CS121@Fall2021 Lab 2
Cuckoo Hashing on GPU using CUDA

Cheng Peng (彭程)* 2020533068

December 30, 2021

Contents

*pengcheng2@shanghaitech.edu.cn

1 Introduction

Being one of the most widely used data structures, hash table serves as a fundamental building block of advanced data structures and algorithms. Since hash table was invented, extensive studies that aim at improving the theoritical complexity and the real-world performance has been done. Collision resolution is the main issue to tackle when designing a hash table. Separate chaining and open addressing are the two main strategy to deal with hash collisions. So far, a perfectly balanced hash table does not exists i.e. people have to make space-time and/or update-lookup tradeoff.

Cuckoo hashing is a space-efficient open addressing hash table first proposed in [PR01] by Rasmus Pagh and Flemming Friche Rodler. It guarantees a worst-case constant lookup time and a expected constant time for insertion.

Cuckoo hashing is a suitable choice for implementing concurrent hash table. In this lab, we implemented a cuckoo hash table on GPU with nVidia CUDA. We did several benchmarks and made empirical performance analysis. Finally, we figured out the bottle neck of our algorithm and proposed a few ways to overcome them.

2 The Cuckoo Hashing

In this section, we describe the orignal cuckoo hashing method. 1 is the algorithm pseudo code.

Cuckoo hashing uses two sub-tables T_1, T_2 and two independent hash functions h_1, h_2. A key k is stored in either $T_1[h_1(k) \bmod |T_1|]$ or $T_2[h_2(k) \bmod |T_2|]$. To insert k, we first try to put k at $T_1[h_1(k) \bmod |T_1|]$. If that location is empty, then we succeed. Otherwise, we evict the key k' that previously occupying this location and try to insert k' into T_2 using the same procedure.

We stop on successful insertion or when a infinity loop is detected. In the latter case, we have to do a *rehash*: The table capacity is enlarged[1], h_1, h_2 are replaced with new hash functions, all the keys are re-inserted.

```
Function CuckooInsert(k)
    // The orignal cuckoo hashing does not support duplicate keys
    if CuckooLookup(k) then
        return failed
    end
    for run = 0 to MaxLoop do
        if is-empty(T1[h1(k)]) then
            T1[h1(x)] ← k
            return succeeded
        end
        swap(T1[h1(x)], x)
        if is-empty(T2[h2(k)]) then
            T2[h2(x)] ← k
            return succeeded
        end
        swap(T2[h2(k)], k)
    end
    // the eviction chain is too long, presumed infinity eviction cycle
    CuckooRehash()
    return CuckooInsert(k)
end
Function CuckooLookup(k)
    if T1[h1(k)] = x ∨ T2[h2(k)] = x then
        return found
    else
        return does not exists
    end
end
```

Algorithm 1: original cuckoo hashing method

[1] typically, grow with a constant factor

3 Algorithm Design

3.1 Parallelized cuckoo hashing

We demonstrate our parallelized version cuckoo hashing with the following pseudo code 3 2.

We take a straight-forward approach to parallize the cuckoo hashing eviction. Atomic operations are employed to prevent concurrent modification on same location.

It turns out that this approach is pretty good. When the load factor λ[2] is low for example $\lambda < 0.5$, collisions are infrequent. Even for large load factor, collisions are spread out over the whole table if we choose a good hash function.

```
Kernel LookupKernel(keys, result)
    Data: keys: keys to find.  result: whether a key exists
    forall k ∈ keys parallel do
        result[k] ← T
        for t = 0 to Subtables do
            if T_t[h_t(k)] = k then
                result[k] ← T
            end
        end
    end
end
Kernel InsertKernel(keys, failed)
    Data: keys: keys to insert  failed: indicator of failed insertion
    forall k ∈ keys parallel do
        for j = 0 to EvictionLimit do
            t ← j mod Subtables
            atomicExch( k, T_t[h_t(t)])
            if is-empty(k) then
                break
            end
        end
        // the eviction chain is too long, insertion for k is failed
        if not is-empty(k) then
            atomicCAS(failed,F,T)
        end
    end
end
```

