Cuckoo Node Hashing on GPUs

Muhammad Javed, Hao Zhou,
David Troendle, Byunghyun Jang

Motivation and Purpose

- Concurrent data structure design is important
- Design fast and efficient Hash Table for GPUs
- Room for improvement in the design space

Background

Hash Tables

- Hash tables are data structures which aim to implement dictionaries
- Supported operations:
  - Insert(key, value)
  - Search(key)
  - Update(key, value)
  - Delete(key)

Cuckoo Hashing

- Hash table scheme that supports O(1) search, update, and delete. Supports O(1) amortized insert as well.
- Two hash functions each with their own indexed table
- Inserting a key can potentially evict older keys from the tables. Similar to how young cuckoo birds will push other young cuckoo birds from the nest.

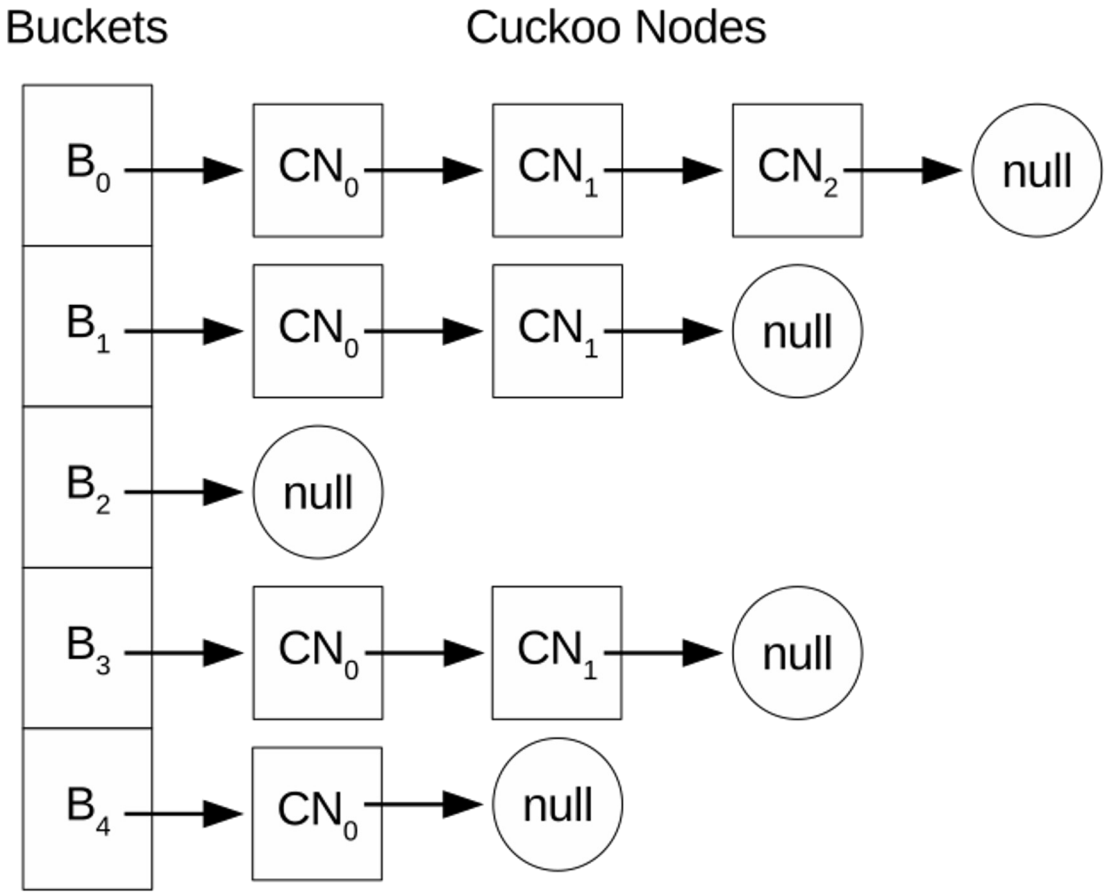
GPU Architecture

- SIMT Architecture
- Warps and Thread Divergence
- CUDA Programming Interface

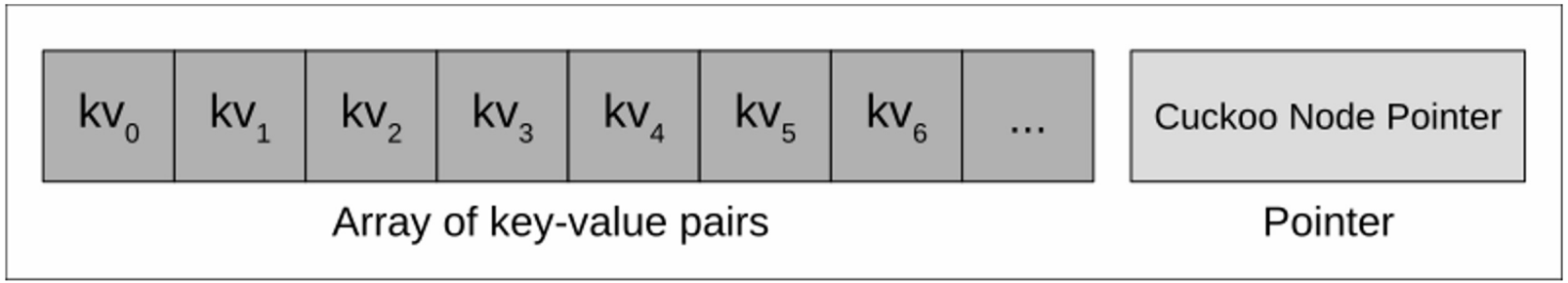
Cuckoo Node Hashing Scheme

Base Design

- The hash table contains a set of buckets where each bucket points to a list of Cuckoo Nodes
- Cuckoo Nodes contain a contiguous chunk of memory for storing key-value pairs and a pointer field for pointing to other Cuckoo Nodes
- A set of hash functions H is associated with the table and a hash function H_B



