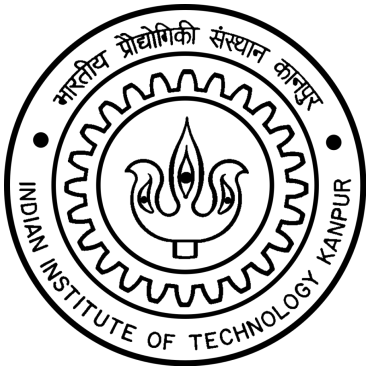


CS636: Analaysis of Concurrent Programs

Insert-Optimised Dynamic Hash Tables for GPUs

Team Members:

Keshav Banka (241110032)
Vinayak Devvrat (241110079)

April 24, 2025

Contents

Chapter 1 Introduction

Hashing tables are an essential part of modern computing because they enable scalability across algorithms, AI, and ML in addition to quick data retrieval and memory efficiency. With uses ranging from duplicate detection in large-scale data processing to feature storage in machine learning and state caching in artificial intelligence, they are an essential tool for optimising performance in computationally demanding tasks.

Hash tables are crucial for algorithm design and applications involving artificial intelligence (AI) and machine learning (ML) due to their versatility. In AI applications, hash tables are widely used for state representation, caching, and memoization. Hash tables, for instance, are used in game-playing algorithms like Minimax with Alpha-Beta Pruning to store previously computed game states in order to avoid redundant calculations and significantly speed up decision-making. Machine learning (ML) uses hash tables to optimise feature storage, nearest neighbour searches, and data preprocessing.

Many machine learning models, including recommender systems, are based on collaborative filtering, where user-item interactions are stored in hash tables for convenient access. Another significant application is feature hashing, sometimes referred to as the "hashing trick," which hashes high-dimensional categorical features into a fixed-size space to minimise memory usage without sacrificing model performance.With the development of parallel computing, GPU-accelerated hash tables have emerged as a powerful tool for processing massive datasets at previously unheard-of speeds. These specialised hash tables make use of the massive parallelism of GPUs to facilitate high-performance lookups, insertions, and deletions in real-time processing applications like deep learning, large-scale graph processing, and high-frequency data analytics.

In this work, we profile the insert, search, and delete operations of four GPU-based hash table implementations (BGHT, SlabHash, WarpCore, and DyCuckoo) across different dataset sizes and duplicate percentages in order to perform a thorough performance analysis. We assess their effectiveness, scalability, and behaviour under various workloads using conventional runtime measurements in conjunction with NVIDIA's Nsight Compute for comprehensive hardware-level profiling.

Building on these discoveries, we create a hybrid hash table that enhances DyCuckoo's insert performance for a range of workloads by leveraging WarpCore's high-throughput operations' advantages through parallelism. The efficacy of the hybrid implementation is

then evaluated using the same exacting profiling. The main results and conclusions are finally presented, along with trade-offs between the various strategies and the situations in which each—or the hybrid—performs best. This work provides valuable insights for selecting or designing GPU-accelerated hash tables based on specific application requirements.

Chapter 2 Related Works

Some rudimentary implementations of GPU based concurrent hash tables utilise the cuckoo hashing algorithm which is performed by a single kernel [1]. The obvious limitation is that a thread block will not complete until all its threads are through. The hash tables are further specialised into multi-valued and compact hash tables. The uncoalesced memory accesses also cause a certain bottleneck as GPUs are optimised for coalesced memory access. These limitations are addressed in some newer implementations.

Three different hash table designs are explored in [2]. The implementations, namely Bucketed Cckoo Hash Table (BCHT), Bucketed Power-of-Two-Choices Hash Table (BP2HT) and Iceberg Hash Table (IHT). The three implementations are based on design strategies such as Cuckoo hashing (BCHT), "least-loaded bucket" (BP2HT) and thresholding (IHT). BCHT emerges a winner and outperforms the other two. However, the high throughput and high load factor benefits are offset by stability issues and possible negative query overheads.

A fully dynamic, concurrent hash table for GPUs, called the Slab Hash is presented in [3], which is built on a custom linked-list structure called the Slab List, which is optimized for GPU execution by reducing memory and control divergence and exploiting warp-synchronous execution. They also introduce a new dynamic memory allocator called SlabAlloc, designed for fast, concurrent allocation on GPUs.

