Better GPU Hash Tables

Muhammad A. Awad

UC Davis

mawad@ucdavis.edu

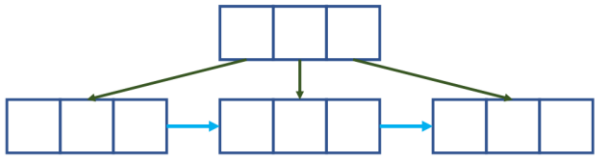
with Saman Ashkiani, Serban D. Porumbescu, Martín Farach-Colton, John D. Owens

About me

- Alexandria University
  - BSc, Naval Architecture and Marine Engineering (2013)
- University of California, Davis
  - PhD, Electrical and Computer Engineering (2016 – June 2022)

About me

- Alexandria University
  - BSc, Naval Architecture and Marine Engineering (2013)
- University of California, Davis
  - PhD, Electrical and Computer Engineering (2016 – June 2022)
- Research interests
  - Parallel algorithms
  - Concurrent data structures



Dynamic GPU B-Tree

```
multiversion_ds ds;
ds.insert(pair);
auto timestamp = ds.take_snapshot();
auto result = ds.find(key, timestamp);
```

Multiversion data structures (*work in progress*)

Hash tables

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

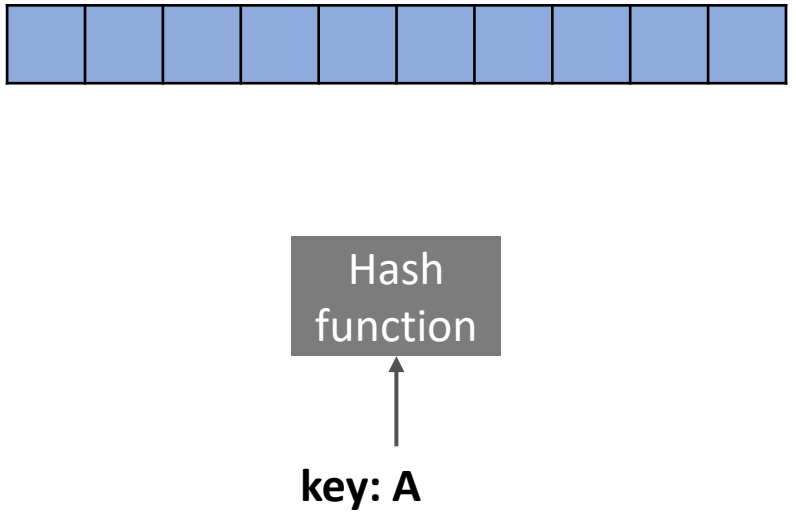
Hash tables

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

Hash function

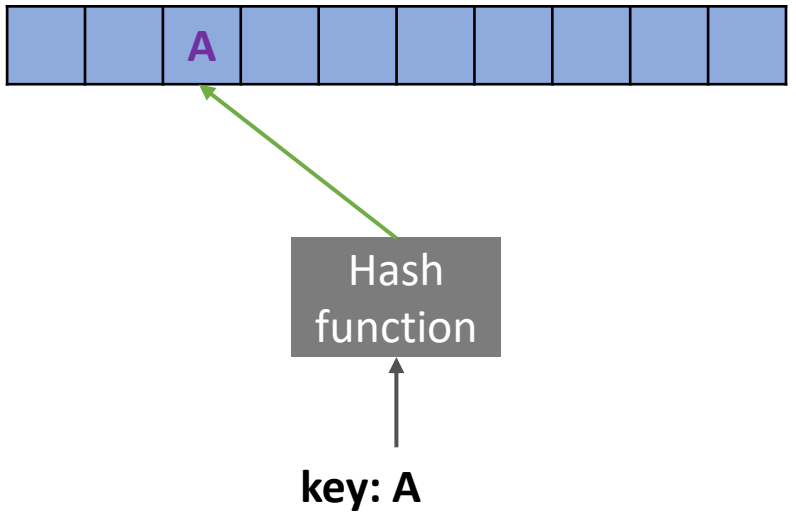
Hash tables

- Insert(A)



