Parallel Computing Lab2: Cuckoo Hashing on CUDA

RYAN ZHANG, Shanghaitech University, Shanghai, China

In this lab, we are required to implement a fast cuckoo hashing algorithm on GPU with CUDA and test its performance.

1 INTRODUCTION

The basic idea of cuckoo hashing is that, we have one set of multiple hash functions, forming several sub tables. Each key can be inserted to several fixed location, which ensures efficiency during lookup.[1]

2 ALGORITHM

2.1 Hash function set

We need to choose proper hash functions to minimize collision numbers. The hash function should map a 32 bit unsigned integer to a number between 0 and *size of hash table* - 1. We need two or three different hash function. Here we use hash functions from *Hash Function Prospector*[2]. This repository provides a series of low biased integer hash function. Here we choose the one listed below, three sets of parameters for the function are listed in Table 1. In cuckoo hashing, we need to rebuild hash table and choose a new set of hash functions if the eviction chain is too long. Here the solution is we rotate the 32-bit number with a shift value called *seed* before hashing it. Now we only need to change *seed* each round, we can reuse these low biased hash functions. *seed* is set to the round number here, starting from 0.

```
uint32_t hash_function(int32_t n, uint32_t seed) {
    uint32_t x = static_cast<unit32_t>(n);
    x = (x << seed) | (x >> (32-seed);
    x ^= x >> a; x *= b;
    x ^= x >> c; x *= d;
    x ^= x >> e; x *= f;
    x ^= x >> g;
    return x % table_size;
}
```

Table 1. Parameters in hash function

Func Id	a	b	c	d	e	f	g
0	17	ed5ad4bb	11	ac4c1b51	15	31848bab	14
1	16	aeccedab	14	ac613e37	16	19c89935	17
2	16	236f7153	12	33cd8663	15	3e06b66b	16

2.2 Insertion

For insertion, we just insert the keys one by on in serial algorithm and count the length of the eviction chain. For parallel algorithm, we have multiple threads modifying the hash table at the same time.

2.3 Look up

The look up algorithm in parallel is quite straight forward, we just assign each keys to a thread, and loop through all hash functions to check all possible locations of the key. If we found the key, return true, else we return false. A bool array with the same length as key array is used to store the result.

3 IMPLEMENTATION ON GPU[3]

3.1 Key generation

As required in this lab, we first need to generate an array of keys for the hash table. These keys should be 32-bit integer numbers and should be as random as possible. In our hash table, the only data structure we need to preserve is a large array of 32-bit integer, where value 0 represents that this entry is empty. The merit of this method is don't need to maintain another array to represent the state of each entry in hash table. But now we need to make sure that there is no 0 value in the keys we generate. Also the keys used for insertion do not contain any repeat value. We simply use a bool array with length 2^{32} to remove repeat value. The random engine we used on host is *mt19937* from standard library.

We generate keys for insertion and look up operation on host and store them in vectors. They are copied to device during tests.

3.2 Block and threads

To do cuckoo hashing given an array of keys, we need to launch a kernel function. We need to give proper gridDim and blockDim such that we can get optimal performance.

In CUDA, threads are lighted weighted and can be used to hide the I/O latency. We need to make sure each block contains enough thread number. Here we assign 1024 threads to each block. The block number depends on the number of keys in each operation. We launch $\frac{\text{key num}}{\text{thread num per block}}$ thread blocks to ensure that each key is assigned to a thread. In our benchmark, we have typically 2^{24} keys, which leads to about 214, which is too much for a GPU. But extra blocks will run in serial automatically, so we do not need to bother that.

3.3 Synchronizations

All threads activated will modify the array representing the hash table concurrently, so it is very important to do synchronization. Here we just use built-in atomic operations of CUDA. For each CUDA thread, it loop through length of max eviction chain and first calculate the location the key should be hashed to. Then it do a atomic exchange and set the item on that location of corresponding table to the newly inserted one. If the key be exchanged out is 0, it means that location is empty before, then the insertion is done. We terminate the for loop. Otherwise, we rehash the victim key with next hash function, keep using atomic exchange. Each time we using the next hash function, mod by number of hash function. If any thread failed to hash all victim keys after *MAX EVICT NUMBER* rounds, we set the input pointer indicating the result to be true, showing that we need to rebuild the hash table.