The DyCuckoo implementation [4], devises a 2-in-d cuckoo hash that ensures a maximum of two lookups for find and deletion operations, while still retaining similar performance for insertions as general cuckoo hash tables. It achieves efficiency while enabling fine-grained memory control. Moreover, the performance of DyCuckoo is not significantly affected by the number of hash tables.

An even more scalable approach is presented in WarpCore [5]. The "GPU-accelerated approach" mitigates the addressing limitations of SlabHash [3]. Moreover, it supports diverse hash table types (single/multi-value, bucket lists, counting tables). The approach proves to be scalable and memory-efficient. It leverages Cooperative Probing Scheme which combines warp-level parallelism with double hashing to improve memory access patterns. It significantly outperforms state-of-the-art libraries and has been successfully bench-marked with real-world applications.

A thorough analysis and benchmarking of any concurrent data structure is essential to gauge its applicability in real world applications. Such parameters have been explored in [6]. These include probe complexity, placement strategy and bucket size.

Chapter 3 Profiling Existing Implementations

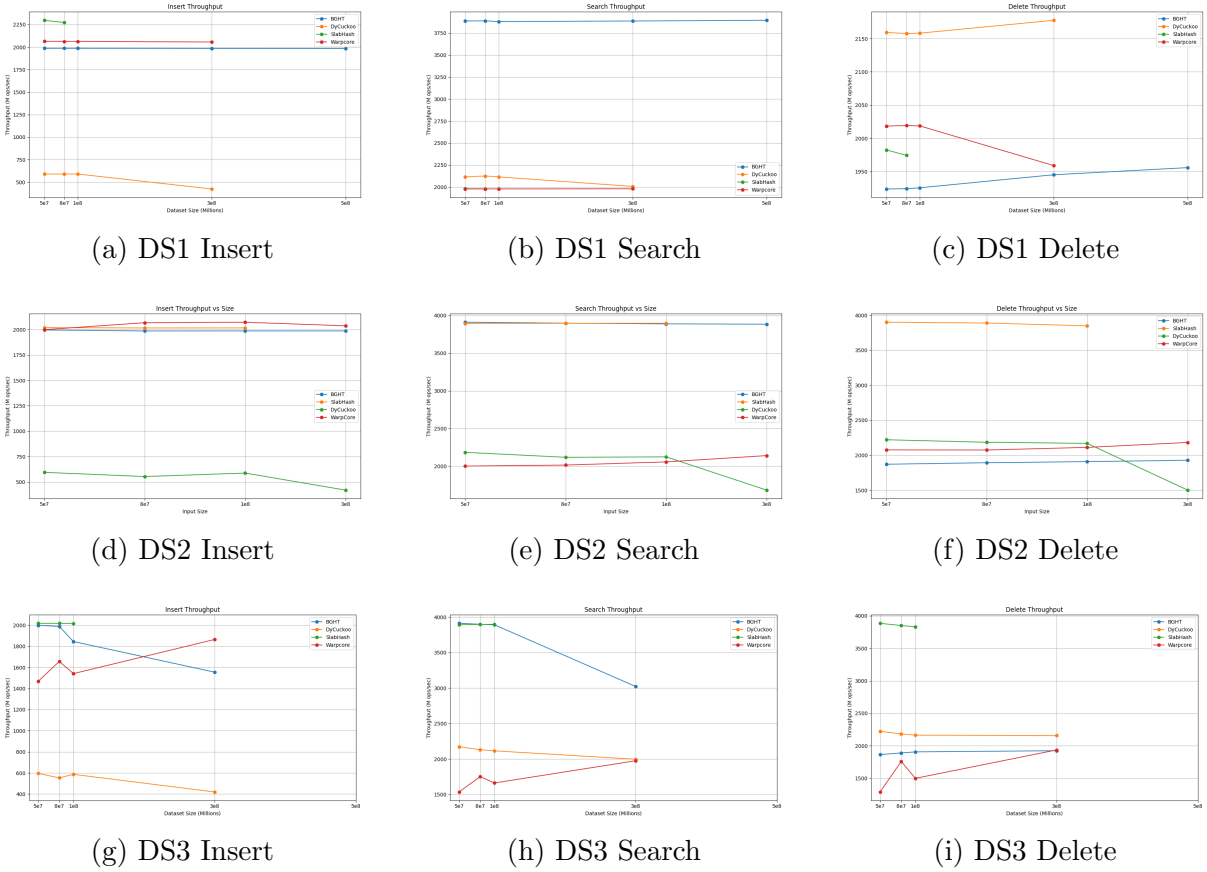
We profiled four existing implementations namely BGHT [2], SlabHash [3], WarpCore [5] and DyCuckoo [4]. This chapter presents a comparative analysis of these implementations based on scalability, memory usage etc.
The specifications of several parameters for the analysis are :

