Cuckoo Hashing Cuda Lab

Yiwei YANG yangyw@shanghaitech.edu.cn
2018533218

1. The introduction of cuckoo hashing

Hash table, one of the most fundamental data structure, have been implemented on Graphics Processing Units (GPUs) to accelerate a wide range of data analytics workloads. Most of the existing works focus on the static scenario and try to occupy as much GPU device memory as possible for maximizing the insertion efficiency. In many cases, the data stored in the hash table gets updated dynamically and existing approaches takes unnecessarily large memory resources, which tend to exclude data from concurrent programs to coexist on the device memory. In this paper, we design and implement a dynamic hash table on GPUs with special consideration for space efficiency. To ensure the insertion performance under high filled factor, a novel coordination strategy is proposed for resolving massive thread conflicts.

1.1 The challenge of Cuckoo Hashing in GPU

1. The aim of a universal hash function set is to distribute the input keys into random entries of a hash table uniformly. The nearly random accessing to memory actually destroy the space locality within the warp which contains the consecutive threads, which makes it hard to manipulate shared memory and cause high hit missing rate of cache.

2. Cuckoo hashing is a variant of open addressing hashing method, the addressing iteration could introduce much divergence between threads in a warp. With the increasing of load factor of a hash table, the probability of the thread divergence grows quickly either.

3. As for a cuckoo hashing table with only two hashing function, any multiple loops in a connected component could lead a failure. For any failure, the original algorithm invokes a rehashing, which could be painful with high expected cost. Besides, we don't double the size of the hash table while rehashing, which can not guarantee the rehashing to work and it will interrupt all working thread.

1.2 The Algorithm of Cuckoo Hashing in GPU

I proposed the cuckoo hashing in Zhou and Zeng (2015).

Algorithm A2 Parallel cuckoo hashing on a GPU

```
procedure Cuckoo-Hash-Insert(𝓗, T, k)
    for i ← 0 to k − 1 do
        𝒯[i] ← ∅
    end for
    for all t ∈ T do                                    ▷ dispatch nodes according to ℏ(x)
        Push-Back(𝒯[ℏ(t)], t)
    end for
    for i ← 0 to k − 1 in parallel do
        for j ← 1 to |𝒯[i]| in parallel do
            call Insert(𝓗[i], 𝒯[i][j]) on block i
        end for
    end for
end procedure
procedure Insert(H, t)
    z ← −1
    while true do
        for i ← 0 to d − 1 do
            if H[h_i(t)] is empty then
                H[h_i(t)] ← t
                z ← i
                break
            end if
        end for
        thread synchronization
        if z ≠ −1 and H[h_z(t)] = t then
            return
        end if
        Let r be a random number in {0, 1, . . . , d − 1}
        Atomic-Swap(H[h_r(t)], t)
    end while
end procedure
```

Figure 1: Parallel cuckoo hashing on a GPU

2. Environment Setup

1. NVIDIA V100, CUDA 11.0

2. The software is cross platform, tested on MSCV on windows, clang on mac and icc on linux.

3. Deploy Mersenne Twister 19937 generator to generate random integers.

To deploy the project, just

```
mkdir build
cd build
cmake ..
make
```

./CuckooHashing.

3. Experiment

3.1 Experiment 1

Cuckoo Hashing	Insertion size	Performance
	2^1	1.665 ms
	2^2	2.291 ms
	2^3	4.257 ms
	2^4	2.616 ms
	2^5	3.858 ms
	2^6	2.589 ms
	2^7	2.315 ms
	2^8	9.164 ms
	2^9	4.281 ms
	2^{10}	2.742 ms
	2^{11}	2.676 ms
	2^{12}	4.700 ms
	2^{13}	7.928 ms
	2^{14}	2.405 ms
	2^{15}	5.150 ms
	2^{16}	7.061 ms
	2^{17}	6.978 ms
	2^{18}	6.582 ms
	2^{19}	6.653 ms
	2^{20}	9.412 ms
	2^{21}	15.54 ms
	2^{22}	25.42 ms
	2^{23}	43.83 ms
	2^{24}	115.5 ms

Here the performance is quite competitive compared to many public benchmark. With the scale of insertion increasing, the inserting speed will go up (Better Occupancy) and finally go down (the high load factor of hash table introduce too much collision).

3.2 Experiment 2

Cuckoo Hashing	percentile	Performance
	S_0	25.062000 ms
	S_1	24.761000 ms
	S_2	24.801001 ms
	S_3	24.785999 ms
	S_4	24.948999 ms
	S_5	25.292000 ms
	S_6	25.968000 ms
	S_7	24.972000 ms
	S_8	24.790001 ms
	S_9	25.865000 ms
	S_{10}	24.771000 ms

Since the lookup operation exactly take $O(1)$ time, here are no obvious difference for random input or existed keys. Besides, here indicates the drawback of my design, the lookup time cost is nearly close to the insert operation, the extra cost is introduced by my auxiliary linear probing table, since it may lead to traverse all the auxiliary table in the worst case.

3.3 Experiment 3

Cuckoo Hashing	Ratios	Performance
	1.9	38.073002 ms
	1.8	23.381001 ms
	1.7	23.188999 ms
	1.6	25.606001 ms
	1.5	23.326000 ms
	1.4	21.899000 ms
	1.3	24.674999 ms
	1.2	23.295000 ms
	1.1	24.481001 ms
	1.05	24.409000 ms
	1.02	30.268000 ms
	1.01	23.343000 ms

The experiment result reveal the rules of efficiency of hashing: low load factor leads to better performance.

3.4 Experiment 4

Cuckoo Hashing	Bound	Performance
	0	39.252998 ms
	1	25.014999 ms
	2	25.364000 ms
	3	29.417999 ms
	4	24.982000 ms
	5	28.659000 ms
	6	24.945000 ms
	7	24.924000 ms
	8	24.877001 ms
	9	24.905001 ms
	10	26.068001 ms

The result of this experiment of mine would be quite different to others. Here you can see the lower bound lead to better performance, which is actually promised by the auxiliary. Most elements will be successfully hashed to proper position at the first time, it's the long tail effect, the lower bound we set, the less divergence in warp either. However, it can't be too small as the auxiliary table can't be too large.

References

Yichao Zhou and Jianyang Zeng. Massively parallel a* search on a gpu. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.