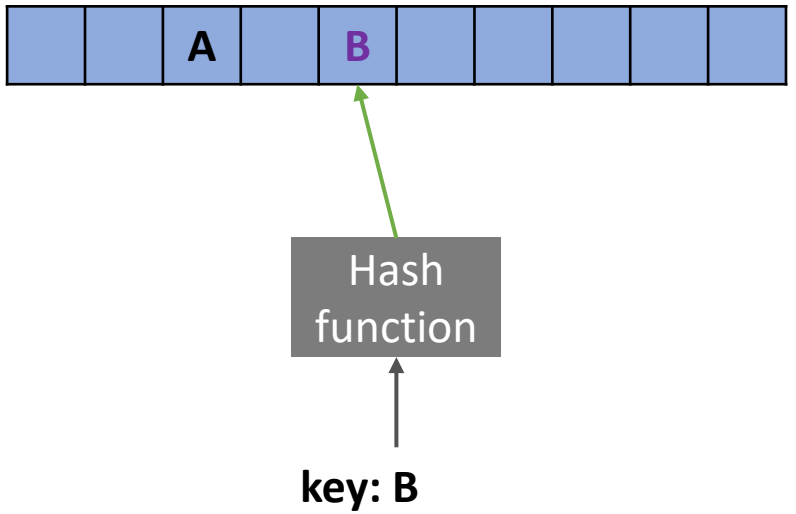
Hash tables

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



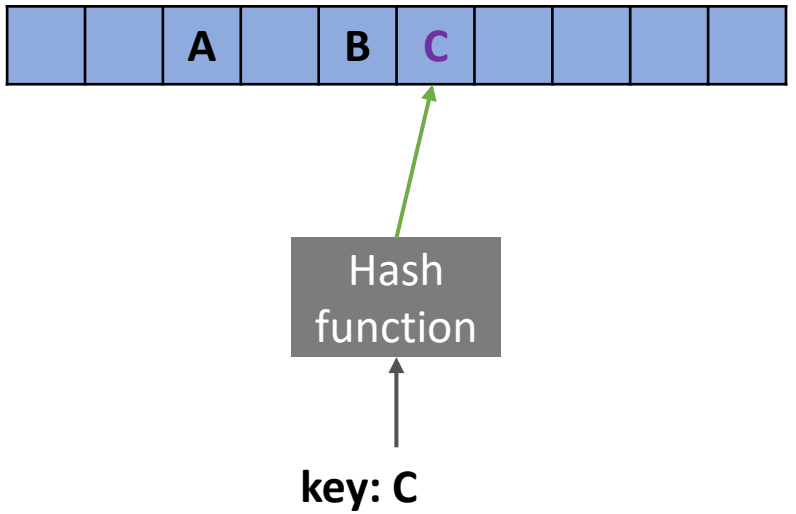
Hash tables

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



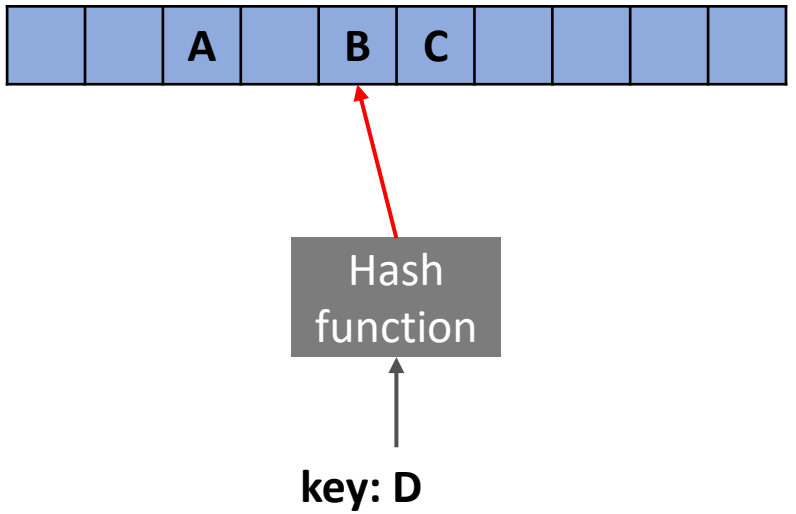
Hash tables

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



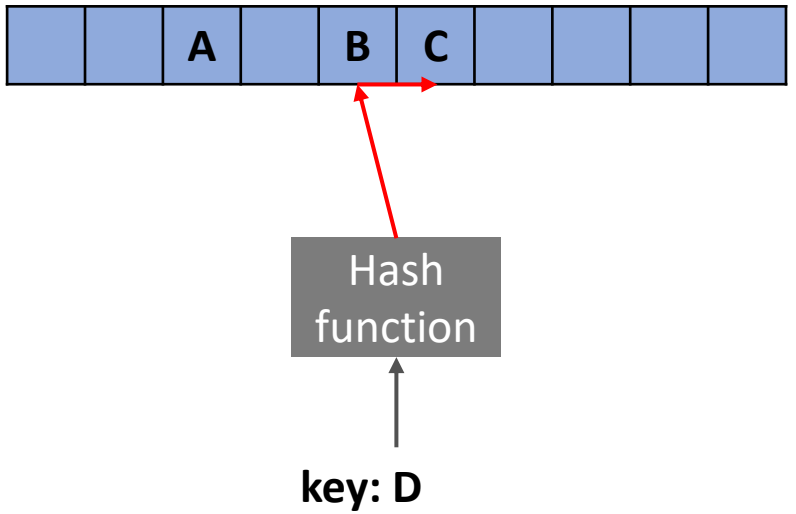
Hash tables

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



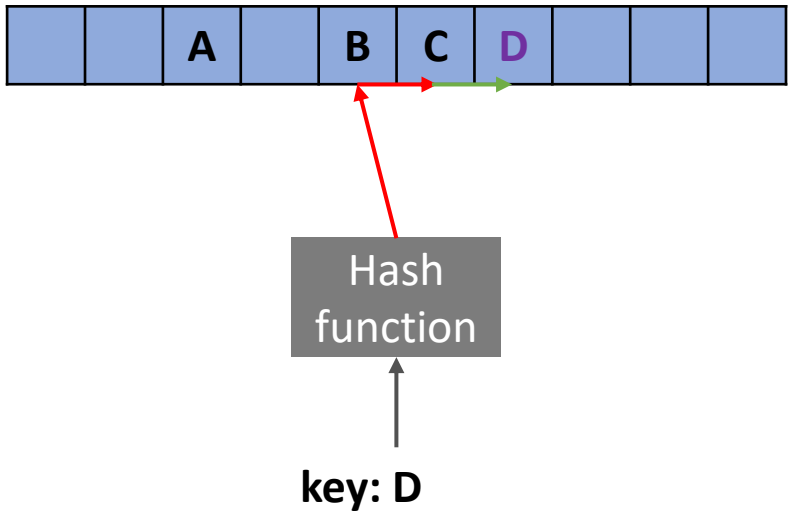
Hash tables

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



Hash tables

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



Design decisions

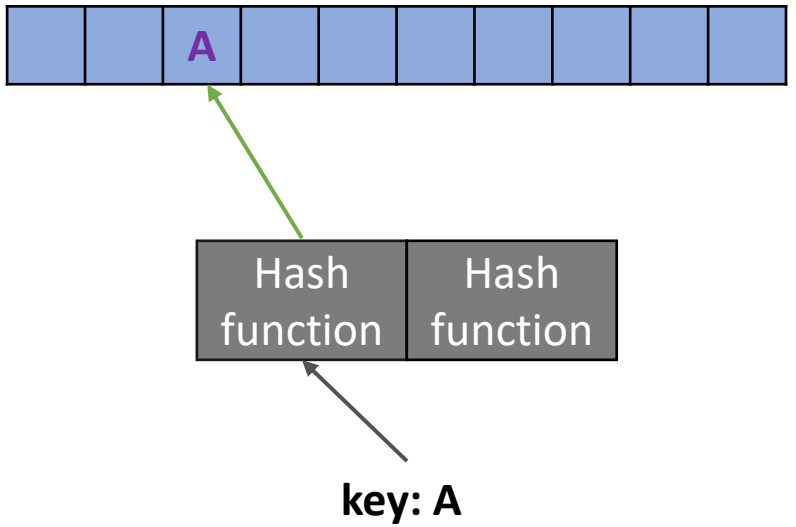
Design decisions

- 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

Probing Scheme

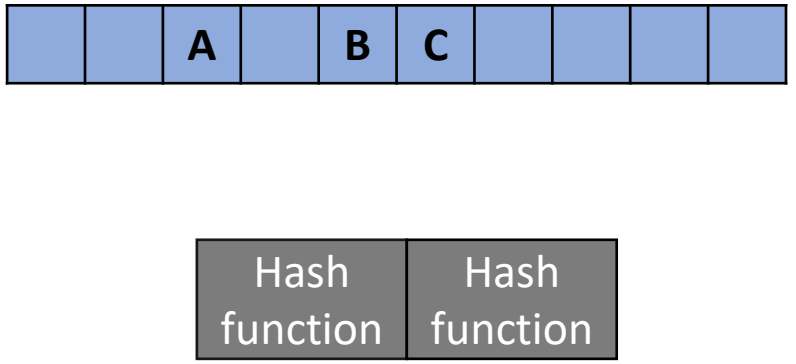
Cuckoo hashing

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



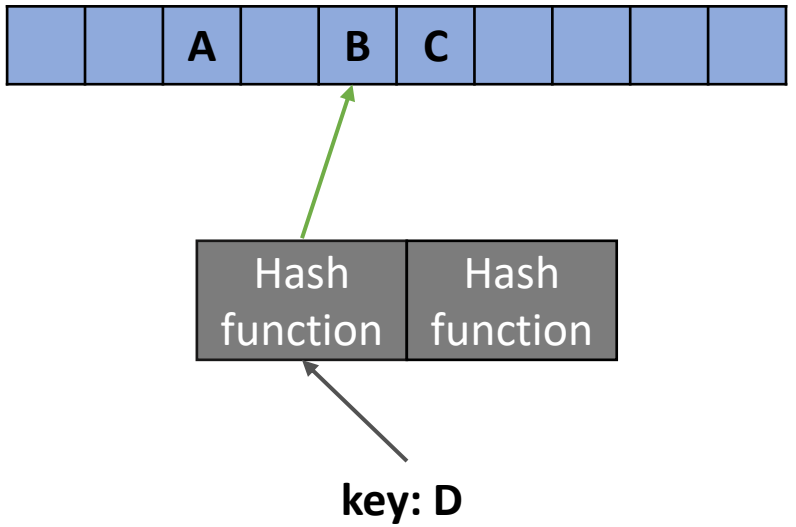
Cuckoo hashing

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



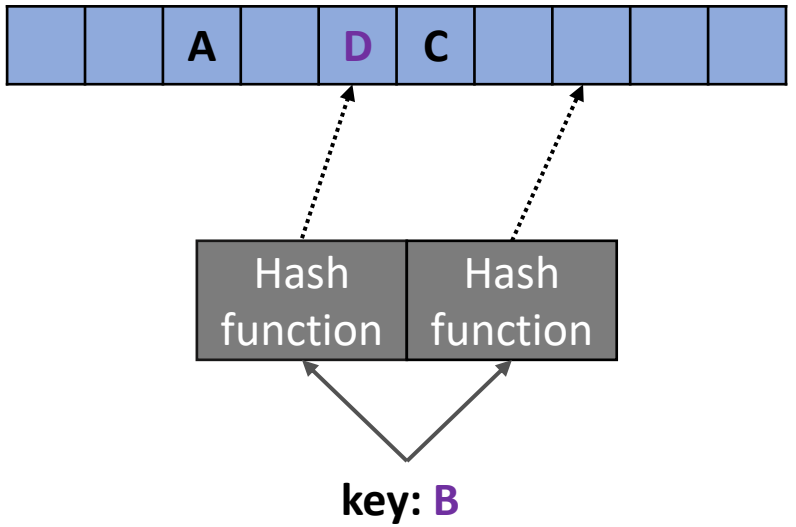
Cuckoo hashing

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



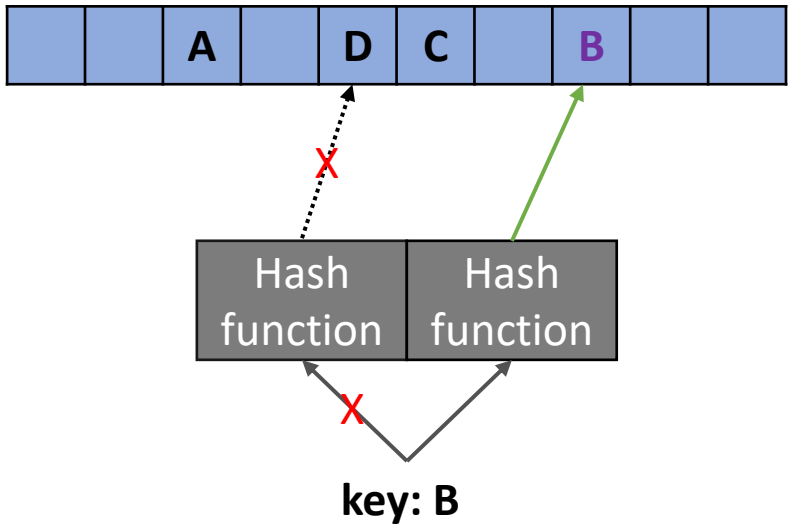
Cuckoo hashing

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



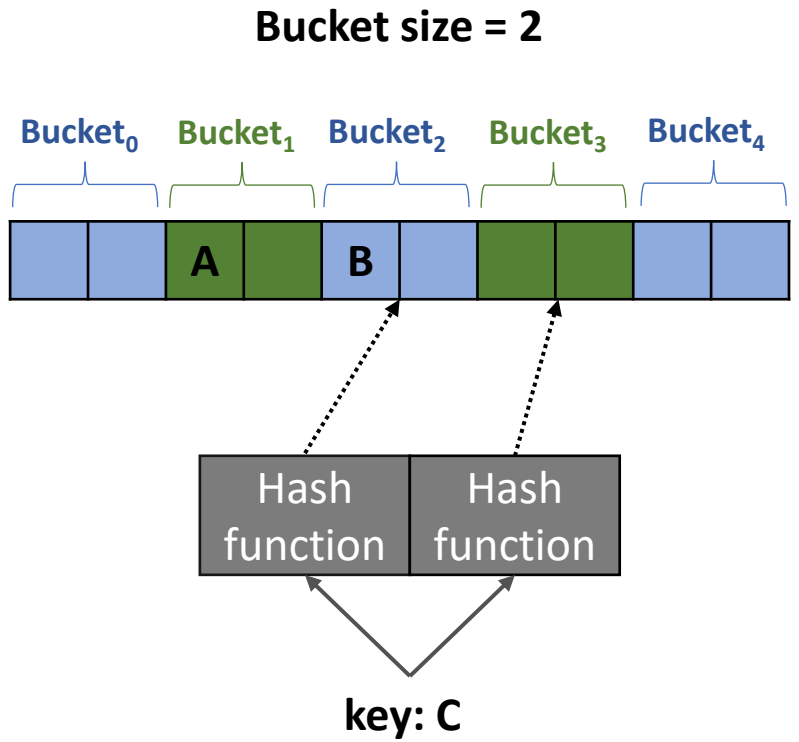
Cuckoo hashing

- 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



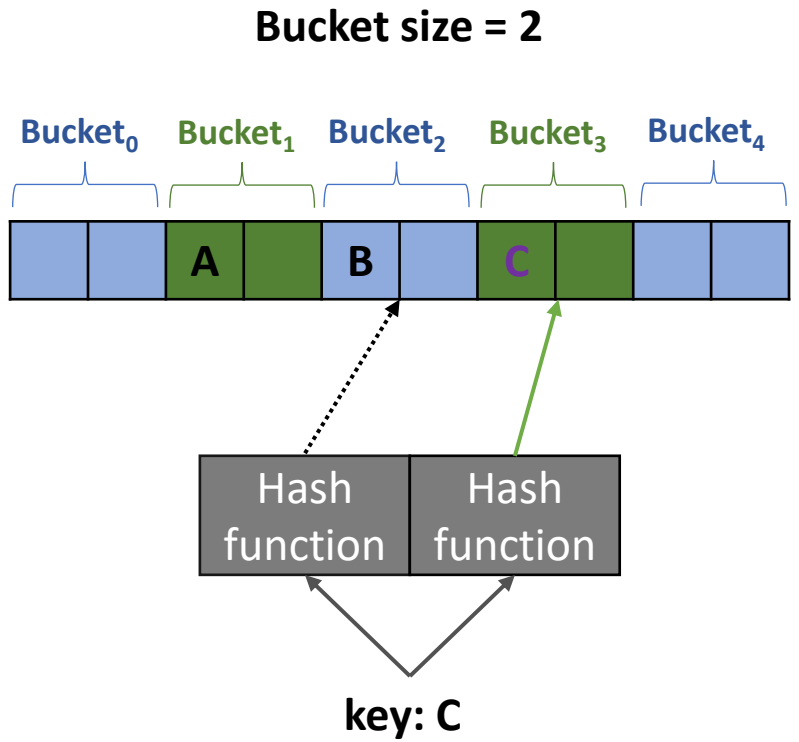
Power of two (or more) choices

- 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 $bucket_3$



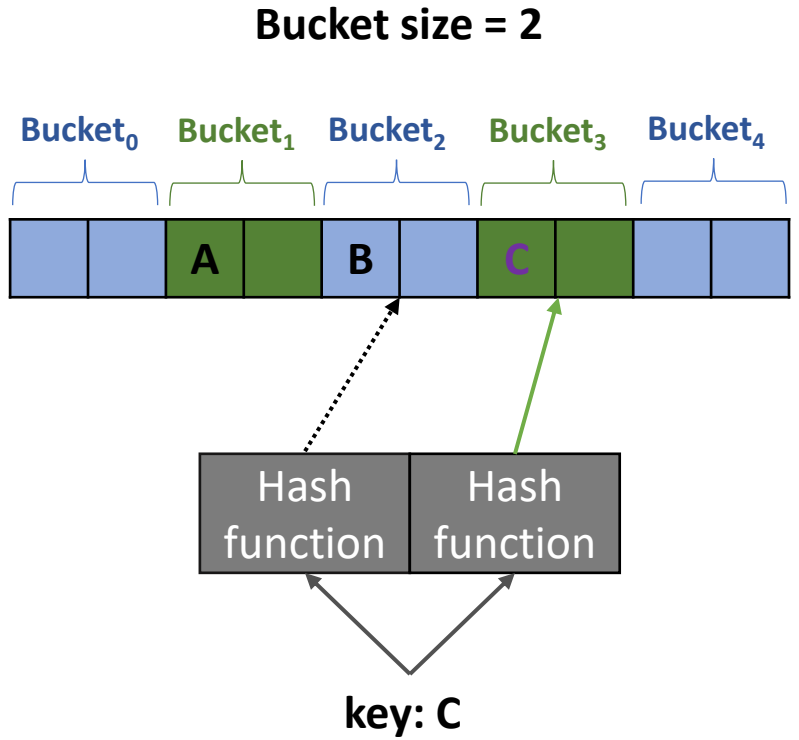
Power of two (or more) choices

- 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 $bucket_3$



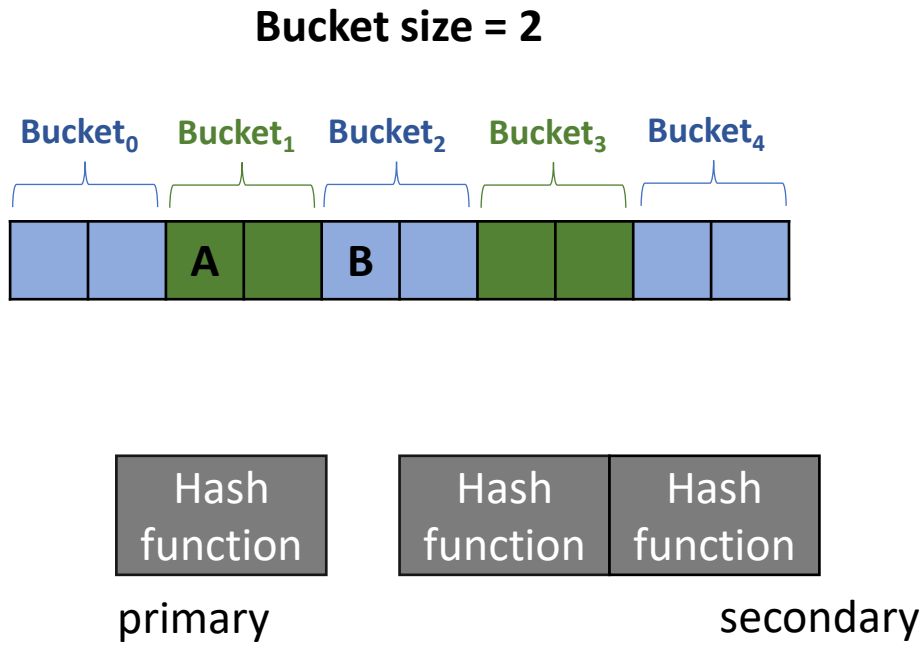
Power of two (or more) choices

- 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



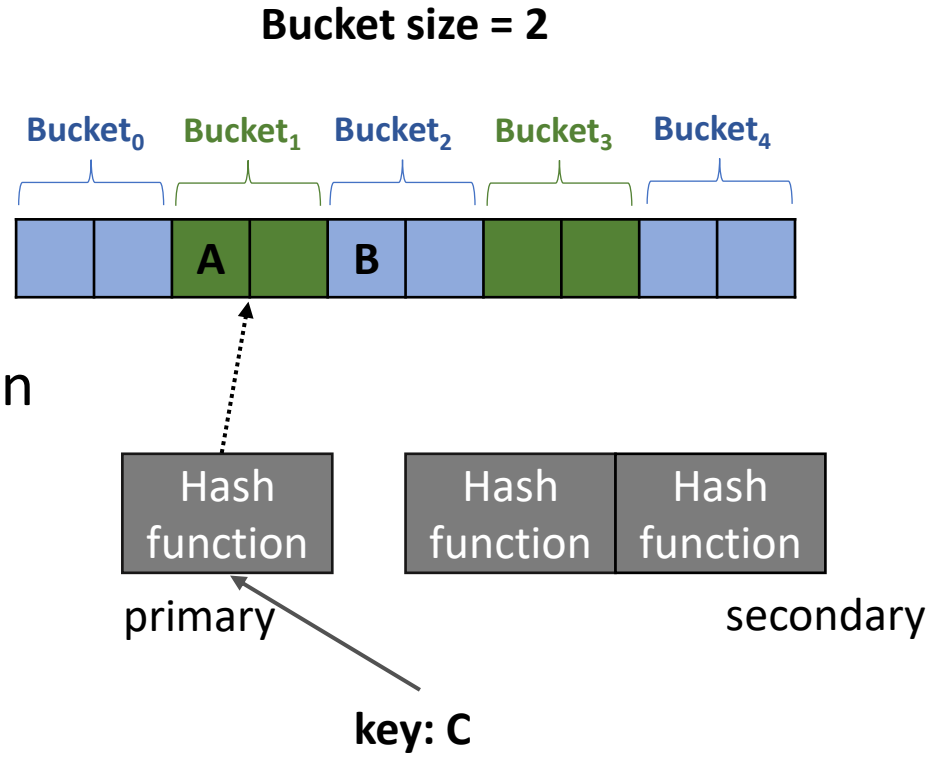
Iceberg hashing

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



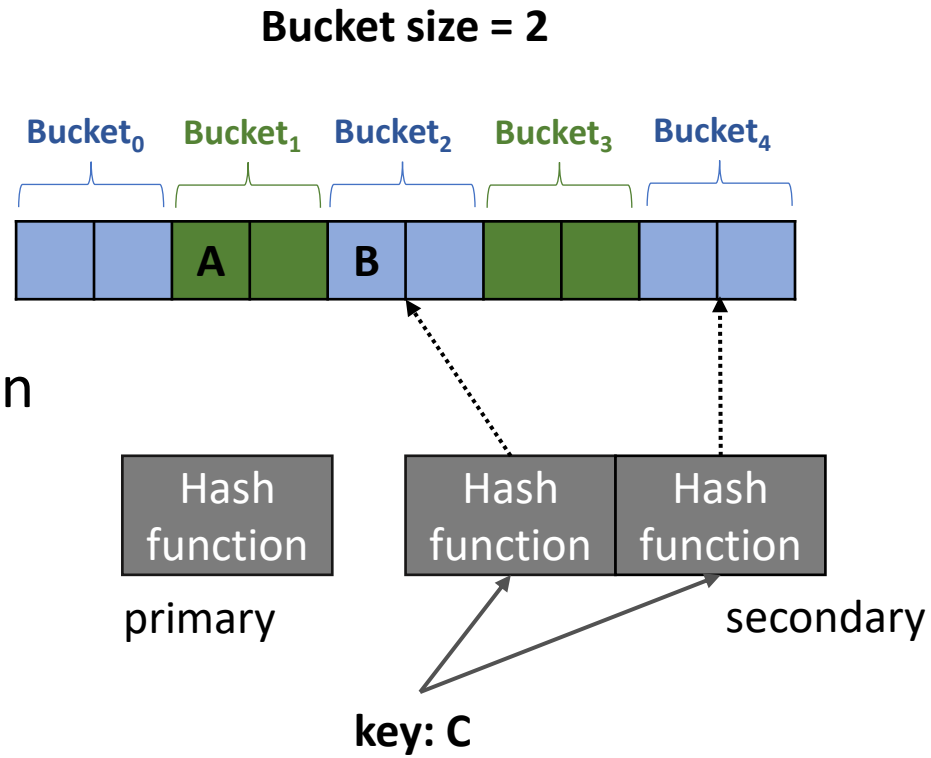
Iceberg hashing

- 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



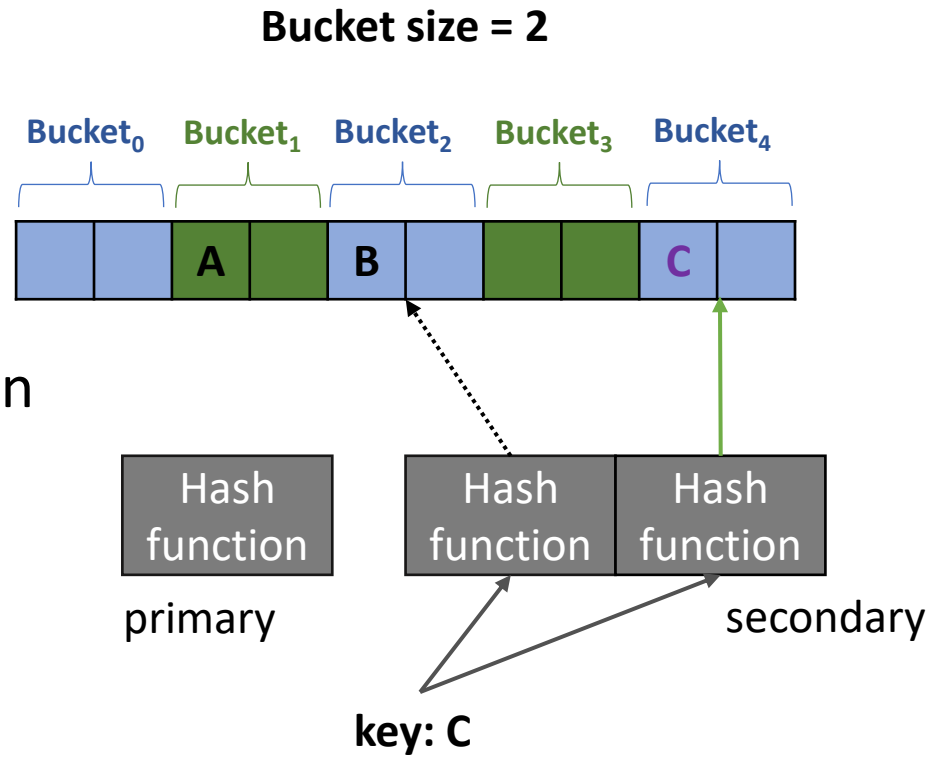
Iceberg hashing

- 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



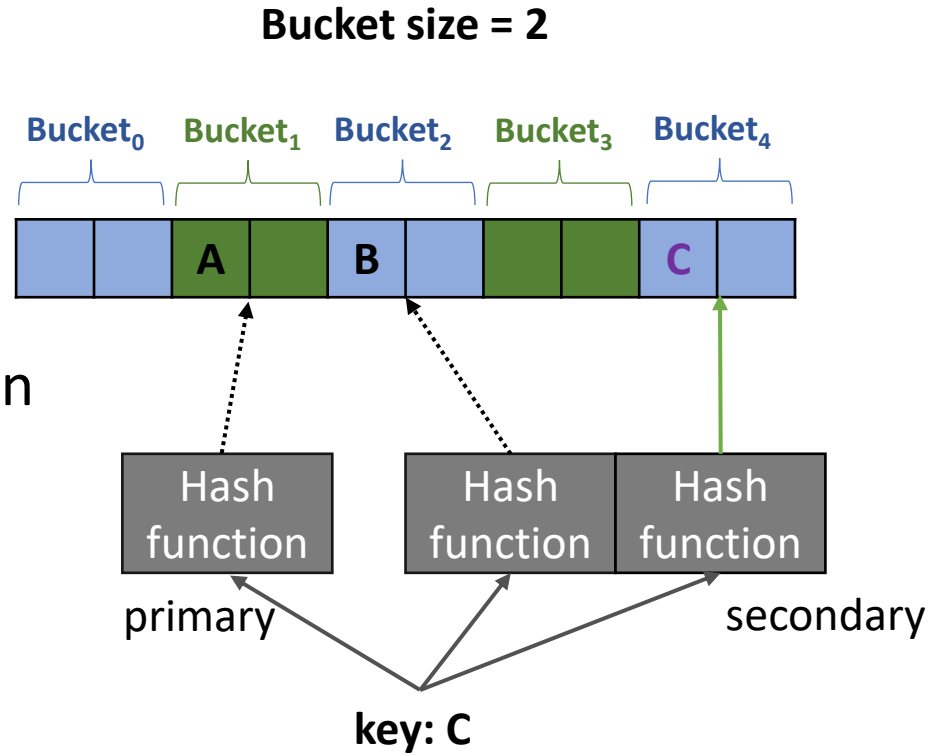
Iceberg hashing

- 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