- Number of operations, $N \in \{5e7, 8e7, 1e8, 3e8, 5e8\}$
- Load factor = 0.85
- Batch size (if applicable) = $1e5$
- **System Specifications.** We profiled all the implementations on GPU3 which has the following specifications :
  - **GPU Model:** NVIDIA A40 (Ampere architecture)
  - **Memory:** 46 GB (FB), 65 GB (BAR1)
  - **Compute Mode:** Default
  - **ECC Mode:** Enabled
  - **Performance State:** P0 (maximum performance)
  - **Driver Version:** 550.54.15
  - **CUDA Version:** 11.4
  - **Temperature:** 57 °C (well within safe range)
  - **Graphics Clock:** 1740 MHz, **Memory Clock:** 7250 MHz
- **Dataset specifications.** We evaluate the performance of each implementation using three distinct datasets. Each dataset consists of three operation trace files — insert, search, and delete — containing key sequences with varying levels of duplication. The datasets are as follows:
  1. **DS1.** Size = $4e9$, insert trace with no key duplication, search trace with 30% of keys present in the insert trace with no duplication, delete trace with 10% of keys present in insert trace with no duplication.

2. **DS2.** Size = $5e9$, insert trace with 10% key duplication, search trace with 40% of keys present in the insert trace of which 20% are duplicates, delete trace with 10% of keys present in insert trace of which 40% are duplicates.
3. **DS3.** Size = $4e9$, insert trace with 20% key duplication, search trace with 40% of keys present in the insert trace of which 40% are duplicates and 10% are not present in the insert trace, delete trace with 10% of keys present in insert trace of which 50% are duplicates and 20% are not present in the insert trace.

The first four sections present a basic profiling wherein the throughput for each kernel (insert, search and delete) is noted over five consecutive runs. These are then plotted alongside other implementations. We also explore the possible reasons for the performance variability among these implementations. In the last section we focus on a more fine-grained profiling using NVIDIA Nsight Compute [7] for a specific dataset.

3.1 Findings



(a) DS1 Insert (b) DS1 Search (c) DS1 Delete

(d) DS2 Insert (e) DS2 Search (f) DS2 Delete

(g) DS3 Insert (h) DS3 Search (i) DS3 Delete

Figure 3.1: Throughput vs Data Sizes

3.2 BGHT

- **Consistent High Performance:** BGHT achieves high throughput across all operations (insert, search, delete) and scales well with dataset size.
- **Scalability:** It remains the only implementation with available results at 5e8, showing reliable performance at large scales.
- **Balanced Throughput:** Performance across operations is balanced, indicating a robust and optimized design.

3.3 SlabHash

- **Insert Bottleneck:** Insert throughput is lower than other implementations and degrades with larger datasets.
- **Good Search Performance:** Maintains decent search throughput, making it suitable for read-heavy workloads.
- **Performance Degradation:** Throughput decreases as dataset size increases, especially for insert operations.

3.4 WarpCore

- **High Insert Throughput:** Achieves top insert throughput at smaller dataset sizes (e.g., 5e7).
- **Sharp Decline in Delete:** Delete throughput drops sharply with larger dataset sizes, revealing a potential weakness.
- **Competitive Search Performance:** Search throughput remains competitive, though not the highest.

3.5 DyCuckoo

- **Stable Throughput:** Shows consistent performance across dataset sizes for all three operations.
- **Efficient Deletes:** Delete throughput is strong, often outperforming SlabHash and WarpCore.
- **Balanced Design:** No significant drop or spike in throughput across operations, indicating a well-rounded approach.

3.6 Nsight Profiling

All four hash table implementations—BGHT, DyCuckoo, SlabHash, and WarpCore—were tested using NVIDIA Nsight Compute. The goal was to understand runtime behaviour across implementations, identify possible performance bottlenecks, and analyse kernel-level execution patterns.