Table 2. Test.1 Results. (t is the number of hash functions)

key number	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}
time/ms(t=2)	0.057	0.058	0.051	0.066	0.059	0.087	0.115	0.167	0.269	0.482	0.915	1.816	3.722	7.681	17.369
MOPS(t=2)	17.9	35.6	80.4	123.6	277.4	375.6	570.7	783.3	976.0	1086.8	1145.7	1154.8	1126.8	1092.1	965.9
time/ms(t=3)	0.064	0.070	0.072	0.072	0.065	0.098	0.109	0.175	0.272	0.494	0.927	2.140	3.842	7.669	17.112
MOPS(t=3)	16.0	29.2	56.6	113.5	250.5	335.4	599.6	749.1	963.3	1061.9	1131.5	980.2	1091.6	1093.8	980.4

Table 3. Test.2 Results. (t is the number of hash functions)

percentage of new keys	100%	90%	80%	70%	60%	50%	40%	30%	20%	10%	0%
time/ms (t=2)	5.961	6.446	7.099	7.403	7.765	8.485	8.666	8.903	9.382	10.025	9.865
MPOS (t=2)	2814.6	2602.9	2363.5	2266.4	2160.7	1977.3	1936.0	1884.4	1788.2	1673.6	1700.7
time/ms (t=3)	6.086	7.656	8.332	9.196	10.051	10.779	11.667	12.926	13.577	14.628	16.603
MPOS (t=3)	2756.9	2191.4	2013.7	1824.3	1669.1	1556.4	1438.0	1297.9	1235.7	1146.9	1010.5

3.4 Verifiaction

We use *unordered set* from STL to verify the result by inserting some keys and them looking them up. All tests below have passed the verifiaction.

4 BENCHMARK AND RESULT

Now we demonstrate our benchmark result and analysis. Test platform are as in Table 4

Table 4. Parameters in hash function

CPU	Intel i7-10700
RAM	DDR4@2933MHz 32G
GPU	NVIDIA RTX2080Ti
OS	Microsoft Windows10 21H1

4.1 Test.1 key insertion test on large table

In this test, we create a hash table with fixed size of 2^{25} entries. We insert 2^i keys into the hash table and check its performance, where $i = \{10, 11, 12 \ldots, 24\}$

The results are shown in Table 2

As we can see from the table, when the size of hash table large enough comparing to the number of inserted keys, the scalability is quite good. In the beginning, as the number of keys increases, we start to use more and more cuda threads to insert them so the performance goes up quickly. After we reached a critical point, we no longer have more threads in GPU to handle more keys simultaneously, and more keys start to collide with others, causing more overhead during atomic exchange. So we will start to have as slight performance drop at last.

4.2 Test.2 key look up test

In this test, we create a hash table with fixed size of 2^{25} entries, then insert 2^{24} keys into the hash table. Then we create 11 sets of keys with size 2^{24} keys and look them up in the table. For these the i^{th} set of look up keys, $10i\%$ of keys are from keys inserted in hash table ($i = \{0, 1, \ldots, 10\}$). We test the performance of look up operation.

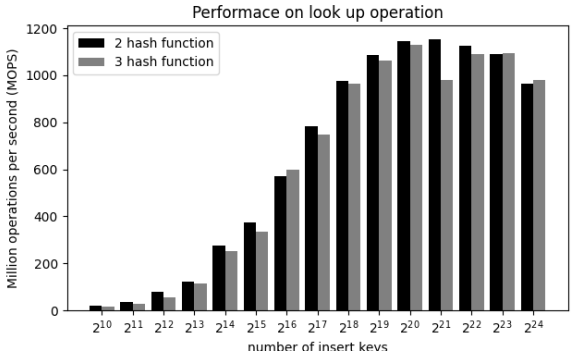

Fig. 1. Result of test.1

The results are shown in Table 3

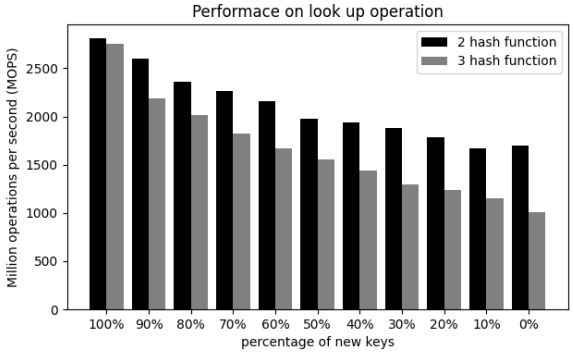

Fig. 2. Result of test.2

Table 5. Test.3 Results. (t is the number of hash functions)