Algorithm 2: GPU kernel functions

[2] $\lambda = \frac{n}{N}$,where n is the number of inserted keys and N is the hash table capacity.

```
Function GpuCuckooLookup(keys)
    Input: a list of keys to lookup in cuckoo hashing table
    Output: a list of 0/1 indicating whether the key can be found
    result ← GpuArray
    LaunchKernel(LookupKernel(keys,result))
    return result
end
Function GpuCuckooInsert(keys)
    Input: a list of unique keys to insert
    failed ← F
    // rehash-insert, until the all the keys are successfully inserted
    while T do
        failed ← F
        LaunchKernel(InsertKernel(k, failed))
        if failed = T then
            GpuCuckooRehash()
        else
            break
        end
    end
end
Function GpuCuckooRehash()
    // save the previously inserted keys
    T'_0, T'_1 ... ← T_0, T_1 ...
    // clear the table; find new hash functions
    clear(T_0, T_1 ...)
    generate-hash-function(h_0, h_1 ...)
    // re-insert the keys
    GpuCuckooInsert(T'_0 ∪ T'_1 ...)
end
```

Algorithm 3: parallizing cuckoo hashing on GPU

3.2 Choice of hash function

To achieve good performance, we have to choose a set of hash function with care. Hash functions of high quality are generally slow to compute, which may limit the throughput. However, using fast hash functions may result in more collisions, which in turns lower our throughput. Finding A hash function that achieves best balance between speed and quality is the key to high performance.

We found a research paper [JO20] on *Journal of Computer Graphics*. A set of cryptographic and non-cryptographic hash functions for their quality and speed in the context of GPU rendering. They concluded that for 1D to 1D hash function, which is the scenario in lab2, the *xxhash32* falls in the middle of the spectrum from the fastest algorithms to the algorithms that yield the best results.

A simple reference implementation [Bru] in cpp was found. We reproduced a CUDA version of it.

The *xxhash32* hash function

The *xxhash32* takes a two input a 4-byte seed and a byte stream, then produces a 32-bit integer as output. Or formally speaking:

$$H : \text{byte}^4 \times \text{byte}^* \to \text{byte}^4$$

Therefore, we can creat a set of hash functions by seeding *xxhash32* with different seeds.

$$H_i : \text{byte}^* \to \text{byte}^4 \qquad H_i(\text{text}) = H(\text{seed}_i, \text{text})$$

Where seed_i are a set of unique random number. They can be generated using any pseudo RNG.

3.3 Key details in implementation

- In *xxhash32* calculation, we only need to evaluate the hash of a 4-byte integer. The loop is manually unrolled for better performance.
- In insertion and lookup kernel, each thread need to access all the seeds for hash function. GPU shared memory is used as a cache to reduce the memory access latency.
- *curand* from the CUDA toolkit is used to generate random numbers rapidly on GPU.
- We launch blocks containing the maximal possible block size to get high occupancy.
- CUDA calls may fail silently, we have to manually check if they failed.
- CUDA kernel calls are asynchronized. `cudaDeviceSynchronize`

4 Performance Evaluation

4.1 Generating unique random array

The original cuckoo hashing and our implementation do not support inserting duplicated keys. So we have to generate array of unique random numbers in order to test the performance. C++ STL and CUDA curand library do not have such feature. We have to implement one.

Suppose that we can generate a uniform random numbers in $[0, L-1]$ in constant time, we want to generate m ($m \lll L$) unique random array such that every number in $[0, L-1]$ appears in the array with equal probability m/L.

One way to generate such samples is to generate the numbers one by one, use a hash table to detect and reject duplicated number. However this naive algorithm is in-efficient and takes too much time. We developed the following sampling algorithm to generate test data rapidly.

We divides $[0, L-1]$ into $B = \sqrt{L}$ blocks, we first generate $k = m/B$ unique random numbers $b_1 \dots b_k$ in $[0, B-1]$. Then the unique sample we generate is $b_1, b_1 + B, b_1 + 2B \dots b_1B(B-1); b_2, b_2 + B, b_2 + 2B \dots b_2B(B-1); \dots b_k, b_k + B, b_k + 2B \dots b_kB(B-1);$. Generating k unique random numbers in $[0, B-1]$ can be done by randomly shuffle the permutation $0, 1, 2 \dots B-1$ can take the first k ones. Therefore, we can generate such unique sample in $O(\sqrt{L} + m)$ time.

