A CUDA-based Cuckoo Hash Table Design

Zhiqiang Xie
School of Information Science and Technology
ShanghaiTech University
xiezhq@shanghaitech.edu.cn

Abstract

This is the report for CS121 Parallel Computing course project assignment. In this report, we'll demonstrate our design of Cuckoo hash table in CUDA framework to utilize the usage of GPU platform.

1 The challenge of Cuckoo Hashing in GPU

1.1 Space Locality

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.

1.2 Thread Divergence

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.

1.3 Penalty of Getting Stuck

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.

2 The design we adopt

2.1 Reduce Amount of Memory Access

As we talked above, rather than improving space locality, reducing the amount of memory access is more reasonable and beneficial. Here we set one more slot for each bucket (the overall capacity of the table stay the same, like the number of buckets is the half), which will introduce a higher probability to cut out the long collision-relocation chain.

2.2 Light-weighted Thread

Since the divergence is hard to avoid, we design a light-weighted thread and small blocks for better occupancy. Like here only 64 threads are in one block, and only a few necessary control flow and operation are introduced.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

2.3 One more Auxiliary Table

Finally, we design an auxiliary table to store the final "black sheep": only a very small extra space cost (1% percent of the hash table space cost) and a few expected time cost in look up phase are introduced, but it help us avoid the disaster of interrupting and rehashing. Specifically, it's the idea of cache, one small hash table with simple addressing strategy (in this project it's just linear probing) take the middle role to prevent the painful rehashing.

2.4 Some More Trade-off

- Thread divergence is more expensive than memory access
  - In modern GPU, the cost of global memory access is lower and lower, and some advanced scheduling mechanism will reduce the idle time of computing resource. As for the divergence and memory accessing trade-off in this project, memory accessing is cheaper, which helped to accelerate the kernel for 60% from my testing.
- More explicit intrinsics rather than C/C++ style math computing.

3 Experiment

3.1 Setup

- NVIDIA Quadro M620, CUDA 8.0
- Deploy Mersenne Twister 19937 generator to generate random integers.
- Apply Round-robin scheduling in switching hash function.
- Note here only three hash function cases presented, cause my auxiliary table will capture too much keys in the situation of only two hash function while inserting, which degrades the meaning of comparison.

3.2 Experiment 1

Cuckoo Hashing	Insertion size	Performance
Insertion million/sec	2^{10}	2.17
	2^{11}	4.21
	2^{12}	4.85
	2^{13}	12.11
	2^{14}	29.23
	2^{15}	51.07
	2^{16}	79.06
	2^{17}	132.5
	2^{18}	178.3
	2^{19}	215.3
	2^{20}	239.6
	2^{21}	243.6
	2^{22}	227.4
	2^{23}	199.8
	2^{24}	129.8

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).

3.3 Experiment 2

Cuckoo Hashing	Lookup Set	Performance
Lookup million/sec	S_0	124.4
	S_1	129.6
	S_2	126.9
	S_3	129.6
	S_4	129.7
	S_5	129.6
	S_6	129.7
	S_7	129.6
	S_8	129.5
	S_9	129.4
	S_{10}	126.1

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.

3.4 Experiment 3

Cuckoo Hashing	Table Size	Performance
Lookup million/sec	$1.01n$	120.7
	$1.02n$	122.5
	$1.03n$	123.6
	$1.04n$	131.1
	$1.05n$	139.8
	$1.1n$	143.9
	$1.2n$	154.2
	$1.3n$	178.4
	$1.4n$	187.8
	$1.5n$	201.1
	$1.6n$	209.5
	$1.7n$	215.8
	$1.8n$	220.5
	$1.9n$	219.9
	$2.0n$	222.3

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

3.5 Experiment 4

Cuckoo Hashing	Bound Stop	Performance
Insertion million/sec	$\log n$	175.7
	$2\log n$	184.9
	$3\log n$	174.2
	$4\log n$	154.2
	$5\log n$	149.2
	$6\log n$	137.1
	$7\log n$	120.9
	$8\log n$	113.3

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.

3.6 Experiment 5

Here we tried various hash function such as murmur hashing, some variants of additive hashing, but all of them not work so well. Here I list some reasons:

- The keys are random numbers which are generated distinctly and relatively uniformly, which expect less to the hash function set, which suggests the advantage of advanced hash functions are not shown.
- Some hash function like murmur hashing is expensive to be deployed in the CUDA threads, which would introduce extra time cost.
- The carefully selected original hash function set (from a universal hashing family) is good enough for the 32-bit integer hashing.

References

[1] Cuckoo Hashing, Stanford CS166 course notes.

[2] Dan Anthony Feliciano Alcantara (2011) *Efficient Hash Tables on the GPU*. University of California Davis.