Design decisions

- 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

Design decisions

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

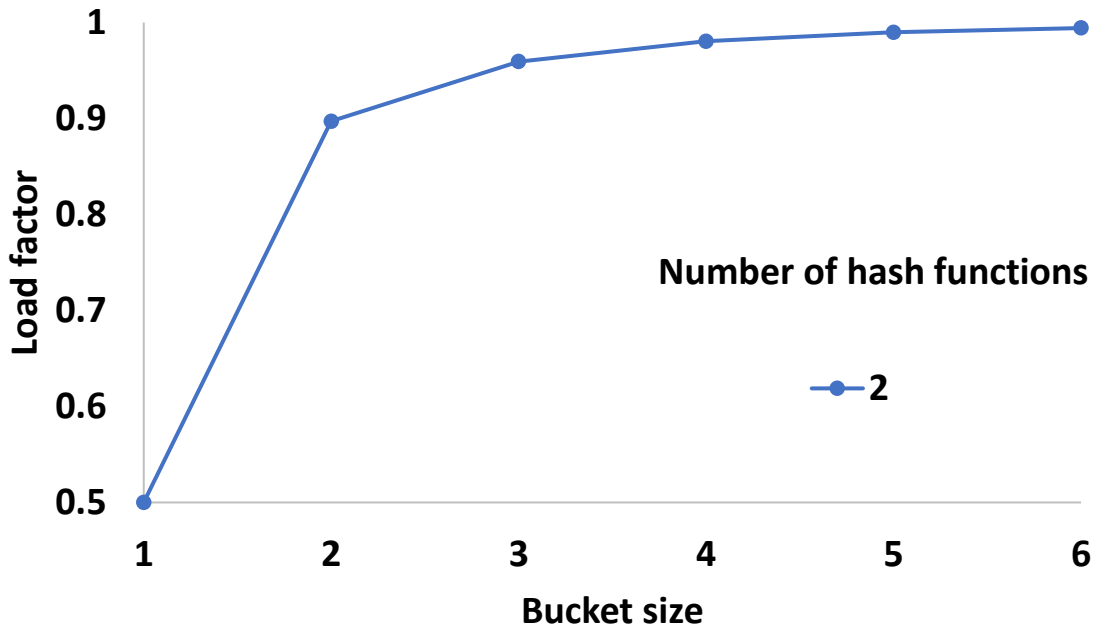


Figure from: **Stefan Walzer,** *Random hypergraphs for hashing-based data structures*. Ph.D. dissertation, TU Ilmenau.

Design decisions

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

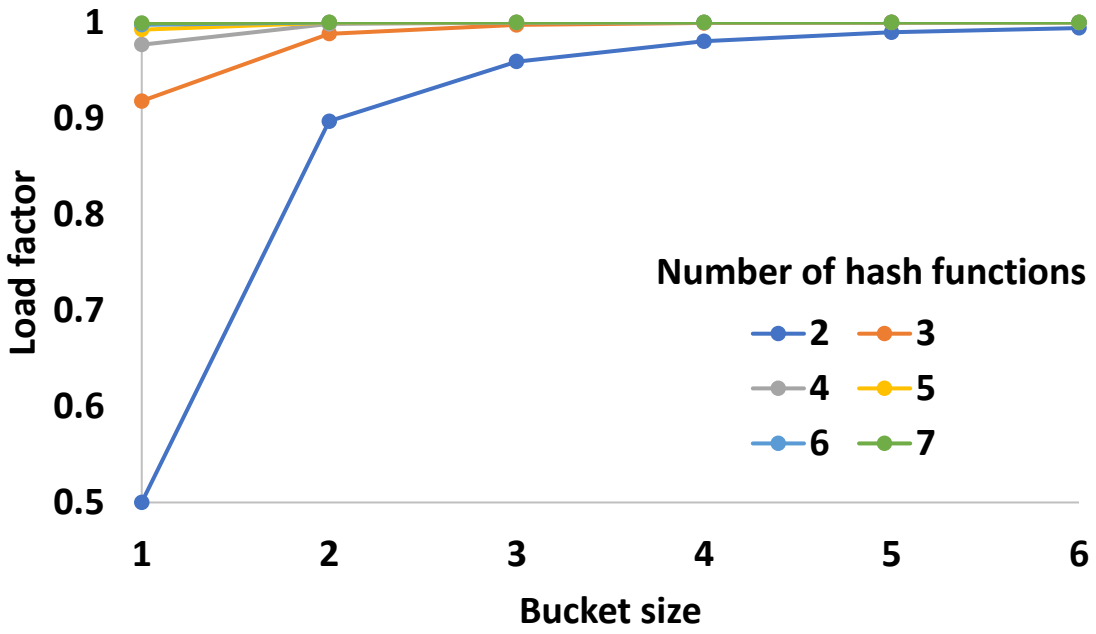


Figure from: **Stefan Walzer,** *Random hypergraphs for hashing-based data structures*. Ph.D. dissertation, TU Ilmenau.

Design decisions

- 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

Design decisions

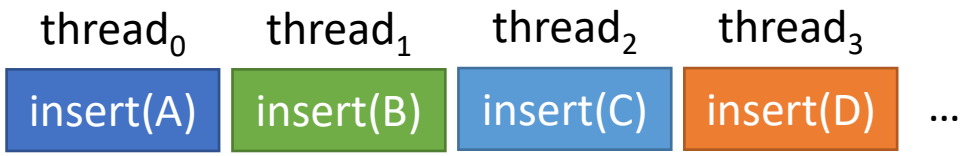
- Probing scheme
  - Linear, quadratic, double hashing, **cuckoo**, **power-of-two choices**, **iceberg**
- Bucket size
- Probe complexity
  - Number of hash functions
- Placement strategy
  - Balanced or not balanced

Implementation

Tile-wide cooperative insertion

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

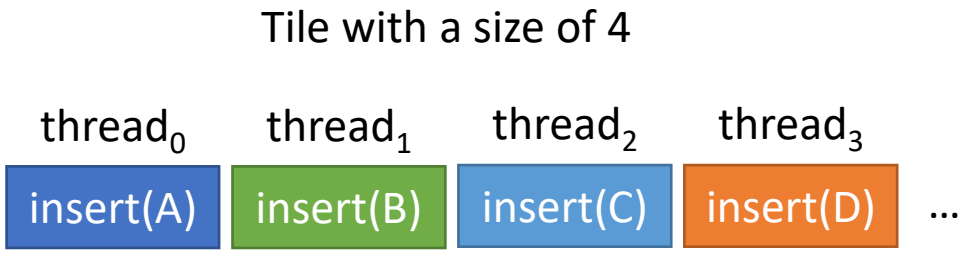
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {









}
```



Tile-wide cooperative insertion

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

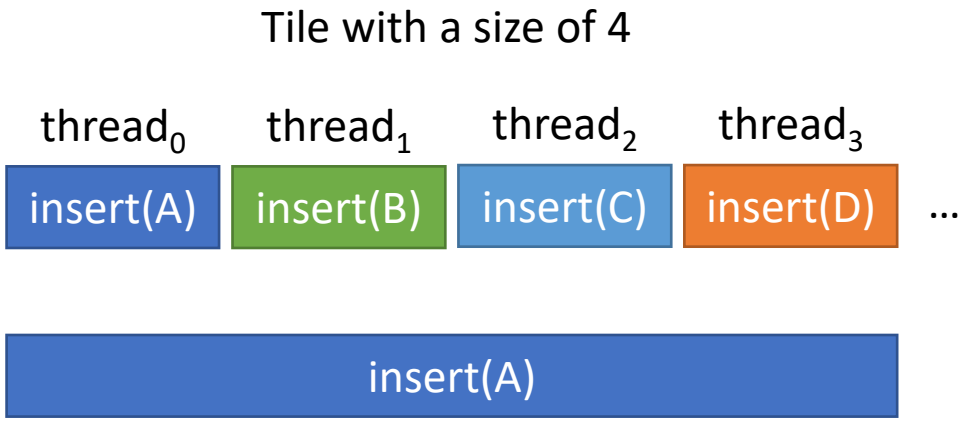
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;









}
```



Tile-wide cooperative insertion

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

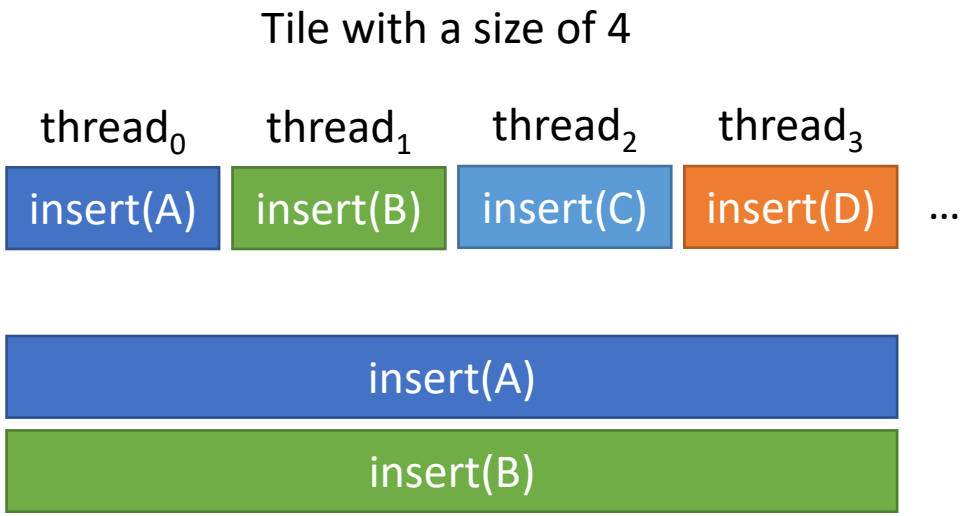
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // 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

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

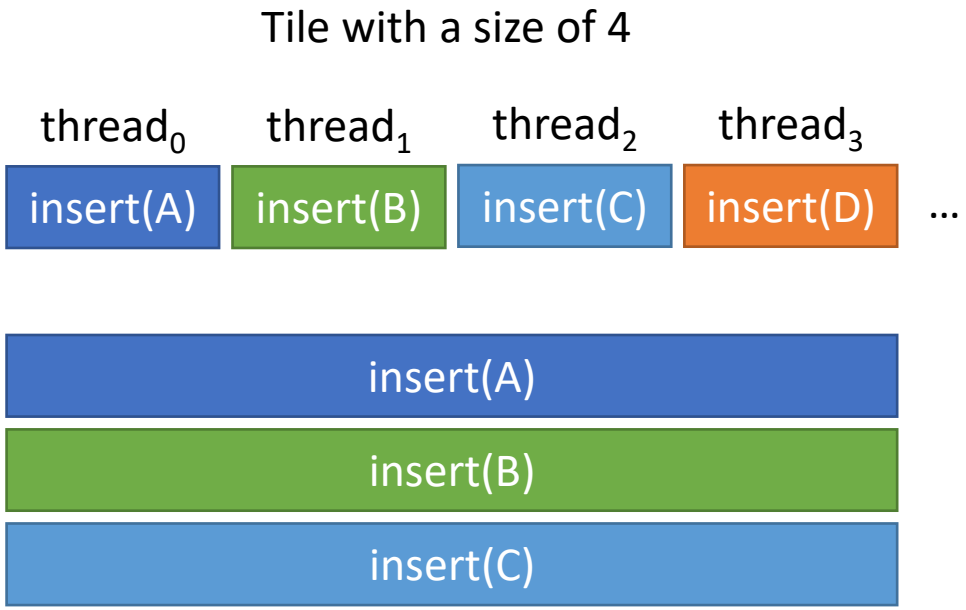
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // 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

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

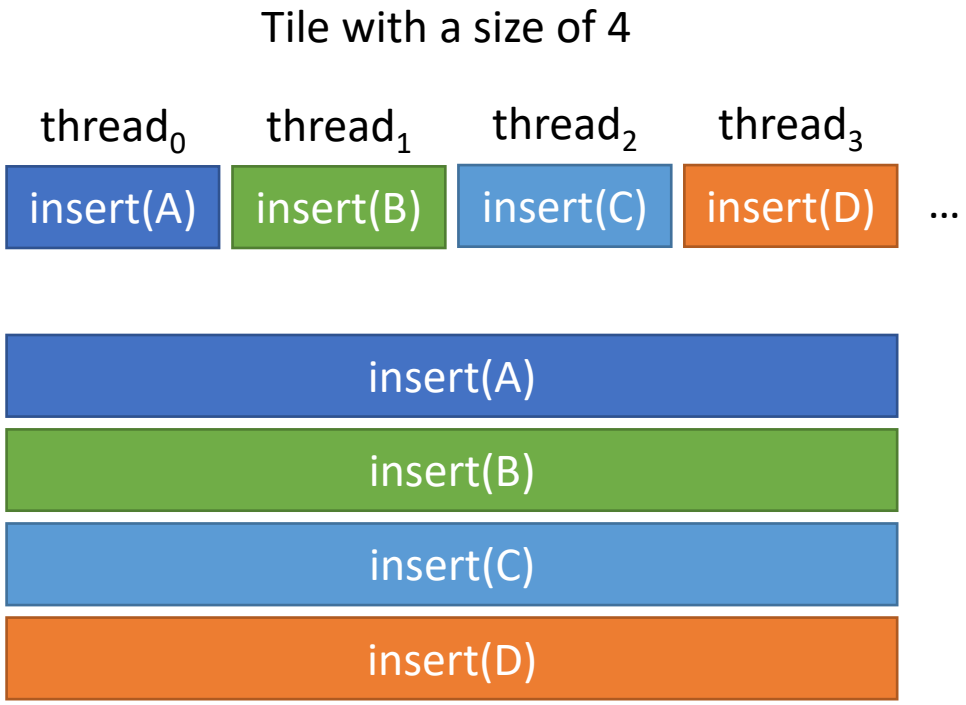
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // 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

- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

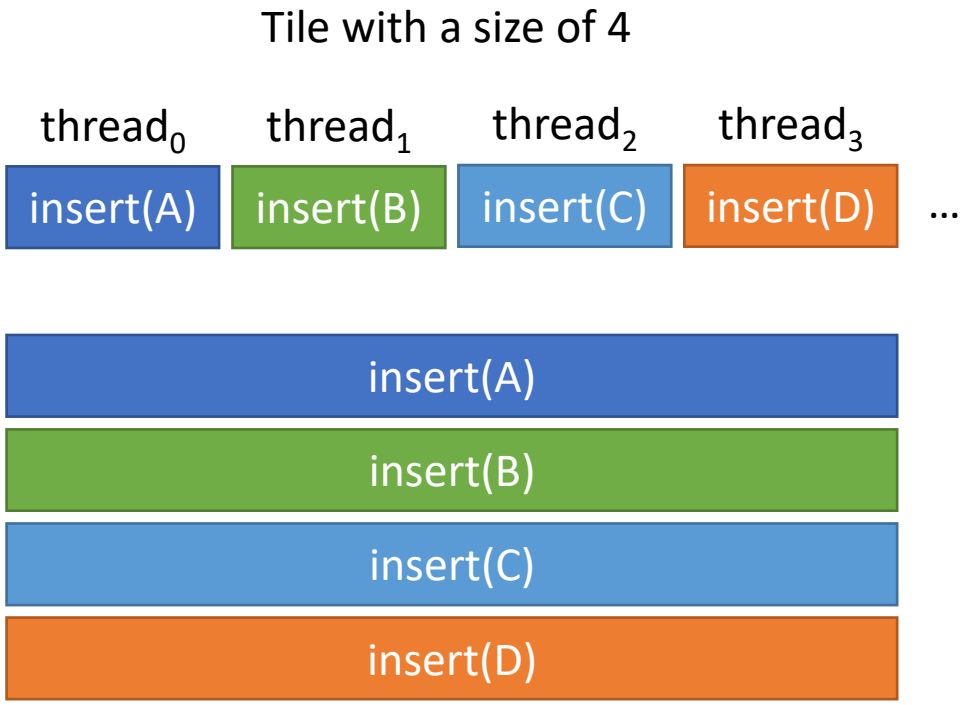
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // 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