4.2 Benchmark setup

Testing environment

We tested our program on a cloud server equipped with the following main hardware.

CPU Intel(R) Xeon(R) CPU E5-2678 v3 @ 2.50GHz

GPU RTX 2080 Ti. Turing SM architecture, 11 GB VRAM.

RAM 62GB. unknown configuration.

The CUDA version is 11.1

Test cases

1. Create a hash table of size 2^{25} in GPU global memory, where each table entry stores a 32-bit integer. Insert a set of 2^s random integer keys into the hash table, for $s = 10, 11 \dots 24$

2. Insert a set S of 2^{24} random keys into a hash table of size 2^{25}, then perform lookups for the following sets of keys $S_0, S_1 \dots S_{10}$. Each set S_i should contain 2^{24} keys, where $(100 - 10i)$ percent of the keys are randomly chosen from S and the remainder are random 32-bit keys.

3. Fix a set of $n = 2^{24}$ random keys, and measure the time to insert the keys into hash tables of sizes $1.1n, 1.2n, \dots 2n$. Also, measure the insertion times for hash tables of sizes $1.01n$, $1.02n$ and $1.05n$. Terminate the experiment if it takes too long.

4. Using $n = 2^{24}$ random keys and a hash table of size $1.4n$, experiment with different bounds on the maximum length of an eviction chain before restarting.

We run each test case at least five times to get consistent performance.

4.3 Benchmark result

insertion test

insertion size	time(ms)[3]	performance(MOPS)[4]
2^{10}	41.6	24.6153
2^{11}	31.8	64.4025
2^{12}	35.4	115.7062
2^{13}	37.8	216.7195
2^{14}	38.2	428.9005
2^{15}	51.2	640.0
2^{16}	82.6	793.4140
2^{17}	136.4	960.9384
2^{18}	1364.0	192.1876
2^{19}	944.4	555.1545
2^{20}	1423.6	736.5664
2^{21}	2406.2	871.5617
2^{22}	4403.0	952.6014
2^{23}	8979.0	934.2474
2^{24}	19222.6	872.7859

When inserting small set of keys, the cost of launching GPU kernels and synchronizing the host and device limit the perform. We can observe that the throughput is relatively low compared to inserting large set of keys.

When $n \geq 2^{16}$, the throughput is steady. Since the load factor λ is less than 0.5, collisions are rare. Insertion for each key roughly can be done in constant time so the running time is propotion to the number of keys to insert.

lookup test

percentage of keys from the inserted keys	time(ms)	performance(MOPS)
0%	579.0	28976.1934
10%	577.0	29076.6308
20%	580.0	28926.2344
30%	577.8	29036.3724
40%	578.4	29006.2517
50%	717.2	23392.6603
60%	712.4	23550.2751
70%	711.0	23596.6469
80%	710.2	23623.2272
90%	711.4	23583.3792
100%	710.8	23603.2864

Cuckoo hashing guarantees worst-case constant time for lookup. The lookup can be perfectly parallelized. Thus, the throughput remain constant.

insertion with high load factor

We terminate the insertion if the keys can not fit into the table after 500 rehash.

load factor	time(ms)	performance(MOPS)
$\frac{10}{11} = 0.909$	91473.0	183.4116
$\frac{10}{12} = 0.833$	52895.4	317.1772
$\frac{10}{13} = 0.769$	32446.2	517.0779
$\frac{10}{14} = 0.714$	85283.2	196.7235
$\frac{10}{15} = 0.666$	51899.4	323.2641
$\frac{10}{16} = 0.625$	52622.8	318.8202
$\frac{10}{17} = 0.588$	31932.4	525.3979
$\frac{10}{18} = 0.555$	25061.8	669.4337
$\frac{10}{19} = 0.526$	38706.6	433.4458
$\frac{10}{20} = 0.5$	59105.0	283.8544
$1/1.01 = 0.99$	NA	NA
$1/1.02 = 0.98$	NA	NA
$1/1.05 = 0.95$	NA	NA

Cuckoo hashing is a space-efficient method, however, high load factor do hurt the performance. When the table is nearly-full, insertion always fails.
We suggest that grow the table size when rehahing is necessary in practice.