Example table of Cuckoo Node Hashing with 5 buckets



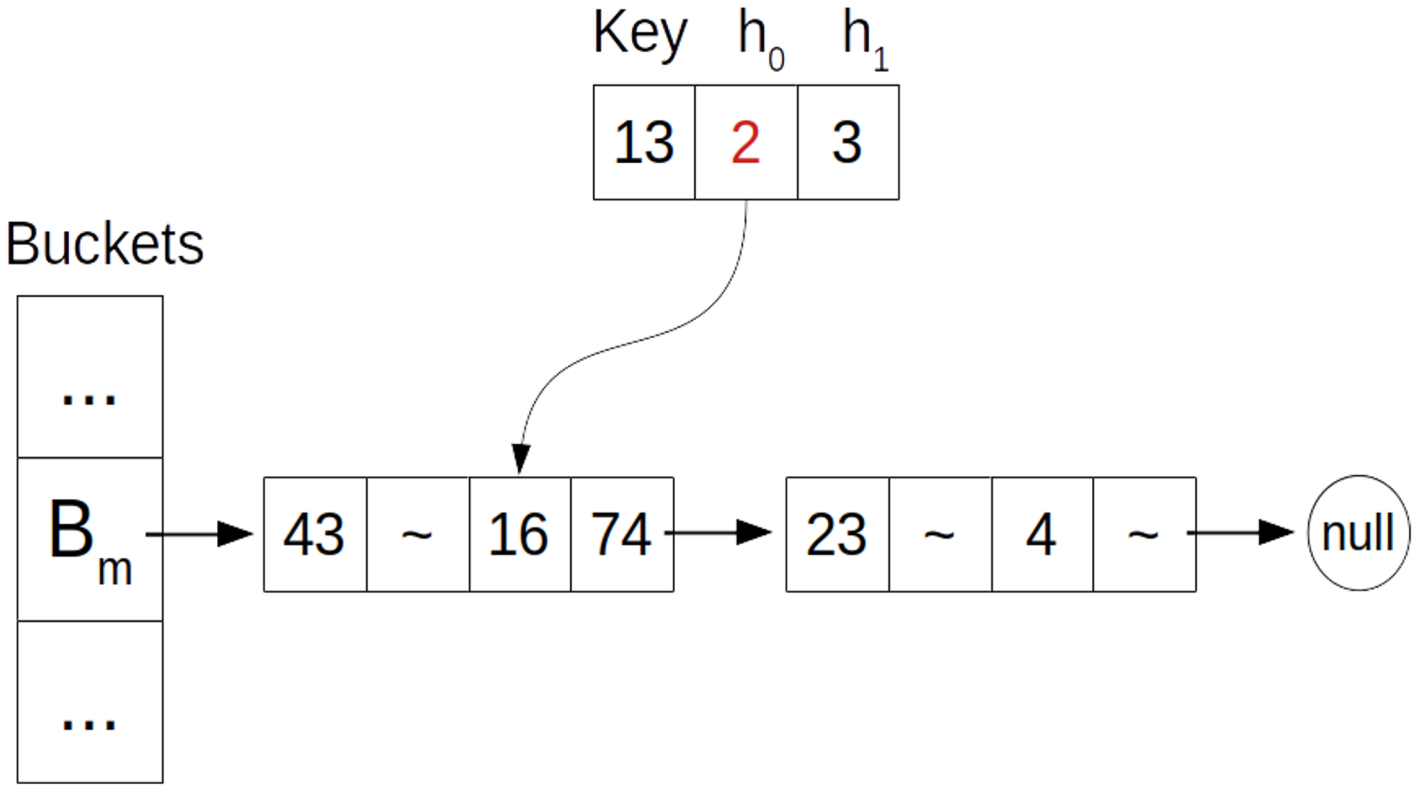
Magnified view of a single Cuckoo Node

The Cuckoo in Cuckoo Node

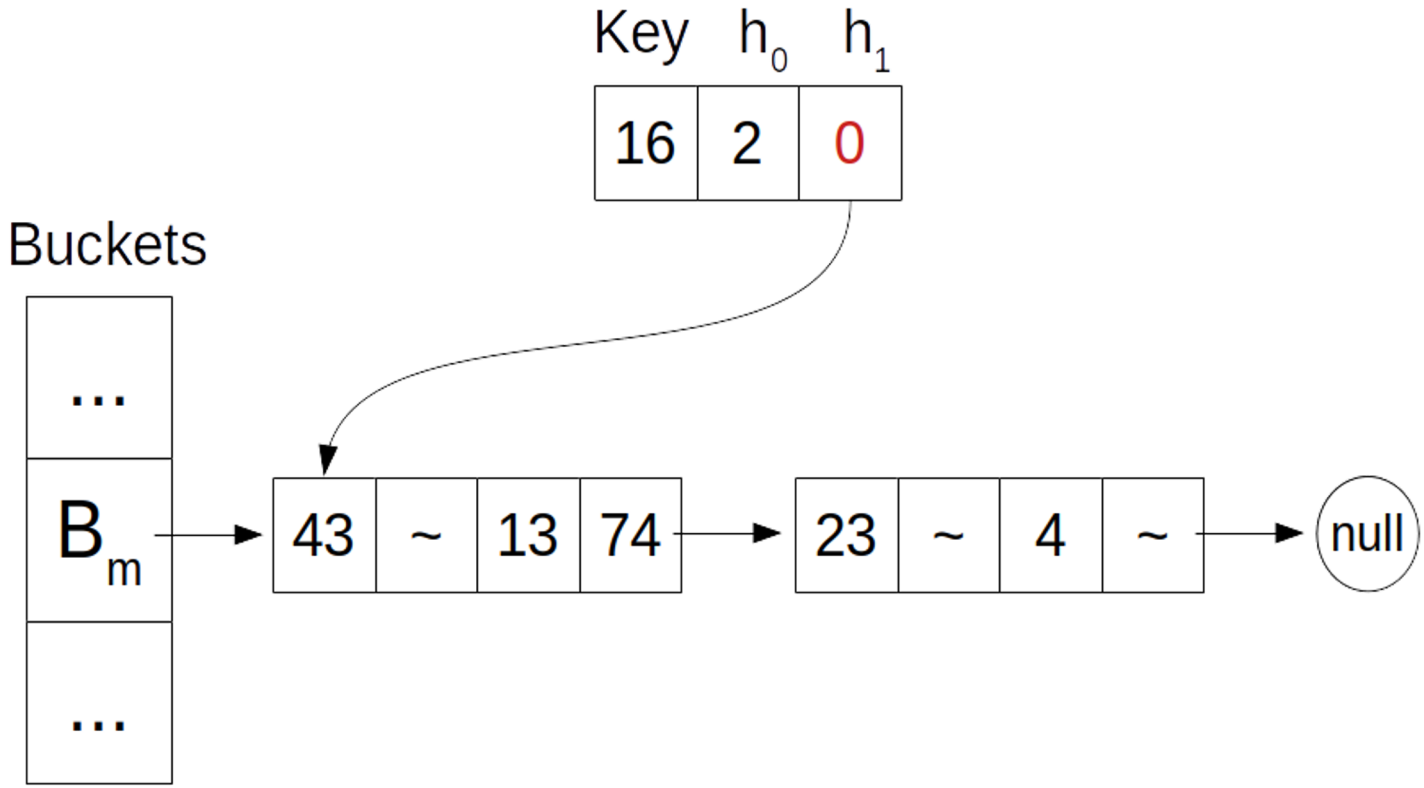
- Why call it Cuckoo Node?
  - Every Cuckoo Node is effectively treated as its own Cuckoo Hash Table
  - Evictions occur within the Cuckoo Node and across the chain
  - Sacrifices constant runtime operations for dynamic hash table

