Ramanujan Numbers

TU Vienna - Efficient Programs - 185.190

Thomas Fenz, 00702096
David Fuchssteiner, 00904294

Problem Definition

The Ramanujan numbers:

$i^3 + j^3 = k^3 + l^3$ and $i^3 + j^3 <= N$

So, the Ramanujan numbers are positive integer that can be expressed as the sum of 2 cubes in 2 ways.

Optimization Overview

based on reference programs

Optimizing Ramasort

Optimizing Ramanujan (hashing based)

3-Nested-Loops-Zero-Mem

Double hashing / No chaining

STL Hashset

Bucket Hashing

new implementations

Optimizing Ramasort

N=10^{13}, MEMORY=9000000

- Optimization of data structure
- Remove “k, l” variables in struct
- Results:
  - ½ memory usage
  - Performance doubled

```
struct entry {
  int k,l;
  long value;
};
```

```
Memory usage: >=3713272656

 Performance counter stats for './ramasort 10000000000000':

      115335927284      cycles
      206884997854      instructions              #    1.79  insn per cycle
         608632006      branch-misses
          50619386      LLC-load-misses
         160180649      LLC-store-misses

      24.861121914 seconds time elapsed

      23.498854000 seconds user
       1.335934000 seconds sys
```

```
Memory usage: >=1856636328

 Performance counter stats for './ramasort_opt 10000000000000':

       58139937226      cycles
      125507760360      instructions              #    2.16  insn per cycle
         619435114      branch-misses
          21887689      LLC-load-misses
          78201161      LLC-store-misses

      12.515058860 seconds time elapsed

      11.841454000 seconds user
       0.660081000 seconds sys
```

Optimizing Ramanujan *(a) Reduce Cube Computations*

- Compute cubes only **once for i, j in outer loop** and **once for j in inner loop.**

```
for (long i = 0; cube( n: i) <= n; i++) {
    for (long j = i + 1; cube( n: i) + cube( n: j) <= n; j++) {
        long sum = cube( n: i) + cube( n: j)
        //...
    }
}
```

```
long cubeI = 0;
for (long i = 0; cubeI <= n; i++) {
    cubeI = cube( n: i);
    long cubeJ = cube( n: i+1);
    for (long j = i + 1; cubeI + cubeJ <= n; j++) {
        cubeJ = cube( n: j);
        long sum = cubeI + cubeJ;
        //...
    }
}
```

could be further improved by reusing result of **cube(i+1)** for next outer loop iteration, but the compiler is doing its job anyhow.

- **The increase in performance is negligible.**
- Ramanujan N=10^{13}: **12.5 vs 12.3** seconds

Optimizing Ramanujan *(b) Better Hash Function*

Problem: too many collisions using the given hash function.

Properties of good hash functions:

- Good statistical distribution.
- Each input bit affects each output bit with about 50% probability.
- Easy to compute.

```
size_t hash(long v, size_t bound){
    return v&(bound-1);
}
```

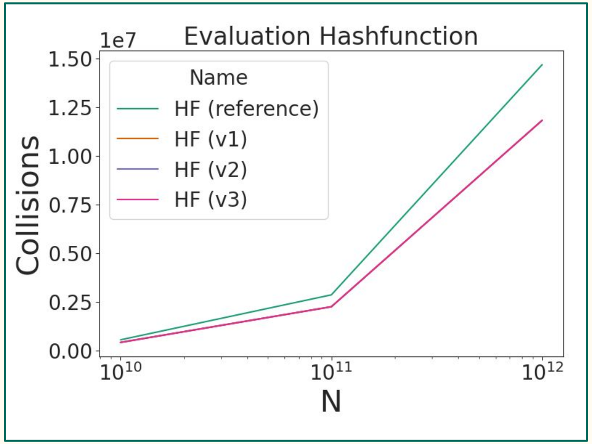
```
size_t hash_opt(long h, size_t bound){
    h ^= (h >> 20) ^ (h >> 12);
    h = h ^ (h >> 7) ^ (h >> 4);
    return h % bound;
}
```

A *experimental evaluation* of 4 hash functions:

