Parallel Computing Final Project – CUDA Cuckoo Hashing Report

NAME: MO XIAOYE
STUDENT NUMBER: 68488464
EMAIL: MOXY@SHANGHAITECH.EDU.CN

1 INTRODUCTION

Cuckoo hashing is named after a type of bird which lays its eggs in other birds' nests, and whose chicks push out other eggs from the nest after they hatch. Cuckoo hashing uses an array of size *n*, allowing up to *n* items to be stored in the hash table. For simplicity, we only consider storing keys in the hash table and ignore any associated data.

For this project, I will implement cuckoo hashing on the GPU using CUDA. First, design a highly parallel cuckoo hashing algorithm optimized for the GPU architecture. My goal is to make the algorithm as fast as possible.

1.1 Enviornment

Windows 10, VS2015

GPU information:
— General Information for device 0 —
Name: GeForce GTX 1060 6GB
Compute capability: 6.1
Clock rate: 1784500
Device copy overlap: Enabled
Kernel execition timeout : Enabled
— Memory Information for device 0 —
Total global mem: -2147483648
Total constant Mem: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
— MP Information for device 0 —
Multiprocessor count: 10
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)

1.2 How to Run?

On Windows, open "ParallelComputingProject.sln" with VS2015.
Run the code, and follow the instructions in the terminal.
Wait for a little bit, and the results will come out.
If you are using Linux, please use "nvcc" to build it.

2 IMPLEMENTATION DETAILS

In my implementation, there are two main CUDA kernel, one is to do the hashing, the other is to check the collison whether the bucket has been inserted. I will introduce these two kernels in details below.

2.1 cuckooHash Kernel

hashTable is the hash table of cuckoo hashing, *a* and *b* is the array of different hash functions' parameters, *entry* is the input random number array, *function* stores each random number in the *entry* using which hash function, *collison* stores each random number in the *entry* meets collision when hashing, *n_function* means using how many hash functions, *n* and *p* are the paramenters of hash functions.

First, calculate the key of the the hash table, assuming one thread one key. Using the corresponding hash function to do the hash, then test if the bucket in the hash table is empty or this key has a collision in previous iteration, and add the key into the bucket. Then, turn the this key's hash function to the next hash function for later operations.

```
__global__ void cuckooHash(
  unsigned* hashTable,
  unsigned* a, unsigned* b,
  unsigned* entry,
  unsigned* function,
  unsigned* collision,
  unsigned n_function, unsigned n, unsigned p)
{
  unsigned k = blockDim.x*blockIdx.x+threadIdx.x;
  unsigned num = function[k];
  unsigned hashValue = ((a[num] * entry[k] +
        b[num]) % p) % n;
  if (collision[k] == 1 ||
        hashTable[hashValue] == 0xffffffff) {
      hashTable[hashValue] = entry[k];
      function[k] = (num + 1) % n_function;
  }
}
```

2.2 detectCollision Kernel

hashTable is the hash table of cuckoo hashing, *a* and *b* is the array of different hash functions' parameters, *entry* is the input random number array, *function* stores each random number in the *entry* using which hash function, *collison* stores each random number in the *entry* meets collision when hashing, *n_function* means using how many hash functions, *n* and *p* are the paramenters of hash functions.

In this part, the goal is to test whether the key in the *entry* meets collisions in previous hashing. At first, calculate each key's hash value, test if the corresponding bucket in the hash table is the key. If true, do nothing. If not, set the collision[key] to 1.

```
__global__ void detectCollision(
unsigned* hashTable,
unsigned* a, unsigned* b,
unsigned* entry,
unsigned* function,
unsigned* collision,
unsigned n_function, unsigned n, unsigned p)
{
  unsigned k = blockDim.x*blockIdx.x+threadIdx.x;
  unsigned num = (function[k] - 1) % n_function;
  unsigned hashValue = ((a[num] * entry[k] +
          b[num]) % p) % n;
  if (hashTable[hashValue] != entry[k]) {
    collision[k] = 1;
  } else {
    collision[k] = 0;
  }
}
```

2.3 lookup Kernel

hashTable is the hash table of cuckoo hashing, *a* and *b* is the array of different hash functions' parameters, *searchEntry* is the input random number array going to be searched, *function* stores each random number in the *searchEntry* using which hash function, *dict* stores if the search get hash hit, *n_function* means using how many hash functions, *n* and *p* are the paramenters of hash functions.