Inserting in the Cuckoo Node Table

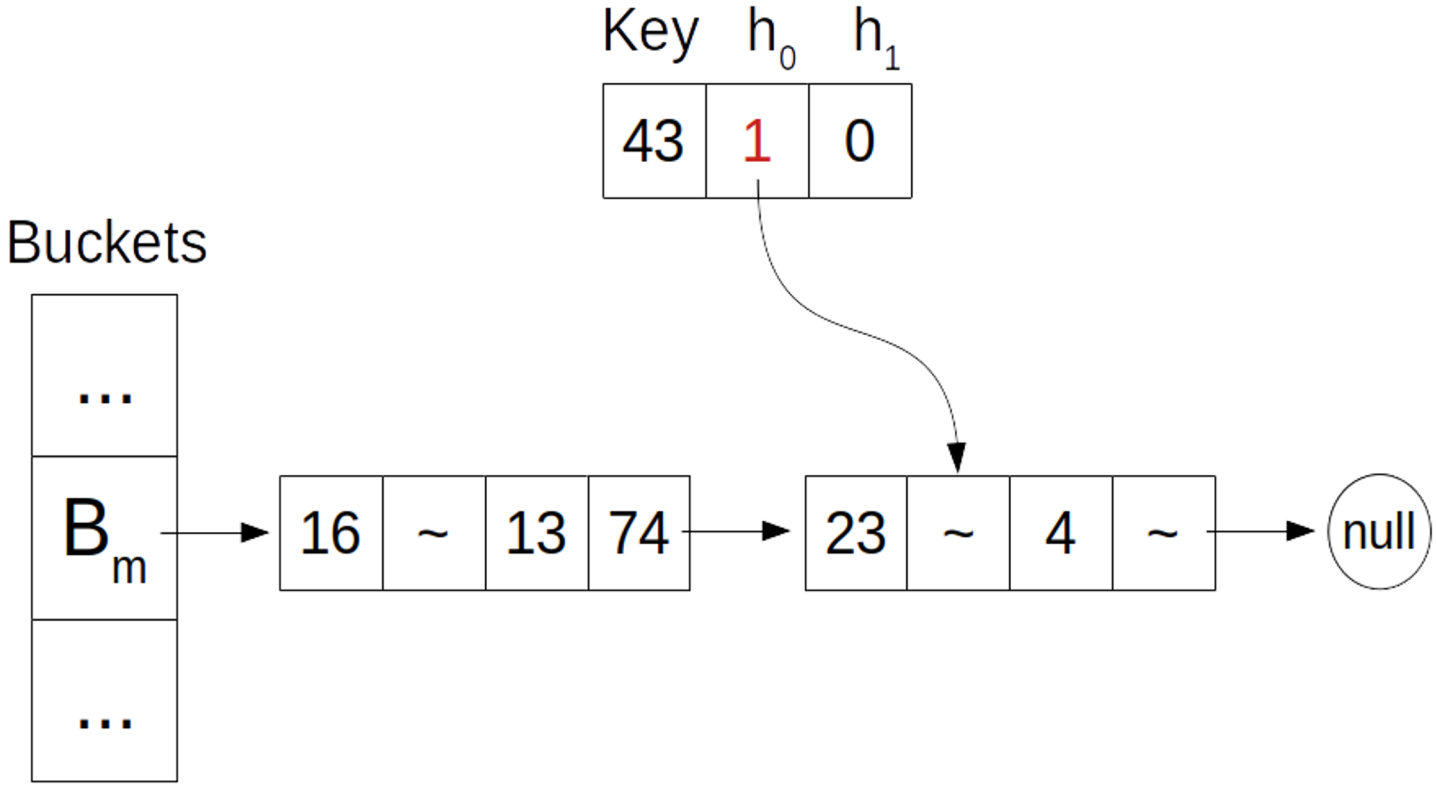
- Determine the bucket the key will go into by applying H_B on the key
- For every Cuckoo Node in the hashed bucket, we insert and evict keys until we insert a key into an empty slot
- This eviction process within a Cuckoo Node occurs Max-Loop number of times
- Once Max-Loop is reached within a Cuckoo Node, the algorithm moves on to the next Cuckoo Node



