CuckooHashing using CUDA

Runze Yuan

July 4, 2020

1 Introduction

This lab aims to use cuda to accelerate the operations(insert, delete and lookup) in cuckoo hashing.

Both serial and parallel version are implemented.

2 Design detail

Note: A complete cuckoo hashing should implement **rehash**, that is, when there are too many evictions, we should use different hash function sets and re-insert all the keys. In this lab, I implemented rehash first but found that the functions set I regenerate still has too many Hash Collisions and it never stops.

So the rehash part is implemented but not used. That means, if we meet the situation that the table should be rebuilt, this hash table shouldn't be used. To finish the lab and observe the speedup that cuda brings, I increase the eviction bound to give a penalty when too many evictions happen.

Again, some keys would be missing when the table should be rehashed.

2.1 Hash Function

When I start this lab, to allow easily rehash, universal hashing is used in this lab. That is

$$f(key) = ((a * key + b) \bmod prime) \bmod table\_size$$

Where

- prime is a large prime number that is larger than table_size. I set it to 50331653
- table_size is the size of the table
- a and b are two random numbers that is less than prime.

2.2 CUDA algorithm

2.2.1 Data structure

In insert part, we should use the function **atomicExch** in cuda and this function can only be applied on basic type like int. But in cuckoo hashing, every node in the table should store two information:

- key
- index of hash function the node currently used.

So, we compressed the data into a uint32_t number, whose 30 bits in the left store the key and the 2 bits in the right store the index of hash function.

These data can be easily extracted using binary operation like << and |, etc.

2.2.2 Insert

The parallelism in insertion is naive:

- Let each thread insert a key into the table
- When one thread has to evict a key that is already in the table, use atomicExch to ensure that there won't be interference.
- The other part is similar to the serial version.

2.2.3 Delete & Lookup

Deletion and lookup part is relative easy to implement. I just let each thread to iterate to use different hash function to determine that if the key in the table. For deletion use atomicExch(key, 0) to delete the key.

3 Experiment Environment

OS	Ubuntu 18.04
CPU	4 Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
Cuda Version	CUDA10.2
GPU	GeForce GTX 1050Ti 4GB

Table 1: testing environment

4 Evaluation

In each experiment, I did it 5 times, I use the average value to represent time that serial and cuda version takes.

4.1 Experiment 1

In the first experiment, I create hash table of size 2^{25} and insert $2^{10}, ..., 2^{24}$ keys into it. Below is the average time that each test takes.