This CUDA kernel is used for Task 2 when testing the performance of lookups. The code is very simple, just test the hash value bucket in hash table whether is equal to the searching element.

```
__global__ void lookup(
  unsigned* hashTable,
  unsigned* a, unsigned* b,
  unsigned* searchEntry,
  unsigned* dict,
  unsigned n_function, unsigned n, unsigned p)
{
  unsigned k = blockDim.x*blockIdx.x+threadIdx.x;
  for (unsigned i = 0; i < n_function; i++) {
    unsigned hashValue = ((a[i]*searchEntry[k] +
          b[i]) % p) % n;
    if (hashTable[hashValue] == searchEntry[k]) {
      dict[k] = 1;
      break;
    }
  }
}
```

2.4 Generate Random Number

I use the *rand()* function in *stdlib.h*, but I found it can only generate the random number between 0 and $2^{15} - 1$. For larger random number, I will use my own random number generating function.

```
unsigned myrand() {
  unsigned a = rand()<<10;
  unsigned b = rand();
  return a + b;
}
```

2.5 Generate a and b Arrays

In my implementation, I set *a* and *b* between 0 to 10, because I found in this case, it has fewer collisions and fewer rehashing conditions, it can accelerate the whole program.

For testing Task 5, just need to modify the last 2 rows in this functions.

```
void generate_a_b(unsigned n_function, unsigned* a,
        unsigned* b) {
  for (unsigned i = 0; i < n_function; i++) {
    a[i] = rand() % 10;
    b[i] = rand() % 10;
    if (i != 0) {
      while (a[i] == a[i - 1] || b[i] == b[i - 1]){
        a[i] = rand() % 10;
        b[i] = rand() % 10;
      }
    }
  }
  /////////////For task 5
  //Modify the bellows
  //a[0] = 232;
  //b[0] = 0;
```

2.6 Main Function

I just show the hash part, for more details please check the source code.

```
...

iteration = 0;
startTime = clock();
do {
  flag = 0;
  //Restarting hash
  if (iteration == limit) {
    iteration = 0;
    std::cout << "....Rehash...." << std::endl;
    generate_a_b(n_function, a, b);
    memset(hashTable, 0xffffffff, N *
            sizeof(unsigned));
    memset(function, 0, entryLength *
            sizeof(unsigned));
    memset(collision, 0, entryLength *
            sizeof(unsigned));

    err = cudaMemcpy(d_hashTable, hashTable,
            N * sizeof(unsigned),
            cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣hashTable"
        << std::endl;
      goto Error;
    }

    err = cudaMemcpy(d_a, a, n_function *
```

```
                    sizeof(unsigned),
                    cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣a[]"
          << std::endl;
      goto Error;
    }

    err = cudaMemcpy(d_b, b, n_function *
                    sizeof(unsigned),
                    cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣b[]"
          << std::endl;
      goto Error;
    }

    err = cudaMemcpy(d_entry, entry, entryLength *
                    sizeof(unsigned),
                    cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣entry[]"
          << std::endl;
      goto Error;
    }

    err = cudaMemcpy(d_function, function,
                    entryLength * sizeof(unsigned),
                    cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣functionIndex[]"
          << std::endl;
      goto Error;
    }

    err = cudaMemcpy(d_collision, collision,
                    entryLength * sizeof(unsigned),
                    cudaMemcpyHostToDevice);
    if (err != cudaSuccess) {
      std::cout << "-->Fail␣to␣copy␣collision[]"
          << std::endl;
      goto Error;
    }
  }

  iteration++;
  cuckooHash << < blockNum, blockSize >> >
  (d_hashTable,
        d_a, d_b,
        d_entry,
        d_function,
        d_collision,
        n_function, N, p);

  detectCollision << < blockNum, blockSize >> >
  (d_hashTable,
        d_a, d_b,
        d_entry,
        d_function,
        d_collision,
        n_function, N, p);


    err = cudaMemcpy(collision, d_collision,
            entryLength * sizeof(unsigned),
            cudaMemcpyDeviceToHost);
    if (err != cudaSuccess) {
      std::cout << "Copy␣collison␣failed␣"
          << std::endl;
      std::cout << cudaGetErrorString(err)
          << std::endl;
      goto Error;
    }

    for (unsigned i = 0; i < entryLength; i++) {
      flag += collision[i];
    }
    std::cout << flag << "␣collisions" << std::endl;

  } while (flag != 0);
  endTime = clock();
  std::cout << "Hash␣Done!" << std::endl;

  ...
```

2.7 Initilization

All keys in *hashTable* are set to 0xffffffff, *function* and *collision* are set to 0.