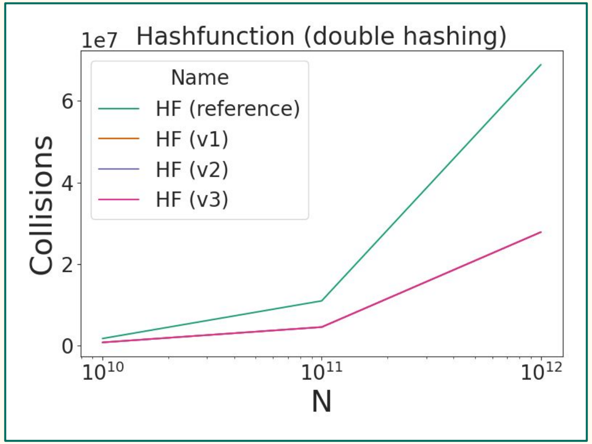
all were equally good - except the function from the reference program

caused a significantly higher amount of collisions.

Optimizing Ramanujan *(b)*



The hash function in the reference implementation produces $> \mathbf{20\%}$ more collisions.



When using double hashing (see later) this factor increased to $> \mathbf{250\%}$!!

```
long hash_ref(long h, long bound) {
    return h & (bound - 1);
}
```

```
long hashV1(long h, long bound) {
    h += ~(h << 15);
    h ^= (h >> 10);
    h += (h << 3);
    h ^= (h >> 6);
    h += ~(h << 11);
    h ^= (h >> 16);
    return h & (bound - 1);
}
```

```
long hashV2(long h, long bound) {
    h ^= (h >> 20) ^ (h >> 12);
    h = h ^ (h >> 7) ^ (h >> 4);
    return h & (bound - 1);
}
```

```
long hashV3(long h, long bound) {
    h = (h ^ 0xdeadbeef) + (h << 4);
    h = h ^ (h >> 10);
    h = h + (h << 7);
    h = h ^ (h >> 13);
    return h & (bound - 1);
}
```

Small study regarding integer hash functions: https://burtleburtle.net/bob/hash/integer.html

Nested Loops - “*Zero*” Memory

```
for (candidate = 0; candidate < N; candidate++) {
    hits = 0;
    for (i = 1; cube( n: i) < candidate; i++) {
        for (j = i + 1; cube( n: i) + cube( n: j) <= candidate; j++) {
            if (cube( n: i) + cube( n: j) == candidate) {
                hits++;
                i++; // one i can only have one hit
            }
        }
    }
    if (hits == 2) {
        checksum += candidate;
        count++;
    }
}
```

- It is possible to compute all Ramanujan Numbers without (< 8 variables) memory consumption.
- Caveat: very inefficient if $N > 10^7$

Nested Loops - “*Zero*” Memory - Results

```
Performance counter stats for './3loops 1000000':

      11906615068      cycles
      37467887778      instructions              #    3.15  insn per cycle
         45411543      branch-misses
             3656      LLC-load-misses
             2881      LLC-store-misses

      2.555593115 seconds time elapsed

      2.555629000 seconds user
      0.000000000 seconds sys
```

Note that N>=10^7 didn’t finish (yet... :)!

Naïve STL Hashset

...we don't have to reinvent the wheel!

(ordered) hashset

std::set<T>

implemented as red-black tree

Problem: insert O(log(tree height))

```
Memory usage: >=1640015576
 Performance counter stats for './naive_hashset 10000000000000':

     793747740298      cycles
     250338308497      instructions              #    0.32  insn per cycle
       2679870753      branch-misses
       2679377341      LLC-load-misses
        286354156      LLC-store-misses

    242.484280557 seconds time elapsed

    237.014700000 seconds user
      5.395059000 seconds sys
```

Runtime: ~242 sec

unordered hashset

std::unordered_set<T>

hashtable

insert: ~O(1)

Speedup: ~1.7

```
Memory usage: >=2148398632
 Performance counter stats for './naive_hashset_unordered 10000000000000':

     407078047103      cycles
     185856675794      instructions              #    0.46  insn per cycle
        452281524      branch-misses
       2158046006      LLC-load-misses
        320069311      LLC-store-misses

    141.386926949 seconds time elapsed

    136.646384000 seconds user
      4.708082000 seconds sys
```

Runtime: ~141 sec

unordered hashset + reserved capacity

std::unordered_set<T>;

.reserve()

no rehashing needed

Speedup: again ~1.7

```
Memory usage: >=2148398632
 Performance counter stats for './naive_hashset_unordered 10000000000000':

     259869084548      cycles
     183477426259      instructions              #    0.71  insn per cycle
        282926217      branch-misses
       1038344413      LLC-load-misses
        320803329      LLC-store-misses

     80.630881418 seconds time elapsed

     76.026427000 seconds user
      4.575905000 seconds sys
```

Runtime: ~81 sec

Naïve STL Hashset

...or do we?!