3.6.1 Kernel Execution Overview

- **BGHT:** With almost half of the total kernel runtime, the kernel `build_table_kernel` dominated execution time. Because of its bucketized design, which necessitates atomic operations or contention-heavy updates, this implies a high computational and memory access load during table construction.
- **DyCuckoo:** The `cuckoo_insert` kernel took up more than 60% of the runtime, making it the main focus of execution time. This could be due to insertions which require multiple displacements and some might even require resizing up the table.
- **SlabHash:** The distribution of kernel time between insertion and search operations was more uniform. Here also, build table took most of the time almost 46% of the time.
- **WarpCore:** According to profiling, the kernel footprint was balanced but comparatively costly, with a considerable amount of time devoted to each of the main functions.

3.6.2 Memory and Launch Behavior

All implementations exhibited consistent temporal patterns for memory operations like `cudaMemcpy` and `cudaMalloc`. Due to repeated allocations and host-device transfers, these contributed to variance in total execution time even though they were not the main causes of total runtime.

Kernel launch configurations differed significantly:

- **BGHT and SlabHash** employed larger grid sizes with modest thread blocks, possibly to ensure high parallelism across buckets or slabs.
- **DyCuckoo** used more compact launches but with denser threads per block, which may favor cache locality but also increase shared memory pressure.
- **WarpCore** utilized persistent kernels, leading to fewer kernel launches but with extended duration per launch. This approach trades off launch overhead against warp idling, depending on key distribution and operation mix.

3.6.3 Observations and Insights

- The concentration of execution time in a single kernel (e.g., `build_table_kernel` in BGHT or `cuckoo_insert` in DyCuckoo) may indicate hotspots due to data contention or complex access logic.
- WarpCore's reliance on persistent kernel execution seems to amortize launch costs but may underperform if active warps are not uniformly engaged across the dataset.
- SlabHash's longer average kernel durations may stem from pointer chasing in slab chains, while maintaining reasonable memory coalescing.
- DyCuckoo's multiple reinsertions and displacements during insertion contributes to higher control flow divergence and cache misses, observable in longer insert kernel durations relative to search or delete.

Chapter 4 Design and Implementation Details

In this chapter, we provide a high level overview of our approach and explain the major implementation changes in the original code via code snippets.

4.1 Design

An insert in DyCuckoo involves the interplay of multiple hash tables. It involves moving a displaced key (due to an insert) to the next hash table until it finds a suitable spot and no more displacement is needed. This process is sequential and leads to slow insertions. Our approach is based on parallelising this particular process (similar to WarpCore). Hence, instead of placing the displaced key directly into the next table, it is kept in a temporary slot. Only on the next insert operation, along with the insert in progress, the displaced key (from the extra slot) is put in a suitable spot in parallel. This leads to faster inserts ($O(1)$) as the cost of displacement is deferred by distributing the work across future insertions while at the same time not incurring much overhead on access.
The idea of using overflow or buffer slots for displaced items has been explored in some concurrent cuckoo hashing schemes [8], but usually as overflow buffers per bucket rather than a single temp slot per table. Our approach is more space-efficient and distributes the cost of displacement.

Figure 4.1, broadly illustrates this approach and how the insertion process in our implementation is different from traditional DyCuckoo.
Our approach, however slightly increases the lookup cost (by checking k extra slots for k tables).

4.2 Implementation

Step 1 : Data Layout Changes

We modified the data structure in `data_layout.cuh` so that each hash table has an extra slot (the "temp" slot) to temporarily store a displaced item.