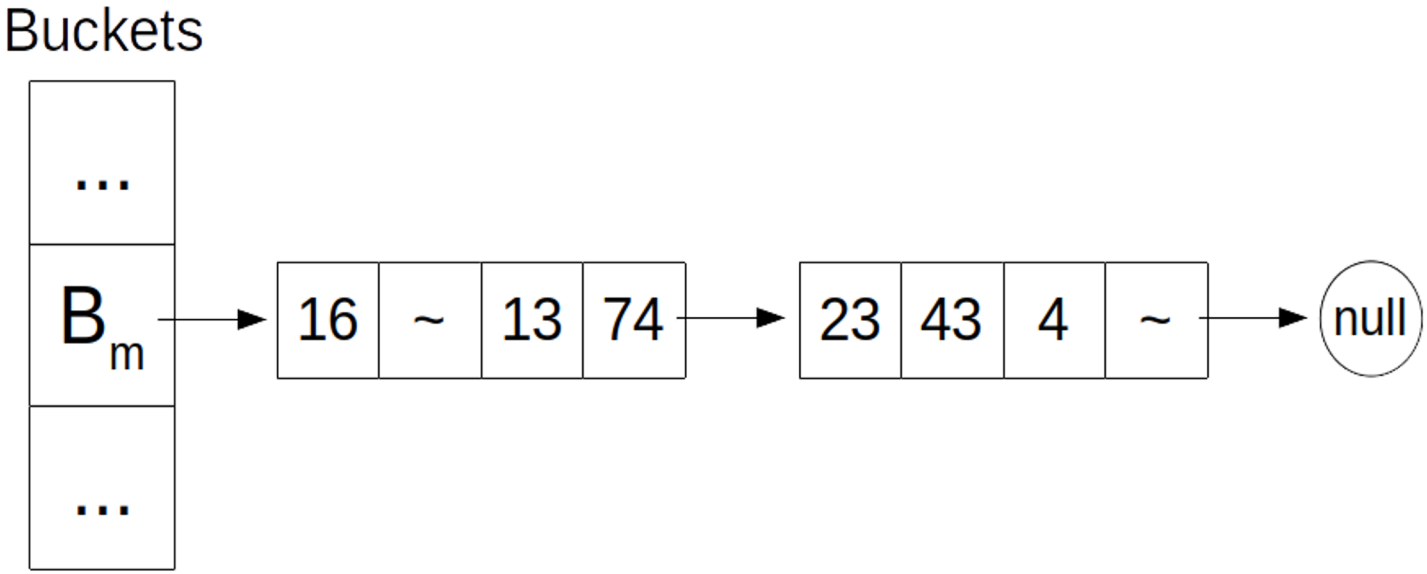
Example table with Max-Loop value of 2 and 2 hash functions (h_0 and h_1 in H). Key 13 has hashed to bucket B_m and an eviction of 16 occurs.



The evicted key 16 is hashed using h_1 in the set H. 43 is evicted and 2 evictions have occurred for this Cuckoo Node. Max-Loop is reached.



