CS121 Parallel Computing Lab 2 Cuckoo Hashing using CUDA

Qihe Chen 2020533088

December 17, 2022

Abstract

A hash table is a data structure that is used to store keys and values, which allows for quick insertion and lookup of the values, making it a useful data structure for a variety of applications. Cuckoo hashing is a technique for implementing a hash table that uses a cuckoo search algorithm to efficiently store and retrieve key-value pairs by mapping keys to two different indices in an array and storing the values at those indices. In this lab, we implement Cuckoo Hashing by CUDA language, which is supported by several test experiments related to algorithm accuracy and performances.

1 Introduction

1.1 Compile & Build

We recommend to compile the project on Windows 10 operation system. For compiling successfully, g++ should support at least C++ 17 standard and nvcc should support at least C++ 17 also.

We recommend we to build the project by CMake with version at least 3.21.

```
mkdir build
cd ./build
cmake ..
cmake --build . --config Release
cd ./tests
./main.exe
```

It should compile successfully and automatically run the test experiments and accuracy test.

1.2 Benchmark Configures

Config	Value
CPU	Intel(R) Core(TM) i9-10900x CPU @ 3.70GHz
GPU	NVIDIA GeForce RTX 3090
Memory	64 GB DDR4 3200 MHz 2 of 8 slots
GPU Memory	24576 MB GDDR6X (Micron)
OS	Windows 10 21H2 19044.2251
Compiler	MSVC 19.33.31630.0, nvcc V11.8.89
Optimize	MSVC: /O2, nvcc: -O3
CUDA Cores	10496
Bandwidth	936.2GB/s
Driver Version	31.0.15.2206 (NVIDIA 522.06) DCH / Win 10 64

2 Implement Details

2.1 Cuckoo Hashing Algorithm with CUDA

Cuckoo hashing is a hashing algorithm that uses two hash functions to place keys into a table. The idea is to use the two hash functions to place a key into one of the two possible positions it could be hashed to, with the goal of minimizing the number of collisions (when two keys are hashed to the same position). If a collision does occur, the algorithm "kicks out" the key that was previously occupying that position and re-hashes it to its other possible position. This process continues until a valid placement for the key is found or a pre-determined number of iterations has been reached.

CUDA is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on graphics processing units (GPUs). It allows developers to use the power of the GPU to accelerate applications, such as machine learning algorithms, image and video processing, and other computationally-intensive tasks.

It is possible to implement the cuckoo hashing algorithm using CUDA to take advantage of the parallel processing capabilities of the GPU. This can potentially improve the performance of the algorithm by allowing it to process multiple keys simultaneously. However, the specific details of how to do this would depend on the specific implementation and the types of data being processed.

The specific details of how to implement the cuckoo hashing algorithm using CUDA would depend on the specific implementation and the types of data being processed. In general, however, the following steps could be used as a starting point:

1. Define the data structures and variables that will be used by the algorithm. This may include the hash table, t hash functions $h_1(k), h_2(k), ..., h_t(k)$, and any other data that is needed to store and process the keys. Furthermore, in this lab, we perform each experiment first with two hash functions (i.e. t=2), then again with three functions.

2. Write the code for the t hash functions that will be used to place the keys into the table. These functions should be able to map a given key to one of the t possible positions it could be placed in the table.

3. Write the code for the main cuckoo hashing algorithm. This should include the logic for inserting a key into the table, handling collisions, and re-hashing keys if necessary.

4. Use CUDA to parallelize the algorithm by dividing the keys into smaller groups and processing each group concurrently on the GPU. This will allow the algorithm to take advantage of the parallel processing capabilities of the GPU to speed up the computation.

5. Optimize the implementation by fine-tuning the data structures and algorithms used, as well as the way the keys are divided and processed on the GPU. This may involve experimentation and testing to determine the best approach for the specific implementation and data being processed.

Overall, implementing the cuckoo hashing algorithm using CUDA can potentially improve its performance by leveraging the parallel processing capabilities of the GPU. However, it will require a deep understanding of both the algorithm and CUDA, as well as careful optimization and testing to achieve the best results.