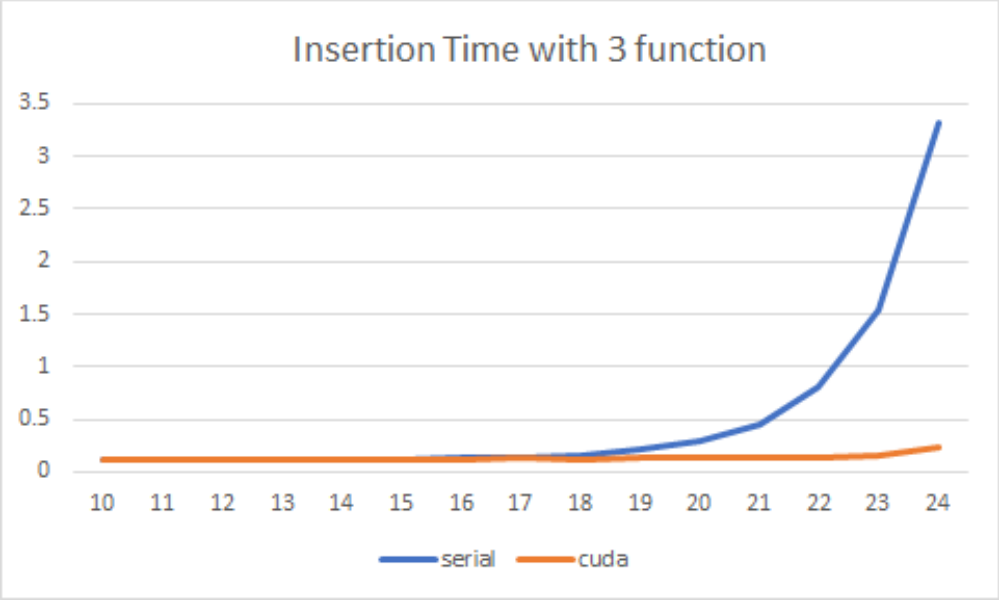

Figure 1: Insertion time with 3 hash functions

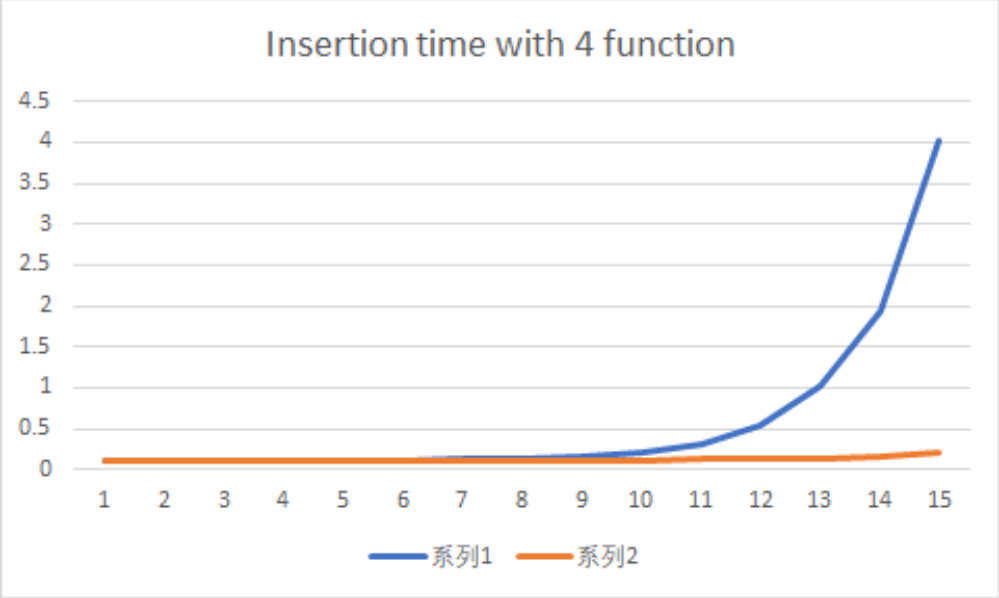

Figure 2: Insertion time with 4 hash functions

As we can see, since we the size of dataset we insert increases exponentially, the time takes by serial version increases exponentially. But the time takes by cuda version, take constant time as the size grows.

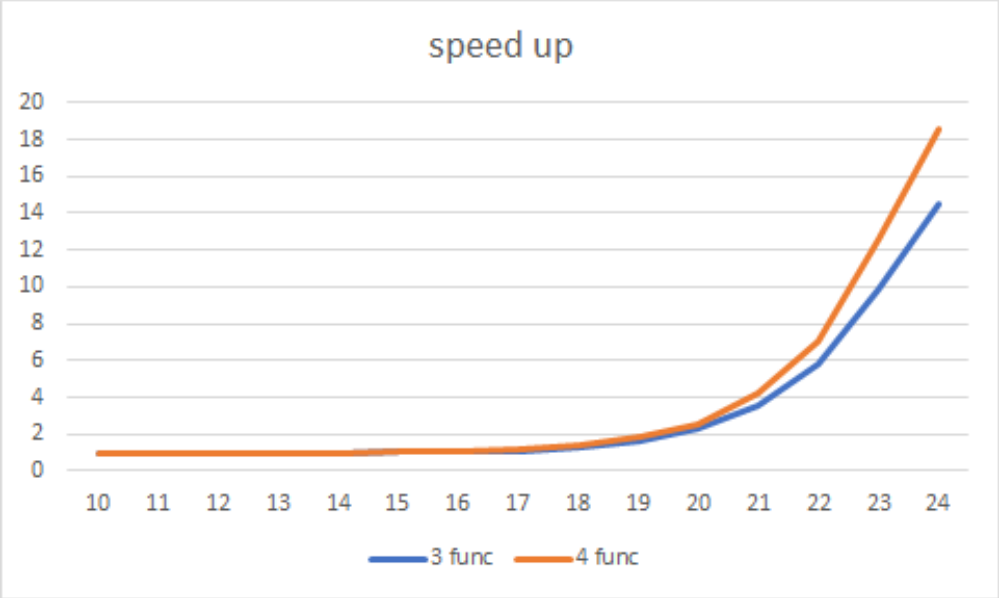

Figure 3: Speedup using 3 hash functions vs Speedup using 4 hash functions

This is reasonable because we let each thread insert a value, even with some atomic operation, the time it takes remain constant.

As we can see, the speed up using 4 hash functions can be larger than using 3 functions. This is because that if we use 4 functions, we will meet less hash collisions that 3 hash functions. Which will reduce the number of atomic operations and achieve better speedup.

4.2 Experiment 2

Num of functions	Serial	CUDA	SpeedUp
3 functions	16.03957	0.125095	128.2194
4 functions	19.1971	0.128934	148.8918

Table 2: Running time(s) of the two algorithm on different mazes

In the experiment 2, we perform looking up on the 10 datasets. The time each takes is as above.