size of table	1.01n	1.02n	1.05n	1.1n	1.2n	1.3n	1.4n	1.5n	1.6n	1.7n	1.8n	1.9n	2.0n
time/ms(t=2)	failed	failed	failed	failed	failed	failed	failed	17.976	17.964	17.672	17.685	17.235	17.395
MPOS(t=2)	–	–	–	–	–	–	–	933.3	933.9	949.3	948.6	973.4	964.5
time/ms(t=3)	19.395	19.226	19.097	18.856	18.490	18.169	17.889	17.656	17.451	17.258	17.091	16.957	16.814
MPOS(t=3)	865.0	872.6	878.5	889.8	907.4	923.4	937.8	950.2	961.4	972.2	981.7	989.4	997.8

Table 6. Test.4 Results. (t is the number of hash functions)

max eviction num	10	15	20	25	30	35	40	45	50	55	60	65	70	75
time/ms(t=2)	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	fail	**18.225**	18.703	18.815
MPOS(t=2)	–	–	–	–	–	–	–	–	–	–	–	**920.6**	897.1	891.7
time/ms(t=3)	fail	fail	fail	**17.557**	17.881	17.928	17.881	17.876	17.945	17.942	17.890	17.866	17.922	17.869
MPOS(t=3)	–	–	–	**955.6**	938.3	935.8	938.2	938.5	934.9	935.1	937.8	939.0	936.1	938.9

The number of total keys we need to look up is fixed at 2^{24}, which is sufficient to fully utilize all cuda threads. The more new keys we got to look up, the more likely it is for us to try all possible locations to determine whether the key is in our hash table. And we need to check more 50% more locations for when we got 3 hash function than when we have 2 hash functions, so the performance is $\frac{2}{3}$ at worst case.

4.3 Test.3 key insertion test on tight table size

In this test, we always want to insert 2^{24} keys into the hash table. Let the number of keys be n, we adjust the size of hash table be $1.01n, 1.02n, 1.05n$ and $1.1n, 1.2n, 1.3n, \ldots, 1.9n, 2.0n$. We test the performance on insertion operation.

The results are shown in Table 5

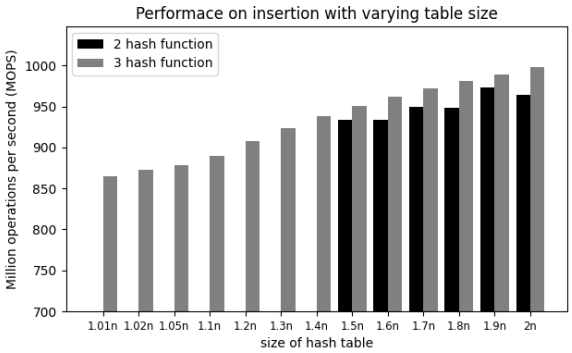

Fig. 3. Result of test.3

When size of hash table is not large enough, we will have many collision during insertion, which leads to repetitive rebuilding of hash table and insertion failure. Increase the number of hash function can efficiently solve the problem. As we can see from the figure, If we use 2 hash functions, we need hash table of size at least 1.5 times of key number to guarantee 5 consecutive insertion. While we only need hash table of size 1.01n to achieve the same goal if we use 3 hash functions.

Also, we notice that, the larger our hash table is, the better performance we get because we will have fewer collision.

4.4 Test.4 eviction chain bound test

In this test, we always insert 2^24 keys into a table of size 1.4×2^{24}. We will try different bound of eviction chain length and see the performance.

The results are shown in Table 6

As we can see from the figure, the best performance occurs when the length of eviction chain is just enough for all keys to be inserted. Since we are basically use the same hash function, just rotate the input with different shift in different rounds, reconstruct the hash table will not help the insertion to much. That is to say, if we can not insert all keys successfully in the first round, then we are not very likely to insert them no with more reconstruction of hash table.

In our test conditions, the best eviction number is 65 and 25 for 2 and 3 hash functions respectively. 3 hash function table can handle tighter bound since it has higher tolerance on collision.

REFERENCES

[1] D. Gries. 2021. *CuckooHashing*. https://www.cs.cornell.edu/courses/JavaAndDS/files/hashing_cuckoo.pdf
[2] skeeto. 2018. *Hash Prospector*. https://github.com/skeeto/hash-prospector
[3] Zhang.C. 2020. *CuckooHashing*. https://github.com/chibinz/CuckooHashing