Comp 480 Final Project: GPU Hash Tables

Alex Artrip, Max Yu

Contents

1 Overview **1**

1.1 Terminology and Notation 2

2 Literature Survey **2**

3 Hypothesis **3**

4 Experimental setting **3**

5 Note on Implementation and learnings **4**

6 Experimental result and plots **4**

6.1 Unsolved Mysteries 6

7 Justification and Conclusion **6**

8 References **7**

1 Overview

We are exploring the implementation and benefits of different hash table schemes on GPUs. Specifically we are integrating and adjusting 2 different hash table variants intended to run on NVidia GPUs into a project and thoroughly benchmarking the speed at which they can handle insertions and lookups. The first is a straightforward linear probing approach, the second is a Cuckoo Hashing implementation used in the CUDPP library. Taking inspiration from both of these approaches we create standard cuckoo hash tables using 2 and 4 hash functions. Then based on the Cuckoo Hashing implementation, we build a third modified Cuckoo Hashing scheme that is inspired by linear probing such that on an attempt to hash x with 2 murmur

hash functions h_1 and h_2, if $h_1(x)$ is occupied we then attempt to add it to slot $h_1(x) + 1$, then $h_2(x)$, then $h_2(x) + 1$. Since the memory handling and processing on GPUs is much different than that of regular CPUs we are thoroughly benchmarking all of these implementations to better our understanding of how hash tables work in this environment.

1.1 Terminology and Notation

In this paper, we will often refer to

- our modified Cuckoo Hashing scheme as **C2h1p**, which stands for Cuckoo 2 Hash Functions and 1 Probing.
- **C4** will refer to the standard CUDPP hash table using 4 hash functions.
- R will refer to the capacity/size of our hash table.
- α refers to the load factor.

2 Literature Survey

Dating its roots back to dictionaries, Hash Tables are one of the most widely used data structures, and are well developed in traditional settings. However, hash tables on GPUs are different as we are in an environment with a massive amount of threads to do work, and a unique memory space layout designed to support mass parallelism. Several GPU hash tables have been proposed, many working with GPU optimized variations of traditional hashing schemes like Cuckoo Hashing in CUDPP, Chaining in SlabHash, and Linear Probing in WarpCore. SlabHash has a notable advantage in its ability to dynamically resize the hash table as more entries are inserted, while the other methods in CUDPP Cuckoo Hashing and Linear probing work over a fixed size hash table. SlabHash is a variant of chaining that is optimized for the exact amount of threads available in a single warp in a GPU, so not much can be changed regarding its performance on our end, and as such we chose not to work with it directly. However, it gives insight into a different way to utilize the massive parallelism of GPUs.

From reading the literature, a common theme in these GPU hash tables is unsurprisingly the attempt to parallelize hash table operations using a multitude of locks and thread safe instructions. But what to parallelize is also a point of contention. Where in some hash tables like **Nosferalatu's SimpleGPUHashTable**, batch operations are parallelized and each thread works with a different element, in SlabHash each element level hash table operation is actually parallelized, and a Warp of threads work on the same element together.

Since we are working with linear probing and Cuckoo Hashing, the SlabHash approach of multiple worker threads for a single element won't obviously buy us anything (It is based on having a preallocated array in memory for chaining), so we focus instead on the performance increases that come with doing batch inserts and lookups.

3 Hypothesis

We hypothesize that our modified Cuckoo Hashing implementation will result in a speedup to insertion relative to the standard Cuckoo Hashing implementation, but lead to an increase in mean lookup time less than 2x (and lookup time for Cuckoo is already extremely low so such a factor can be acceptable) relative to the standard Cuckoo Hashing implementation with 2 hashes and has less lookup time than standard Cuckoo with 4 hash functions.

4 Experimental setting

All of these algorithms were created to be run on Nvidia GPUs, and as such they are implemented in C++ and CUDA. The results we record will be from running these tests on a Nvidia GTX 1060, but these tests can be run from any Nvidia GPU. Insertion time for all 4 variations should depend on the α where insertion time is correlated positively with α since extra time must be taken to resolve collisions in the tables. Although lookup cost for both variants of Cuckoo Hashing should be constant across varying α, we still test lookups for all combinations of R and α to confirm this as well as to give a comparison between the lookup costs of the two different Cuckoo hash tables. On the other hand, with Linear Probing we expect to see an increase in lookup time for a single key as the α increases. We are also interested in evaluating the scalability of the hash table implementations as they relate to the GPUs memory management so we will vary the number of slots in each table which we call R.