2.2 The reason why two hash functions is enough

Cuckoo hashing only uses two hash functions because this is the minimum number of hash functions needed to guarantee that each item is mapped to two different positions in the hash table. Using more than two hash functions would not provide any additional benefits and would only increase the complexity of the algorithm without improving its performance. In fact, using more than two hash functions could potentially decrease the performance of the algorithm because it would increase the number of possible configurations for the hash table, making it more difficult to find a good set of hash functions that result in well-distributed items in the table.

2.3 Cuckoo Hashing Algorithm: Insert a key

To insert a key into a cuckoo hashing table, we would first use the two hash functions to map the key to two different positions in the hash table. Then we would check to see if either of these positions is empty. If one of the positions is empty, we would simply insert the key into that position. If both positions are already occupied, we would "kick out" the item that is currently in the first position and insert the new key in its place. Then we would take the item that was kicked out and try to insert it into the other position using the same process. This process is repeated until an empty position is found or a predetermined number of iterations has been reached.

The time complexity of this process is O(1) on average, meaning that it takes a constant amount of time to insert a key into the table on average. However, in the worst case, the time complexity can be as high as O(n) if all of the positions in the table are already occupied and a long chain of items being kicked out and inserted into different positions is created. This worst-case scenario is unlikely to occur in practice, but it is something to be aware of when using cuckoo hashing. Thus, in order to avoid the worst case from happening, the length of the kicking chain in cuckoo hashing can be set by choosing the size of the hash table and the number of iterations to a value of 4logn, where n is the number of items to be inserted into the table. This value ensures that the expected number of iterations needed to insert a new item into the table is a constant, which means that the average time complexity of the algorithm is O(1). In other words, this value ensures that the algorithm will perform well on average, even though it may not perform as well in the worst case.

Algorithm 1 Insert

$h_1, h_2, \cdots, h_t$ are t hash functions
$T_1, T_2, \cdots, T_t$ are corresponding tables
C is the size of the table (capacity)
input: a key k
for $i = 0 \rightarrow$ iter-bound **do**
  loc $\leftarrow h_{i \bmod t}(k) \bmod C$
  if $T_{i \bmod t}$[loc] = empty **then**
    $T_{i \bmod t}$[loc] $\leftarrow k$
    return
  else
    swap$(k, T_{i \bmod t}$[loc]$)$
  end if
end for
rehash()
insert(k)

2.4 Cuckoo Hashing algorithm: Lookup a key

To look up a key in a cuckoo hashing table, we would first use the two hash functions to map the key to two different positions in the hash table. Then we would check to see if the key is present in either of these positions. If the key is found in one of the positions, we can return the corresponding value immediately. If the key is not found in either of the positions, we can conclude that the key is not present in the table.

The time complexity of this process is O(1) on average, meaning that it takes a constant amount of time to look up a key in the table on average.

2.5 Data structure fine-tuning

Cuckoo hashing uses multiple tables to store the items in the hash table, where the number of tables is equal to the number of hash functions used. Storing these tables in a single, big linear array can be inefficient because it does not make use of the independence of the tables and can result in poor performance due to suboptimal pipeline and cache utilization.

To improve the performance of cuckoo hashing, it is better to store the tables in separate linear arrays. This allows the tables to be accessed independently and makes better use of the pipeline and

Algorithm 2 Lookup

$h_1, h_2, \cdots, h_t$ are t hash functions
$T_1, T_2, \cdots, T_t$ are corresponding tables
C is the size of the table (capacity)
input: a key k
for $i = 0 \to t - 1$ **do**
  loc $= h_i(k)$ mod C
  if T_i[loc] $= k$ **then**
    return true
  end if
end for
return false

cache. As a result, the algorithm can be expected to perform better when using separate linear arrays for the tables.

Also, we implement a data structure that can be put onto cpu or gpu and converted between them easily, which is called arrays. For each kind of hashing function, we construct a single array. Then, we can stack these arrays together and put them into a hash table. And in a hash table, we call these arrays by slots.

The most important part is how we can manage the hash table to perform insert and lookup operation.