In most cases, the key values are set from 0 to 10^7.

All the running time does not include the time of copying data between device and host.

3 RESULTS

There are 5 tasks for us to do, each task using 2 or 3 hash functions and repeat for 5 times.

3.1 Task 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, ... 24.

At first, I test on key values between 0 to $2^{15} - 1$, then I test on key values between 0 to 10^7, and the result is shown in Fig.1 and Fig.2

3.2 Task 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_{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.

I test the running time and hit rate, the result is shown in Fig.3 and Fig.4. From the result, we can see the lookup time is very fast, also the hit rate looks well. The hash values are not so large may cause this result.

Name: MO Xiaoye
student number: 68488464
email: moxy@shanghaitech.edu.cn

keyValue = [0, 2^15]

num_hashFunction = 2

s	1	2	3	4	5	avg	speed
10	2ms	1ms	1ms	2ms	1ms	1ms	1.024
11	1ms	1ms	2ms	2ms	1ms	1ms	2.048
12	2ms	2ms	1ms	1ms	1ms	1ms	4.096
13	4ms	3ms	3ms	3ms	3ms	3ms	2.731
14	3ms	3ms	2ms	3ms	4ms	3ms	5.461
15	3ms	4ms	4ms	4ms	3ms	4ms	8.192
16	71ms	4ms	3ms	3ms	4ms	17ms	3.855
17	5ms	5ms	5ms	5ms	163ms	37ms	3.542
18	5ms	4ms	6ms	6ms	5ms	5ms	52.429
19	11ms	10ms	9ms	149ms	11ms	38ms	13.797
20	17ms	17ms	20ms	18ms	19ms	18ms	58.254
21	33ms	33ms	33ms	32ms	33ms	33ms	63.550
22	698ms	59ms	60ms	58ms	60ms	187ms	22.429
23	1277ms	113ms	1270ms	103ms	102ms	573ms	14.640
24	199ms	196ms	210ms	212ms	200ms	203ms	82.646

num_hashFunction = 3

s	1	2	3	4	5	avg	speed
10	1ms	3ms	2ms	1ms	1ms	2ms	0.512
11	2ms	1ms	2ms	2ms	1ms	2ms	1.024
12	2ms	5ms	4ms	3ms	3ms	3ms	1.365
13	2ms	3ms	3ms	3ms	3ms	3ms	2.731
14	11ms	3ms	3ms	7ms	3ms	5ms	3.277
15	6ms	3ms	4ms	2ms	3ms	4ms	8.192
16	10ms	3ms	7ms	4ms	4ms	6ms	10.923
17	4ms	4ms	4ms	4ms	4ms	4ms	32.768
18	5ms	4ms	14ms	6ms	5ms	7ms	37.449
19	8ms	6ms	9ms	7ms	7ms	7ms	74.898
20	13ms	12ms	13ms	13ms	13ms	13ms	80.660
21	21ms	20ms	21ms	21ms	20ms	21ms	99.864
22	40ms	41ms	40ms	40ms	42ms	41ms	102.300
23	75ms	73ms	69ms	70ms	220ms	101ms	83.056
24	134ms	133ms	133ms	131ms	135ms	133ms	126.144