Since all 4 of the implementations are utilizing Murmurhash which is a "good" hash function, there is little benefit to testing sets of data with varying levels of similarity, so all tests are run with randomly generated sequences of 32 bits for both keys and values. For each implementation we test a series of sizes for R ($R = 2^{10}, 2^{14}, 2^{28}, 2^{22}, 2^{26}$). The Load Factor of the hash tables affects performance related to insertions and lookups for all 4 implementations as mentioned earlier, so for all values of R, we run the following tests varying load factor between .1 and .9 by increments of .1 ($\alpha = .1, .2, .3, .4, .5, .6, .7, .8, .9$). To thoroughly test the 4 implementations of hash tables we test the speed of insertions, and lookups for all 4 implementations of hash tables for all combinations of (R, α). To test insertion time for any (R, α) combination we generated

$ceil(R \times \alpha)$ key value pairs, and for each implementation we start a timer, insert all key value pairs into the hash table then stop the timer and report time elapsed. Similarly, to test lookup time, using the same key value pairs, for each hash table implementation we start a timer, query every key from the key value pairs we generated earlier, and after all keys have been queried, stop the timer and report the rate of time elapsed. All reported data is stored in text files that can later be processed by a python program into plots. Specifically this python program creates the following 4 plots seen in section 6.

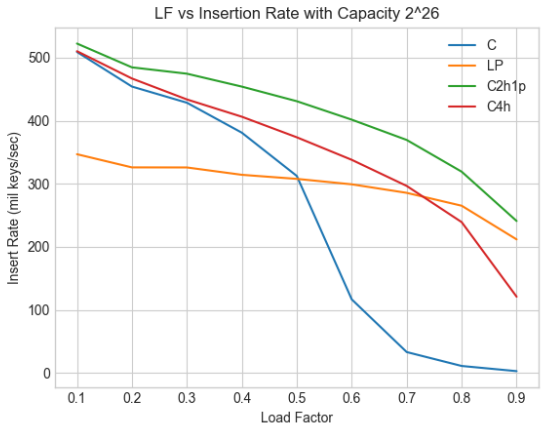
5 Note on Implementation and learnings

For all versions of Cuckoo hashing we create a stash immediately following the memory of the main hashtable, which stores a small constant number of keys (101 was the number CUDPP chose) that failed to be inserted after exceeding the max iterations defined for the Cuckoo hashing eviction chains. There is a separate hash function for this stash, and its existence allows a small tolerance of failures without needing to rehash the hash table. This was present in the CUDPP Cuckoo hash table, and we chose to include it in all the Cuckoo variants we ran.

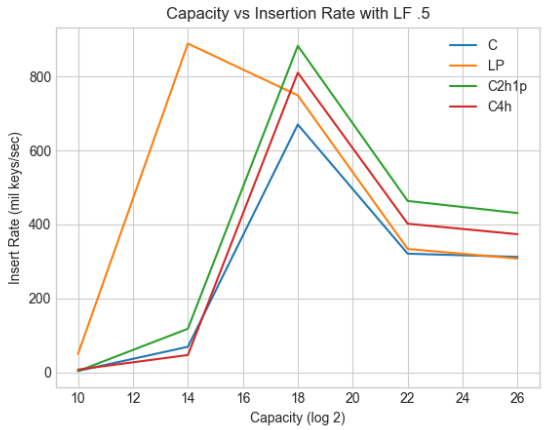
As we were building and testing Cuckoo hashing with 2 hash functions, we realized that for $\alpha > 0.5$, we were incurring exponential growth of our failure count. (The situation where a hash chain cannot complete within a fixed number of iterations). This means that regardless of the insertion and lookup time we report, in reality such a scheme is not very useful if the load factor is expected to exceed this due to the fact that some keys will never be found since they couldn't fit in the limited stash memory. This is one of the problems that our modified scheme helped improve since the additional locations next to the 2 hash functions result in less repeating chains that lead to failure in Cuckoo schemes.

6 Experimental result and plots

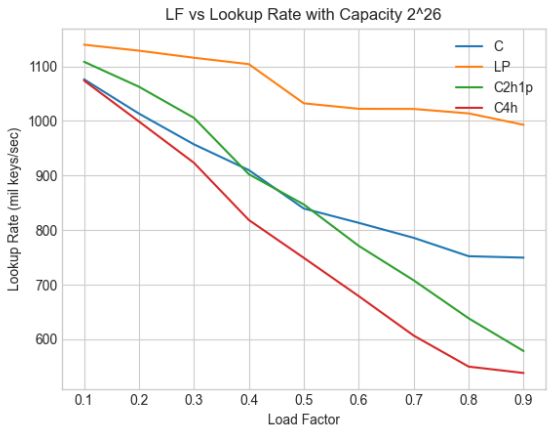
Contrary to our initial expectations, lookup times for all types of Cuckoo hashing were not constant and did increase linearly with load factor. We believe this is due to how we always start the chain of insertion by taking the slot hashed to by our first hash function. Within lookup we check this same slot hashed to by this hash function first, so for most cases given a low load factor we return immediately because we find it in the first place we look. This means that for larger load factor, when many of the keys have been displaced from their first hashes, we need to also check the slots hashed to by the other hash function which adds time to the lookup for this key (It also means we should see an increase in lookup time for more hash functions).Furthermore, for a Cuckoo scheme if all of the normal hash locations don't contain the object