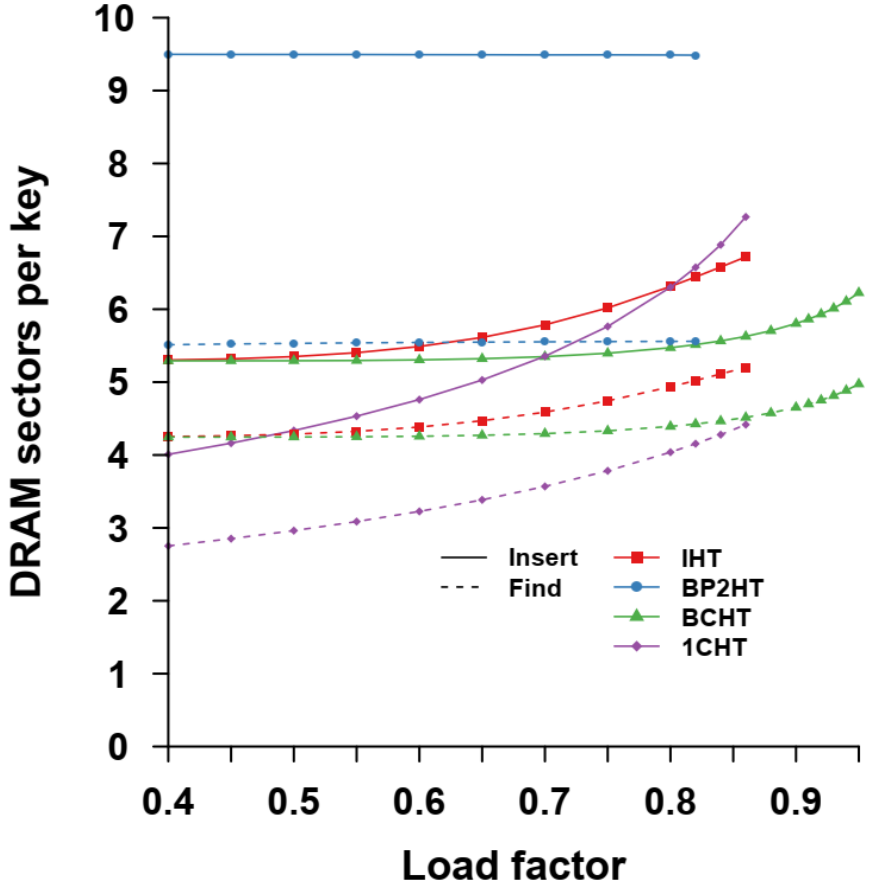
- An efficient implementation will:
  - Avoid branch divergence
  - Achieve coalesced memory access

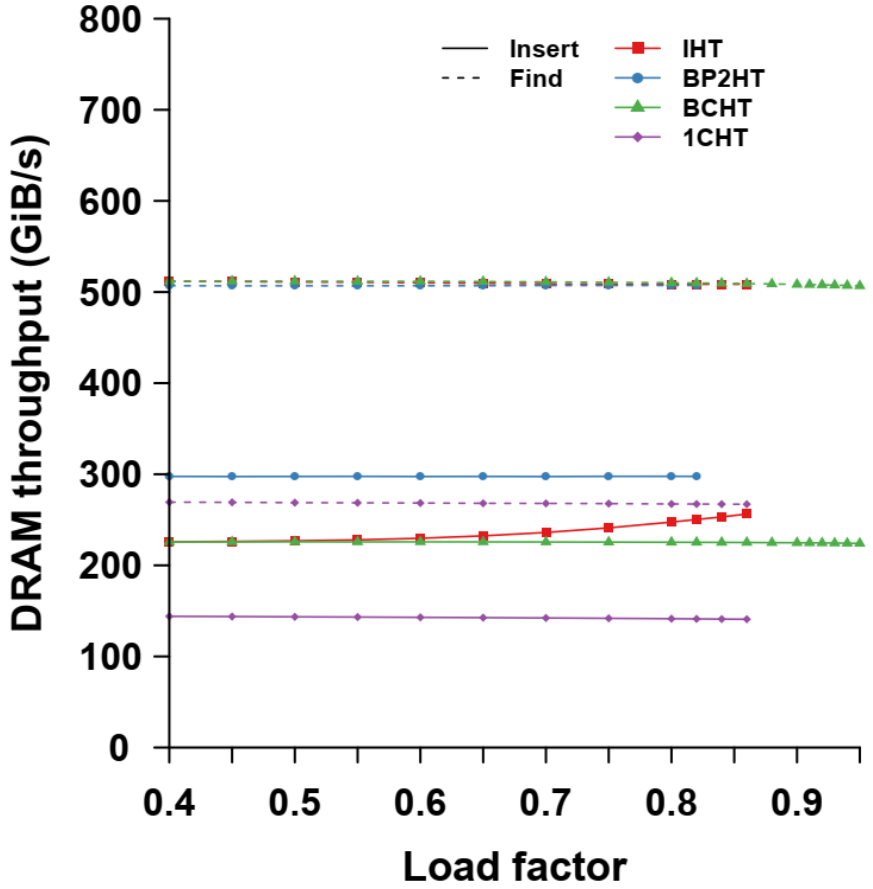
```
bool cooperative_insert(bool to_insert, pair_type pair, pair_type* table) {
    // Construct the work tile
    cg::thread_block thb = cg::this_thread_block();
    auto tile = cg::tiled_partition<bucket_size>(thb);
    auto thread_rank = tile.thread_rank();
    bool success = true;

    // 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;
}
```



Results

Memory transfers and throughput

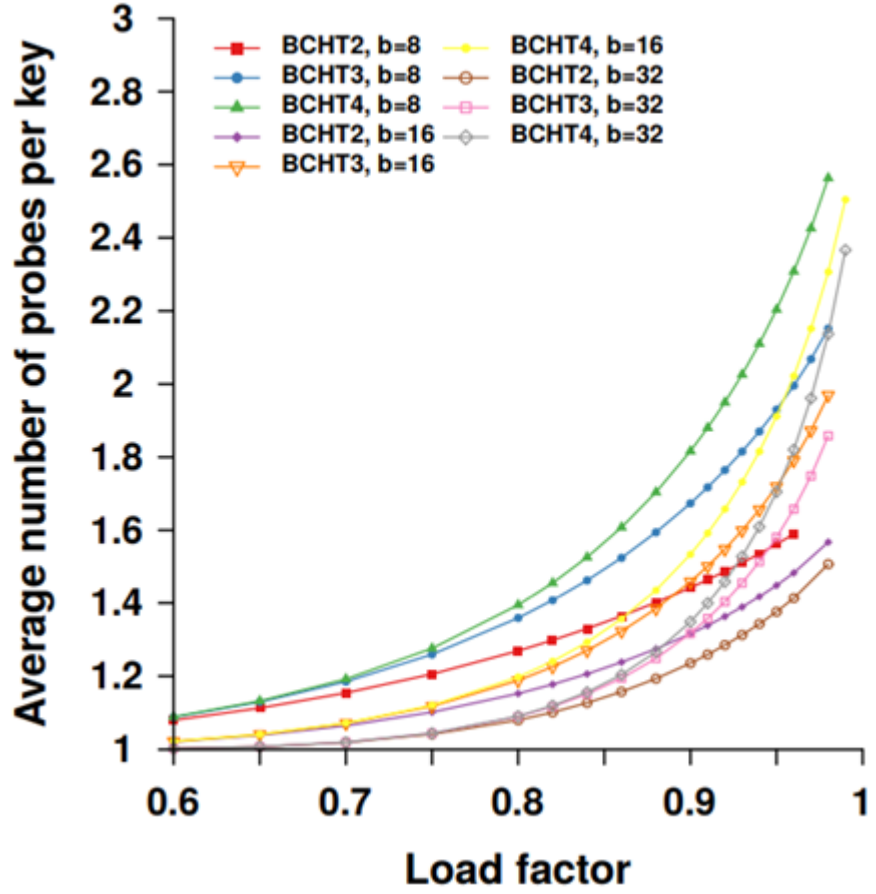


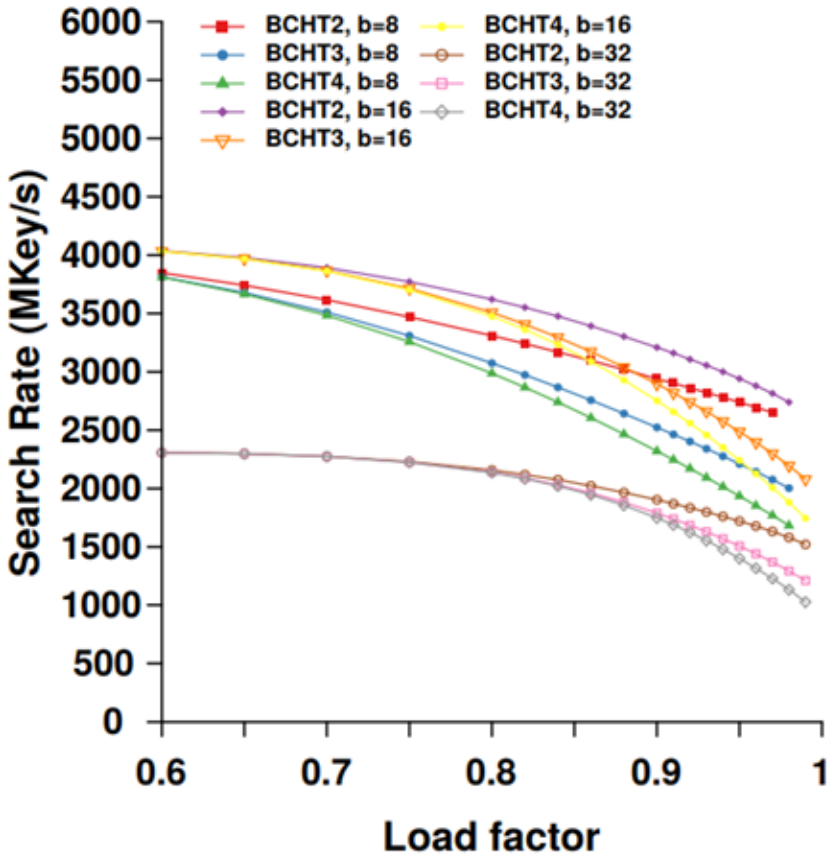


1CHT → *Cuckoo HT, b = 1*
BCHT → *Cuckoo HT, b = 16*
BP2HT → *Power of two HT, b = 16*
IHT → *Iceberg HT, b = 16*