Fig. 1. For key values between 0 to $2^{15} - 1$

keyValue = [0, 10^7]

num_hashFunction = 2-1

s	1	2	3	4	5	avg	speed
10	1ms	2ms	1ms	1ms	2ms	1ms	1.024
11	1ms	2ms	2ms	2ms	1ms	2ms	1.024
12	1ms	2ms	1ms	3ms	1ms	2ms	2.048
13	69ms	1ms	4ms	2ms	1ms	15ms	0.546
14	70ms	4ms	4ms	4ms	4ms	17ms	0.964
15	69ms	4ms	4ms	4ms	5ms	17ms	1.928
16	6ms	6ms	5ms	4ms	4ms	5ms	13.107
17	6ms	8ms	4ms	5ms	5ms	6ms	21.845
18	88ms	7ms	7ms	8ms	7ms	23ms	11.398
19	10ms	8ms	13ms	10ms	12ms	11ms	47.663
20	23ms	18ms	19ms	18ms	23ms	20ms	52.429
21	264ms	41ms	40ms	75ms	26ms	89ms	23.564
22	52ms	715ms	52ms	74ms	52ms	189ms	22.192
23	98ms	128ms	99ms	1169ms	101ms	319ms	26.297
24	257ms	2454ms	249ms	4560ms	239ms	1552ms	10.810

num_hashFunction = 3-1

s	1	2	3	4	5	avg	speed
10	1ms	1ms	2ms	1ms	1ms	1ms	1.024
11	2ms	1ms	2ms	2ms	1ms	2ms	1.024
12	2ms	1ms	2ms	2ms	2ms	2ms	2.048
13	2ms	2ms	5ms	3ms	2ms	3ms	2.731
14	3ms	3ms	3ms	7ms	5ms	4ms	4.096
15	6ms	7ms	3ms	4ms	4ms	5ms	8.192
16	9ms	10ms	6ms	3ms	9ms	7ms	9.362
17	4ms	7ms	11ms	11ms	11ms	9ms	14.564
18	7ms	6ms	6ms	27ms	17ms	13ms	20.165
19	4ms	5ms	5ms	43ms	4ms	12ms	74.898
20	10ms	7ms	67ms	7ms	40ms	26ms	40.330
21	14ms	77ms	74ms	13ms	15ms	39ms	53.773
22	136ms	27ms	26ms	26ms	26ms	48ms	87.381
23	48ms	49ms	255ms	49ms	50ms	90ms	93.207
24	234ms	2224ms	1928ms	171ms	2927ms	1497ms	11.207

Fig. 2. For key values between 0 to 10^7

keyValue = [0, 10^7]

time, num_hashFunction = 2

i	1	2	3	4	5	avg
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0

hitRate, num_hashFuncion = 2

i	1	2	3	4	5	avg
0	100%	100%	100%	100%	100%	100%
1	92.4067%	92.4027%	92.3948%	92.4101%	92.4064%	92.40414%
2	84.8025%	84.8031%	84.8044%	84.8103%	84.8104%	84.80614%
3	77.2113%	77.2019%	77.2059%	77.2096%	77.215%	77.20874%
4	69.6147%	69.608%	69.6198%	69.6185%	69.6128%	69.61476%
5	62.0259%	62.0105%	61.9937%	62.0277%	62.0185%	62.01526%
6	54.4121%	54.402%	54.4068%	54.4117%	54.4154%	54.4096%
7	46.8276%	46.8184%	46.8379%	46.8315%	46.8369%	46.83046%
8	39.2169%	39.2173%	39.2121%	39.2064%	39.2379%	39.21812%
9	31.63%	31.6317%	31.6383%	31.6451%	31.6127%	31.63156%
10	24.0285%	24.0391%	24.0081%	24.0192%	24.0286%	24.0247%

Fig. 3. Using 2 hash functions

time, num_hashFunction = 3

i	1	2	3	4	5	avg
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	0	0
7	0	0	0	0	0	0
8	0	0	0	0	0	0
9	0	0	0	0	0	0
10	0	0	0	0	0	0

hitRate, num_hashFuncion = 3

i	1	2	3	4	5	avg
0	100%	100%	100%	100%	100%	100%
1	92.403%	92.4014%	92.4027%	92.4035%	92.4014%	92.4024%
2	84.8019%	84.8007%	84.8041%	84.8056%	84.8043%	84.80332%
3	77.2037%	77.2121%	77.2086%	77.2117%	77.2179%	77.2108%
4	69.6%	69.6067%	69.6066%	69.6127%	69.609%	69.607%
5	62.0126%	62.0019%	62.0106%	62.025%	62.0136%	62.01274%
6	54.4174%	54.4186%	54.4281%	54.4301%	54.4331%	54.42546%
7	46.8392%	46.825%	46.8206%	46.8126%	46.821%	46.82368%
8	39.2251%	39.2117%	39.2296%	39.2293%	39.2373%	39.2266%
9	31.6257%	31.6373%	31.6226%	31.6335%	31.6214%	31.6281%
10	24.0276%	24.0206%	24.0315%	24.0369%	24.0404%	24.0314%