(ordered) hashset

std::set<T>

implemented as red-black tree

unordered hashset

std::unordered_set<T>

hashtable

insert: ~O(1)

Speedup: ~1.7

unordered hashset + reserved capacity

std::unordered_set<T>;

.reserve()

no rehashing needed

Speedup: again 1.7

```
template<class Iter, class NodeType>
struct /*unspecified*/
{
    Iter     position;
    bool      inserted;
    NodeType node;
};
```

Problem with STL implementation

too universal => unneeded overhead

Double Hashing / No Chaining

```
long *candidates = new long[bound];
char *counts = new char[bound];
```

Idea: Double hashing instead of separate chaining

pros:

- No pointer in data structure
- Single dereference operation
- Separate arrays for *counts* and *candidates*
  => *counts* can be a char (1 byte) array
- High utilization of space in hash table

cons:

- More collisions => **quality of hash function even more important!!**

```
// initial hashing
long idx = hash_improved( h: current_candidate, bound);

// rehash if collision
int collision_count = 0;
while ((candidates[idx]) && candidates[idx] != current_candidate) {
    idx = hash_improved( h: current_candidate + (++collision_count) * 51679, bound);
}
```

hash candidate

double hash on collision until free slot is found

```
Memory usage: >=4294967296

 Performance counter stats for './ramanujan_rehashing 10000000000000':

      42831688622      cycles
      12776557723      instructions              #    0.30  insn per cycle
        136168400      branch-misses
        517202065      LLC-load-misses
        374485375      LLC-store-misses

     12.050636884 seconds time elapsed

     11.394585000 seconds user
      0.656148000 seconds sys
```

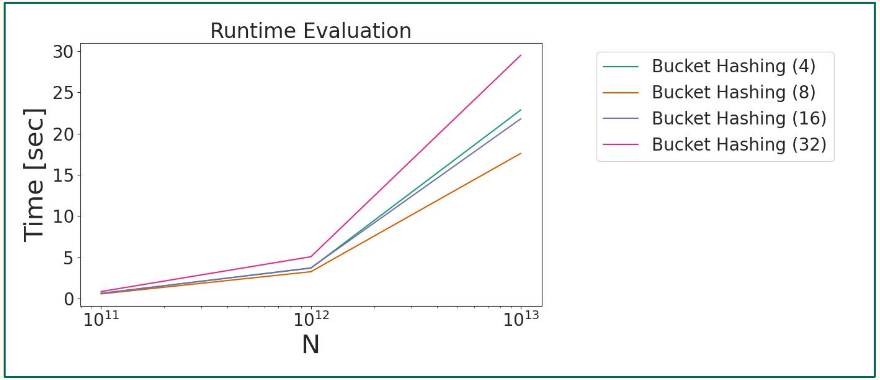
Bucket Hashing

- Table of fixed size
- Each table index contains a dynamic Vector (bucket)
- Hash candidate values into bucket: candidate % table_size
- Hash collision: add value at the end of bucket
- Caveat: bucket size should be rather small (e.g., lookup should be almost O(1))

```
template<typename ramanujan_candidate>
void cache_set<ramanujan_candidate>::init_cache_sections() {
    this->num_cache_buckets = std::ceil(this->ramanujan_candidates_bound / float(this->avg_bucket_size));

    this->caches.reserve( n: this->num_cache_buckets);
    for (size_t i = 0; i < this->caches.capacity(); ++i) {
        this->caches[i].reserve(this->avg_bucket_size);
    }
}
```

Bucket-Hashing

- Memory: reduction of over 50 %
- Runtime: almost twice as fast compared to the original hash-table based implementation