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

It’s all about the number of probes

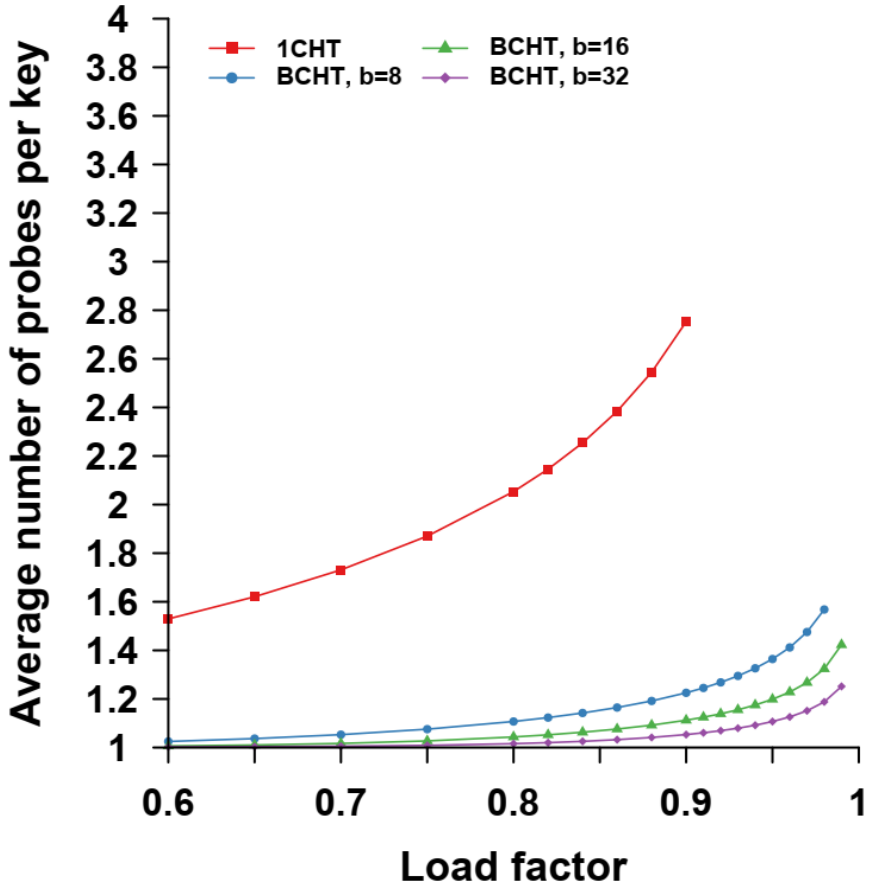


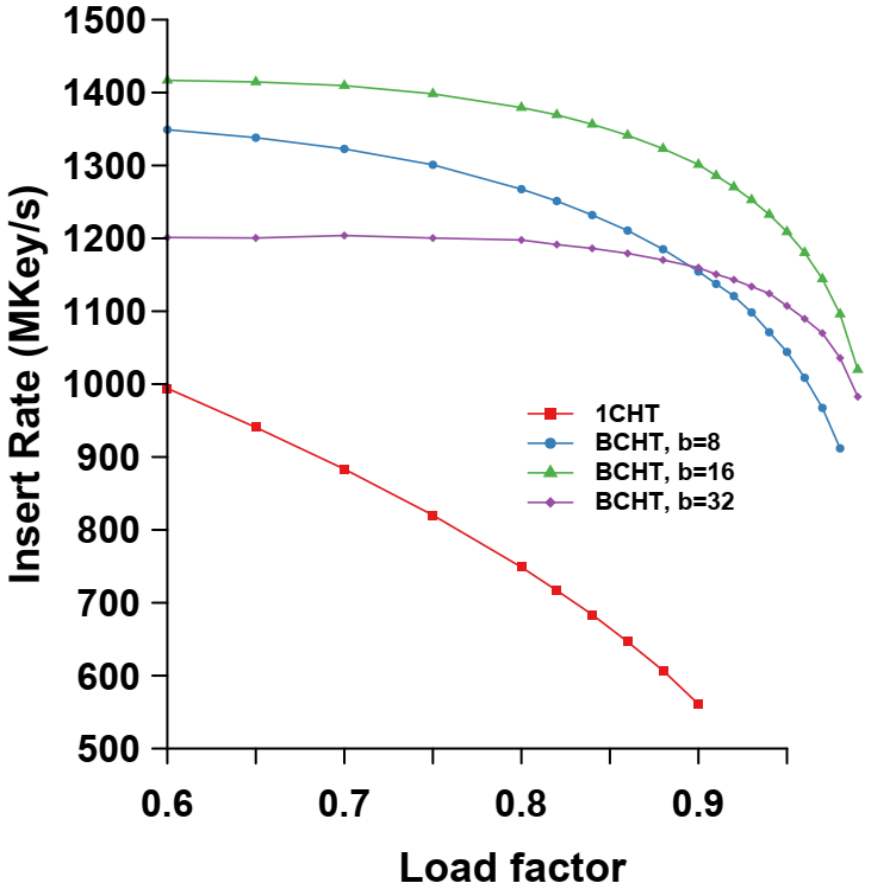


BCHT → *Cuckoo HT, b = 8, 16, 32 and different # of hash functions*

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

BCHT insertion performance



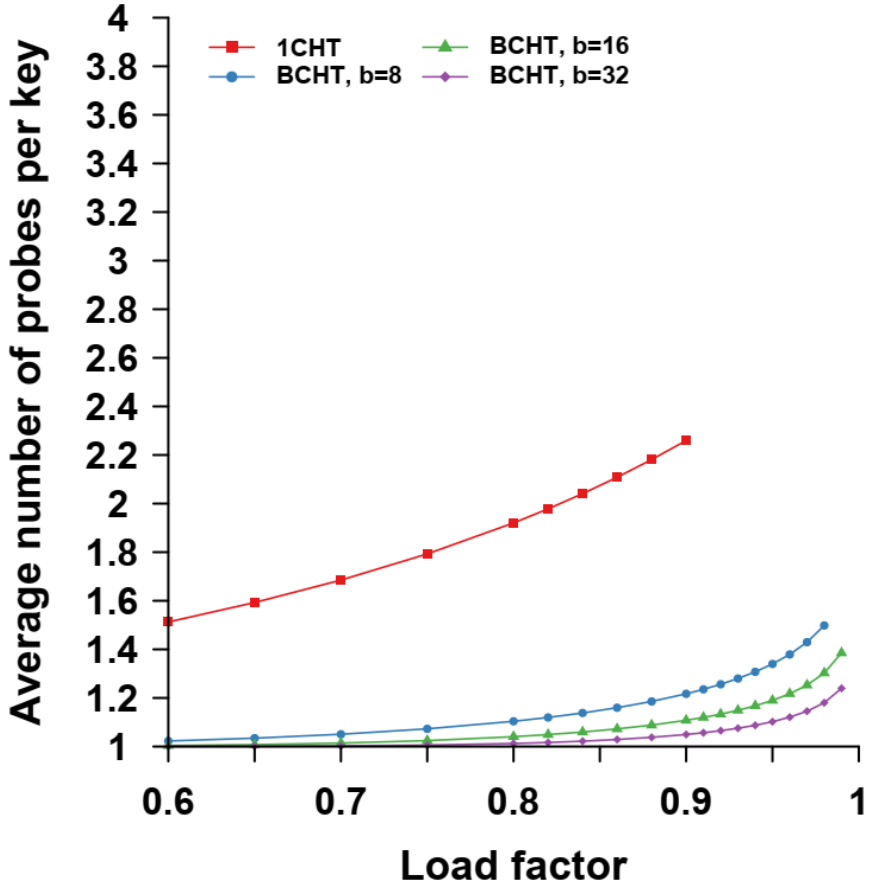


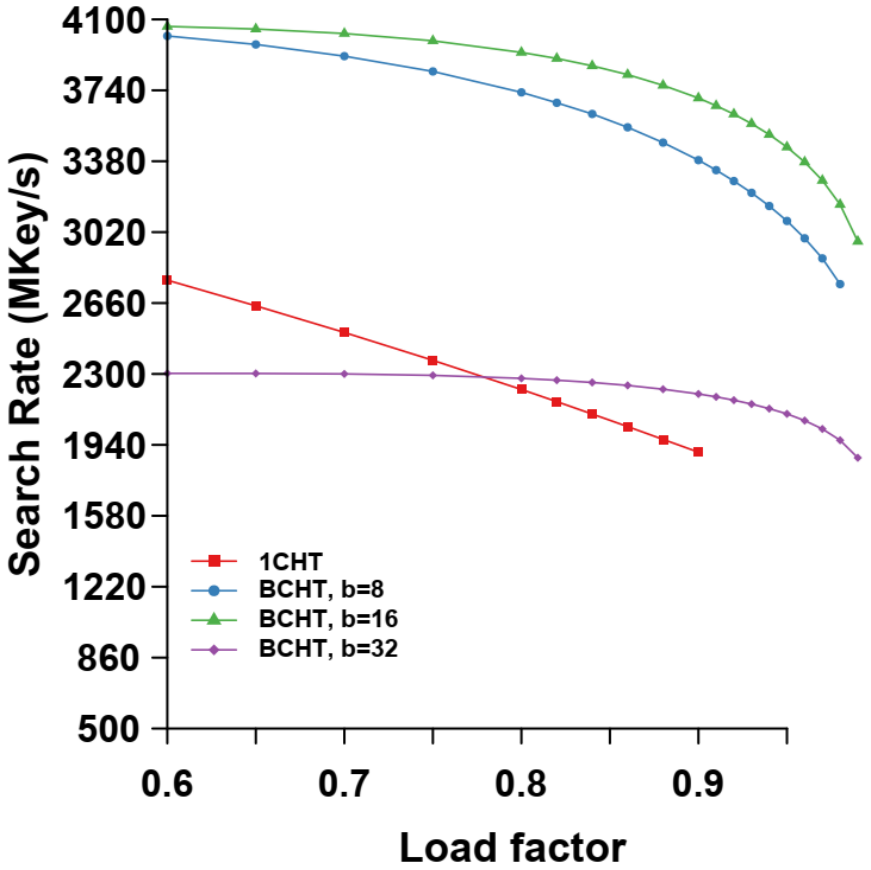
1CHT → *Cuckoo HT, b = 1*

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)



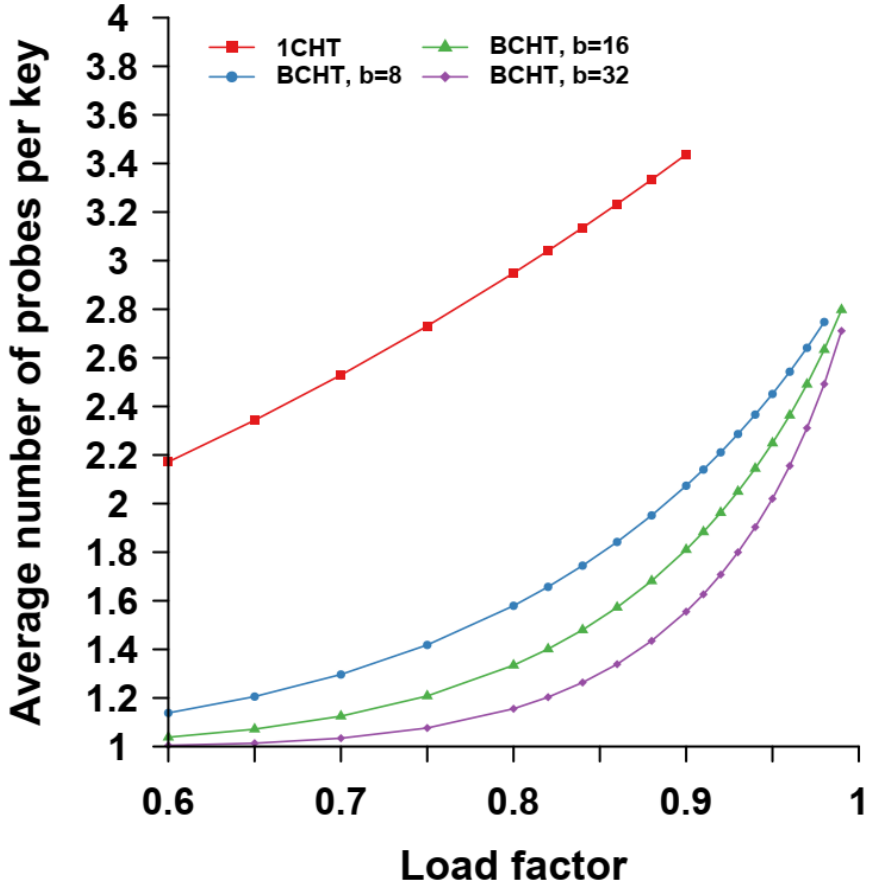


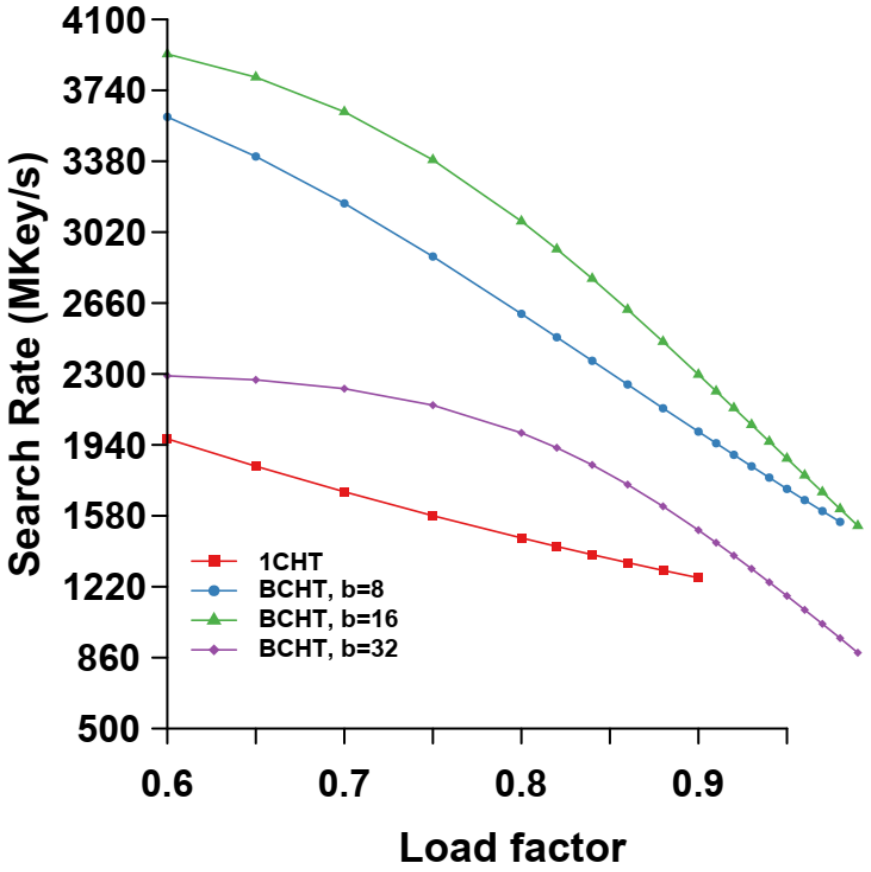
1CHT → *Cuckoo HT, b = 1*

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)



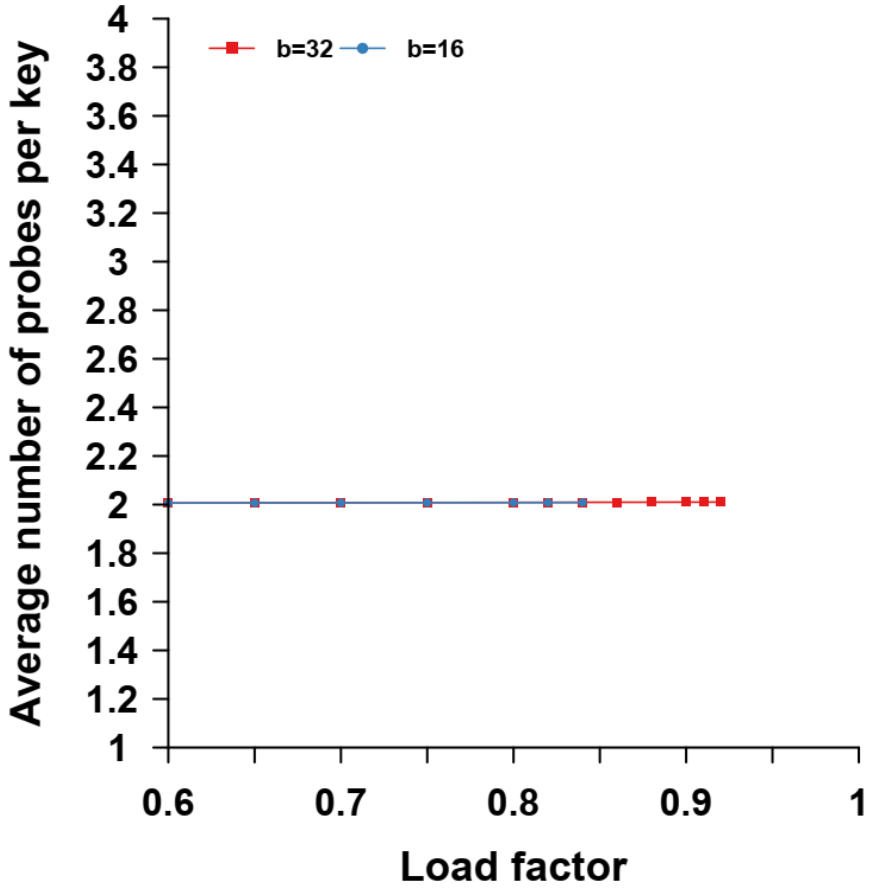


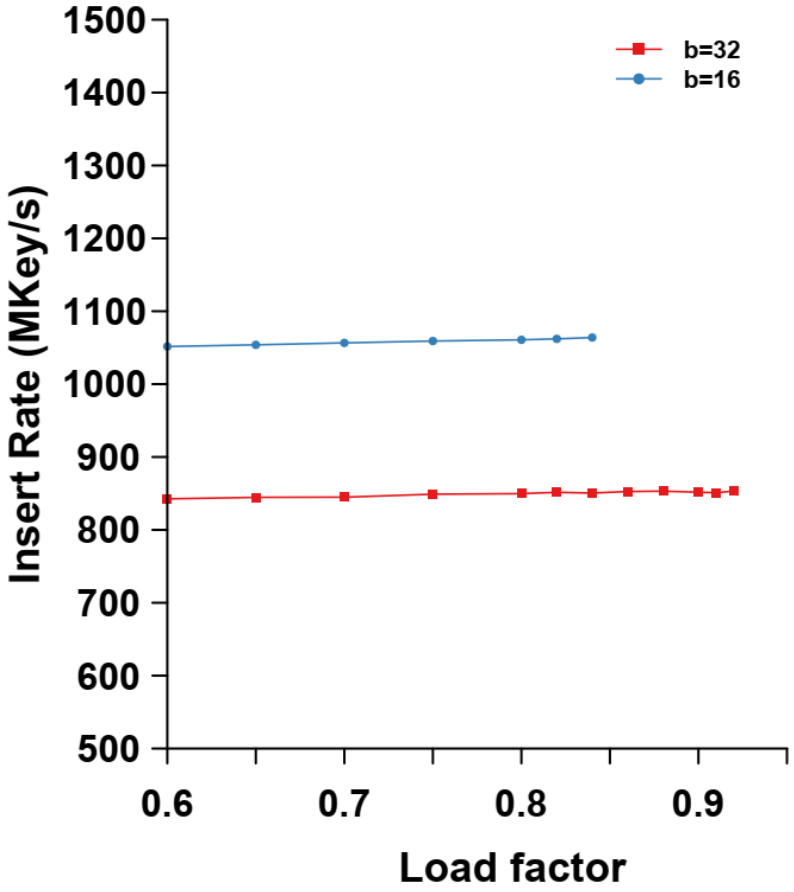
1CHT → *Cuckoo HT, b = 1*

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

BP2HT insertion performance

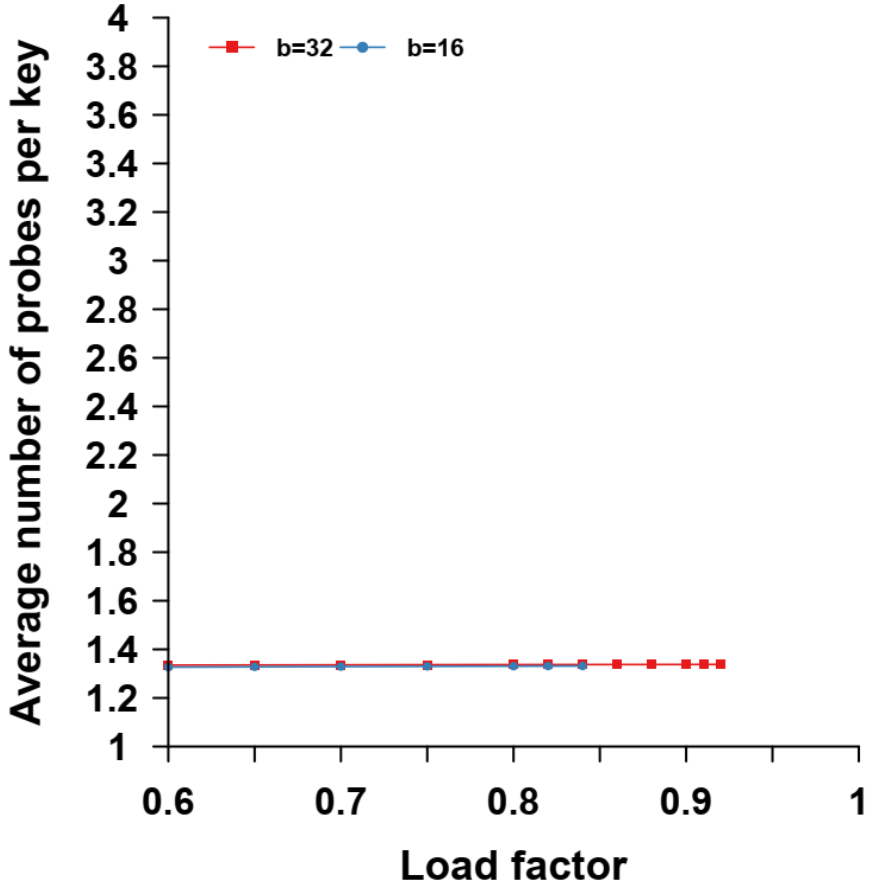


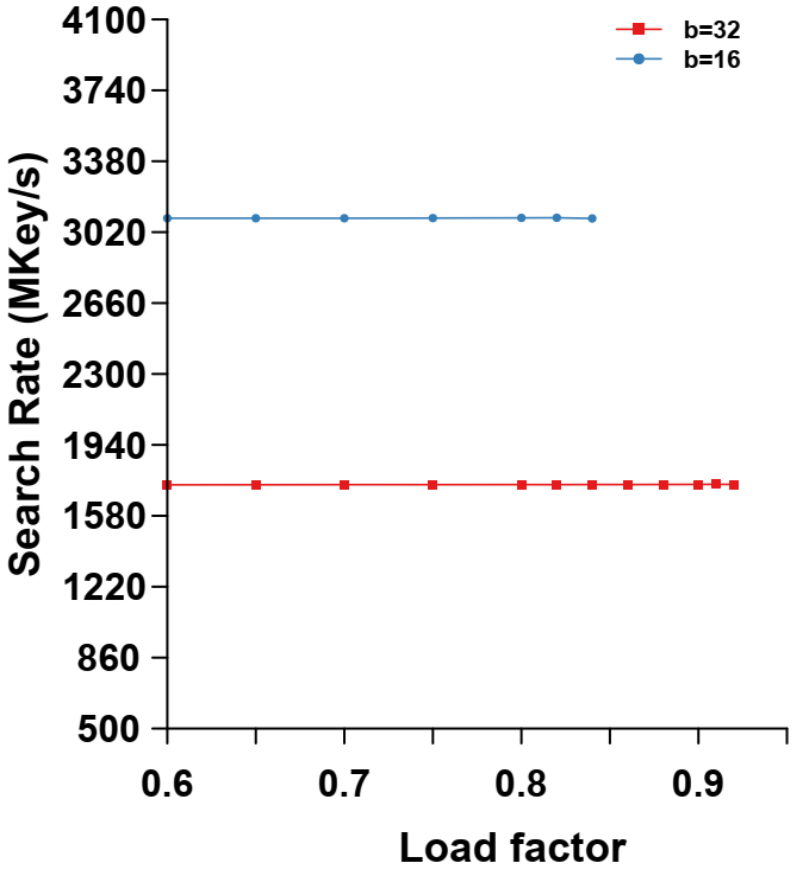


BP2HT → *Power of two HT, b = 16, 32*

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

P2HT find performance (positive queries)