The first thing is that we inference a specific hash table by four arguments, table capacity, table searching bound, number of hash functions and init constant value for slots.

```
// hash table contains:
// 1. C             : capacity for each slot (cuda array)
// 2. bound         : searching bound
// 3. t             : number of hash functions (slots)
// 4. value         : init constant value for slots
//! THESE PARAMETERS CAN NOT BE CHANGED
#define hash_table_head template<uint C,                          \
                                 uint bound,                      \
                                 uint t,                          \
                                 uint value>
hash_table_head
class HashTable {
public:
    DeviceArray<uint, C> slots[t];
    DeviceArray<uint, 1> collisions; //
    ...
};
```

The second thing is that we perform insert operation by five arguments, array capacity, number of hashing function, number of keys needed to be inserted, max length of searching chains, index of current used hashing function. We notice that we need to consider the elements in array level or hash function level instead of table level. Also, we need to limit the depth of evicting chain and be aware of the current used hashing function.

The third thing is that we perform lookup operation by the same arguments like insert operation.

2.6 Test data generation

To generate a random set of 2^s items to test a hash table, for $s = 10, 11, ..., 24.$,we can use a random number generator to create a set of 32-bit integer that we want to insert into the hash table.

It is important to note that the randomness of the set of items is an important factor in the performance of the hash table. If the set of items is not truly random, it may not accurately reflect the performance of the hash table in a real-world scenario, where the items are likely to be distributed randomly. Therefore, it is important to use a good random number generator to create the set of

items, and to test the hash table using a large enough set of items to get a good representation of its performance.

To generate truly random numbers in C++, we can use a hardware-based random number generator (RNG), i.e. std::random_device. However, it runs not efficiently enough when we deal with a variety of random number generation task. Thus we choose std::mt19937 as our final random number generator. To generate the test files, we refer to Github. It provides concrete useful test cases for this lab.

2.7 Hashing Fucntion design: xxhash

XXHash is a fast, non-cryptographic hash algorithm designed for high-speed data hashing. XXHash is based on a variant of the 32-bit Fowler-Noll-Vo hash function, but with additional optimizations to improve performance. It is a pure hash function, which means that it does not use any cryptographic operations and is not suitable for use in cryptographic applications. However, it has a very low collision rate and is generally considered to be a reliable hash function for non-cryptographic purposes.

XXHash is a streamable hash algorithm, meaning that it can be used to compute the hash of a data stream by dividing the stream into fixed-sized blocks and hashing each block separately. This approach allows for efficient, incremental hashing of large data streams without incurring the memory overhead of hashing the entire stream at once.

To implement xxhash algorithm, we refer to the official implemention on Github. And we find that there are two versions of implemention which is telled from whether we divide the stream into fixed-sized blocks and hashing each block separately.

Algorithm 3 xxhash: low-speed version

```
PRIME32_1, PRIME32_2, ... PRIME32_5 are five primes
rotate_left are function to rotate the bits of a value to the left by a specified number of places
input: a key v and a seed
// XXH32_endian_align
  hash=seed+PRIME32_5
  hash+=4u
// XXH32_finalize
  hash+=v*PRIME32_3
  hash=rotate_left(hash, 17)*PRIME32_4
// XXH32_avalanche
  hash^=hash>>15
  hash*=PRIME32_2
  hash^=hash>>13
  hash*=PRIME32_3
  hash^=hash>>16
```

3 Results

3.1 Experiment 1: Insertion

Demand: 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 $s = 10, 11, ..., 24$.

We find that our algorithm performs stable and fast on all the test cases. And the performance with three hashing functions is better than with two. It may be resulted from that the increasing number of hashing functions enlarges the hash table to decrease the number of evictions.

3.2 Experiment 2: Lookup

Demand: Insert a set of S of 2^{24} random keys into a hash table of size 2^{25}, then perform lookups for the following sets of keys $S_0, ..., S_{10}$. 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. For example, S_0

Algorithm 4 xxhash: high-speed version

```
PRIME32_1, PRIME32_2, ... PRIME32_5 are five primes
rotate_left are function to rotate the bits of a value to the left by a specified number of places
input: a key v and a seed
// XXH32_endian_align
  hash=seed+PRIME32_5
  hash+=4u
// XXH32_finalize
  bytes ← change v to bytes
for i = 0 → 4 do
  hash+=bytes[i]*PRIME32_5
  hash=rotate_left(hash,11)*PRIME32_1
end for
// XXH32_avalanche
  hash^=hash>>15
  hash*=PRIME32_2
  hash^=hash>>13
  hash*=PRIME32_3
  hash^=hash>>16
```

should contain only random keys from S, while S_5 should 50% random keys from S and 50% completely random keys.