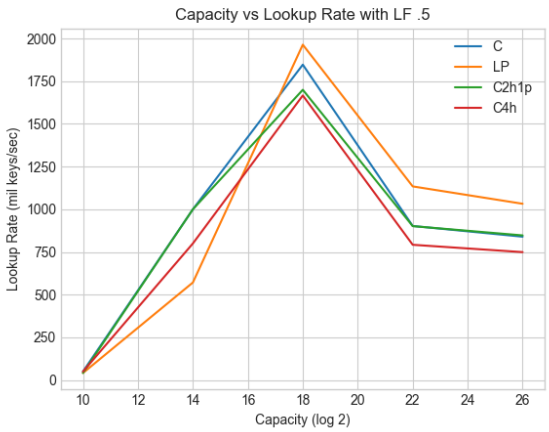
(a) Load Factor on Insertion Rate



(b) Capacity on Insertion Rate



(c) Load Factor on Lookup Rate



(d) Capacity on Lookup Rate

we then have to check the stash hash function which also adds to lookup time. Altogether this makes hash lookup time worse for high load factor lookup queries and queries that are not present altogether. The effects of this can be seen in α vs lookup rate for $R = 2^{26}$. As expected the scheme with 4 hash functions was the slowest, and the modified **C2h1p** was faster, and the standard with 2 Cuckoo was faster than that. The linear probing was the fastest on lookup even for high α which is entirely unexpected.

For all hashtable variations we see a negative correlation between load factor and insertion rate (illustrated by the load factor vs insertion time for fixed capacity graph) as we expected. Exponential decay of insertion rate for the Cuckoo scheme with 2 hash functions can be attributed to the eviction loops that are formed leading to failures to insert (detecting these loops always requires hitting max iterations which is very time intensive when such a probability is non trivial). This is observed with Cuckoo for 2 hash functions once it reaches $\alpha > 0.5$ as mentioned earlier.

There is a strong negative correlation between α and Insertion Rate as expected (Observe in LF vs Insertion Rate with Capacity 2^{26} plot). For all values of α, **C2h1p** perform the best and trends similarly to that of **C4** which is also not surprising since **C2h1p** is essentially an emulation of 4 hash functions that takes less time to compute the results of 2 of these functions. One important thing to note is that failure count for **C4** and **C2h1p** are both 0 even for high load factors (this is from the raw data since a graph from this data of this would just be a single line for Cuckoo with 2 hashes which is hardly informative).

However, linear probing does outperform **C4** for high load factor. We attribute this to the growth of the expected eviction chain time on insertion which leads to many hash function computations whereas linear probing only needs to add one to its current hash_index to check its next slot (This is also why insertion is faster for **C2h1p**, for similar length eviction chains we compute the hashes for each eviction in less time than **C4** would).

6.1 Unsolved Mysteries

It is much of a mystery why for both insertion and lookup, there is a peak in rate around capacity $R = 2^{18}$ consistently for all 4 hash tables. One explanation is that at lower capacities, there are many unused thread resources in the GPU, so we see the slope upwards from 2^{10} to 2^{18}. Then we reach a bottle neck in terms of working threads, and and spatial locality also kicks in; the bigger the hash table, the more often we would need to jump from one end to the other when accessing it. These are however fairly uneducated guesses.

There is another weird phenomenon, which is that Linear Probing outperformed all 3 Cuckoo hashtables on lookup rates, especially on high load factors. We have no good explanation, other than we may have not been as efficient with our Cuckoo hashing implementations. (We copied the exact scheme of **CUDPP's simple hash table**, but the actual implementations differ.)

7 Justification and Conclusion

To highlight, the most significant result we have is comparing C2h1p and C4, we showed a possibility of improving all aspects of Cuckoo Hashing schemes by substituting some number of hash functions for probing slots.

We were also correct in our hypothesis that the **C2h1p** scheme would result in a speed up to the insertion time relative to both standard Cuckoo hash tables. It also resulted in a lookup rate greater than that of **C4** but less than that **C2** (Standard with 2 hash functions) which is also consistent with our hypothesis. These results show that **C2h1p** is a viable hash table algorithm on GPUs and is advantageous in some situations relative to the standard Cuckoo **C4** approach and linear probing. In general, linear probing and Cuckoo

hashing (assuming you use 4 hash functions like **C4** and **C2h1p**) are both effective hashing schemes on GPUs depending on what a program on the GPU is trying to achieve. If speed is of utmost priority, the programmer should find the table size for their specific GPU that yields the best lookup and insertion rates since the size of the table has such a significant impact on these rates (this assuming they have a manageable load factor, for the Nvidia GTX 1060 this corresponds to the $R = 2^{18}$ peak in the varying capacity graphs).

8 References

Cudpp: `https://escholarship.org/uc/item/445536d6`

`https://github.com/cudpp/cudpp/tree/master/src/cudpp_hash`

WarpCore: `https://arxiv.org/pdf/2009.07914.pdf`

SlabHash: `https://par.nsf.gov/servlets/purl/10062407`

`https://github.com/owensgroup/SlabHash`

StateSpace: `https://arxiv.org/pdf/1712.09494.pdf`