Since Max-Loop is reached, we move on to the next Cuckoo Node. The hashed location for the key 43 is empty, so we can place it there.



The insertion process for the key 13 is complete.

Searching, Updating and Deleting

- The search algorithm finds the bucket for the key and applies all hash functions in H to begin comparisons in the Cuckoo Nodes
- Update is similar to search and is trivial once search is implemented
- Delete is also similar to search and marks the key-value pair as logically deleted

Free Cuckoo Node Management

- When the Cuckoo Node Table is initialized, a concurrent stack has a set of Cuckoo Nodes loaded into it
- When a thread that is performing an insert needs a Cuckoo Node it can pop the stack
- A clean operation, which cleans the table of empty Cuckoo Nodes, pushes the Cuckoo Nodes onto the stack

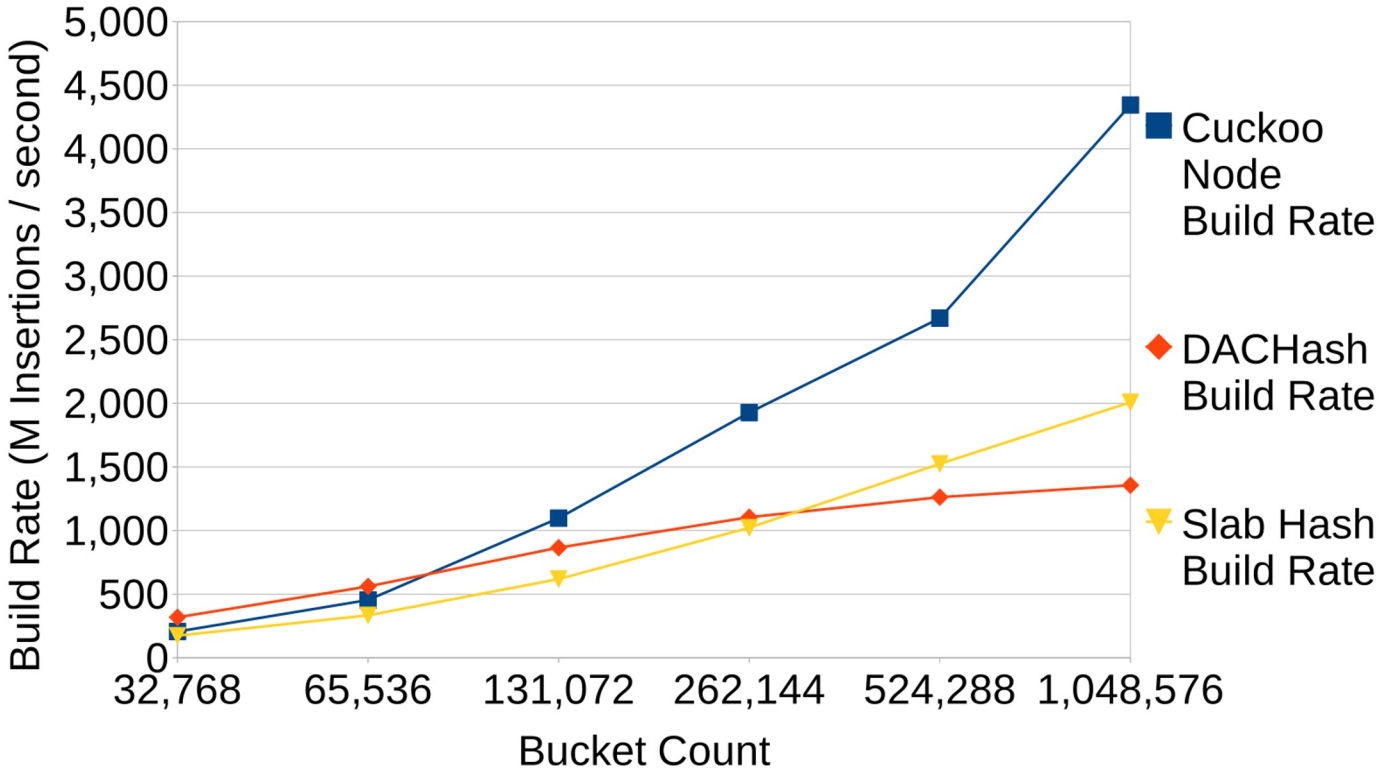
Data Reordering Algorithm

- Occurs before every hash table operation
- Reorders input data based on which buckets the keys hash to
- Heuristic to sort keys by hash values
- Utilizes coalesced memory accesses and warp-level primitives to achieve an efficient GPU solution
- Drops keys into buckets which hash values are mapped to