We find that the hash table with $t = 2$ hash functions performs better than the one with $t = 3$ hash functions. And the average time for lookup has the ratio $\frac{2}{3}$, the average MOPS for lookup has the ratio $\frac{3}{2}$, which is identical to the fact that we only need to traversal the whole hash table whether it contains the keys.

3.3 Experiment 3: sizes of hash tables

Demand: Fix a set of $n = 2^{24}$ random keys, and measure the time to insert the keys into hash tables of sizes $1.1n, 1.2n, ..., 2n$. Also, measure the insertions times for hash tables of sizes $1.01n, 1.02nand1.05n$. Terminate the experiment if it takes too long and report the time used.

We can see that generally, the performance increases as the size increases because there will be less evictions and rehashes. However, when $n = 1.2$, the performance drops hugely since we counter with rehash frequently. With the table size increasing, the size of hash table does not influence our insertions too much since the number of rehashing decreasing convergingly.

The differences of performance for the insertion operation with $t = 3$ hashing function is small, since the eviction chain is small and result in rehashing very unfrequently.

3.4 Experiment 4: Evict Bound

Demand: 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. Which bound gives the best running time for constructing the hash table? Note however you are not required to find the optimal bound.

Here l is a factor on $\log(n)$, i.e. given l, the evict bound is $M = l\log(n)$. For $t = 2$ hashing function, generally the performance (measured by MOPS) increases with l increasing. And only $l \geq 0.8$ gives good performance. This may because we choose a relative good hash function, xxHash, whose quality is excellent to easily pass Google hash test. And using $t = 3$ hashing cuntion itself makes the number of evictions much fewer than using 2 hash functions, which may result in not countering with $l \geq 0.8$ limit situation.

References

s	t	MOPS	Time (ms)	StdDev
10	2	1.851166	0.553165	0.542525
11	2	25.630756	0.079904	0.016706
12	2	42.454395	0.096480	0.016922
13	2	93.896712	0.087245	0.048093
14	2	190.433681	0.086035	0.019646
15	2	426.346909	0.076858	0.011192
16	2	844.327183	0.077619	0.004116
17	2	1323.510423	0.099034	0.003093
18	2	1848.876039	0.141786	0.003664
19	2	2279.100831	0.230042	0.010700
20	2	2606.758701	0.402253	0.009294
21	2	2693.872848	0.778490	0.090119
22	2	2510.332651	1.670816	0.454907
23	2	2533.977262	3.310451	0.378026

Table 1: Insertion test on NVIDIA GeForce RTX 3090 using $t = 2$ hash functions

s	t	MOPS	Time (ms)	StdDev
10	3	16.549442	0.061875	0.051532
11	3	30.191528	0.067834	0.050062
12	3	79.522861	0.051507	0.002685
13	3	112.567057	0.072774	0.025473
14	3	247.462546	0.066208	0.005980
15	3	440.127229	0.074451	0.008249
16	3	597.014926	0.109773	0.047412
17	3	1319.332630	0.099347	0.001691
18	3	1811.748066	0.144691	0.005989
19	3	2384.792285	0.219846	0.003305
20	3	2714.740173	0.386253	0.002542
21	3	2875.495789	0.729318	0.007748
22	3	2800.109362	1.497907	0.087006
23	3	2425.328004	3.458752	0.639231
24	3	2359.757076	7.109722	0.697113

Table 2: Insertion test on NVIDIA GeForce RTX 3090 using $t = 3$ hash functions

i	t	MOPS	Time (ms)	StdDev
0	2	6222.087130	2.696397	0.001505
1	2	5225.489516	3.210650	0.711989
2	2	5194.344066	3.229901	0.321954
3	2	4710.457034	3.561696	0.650128
4	2	4503.325940	3.725517	0.644506
5	2	4427.629440	3.789210	0.260980
6	2	4119.714095	4.072422	0.644109
7	2	3953.465170	4.243674	0.638899
8	2	3699.923254	4.534477	0.626630
9	2	3598.965943	4.661677	0.630336
10	2	3390.588100	4.948173	0.647018
0	3	5013.550306	3.346374	0.702805
1	3	4540.587412	3.694944	0.601766
2	3	4407.796557	3.806259	0.414752
3	3	3808.461203	4.405248	0.649267
4	3	3543.736439	4.734330	0.604948
5	3	3345.606522	5.014701	0.654106
6	3	3165.868036	5.299405	0.434075
7	3	2845.932858	5.895155	0.629987
8	3	2506.789441	6.692710	0.730375
9	3	2675.812546	6.269952	0.003106
10	3	2209.502567	7.593210	0.635024