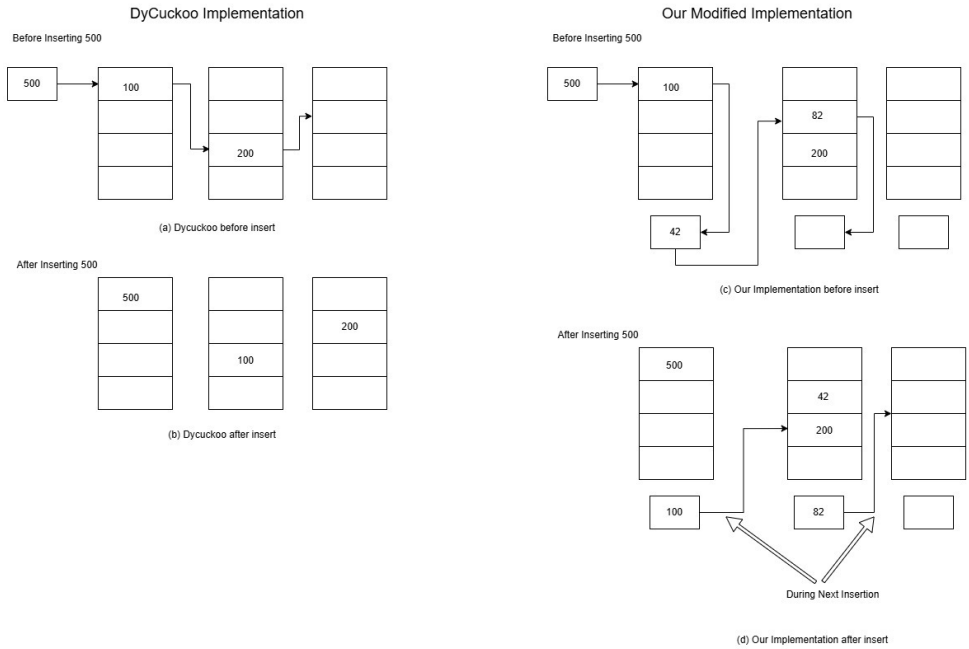


Figure 4.1: Traditional vs. Enhanced DyCuckoo

The modifications are as follows :

```
class cuckoo_t{
    public:
        key_bucket_t* key_table_group[table_num];
        value_bucket_t* value_table_group[table_num];
        // one bucket one lock
        uint32_t *bucket_lock[table_num];
        // count bucket num in single table
        uint32_t table_size[table_num];

        key_t* temp_key_slot[table_num];   // 1 key slot per table
        value_t* temp_value_slot[table_num]; // 1 value slot per table
        uint32_t* temp_slot_occupied[table_num];// 1 flag per table
        ...
```

Listing 4.1: Modification of the Data Layout

The `temp_slot_occupied` is an array of flags that indicate if the temp slot is occupied.

Step 2 : Modifying the Insertion Logic

When an insertion causes a displacement, the displaced item is placed in the temp slot of the next table instead of immediately displacing it further. On each new insert, before inserting, an attempt is made to flush any occupied temp slots into their respective tables.

```
void cuckoo_insert(key_t *keys, value_t *values, uint32_t data_num){
    ...
    for(; tid < data_num ; tid += step){
```

```
        flush_temp_slots(group); // flush temp slots before each insert
        // Get the correct table
        ...
        cg_insert_with_lock(key, values[tid], pre_table_no, true, group
    );
    }
}

void cg_insert_with_lock(key_t &key, value_t value, uint32_t
    pre_table_no, bool active, thread_block_tile<cg_size> group){
    ...
    while(group.any(active == true)){
        // Case of no evicions
        ...
        }else{
            //evict
                key_t displaced_key = (key_insert_bucket->bucket_data)[
    leader];
                value_t displaced_value = (value_bucket->bucket_data)[
    leader];
                //Logic to compute table index
                ...
                // Store in temp slot:
                *(cuckoo_table.temp_key_slot[next_table_no]) =
    displaced_key;
                *(cuckoo_table.temp_value_slot[next_table_no]) =
    displaced_value;
                *(cuckoo_table.temp_slot_occupied[next_table_no]) = 1;

                // Stop further displacement for this insert
                active = false;

        }

}
```

Listing 4.2: Modifying the Insertion Logic

Step 3 : Update the Search and Deletion Logic

Since we may have displaced data in temp slots, both `cuckoo_search()` and `cuckoo_delete()` to check temp slots as well. These modifications are shown below.

```
void cuckoo_search(key_t* keys, value_t* values, uint32_t size){
    ...
    //flag indicates if a templ slot is occupied
    if (!flag) {
```

```
        // Check temp slots
        for (uint32_t t = 0; t < DataLayout<>::table_num; ++t) {
            if (*(cuckoo_table.temp_slot_occupied[t]) &&
                *(cuckoo_table.temp_key_slot[t]) == key) {
                values[group_index_in_all] = *(cuckoo_table.
    temp_value_slot[t]);
                flag = true;
                break;
            }
        }
    }
    ...
}
```

Listing 4.3: Modifying the Search Logic

```
void cuckoo_delete(key_t* keys, value_t* values, uint32_t size){
    ...
        if (!flag) {
            // Check and clear temp slots (if needed)
            for (uint32_t t = 0; t < DataLayout<>::table_num; ++t) {
                if (*(cuckoo_table.temp_slot_occupied[t]) &&
                    *(cuckoo_table.temp_key_slot[t]) == key) {
                    *(cuckoo_table.temp_slot_occupied[t]) = 0;
                    break;
                }
            }
        }
    ...
}
```