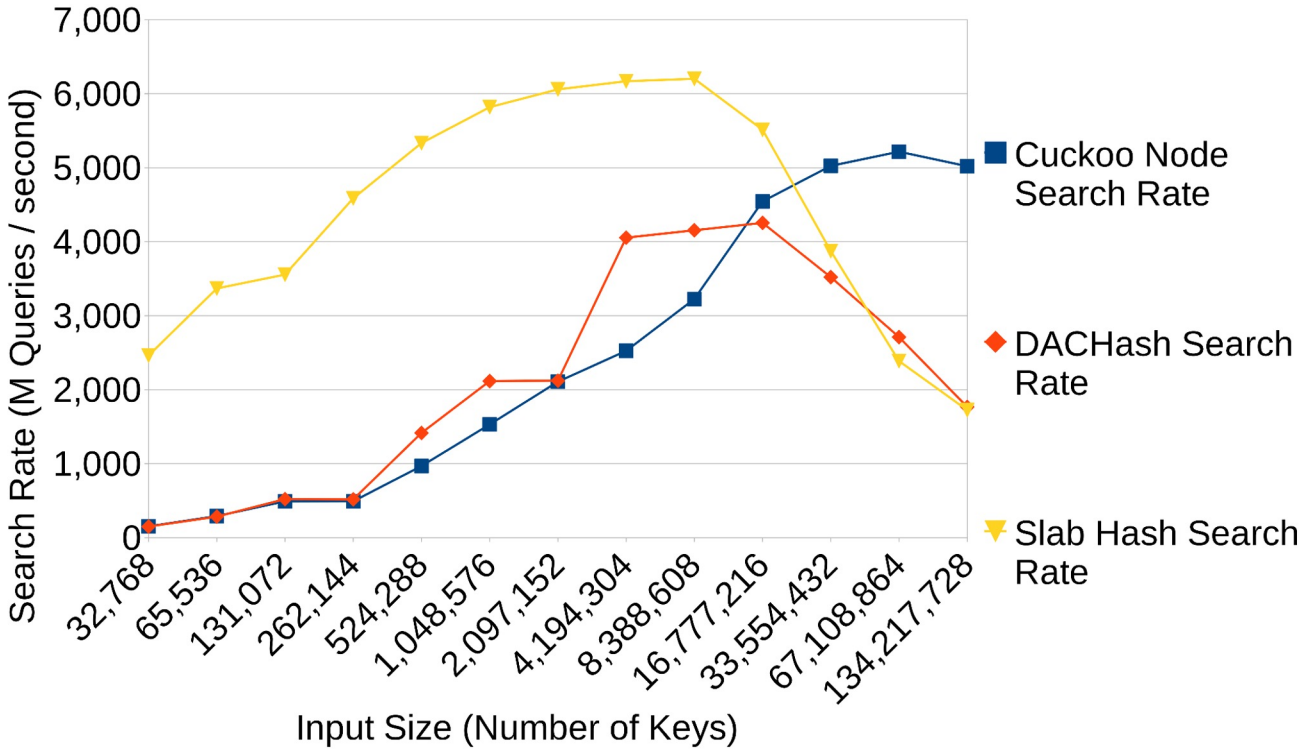
Experimental Results

Experimental Environments and Comparisons

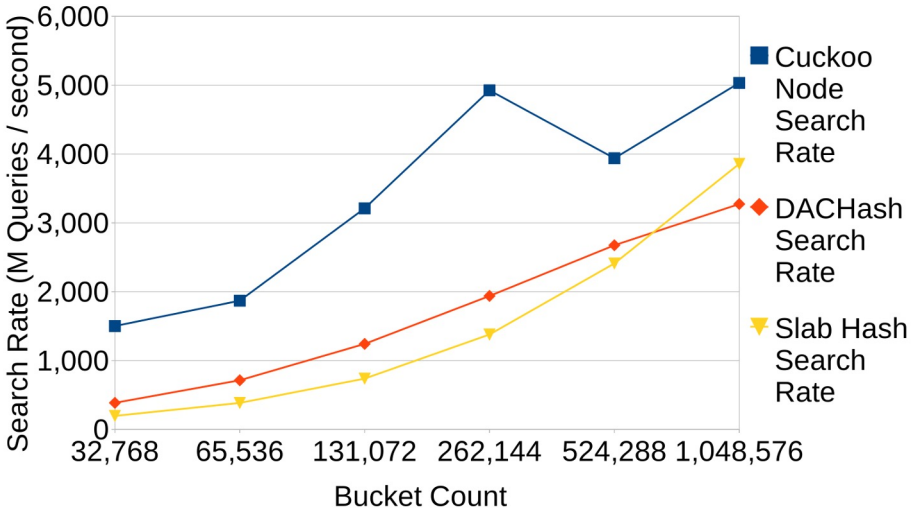
- Environment:
  - NVIDIA RTX 3090 GPU
  - Ubuntu 20.04
  - CUDA 11.4
- Comparisons made to state-of-the-art GPU hash tables:
  - Slab Hash
  - DACHash

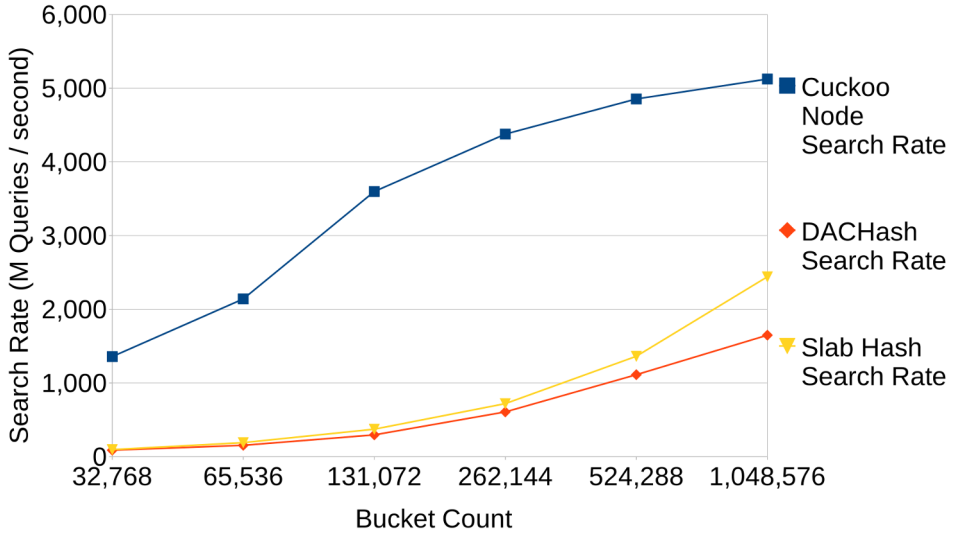


Results from an experiment where 2^{25} elements were inserted into the table across varying bucket counts. Cuckoo Nodes were configured to hold 128 key-value pairs, Max-Loop was set to 64, and 4 hash functions were in the set H.



Results from an experiment where 2^{25} elements were searched for in the table across varying input sizes. Same configuration as before except bucket count is 2^{20} for all tables.





Results from an experiment where 2^{25} elements were search for in the table across varying bucket counts. The left graph is when the keys are guaranteed to exist in the table and the right graph is when keys do not exist in the table. Table configurations are same as previously mentioned.

Buckets	Cuckoo Node Hashing		DACHash	
	Key-Found	Key-Not-Found	Key-Found	Key-Not-Found
32768	16.92	36.59	512.99	1039.06
65536	8.89	20.15	256.99	527.04
131072	4.82	12.03	128.99	271.03
262144	2.65	8.00	64.99	143.40
524288	1.52	5.32	32.99	79.61
1048576	1.19	4.05	16.99	47.26

Probes performed by two different hash table schemes during searches across varying bucket counts.

Conclusions

- We proposed a dynamic and efficient hash table for GPU systems which outperforms other state-of-the-art GPU hash tables
- We proposed a data reordering algorithm for hash table operations which efficiently utilizes the power offered by GPUs

End