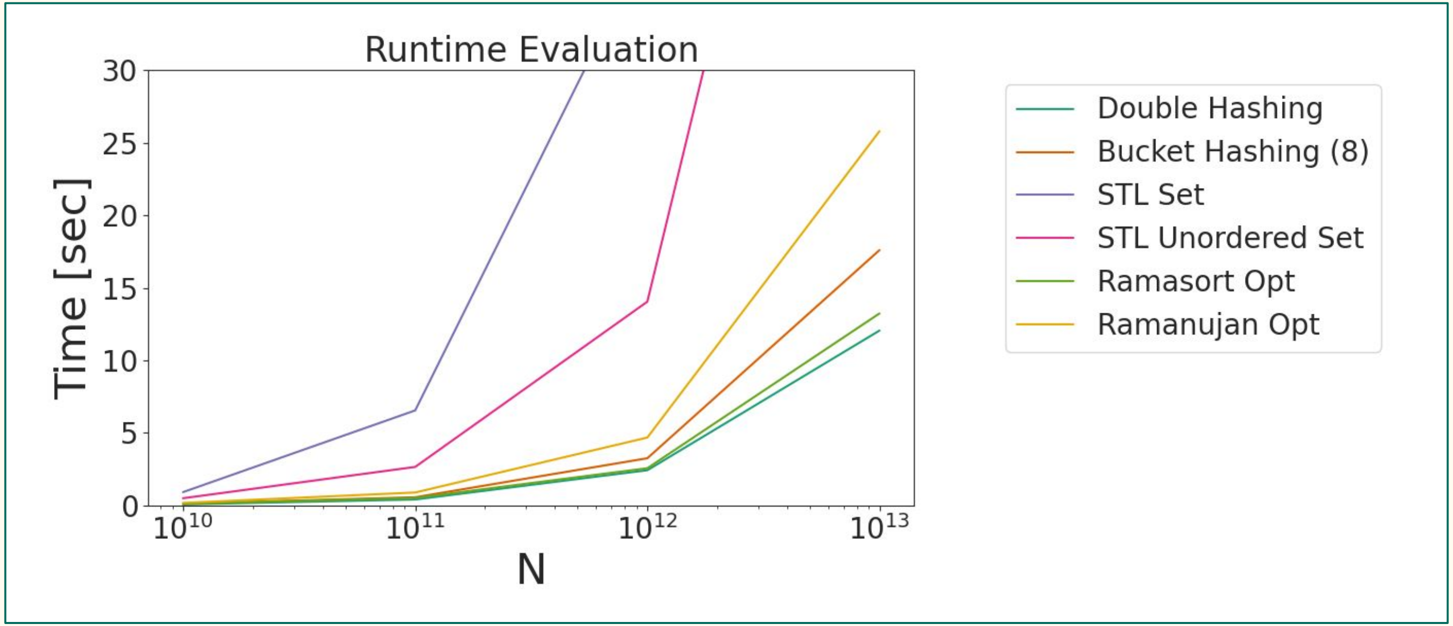


N=10^{13}, MEMORY=9000000

```
114359 Ramanujan numbers up to 10000000000000, checksum=355033384379411459
Memory usage> >=3278201296

 Performance counter stats for './bucket_hashing 10000000000000 8':

       56681785835      cycles
       32950058871      instructions              #    0.58  insn per cycle
         194498396      branch-misses
         670671929      LLC-load-misses
         147337037      LLC-store-misses

      17.599256810 seconds time elapsed

      15.934578000 seconds user
       1.652267000 seconds sys
```

```
114359 Ramanujan numbers up to 10000000000000, checksum=355033384379411459
Memory usage: >=7064785472

 Performance counter stats for './ramanujan 10000000000000':

       98620683446      cycles
       57886988091      instructions              #    0.59  insn per cycle
         130557718      branch-misses
         381598428      LLC-load-misses
         129502573      LLC-store-misses

      30.311566319 seconds time elapsed

      28.057505000 seconds user
       2.240120000 seconds sys
```

Runtime Comparison



Conclusion

- **Compiler:** optimizations are somewhat a black-box
- **C++ Standard library:** data structures are surprisingly inefficient (e.g., node base std::set implementation)
- **Speed vs. memory consumption**: quite difficult to jointly optimize both parameters.