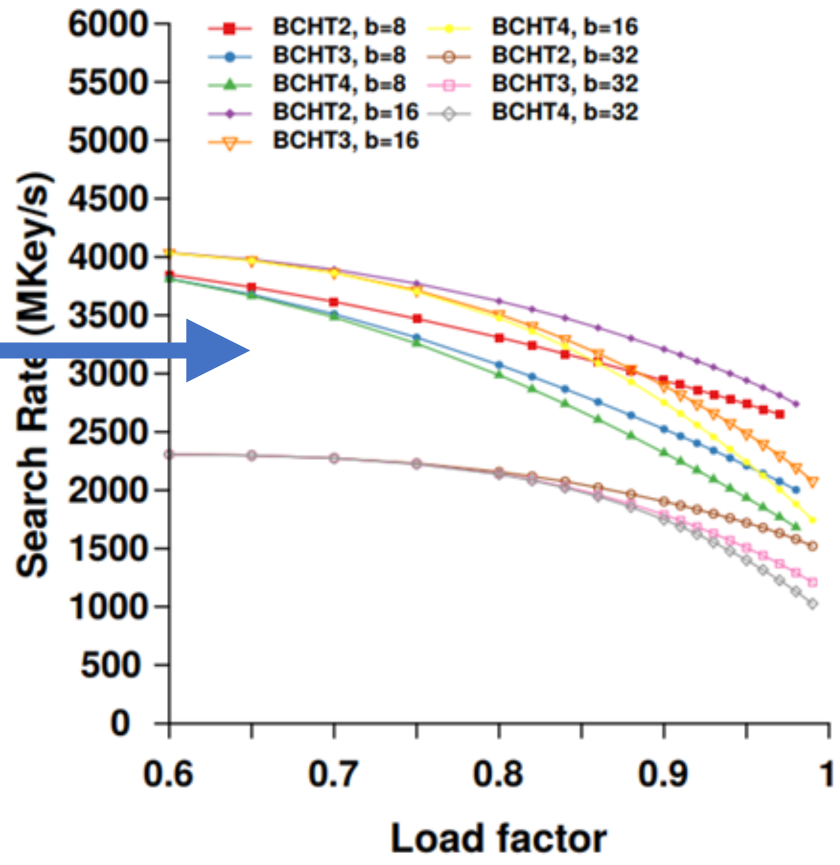


BCHT → *Cuckoo HT, b = 8, 16, 32 and different # of hash functions*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

It's all about the number of probes





BCHT → *Cuckoo HT, b = 8, 16, 32 and different # of hash functions*

GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values