Listing 4.4: Modifying the Deletion Logic

4.3 Improvement Over Original DyCuckoo Hashing

Our deferred displacement with temp slots implementation offers several key advantages over the original DyCuckoo hashing approach:

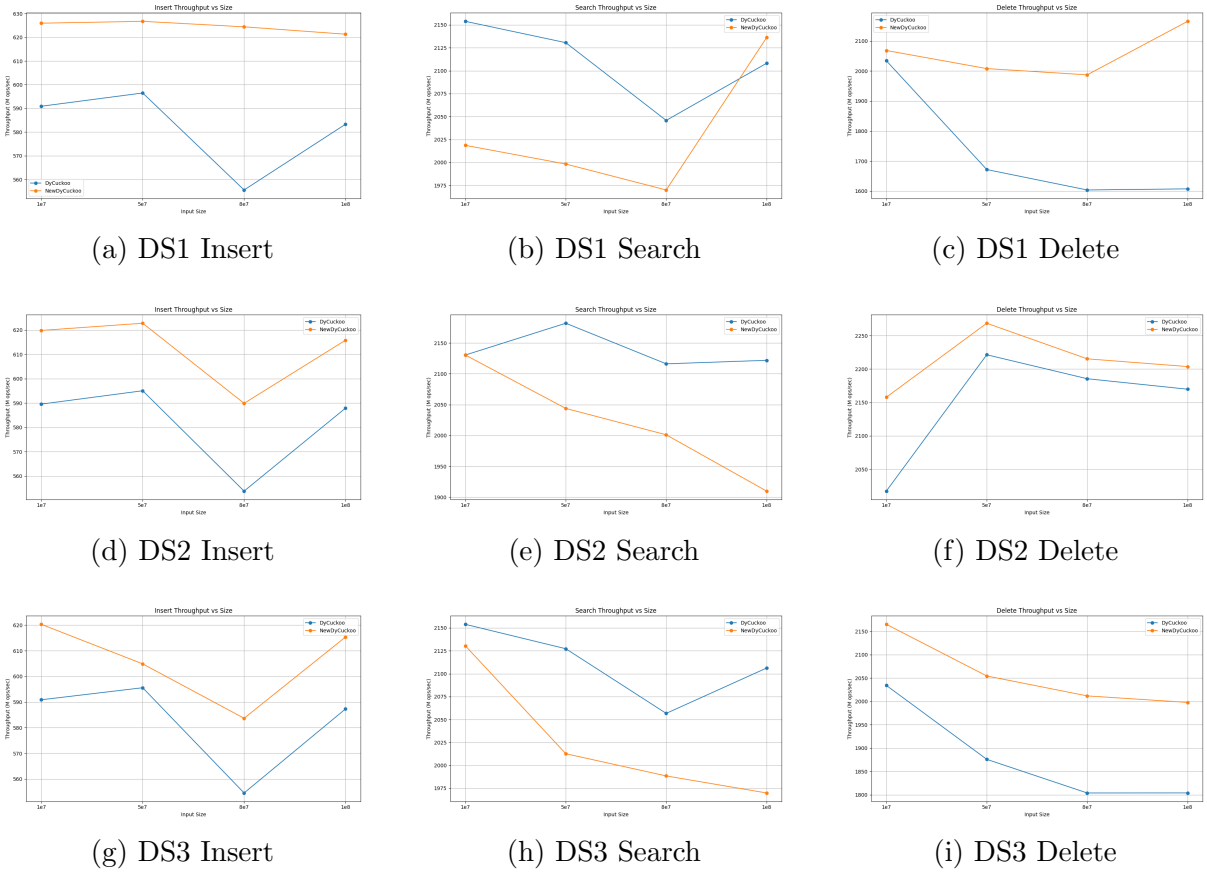
- **Faster Insertions (O(1) in Most Cases):** Unlike the original DyCuckoo, which suffers from long, recursive displacement chains at high load, our implementation defers the placement of displaced items using a temp slot, making insertions almost always a simple O(1) operation and improving throughput.
- **Reduced Lock Contention and Improved Concurrency:** By limiting each insertion to a single table and handling displaced items incrementally, our method reduces lock contention, enhancing scalability and multi-threaded performance.