Fig. 4. Using 3 hash functions

3.3 Task 3

Fix a set of n = 2^{24} random keys, and measure the time to insert the keys into hash tablesofsizes 1.1n, 1.2n, ... , 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 and report the time used.

Because the hash values are not very large, the result in Fig.5 and Fig.6 seems not bad. If meets rehash conditions, the running time will be very slow.

num_hashFunction = 2

	1	2	3	4	5	avg
1.1n	172ms	177ms	258ms	263ms	181ms	210ms
1.2n	179ms	2064ms	1333ms	185ms	251ms	802ms
1.3n	5231ms	1165ms	1340ms	166ms	237ms	1628ms
1.4n	184ms	181ms	188ms	179ms	244ms	195ms
1.5n	1995ms	183ms	680ms	258ms	924ms	808ms
1.6n	182ms	180ms	182ms	181ms	180ms	181ms
1.7n	252ms	264ms	1498ms	315ms	435ms	553ms
1.8n	168ms	1248ms	427ms	266ms	766ms	575ms
1.9n	236ms	247ms	264ms	2502ms	184ms	687ms
2.0n	1080ms	182ms	1532ms	918ms	259ms	794ms

num_hashFunction = 3

	1	2	3	4	5	avg
1.1n	7498ms	2577ms	5135ms	91ms	93ms	3079ms
1.2n	7623ms	4784ms	4620ms	89ms	222ms	3468ms
1.3n	93ms	2583ms	97ms	1097ms	1426ms	1059ms
1.4n	93ms	565ms	92ms	91ms	93ms	187ms
1.5n	92ms	4173ms	93ms	2973ms	336ms	1533ms
1.6n	90ms	93ms	90ms	90ms	93ms	91ms
1.7n	94ms	88ms	478ms	90ms	91ms	168ms
1.8n	90ms	92ms	484ms	2672ms	91ms	686ms
1.9n	2593ms	91ms	93ms	94ms	95ms	593ms
2.0n	89ms	93ms	92ms	90ms	2601ms	593ms

Fig. 5. 1.1n to 2n

num_hashFunction = 2-1

	1	2	3	4	5	avg
1.01n	180ms	5947ms	177ms	243ms	262ms	1362ms
1.02n	6488ms	4294ms	255ms	1089ms	5727ms	3571ms
1.03n	4080ms	3894ms	2609ms	3580ms	3816ms	3596ms
1.04n	258ms	3443ms	5097ms	259ms	1572ms	2126ms
1.05n	1797ms	263ms	1726ms	1238ms	1483ms	1301ms

num_hashFunction = 3-1

	1	2	3	4	5	avg
1.01n	89ms	5388ms	2962ms	2561ms	90ms	2218ms
1.02n	95ms	2609ms	2575ms	727ms	7606ms	2722ms
1.03n	90ms	7276ms	88ms	7281ms	7263ms	4400ms
1.04n	92ms	9804ms	88ms	90ms	90ms	2033ms
1.05n	2581ms	2415ms	90ms	7422ms	2568ms	3015ms

Fig. 6. 1.01n to 1.05n

3.4 Task 4

Using n = 2^{24} random keys and a hash table of size 1.2n, 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.

Well, the optimal bound is hard to find, because the random fator. If we get a good random hash function, it will hash very fast, but if we do not get a good hash function, it will be very very slow even change the hash function. But from the result in Fig.7, it seems set the coefficient to 5 or 5.5 is better.

3.5 Task 5

Finally, experiment with the type of hash functions you use. You are free to select your own hash functions, which should balance the amount of evictions they cause with the functionŠs complexity. Report on the hash functions you used, and the amount of time to insert n = 2^{24} random keys into a hash table of size 1.2n, using a fixed bound you select for the eviction chain length.