We can see the speedup is significant. The time cost using 4 functions is larger that using 3 functions, this is reasonable because we have to check at most 4 times for each key instead of 3 times.

But for each thread in gpu, there is no big difference between checking 3 or 4 times. So we can achieve better performance on optimizing hash table using 4 hash functions.

4.3 Experiment 3

In this experiment, we test the insertion time and speedup using hash table with different size

Since my rehashing part does not work, I give a penalty to represent the time of rehashing time. This naive assumption is that each rehashing take the same time. The figure above

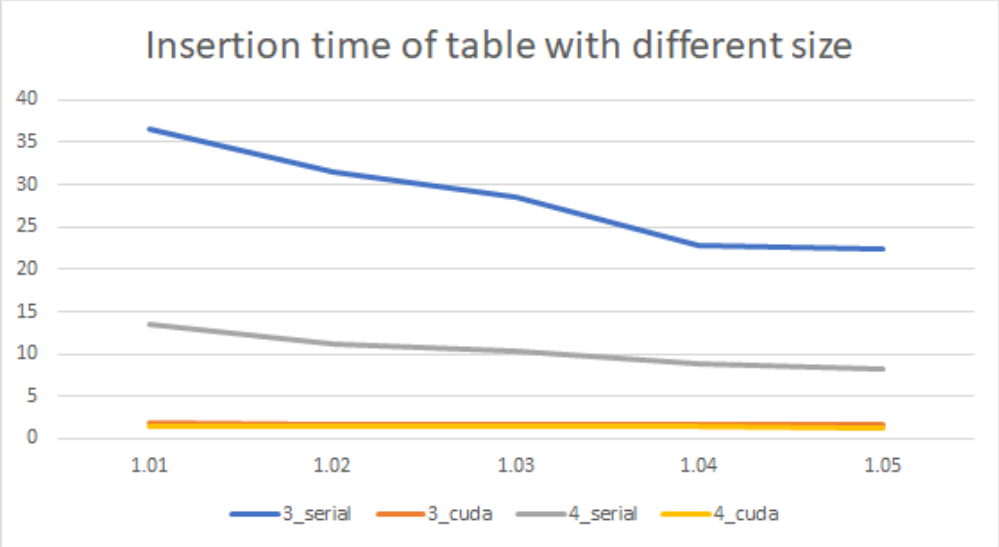

Figure 4: Insertion time using 3 hash functions vs Speedup using 4 hash functions

shows that, when the size of hash table is $1.01, 1.02...$ times of the size of dataset, which is very close to size of dataset, there is a big difference between using 3 and 4 functions in serial version. But there is no difference in CUDA version.

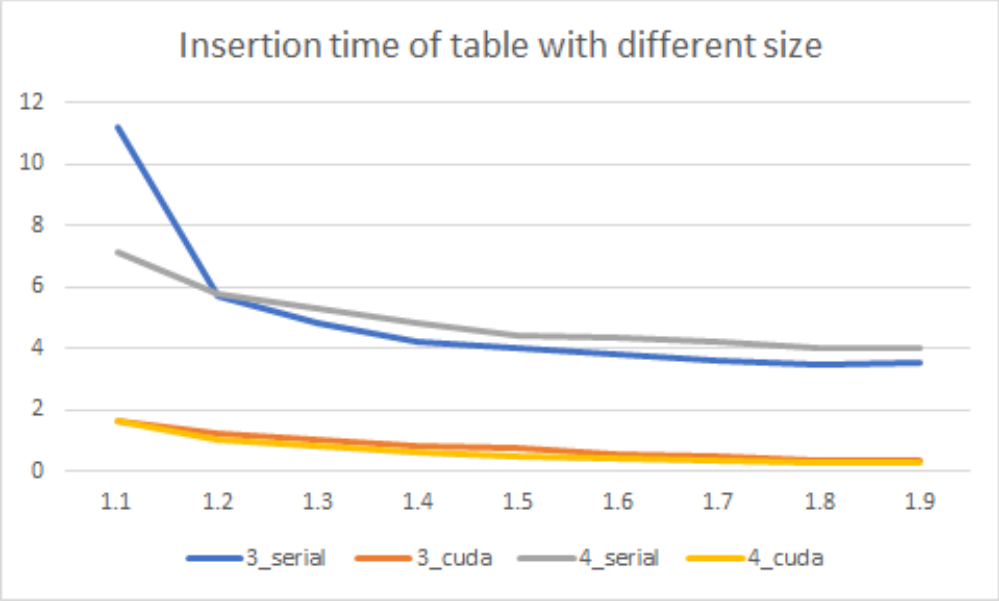

Figure 5: Insertion time using 3 hash functions vs Speedup using 4 hash functions

The figure above shows that, when the size of hash table is $1.1, 1.2...$ times of the size of dataset, the difference between using 3 and 4 functions is not that large in serial version.