upperbound on eviction chain length

eviction chain limit	time(ms)	performance(MOPS)
$1.0 \log_2 n$	24727.8	678.4758
$1.5 \log_2 n$	25054.4	669.6315
$2.0 \log_2 n$	32479.4	516.5494
$2.5 \log_2 n$	25055.6	669.5994
$3.0 \log_2 n$	25241.4	664.6705
$3.5 \log_2 n$	25129.8	667.6223
$4.0 \log_2 n$	25703.4	652.7236
$4.5 \log_2 n$	32322.8	519.0520
$5.0 \log_2 n$	25097.4	668.4842
$5.5 \log_2 n$	25116.4	667.9785
$6.0 \log_2 n$	25053.4	669.6582
$6.5 \log_2 n$	25135.8	667.4629
$7.0 \log_2 n$	25068.8	669.2468
$7.5 \log_2 n$	25050.2	669.7437

If we using a small limit, frequent rehash operations lower the performance. Conversely, if we using a large limit, too much collision would saturate stream multiprocessors.
The cuckoo hashing have too be tuned with care to maximize the performance.

5 Futher Improvement

Busying ourselves with final projects, we haven't done much work on optimizing performance and extending the functionality. We list a few ways to improve the performance and several possible new features.

5.1 Duplicated keys

The original cuckoo hashing and our parallelized version do not support duplicated keys. In graph algorithm and GPU rendering, this is acceptable. However, duplication lies in the root of some problems. For example, in text processing tasks, some words e.g. *the, in* may frequently appear.
We found that [JKD18] tries modify cuckoo hashing for data deduplicatio. They proposed a simple solution to this issue.
When an eviction occurs, compare the key inserted and the evicted key, if they are equal, then stop the eviction chain.

5.2 Open addressing hash table with a stash

[KMW10] proposed a variant of cuckoo hashing where a small stash is used to store the keys can not be inserted because of collision. This can help to prevent rehashing when the load factor is load. However, the lookup time will increase a bit.

5.3 A two-phase insertion approach

Warp divergent and uncoalesced memory access hurt GPU parallelism, however we want the hash function to distribute the keys evenly. This is sort of paradoxical. We found that [Alc+09] proposed an alternative insertion algorithm to address this issue.

The devides the table into buckets of 512 slots. The insertion is sub divides into two phases: In the first phase, a hash function is used to distribute the keys into buckets. In the second phase, they launch a block of thread to execute the cuckoo insertion in each bucket.

This two-level hashing approach can reduce contention, reduce divergent and uncoalesced memory accesses.

5.4 Alternative structures

Besides cuckoo hashing, people have been trying to develop other hashing schemes. [Bor14] describes and compare several popular approaches and proposed a combination of multi-level, cuckoo and linear probing.

A Fun Facts

- According to https://developer.nvidia.com/blog/inside-pascal/, atomic operations are optimized for global memory not the shared memory since Kepler SM architecture.
  However, we get neither a boost nor a degrade in performance when testing our program on RTX 2080 Ti with Turing architecture SM.

reference

[PR01] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo Hashing". In: *Algorithms — ESA 2001*. Ed. by Friedhelm Meyer auf der Heide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 121–133. ISBN: 978-3-540-44676-7.

[Alc+09] Dan A Alcantara et al. "Real-time parallel hashing on the GPU". In: *ACM SIGGRAPH Asia 2009 papers*. 2009, pp. 1–9.

[KMW10] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. "More robust hashing: Cuckoo hashing with a stash". In: *SIAM Journal on Computing* 39.4 (2010), pp. 1543–1561.

[Bor14] Rajesh Bordawekar. "Evaluation of parallel hashing techniques". In: *GPU Technology Conference*. 2014.

[JKD18] Jane Rubel Angelina Jeyaraj, Sundarakantham Kambaraj, and Velmurugan Dharmarajan. "High-speed data deduplication using parallelized cuckoo hashing". In: *Turkish Journal of Electrical Engineering & Computer Sciences* 26.3 (2018), pp. 1417–1429.

[JO20] Mark Jarzynski and Marc Olano. "Hash Functions for GPU Rendering". In: *UMBC Computer Science and Electrical Engineering Department* (2020).

[Bru] Stephan Brumme. *xxHash - a hash algorithm as fast as memcpy*. URL: https://create.stephan-brumme.com/xxhash/.