In my case, I just test different *a* and *b* how to influnce the run time, the result is shown in Fig.8 and Fig.9.

From the result, we can see, *a* needs to be small, otherwise the running time will be slow.

num_hashFunction = 2

	1	2	3	4	5	avg
1	975ms	182ms	91ms	312ms	179ms	348ms
1.5	185ms	248ms	181ms	1167ms	263ms	409ms
2	2757ms	179ms	1164ms	182ms	178ms	892ms
2.5	265ms	181ms	170ms	241ms	181ms	208ms
3	1177ms	183ms	260ms	181ms	180ms	396ms
3.5	1138ms	1409ms	264ms	261ms	177ms	650ms
4	258ms	178ms	183ms	1317ms	245ms	436ms
4.5	245ms	169ms	183ms	1495ms	265ms	471ms
5	184ms	183ms	168ms	1328ms	183ms	409ms
5.5	1251ms	180ms	261ms	181ms	1328ms	640ms
6	181ms	246ms	260ms	1173ms	1336ms	639ms

num_hashFunction = 3

	1	2	3	4	5	avg
1	372ms	533ms	86ms	779ms	2842ms	922ms
1.5	84ms	84ms	962ms	1939ms	86ms	631ms
2	6133ms	85ms	85ms	87ms	86ms	1295ms
2.5	86ms	1636ms	462ms	87ms	86ms	471ms
3	87ms	88ms	85ms	461ms	3553ms	855ms
3.5	86ms	89ms	85ms	2159ms	86ms	501ms
4	85ms	86ms	2363ms	2473ms	86ms	1019ms
4.5	85ms	84ms	87ms	2692ms	86ms	607ms
5	3020ms	5793ms	88ms	86ms	87ms	1815ms
5.5	444ms	458ms	85ms	85ms	86ms	232ms
6	88ms	540ms	3398ms	85ms	86ms	839ms

Fig. 7. Different bound coefficient

num_hashFunction = 2, a1=1, b1=0, bound=6logN

	1	2	3	4	5	avg
1,0	164ms	165ms	242ms	244ms	245ms	212ms
1,1	166ms	242ms	169ms	170ms	246ms	199ms
1,2	167ms	169ms	171ms	244ms	168ms	184ms
1,3	170ms	249ms	166ms	168ms	170ms	185ms
2,0	860ms	169ms	236ms	171ms	172ms	322ms
3,0	169ms	243ms	169ms	169ms	166ms	183ms
4,0	244ms	246ms	246ms	170ms	172ms	216ms
5,0	233ms	170ms	169ms	169ms	172ms	183ms
6,0	241ms	172ms	240ms	169ms	168ms	198ms
100,15008	1387ms	1086ms	1006ms	1013ms	1081ms	1115ms
185,0	1241ms	1243ms	1240ms	1090ms	1081ms	1179ms
1,208	172ms	168ms	170ms	171ms	172ms	171ms
1,10891	167ms	168ms	169ms	171ms	170ms	169ms
232,0	2534ms	6438ms	1388ms	1167ms	7804ms	3866ms

Fig. 8. Using 2 hash funtions

num_hashFunction = 3, a1=1, b1=0, bound=6logN

	1	2	3	4	5	avg
1,0	84ms	84ms	85ms	84ms	85ms	84ms
1,1	85ms	84ms	86ms	86ms	88ms	86ms
1,2	88ms	86ms	85ms	84ms	85ms	86ms
1,3	85ms	86ms	85ms	86ms	85ms	85ms
2,0	85ms	86ms	87ms	86ms	85ms	86ms
3,0	84ms	84ms	87ms	85ms	87ms	85ms
4,0	86ms	86ms	86ms	86ms	85ms	86ms
5,0	85ms	89ms	88ms	84ms	85ms	86ms
6,0	85ms	85ms	86ms	85ms	86ms	85ms
100,15008	-	-	-	-	-	-
185,0	6319ms	20256ms	12030ms	10108ms	10099ms	11762ms
1,208	86ms	85ms	85ms	84ms	85ms	85ms
1,10891	85ms	86ms	85ms	84ms	85ms	85ms
232,0	-	-	-	-	-	-

Fig. 9. Using 3 hash funtions