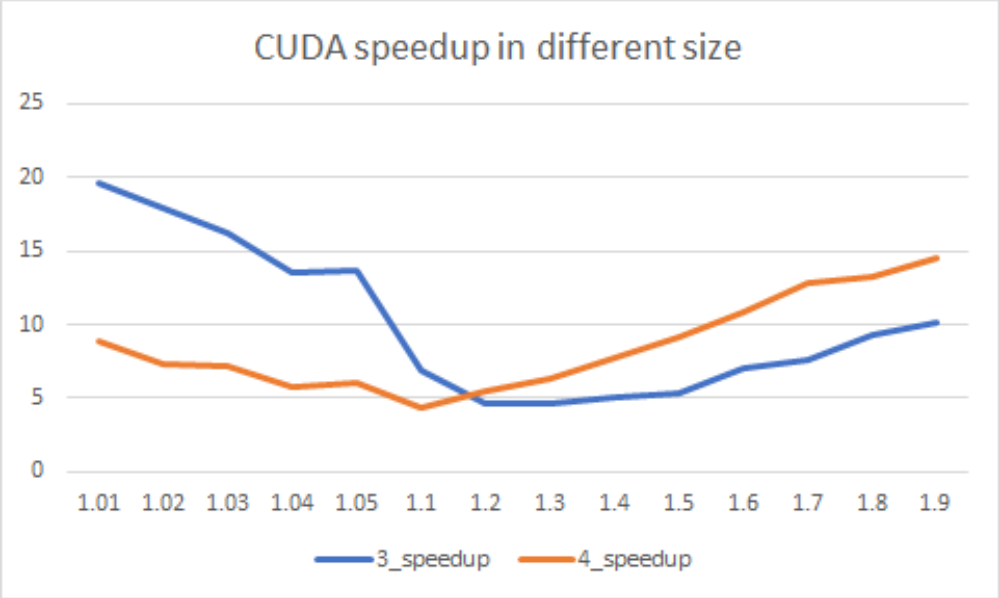

Figure 6: Speedup using 3 hash functions vs Speedup using 4 hash functions

The figure shows that the speedup of func3 and func4 in terms of the size of hash table. As we can see, when their size are extremely similar, the speedup decrease as the table size increase. But when the difference increase, the speedup start to increase.

4.4 Experiment 4

In this Experiment we try to find the best eviction bound. The eviction bound we use is

$$bound = n * log_2(tablesize)$$

The figure above shows the speedup as n increases. It shows that when $n = 3$ it gets the

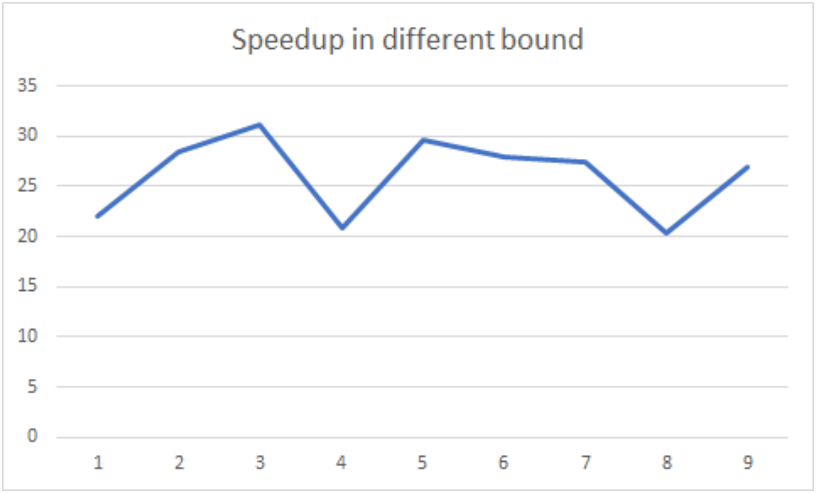

Figure 7: Speedup using 3 hash functions vs Speedup using 4 hash functions

best speedup.

5 Discussion

The number of hash function used some times matters. When using many functions, the probability of hash collision will decrease. But in that case, the time of looking up will increase. So there is a tradeoff between them.

With cuda optimization, we can make lookup or insertion in even constant time, which significantly improve the performance. So, we don't need to worry about the problem above.

References

[1] *Cuckoo Hashing*, available at `https://www.wikiwand.com/en/Cuckoo_hashing`..