Table 3: Lookup test on NVIDIA GeForce RTX 3090 using 2 and 3 hash functions

s	t	MOPS	Time (ms)	StdDev
2.0	2	1914.092749	8.765101	0.661378
1.9	2	1890.463829	8.874656	0.715771
1.8	2	1883.947859	8.905350	0.828579
1.7	2	2256.727959	7.434310	0.591259
1.6	2	2148.149097	7.810080	0.746547
1.5	2	2249.545229	7.458048	0.346503
1.4	2	2144.334209	7.823974	0.716834
1.3	2	2121.280321	7.909005	0.754375
1.2	2	2084.768434	8.047520	0.626051
1.1	2	2016.189889	8.321248	0.723715
1.05	2	2068.923598	8.109152	0.517474
1.02	2	2017.021390	8.317818	0.631189
1.01	2	1943.560927	8.632205	0.671189

Table 4: Size test on NVIDIA GeForce RTX 3090 using 2 hash functions

s	t	MOPS	Time (ms)	StdDev
2.0	3	50.112802	334.789020	1.579255
1.9	3	59.195243	283.421692	0.991593
1.8	3	57.398944	292.291370	1.156639
1.7	3	2078.080208	8.073421	0.747715
1.6	3	2068.164588	8.112128	0.746399
1.5	3	2042.988241	8.212096	0.780235
1.4	3	60.418237	277.684631	3.144495
1.3	3	2029.377284	8.267174	0.662690
1.2	3	1957.107606	8.572455	0.742340
1.1	3	1987.335035	8.442067	0.561624
1.05	3	1891.565991	8.869485	0.702452
1.02	3	1831.829772	9.158720	0.755790
1.01	3	362.472722	46.285458	55.924239

Table 5: Size test on NVIDIA GeForce RTX 3090 using 3 hash functions

l	t	MOPS	Time (ms)	StdDev
0.2	2	171.475458	97.840334	75.369272
0.4	2	122.833874	136.584604	1.125857
0.6	2	82.949906	202.257202	1.032231
0.8	2	67.917909	247.021976	1.016377
1.0	2	65.395948	256.548248	0.852911
1.2	2	65.127315	257.606445	1.320518
1.4	2	55.426514	302.692969	1.389676
1.6	2	1886.208982	8.894675	1.033725
1.8	2	1951.643663	8.596455	0.744806
2.0	2	1900.591315	8.827366	1.072119
2.2	2	1909.201048	8.787559	0.957352
2.4	2	1915.627189	8.758080	0.796888
3.6	2	1890.116202	8.876288	0.860378
4.8	2	1937.003614	8.661427	1.036297
7.2	2	1899.329953	8.833229	0.904491
9.6	2	1899.149743	8.834067	0.947624
14.4	2	1901.087507	8.825062	0.927500
19.2	2	1965.290451	8.536762	0.895939

Table 6: Bound test on NVIDIA 3090 using 2 hash functions

l	t	MOPS	Time (ms)	StdDev
0.2	3	62.172992	269.847333	4.011280
0.4	3	1958.441104	8.566618	0.915396
0.6	3	1990.414775	8.429005	0.654569
0.8	3	2057.231818	8.155238	0.694350
1.0	3	2026.667754	8.278227	1.034394
1.2	3	2059.355404	8.146829	0.760283
1.4	3	1972.075125	8.507392	0.848747
1.6	3	2017.379930	8.316339	0.549848
1.8	3	2037.351636	8.234816	0.593279
2.0	3	2017.137813	8.317338	0.853806
2.2	3	2033.257326	8.251398	0.636399
2.4	3	2028.705118	8.269914	0.669827
3.6	3	2010.925139	8.343034	0.816520
4.8	3	1956.705920	8.574214	0.863485
7.2	3	1969.394977	8.518970	0.851805
9.6	3	2101.092013	7.984998	0.327216
14.4	3	2003.757626	8.372877	1.015644
19.2	3	1945.375367	8.624154	1.025929

Table 7: Bound test on NVIDIA 3090 using 3 hash functions