- **Lower Probability of Endless Loops and Insert Failures:** The temp slot buffers displaced items, reducing the likelihood of cycles or endless loops and making the table more robust, especially under high load.
- **More Predictable Latency:** Since recursive displacement chains are avoided, insertion latency is more predictable and consistent, which is crucial for real-time applications.
- **Minimal Impact on Lookup Performance:** Lookups now check both main and temp slots, introducing a small constant increase, but overall lookup performance remains O(1), competitive with the original.
- **Better Performance at High Load Factors:** The temp slot buffer helps mitigate insertion pressure, allowing the table to function efficiently at higher load factors before rehashing is needed.
- **Simplicity and Flexibility:** Our approach simplifies the logic for deferring displaced items, making it easier to implement and tune (e.g., adding multiple temp slots or per-bucket stashes).
- **Inspired by Proven Techniques:** Similar approaches, such as stash-based cuckoo hashing, have demonstrated improvements in robustness and performance by adding small auxiliary structures to buffer problematic insertions.

In Summary: Our implementation overcomes key challenges in classic cuckoo hashing, such as slow insertions and poor concurrency, while maintaining O(1) lookups and high space efficiency. It is particularly effective for workloads with high insertion rates, high load factors, or multi-threaded access.

Chapter 5 Results

5.1 Basic Profiling



(a) DS1 Insert (b) DS1 Search (c) DS1 Delete

(d) DS2 Insert (e) DS2 Search (f) DS2 Delete

(g) DS3 Insert (h) DS3 Search (i) DS3 Delete

Figure 5.1: Throughput vs Data Sizes

5.1.1 Insert Throughput

The new DyCuckoo implementation shows a marked improvement in insert throughput across all data sizes and datasets. This suggests that the updated design introduces enhancements targeted specifically at optimizing the insertion path, potentially through better memory coalescing, reduced contention, or more efficient load balancing across

threads. These improvements consistently outperform the older implementation, indicating a successful trade-off strategy focused on maximizing insert performance.

5.1.2 Search Throughput

The improved insert performance comes with a noticeable reduction in search throughput. In nearly all dataset configurations, the new DyCuckoo performs worse in search compared to the old version. This indicates that the architectural changes favor insert-heavy workloads at the expense of search efficiency. The reduced throughput may be due to more complex probe sequences, additional metadata checks, or changes in memory layout that hinder fast lookups.

5.1.3 Delete Throughput

Similar to the search operation, delete throughput is generally lower in the new DyCuckoo implementation. While the degradation is not as severe as in search, the performance drop is consistent across different data sizes. This suggests that the new design, while optimized for fast inserts, introduces overheads that impact the efficiency of deletion operations. These may stem from increased bookkeeping or synchronization requirements introduced by the new data structure layout.

Overall, the new DyCuckoo implementation reflects a design choice that prioritizes insert throughput, making it suitable for insert-intensive applications while accepting moderate penalties in search and delete performance.

5.2 Nsight Profiling

Kernel Execution Time Distribution:

- **Original DyCuckoo:**
  - Dominated by `DyCuckooHash::insert()` (62.5%), followed by `search()` (17.6%) and `delete()` (17.5%).
- **Updated Implementation:**
  - `cuckoo_insert()` still dominates (59.9%), but now `delete()` takes up more time (21.6%) than `search()` (17.9%).

This change suggests modified deletion mechanisms possibly incurring higher complexity.

Memory Usage and Transfer Behavior:

- **Original:**
  - Memcpy (HtoD) contributes ˜45% of total time.
  - Malloc is also significant (42%).
- **Updated:**
  - Memcpy still dominates (47.1%) but now malloc time is reduced to 37.3%.
  - HtoD transfer is higher in absolute time (257M ns vs. 254M ns), possibly reflecting increased data moved per insert.

Resize Overhead:

- Resize kernel in the updated implementation contributes only 0.6% of time with very short duration ($\sim$ 446K ns), indicating infrequent resizing.

Synchronization and Events:

- Both implementations rely on a few calls to `cudaDeviceSynchronize`, but the updated version shows slightly more skewed median-to-max time gap.
- Event recording, creation, and synchronization remain negligible in both cases.