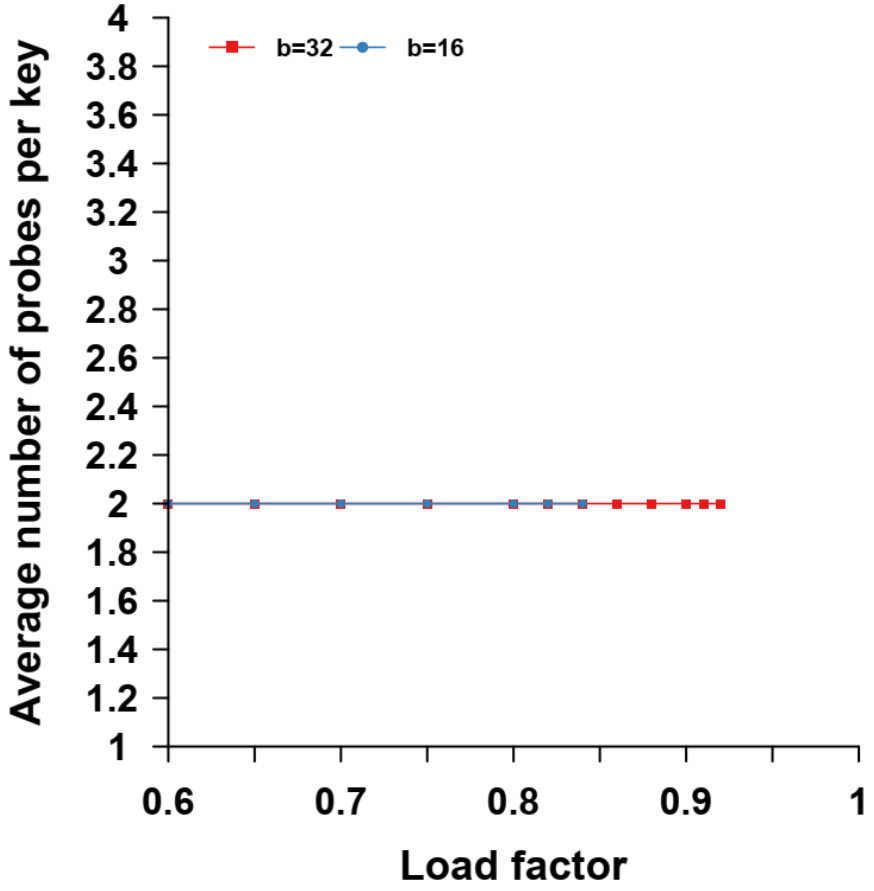


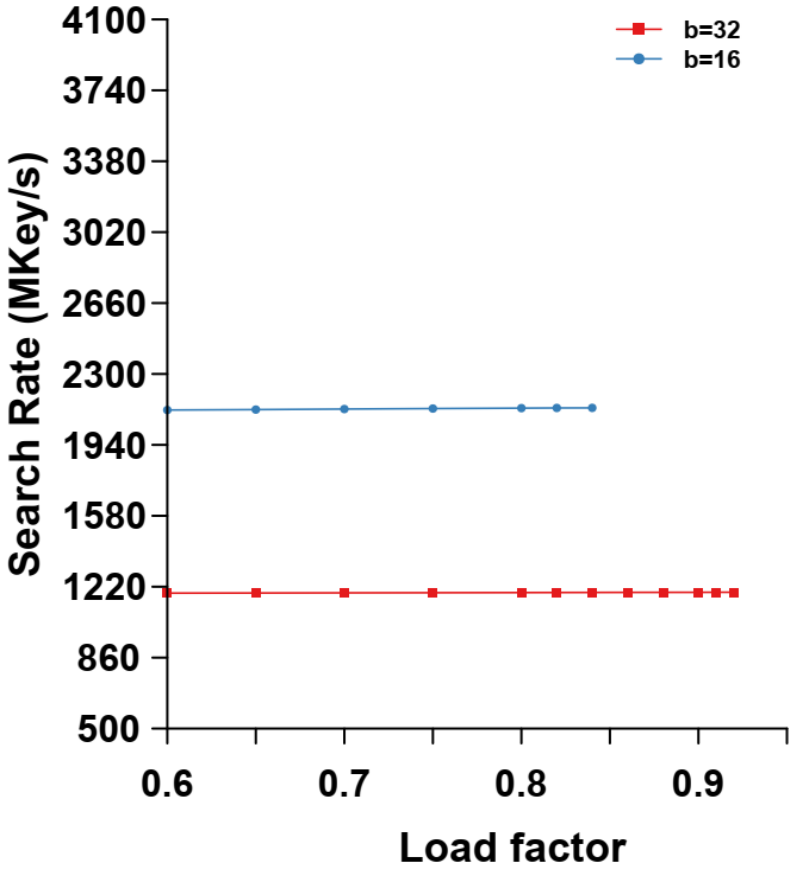


BP2HT → *Power of two HT, b = 16, 32*

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

P2HT find performance (negative queries)

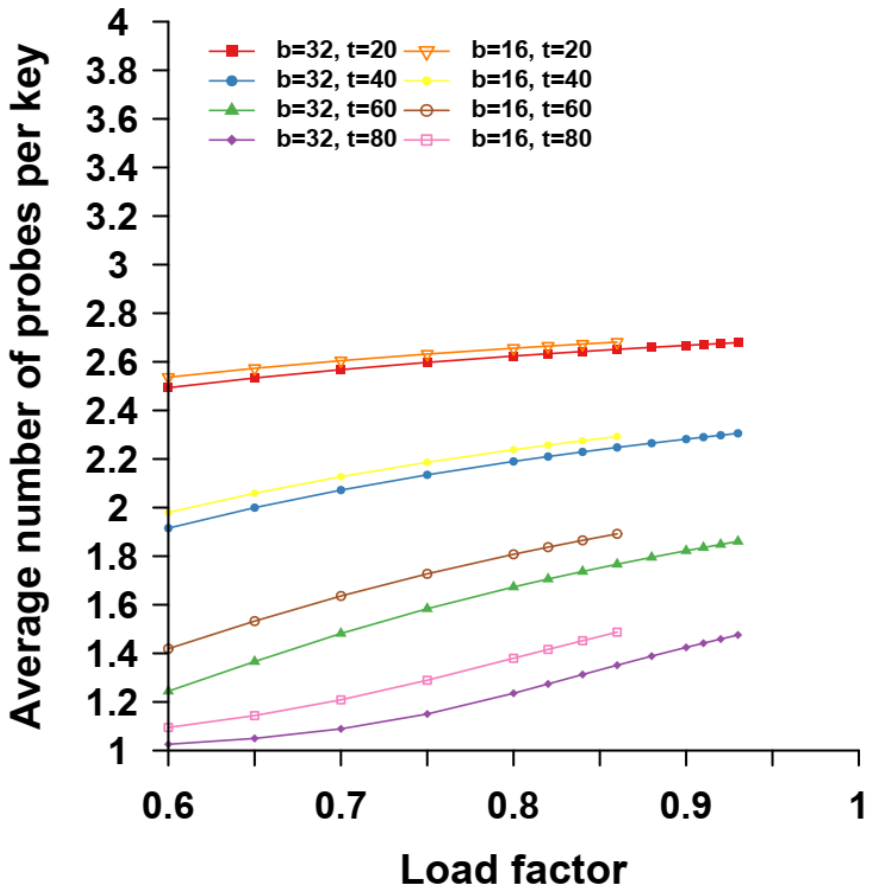


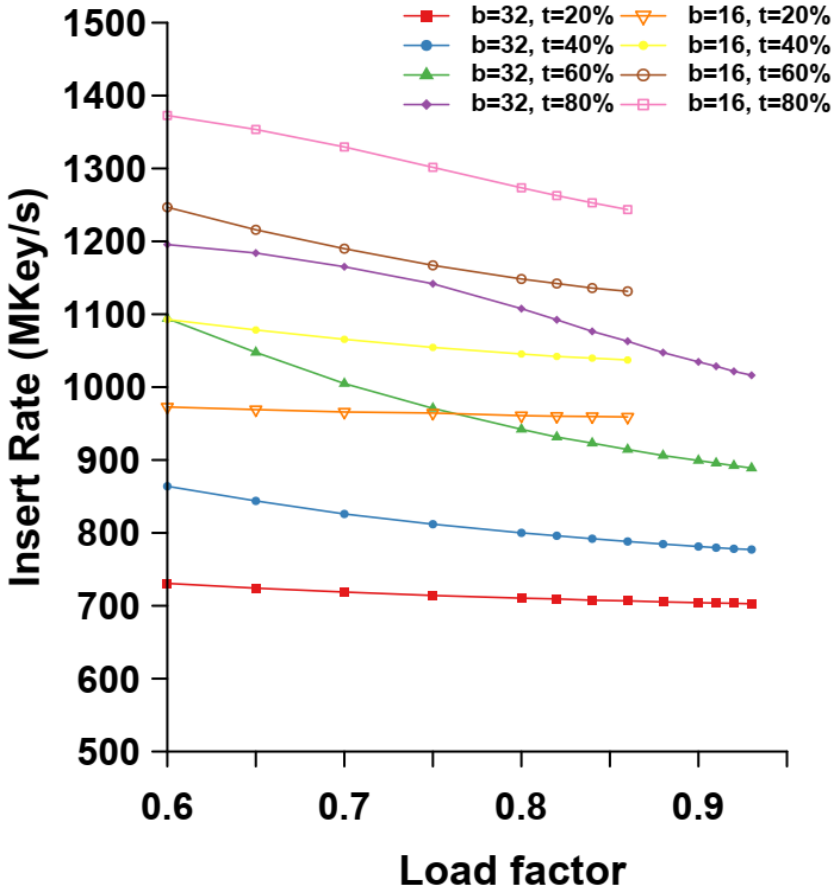


BP2HT → *Power of two HT, b = 16, 32*

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

IHT insertion performance

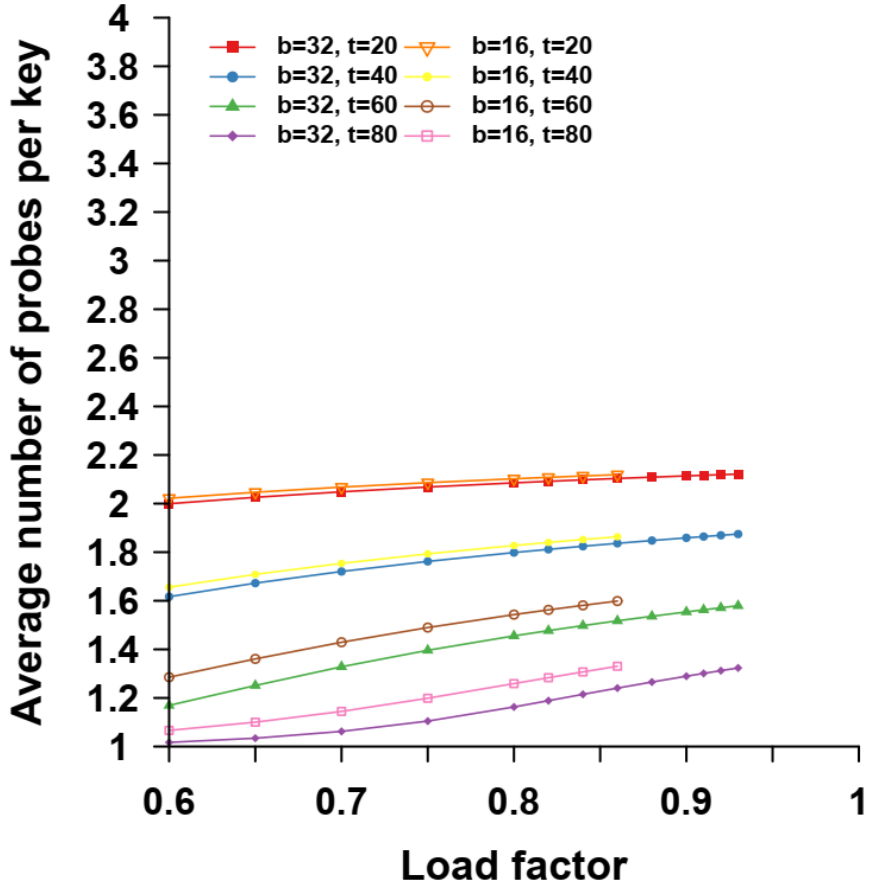


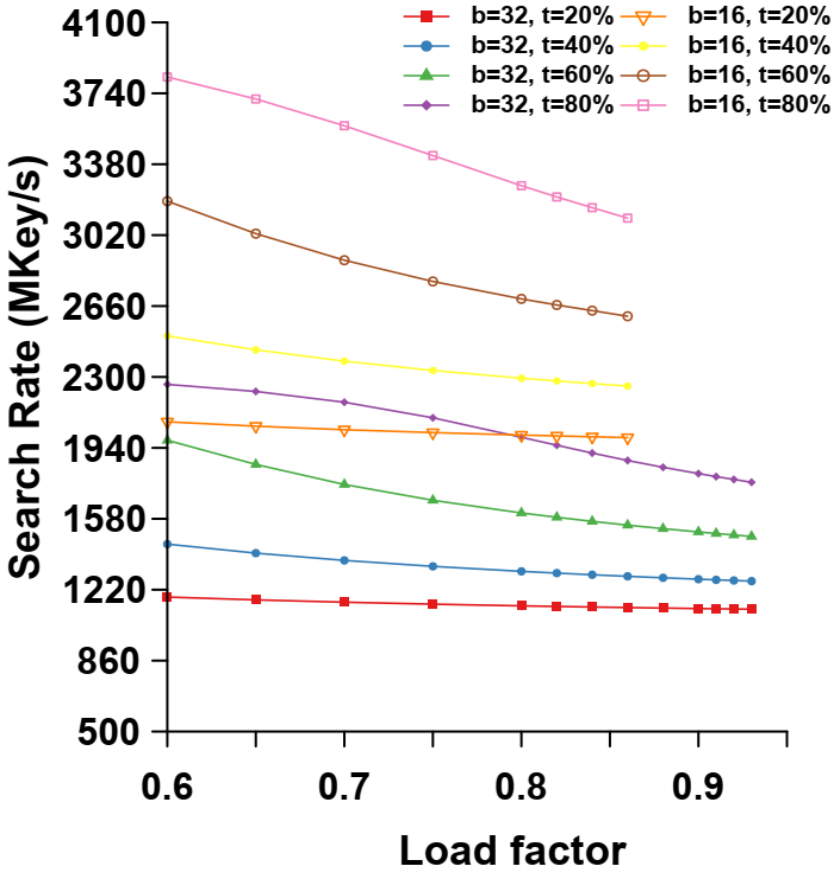


IHT → *Iceberg HT, b = 16, 32, threshold t*

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

IHT find performance (positive queries)