Insights:

- The shift in kernel time distribution suggests a rebalancing of operations—possibly due to better optimized search but more complex deletion logic.
- Increased memcpy and large max time spikes (up to 175M ns) might indicate that insert operations now involve bulk writes or prefetch-heavy paths.
- Persistent launch parameters with significant differences in time suggest either loop unrolling, warp scheduling differences, or memory contention being introduced in the updated variant.
- Resize latency is low in both, but particularly in the new version, which might reflect an improvement in amortized cost strategy.

5.3 Comparative Analysis using Kaggle Dataset

We also tried working on a Kaggle Dataset that included Amazon Review on books. We hashed the Referece ID and the review-text into 32 bit numbers to work on them for our implementation. We used python script to extract the data in binary format containing only hash values. Our implementation was then compared with DyCuckoo implementation over that real-world dataset. It showed outstanding performance over insert kernel while maintaining the search and delete throughput comparable to original implementation. For this dataset, we achieved ∼45x speed up for the insert kernel. The search and delete kernels give a releatively lower performance but not significantly.

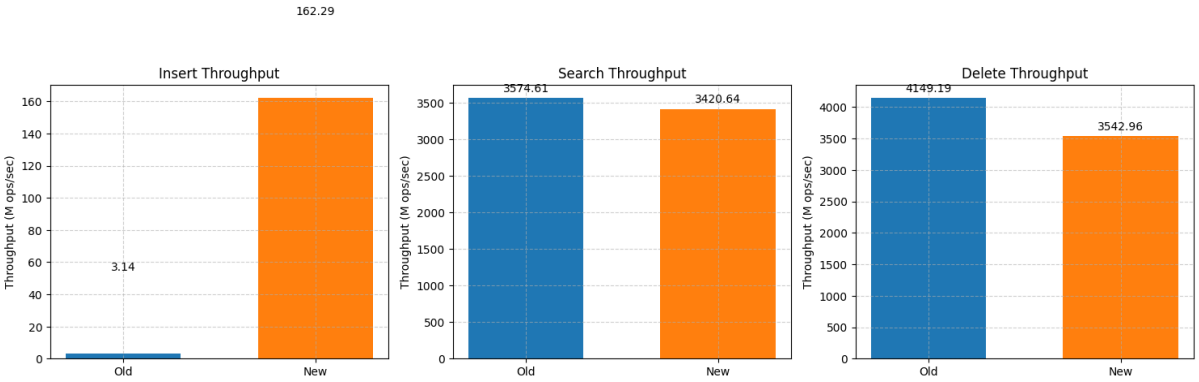


Figure 5.2: Old vs New Dycuckoo implementation

Chapter 6 Conclusion and Future Work

We have successfully profiled four recent implementations of GPU Hash Tables. We used various profiling techniques to assess their performance and made reasonable conclusions regarding their performance and also made estimates about their bottlenecks. Based on these estimates, we implemented a DyCuckoo-WarpCore Hybrid which is subjected to the same rigorous profiling. This hybrid implementation has a significant improvement over the insert kernel of traditional DyCuckoo. We also profiled this implementation with a Kaggle dataset.

In the future work, we would like to revisit the search and delete kernels of our implementation and improve their throughput. We would also like to enhance our existing implementation by varying parameters such as hash functions, bucket size, load factor and probing scheme.

Bibliography

[1] D. A. Alcantara, V. Volkov, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Chapter 4 - building an efficient hash table on the gpu," in *GPU Computing Gems Jade Edition* (W. mei W. Hwu, ed.), Applications of GPU Computing Series, pp. 39–53, Boston: Morgan Kaufmann, 2012.

[2] M. A. Awad, S. Ashkiani, S. D. Porumbescu, M. Farach-Colton, and J. D. Owens, "Better gpu hash tables," 2022.

[3] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," 2018.

[4] Y. Li, Q. Zhu, Z. Lyu, Z. Huang, and J. Sun, "Dycuckoo: Dynamic hash tables on gpus," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 744–755, 2021.

[5] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "Warpcore: A library for fast hash tables on gpus," *CoRR*, vol. abs/2009.07914, 2020.

[6] M. Awad, S. Ashkiani, S. Porumbescu, M. Farach-Colton, and J. Owens, *Analyzing and Implementing GPU Hash Tables*, pp. 33–50. 01 2023.

[7] NVIDIA Corporation, "NVIDIA Nsight Compute," 2025. Accessed: 2025-04-23.

[8] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: compact and concurrent memcache with dumber caching and smarter hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, (USA), p. 371–384, USENIX Association, 2013.