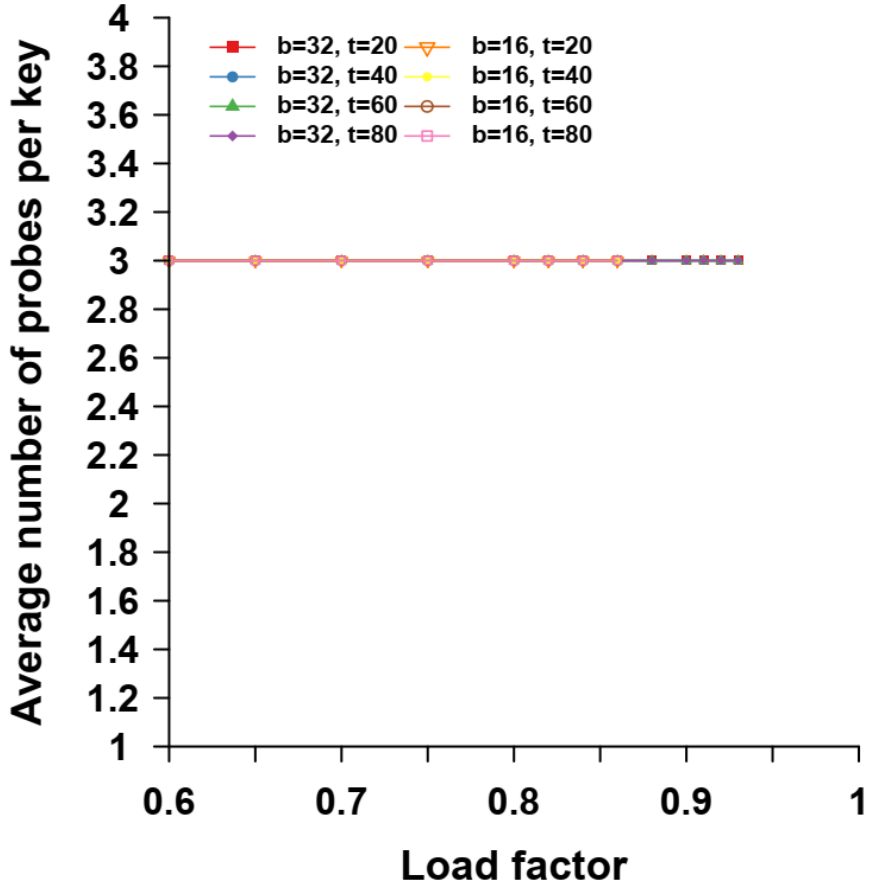


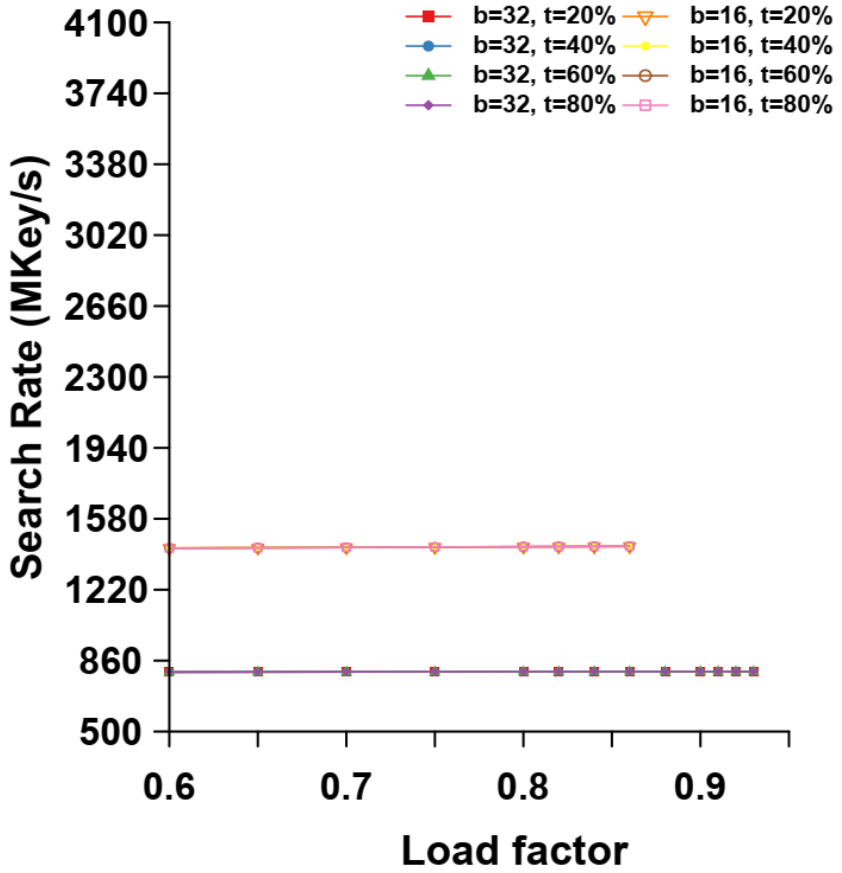


IHT → *Iceberg HT, b = 16, 32, threshold t*

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

IHT find performance (negative queries)

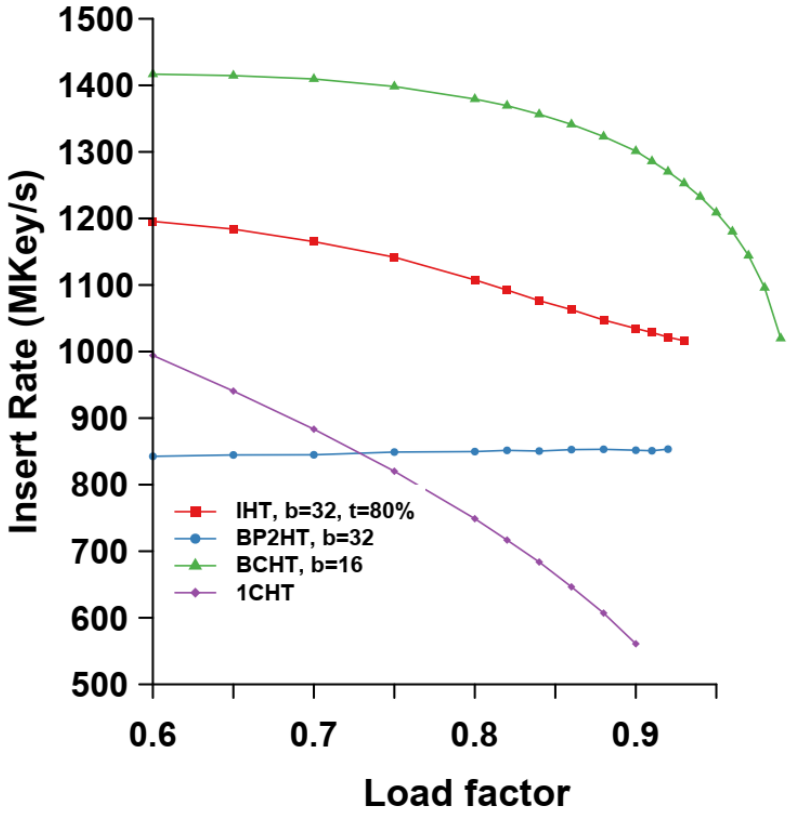


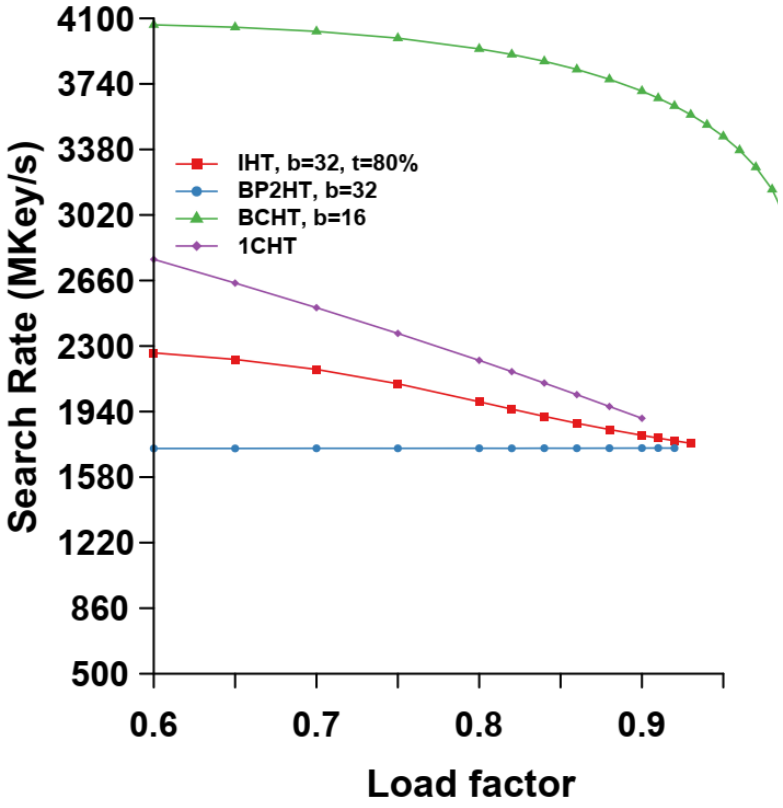


IHT → *Iceberg HT, b = 16, 32, threshold t*

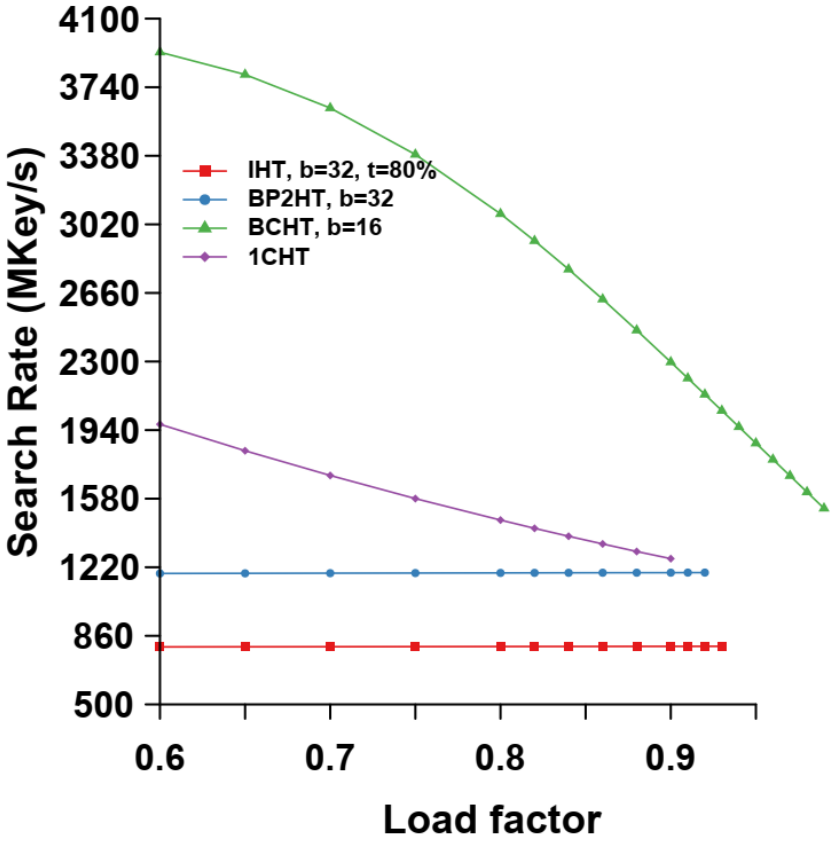
GPU is TITAN V, peak bandwidth 652.8 GB/s
4 bytes keys (uniform random), 4 bytes values

Query and build rates (across implementations)





Positive queries



Negative queries

1CHT → *Cuckoo HT, b = 1*
BCHT → *Cuckoo HT, b = 16*
BP2HT → *Power of two HT, b = 32*
IHT → *Iceberg HT, b = 32, t = 80%*
"best" from each family

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

Acknowledgments


Adobe


- 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?

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
BP2HT	0.92	2	up to 2	yes
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, $b = 16$	BP2HT, $b = 16$

Recommendations

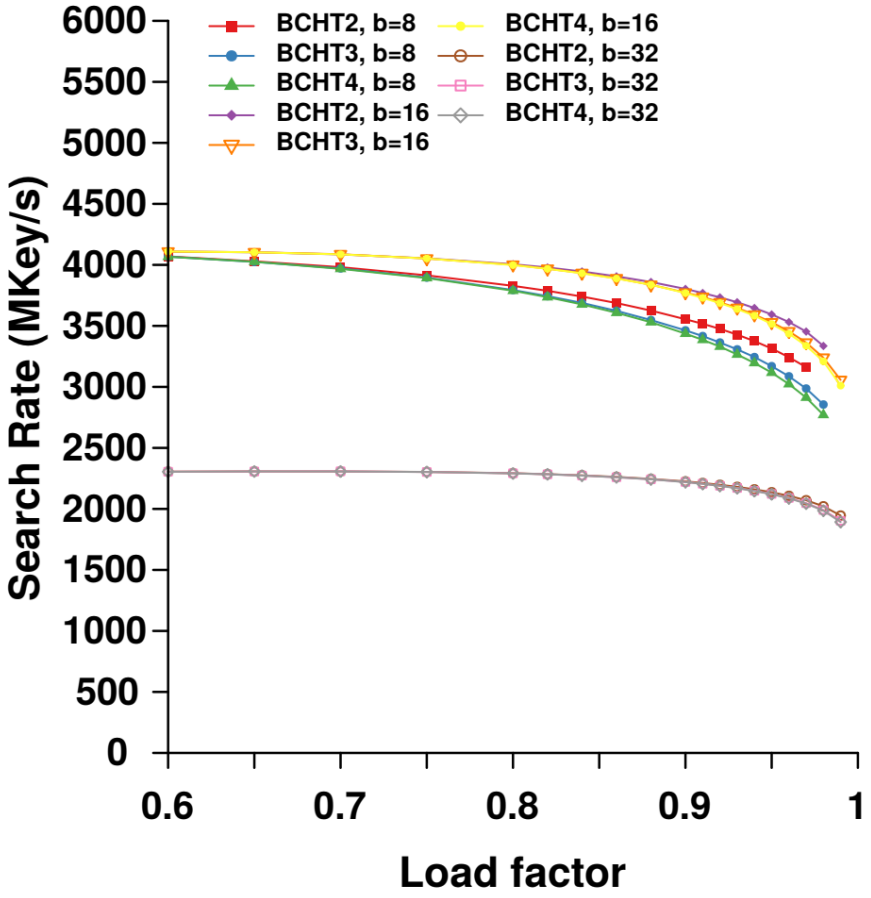
Summary and future work

- 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?

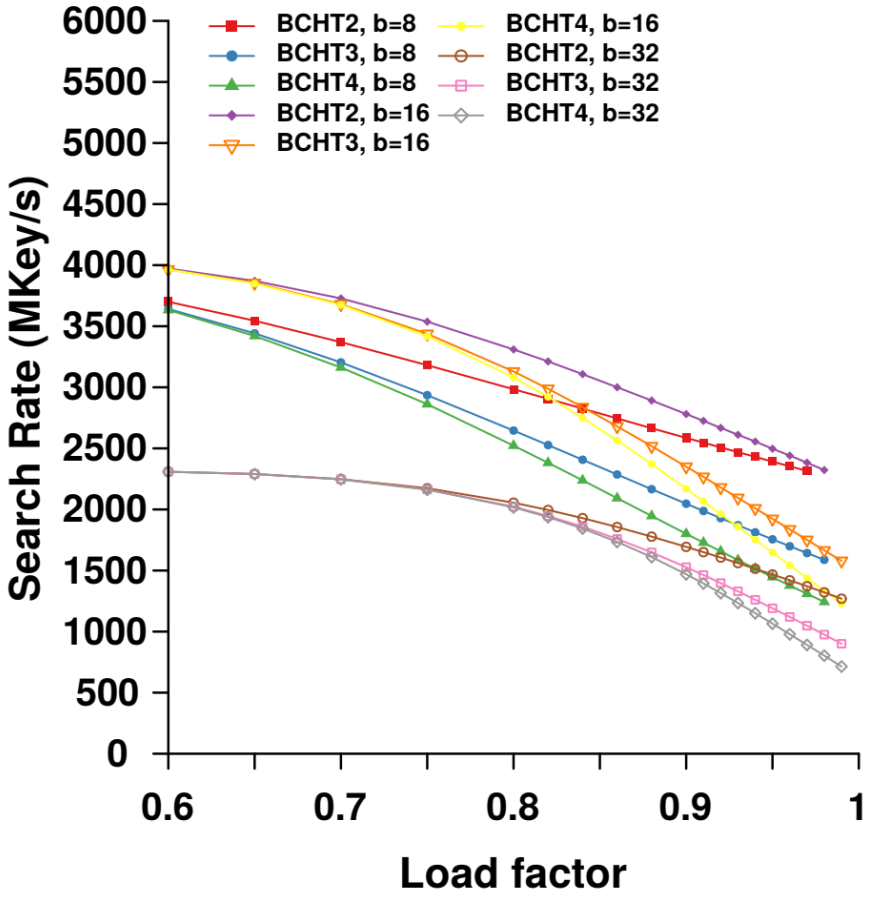
- Iceberg hashing:
  - High load factors: different secondary probing scheme
  - Improve negative queries performance: use a quotient filter

Slides++

Variable number of hash functions

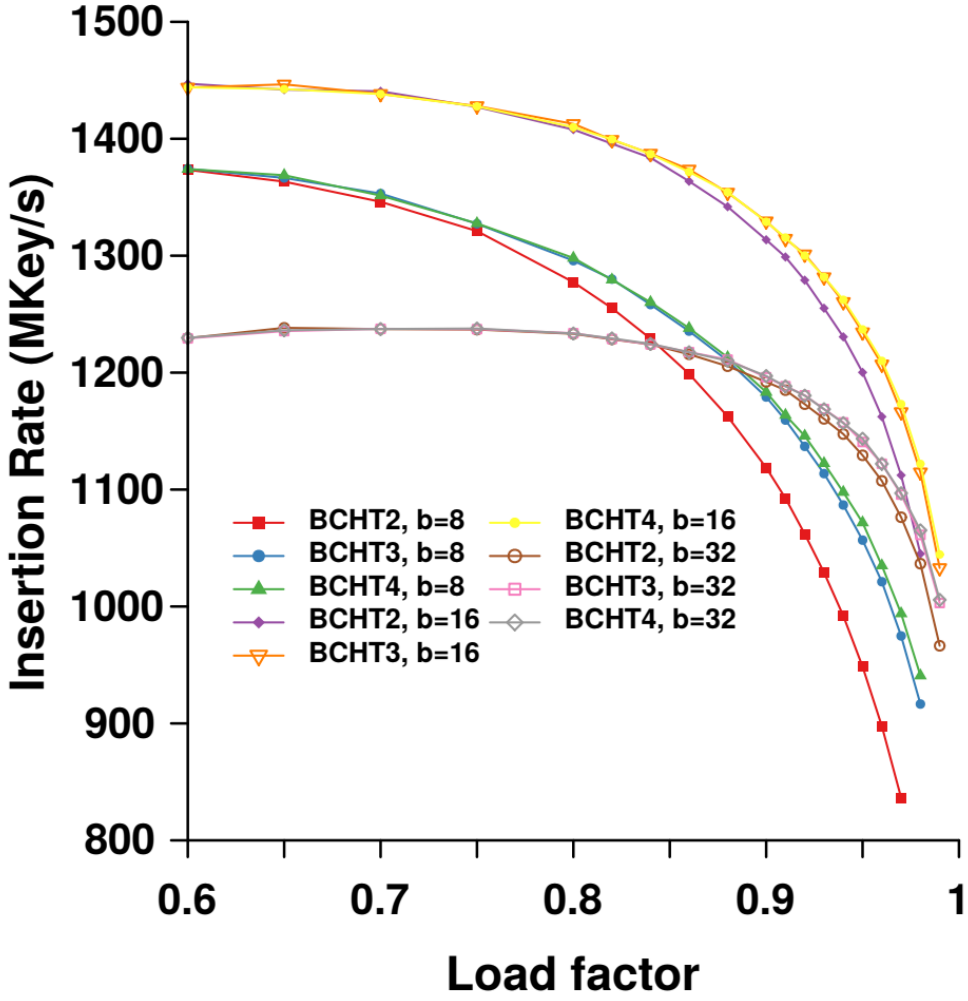


+ve queries



-ve queries

Variable number of hash functions



BCHT: Insertion rates for constant load factors

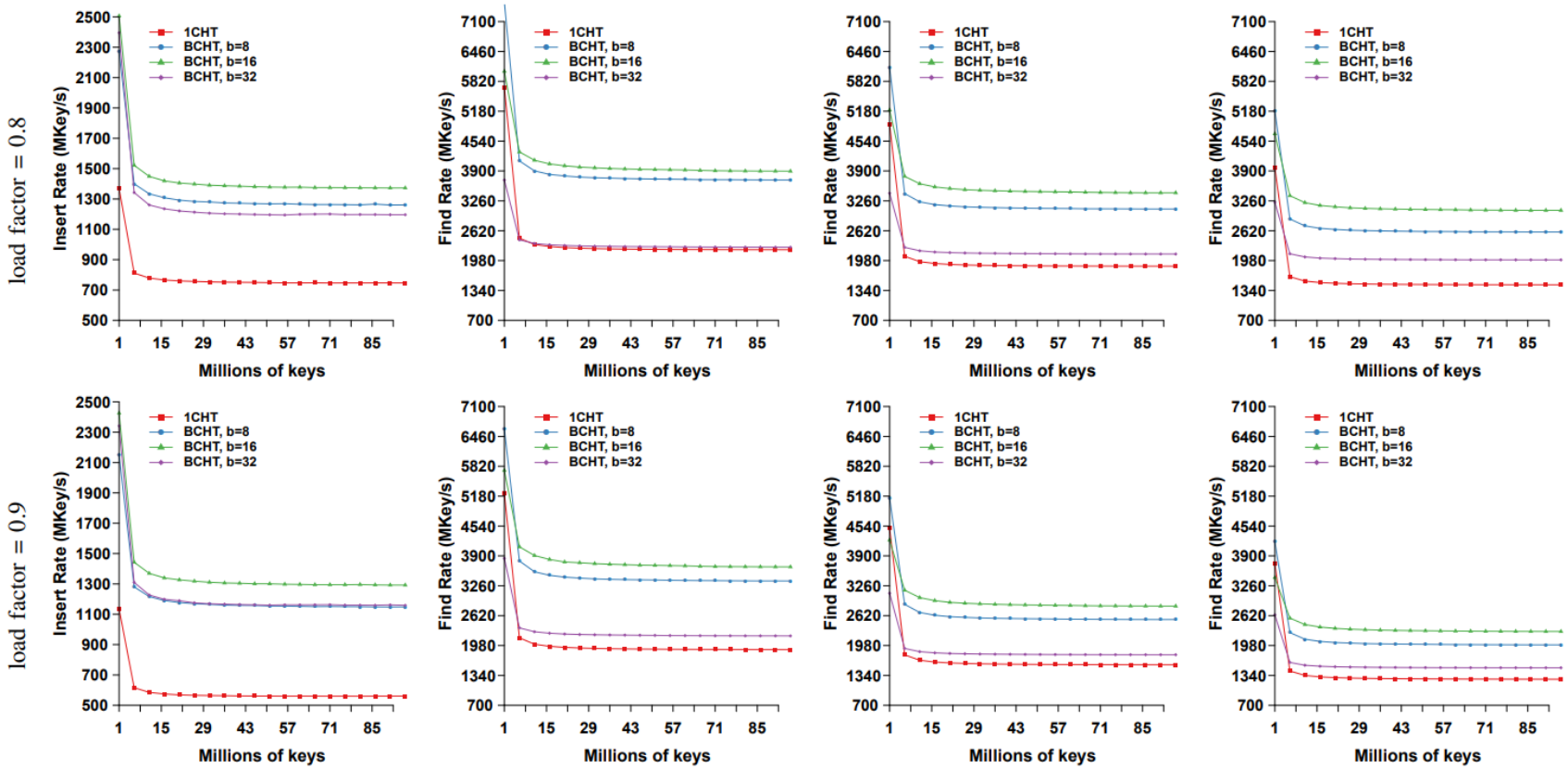


Fig. 4: BCHT throughput for insertion, 100%, 50%, and 0% positive queries (from left to right).

P2HT: Insertion rates for constant load factors

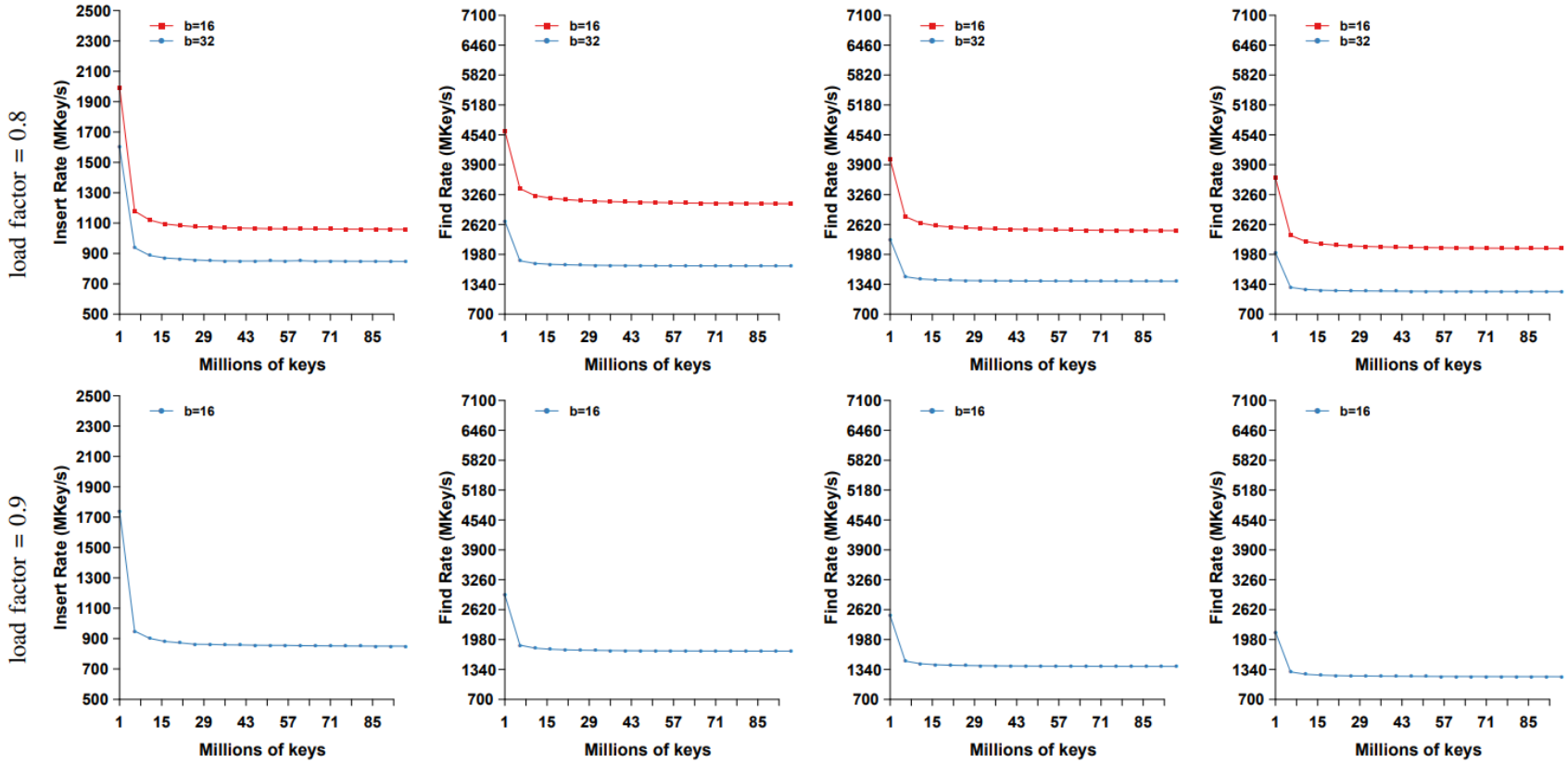


Fig. 7: BP2HT throughput for insertion, 100%, 50%, and 0% positive queries (from left to right).

IHT: Insertion rates for constant load factors

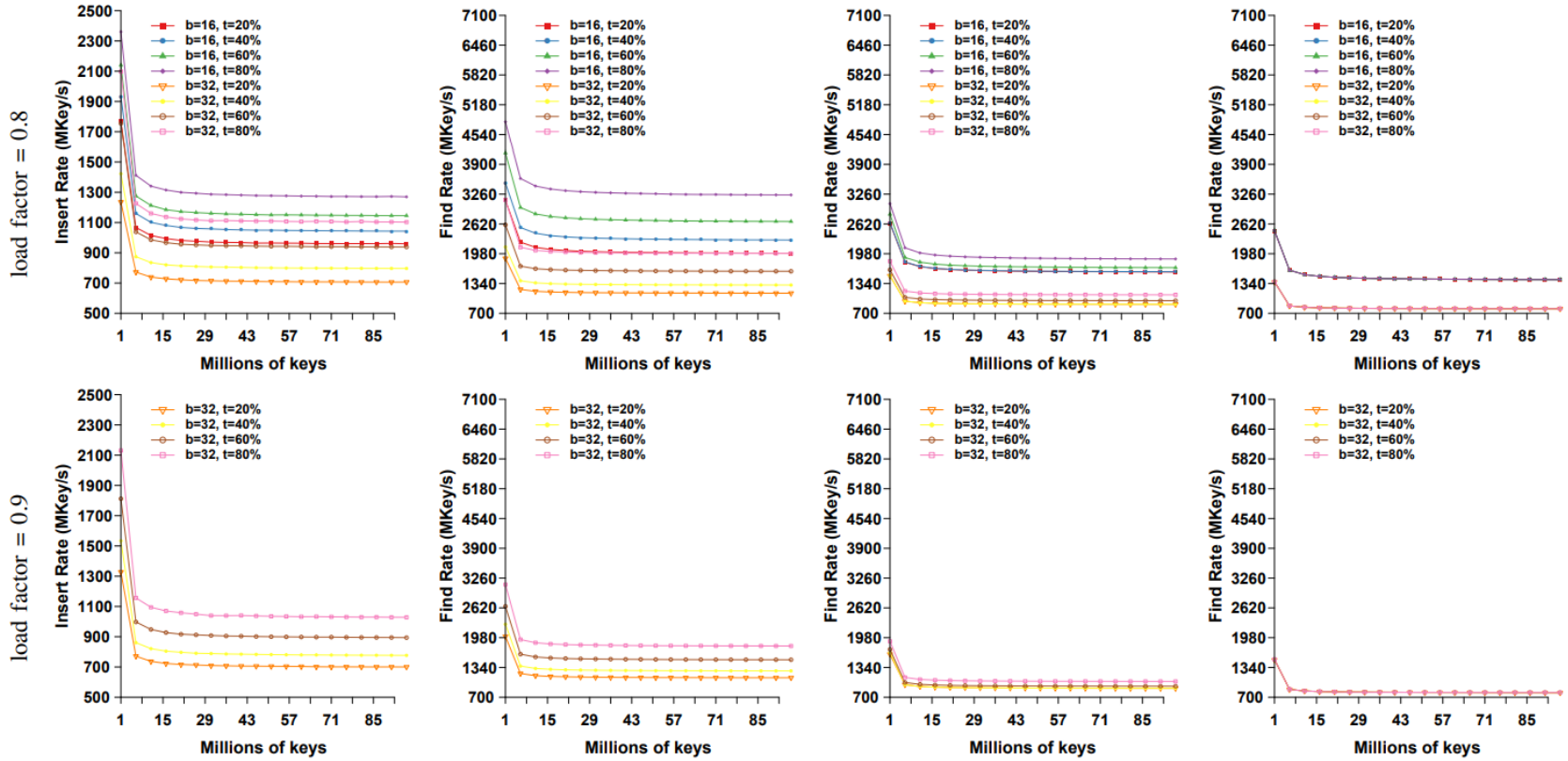


Fig. 10: IHT throughput for insertion, 100%, 50%, and 0% positive queries (from left to right).

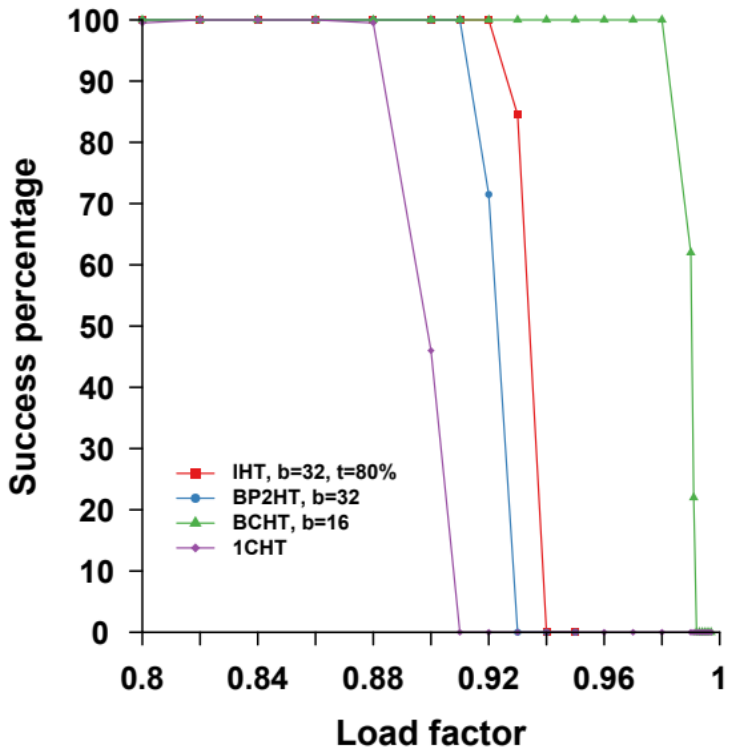


Fig. 2: Comparison of success rates across the recommended hash table variants. The input is 50M keys, run over 200 experiments.