Multiplication Horse-Race

Marius-Florin Cristian; Sidharth Singhal

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
wdx186@alumni.ku.dk; thx889@alumni.ku.dk

Abstract. This progress report contains the implementation and benchmarks for different multiplication algorithms (naive, naive-encoded and karatsuba).

Keywords. Multiplication, naive, encoded, Karatsuba, large numbers

2017, October 2017.

Table of contents

Naive and Naive-encoded Multiplication

Karatsuba Multiplication

Goals of experiment

Approach

Measures

Factors and Sampling Points

Generation of input (test data)

Results

Conclusion

Future Work

1. Naive and Naive-encoded Multiplication

The naive multiplication stores each digit in an *int*, and performs the school book $O(n^2)$ digit multiplications on $std :: vector < std ::< vector < int >>$ (it was easier to use a matrix apparently).
The naive-encoded algorithm, stores one base 10 digit in a nibble, and 4 digits in a $uint16\_t$. The file $mul\_optimisation1.cpp$ exposes all of the low level helper functions, that deal with "cells" (group of 4 digits) and implements, addition, substraction, and naive multiplication at cell level. There is a higher level method called mul_1 that does the same thing for two vectors of $std :: vector < uint16\_t >$ with respect to carry of each round. The result is built in a new $std :: vector < uint16\_t >$ with the operations being done in place (vector cells are passed as refference).

2. Karatsuba Multiplication

Karatsuba multiplication algorithm reduces the multiplication of two n-digit numbers to at most $n^{\log_2 3} \approx n^{1.585} n^{\log_2 3} \approx n^{1.585}$ single-digit multiplications in general (and exactly $n^{\log_2 3} n^{\log_2 3}$ when n is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. Using Karatsuba's algorithm one can find the result of multiplication of two large numbers x and y using three multiplications of smaller numbers, each with about half as many digits as x or y, plus some additions and digit shifts.

3. Goals of experiment

a) Comparison of algorithms: Benchmarking multiplication algorithms: naive and karatsuba.
b) Comparison of data structures: We discuss about two implementations of naive algorithm. They differ in the choice of data structure used to store a digit. The first one uses an integer to store a digit. This makes it easy to implement the algorithm. We try to improve the performance, both in terms of memory and speed, by using uint16_t to store 4 digits, i.e., two digits per byte.
c) Newsworthiness of experiment: Is it worth the extra effort of using uint16_t?

4. Approach

In our initial approach, we tried to work with the metaprogramming template provided, but we failed to cast data types from $Z < n >$ to $Z < m >$; Therefore, we had a change of direction and switched to an approach similar to python. The inputs are treated as strings and they are taken at run time. The first problem we encountered was efficiently storing the result in memory; we initially chose a $std :: vector < int >$ to build our result, where one

cell represents one digit. The major drawback is obvious as one digit can have values from 0 to 9; thus occupying at most 4 bits. Thus our results were capped at around 8000 digits output before running out of memory.
To overcome this problem, we chose to encode our one base 10 digit on a nibble; and to store 2 digits on an unsigned char. The problem is that *char* data type does not support multiplication. Thus we used $uint16\_t$ data structure to store 4 digits (4 nibbles). The drawback of this approach is, that on 4 bits we can represent values up to 15, and we only represent up to 9.

By using this encoding, we expected our result to store more digits. However, the output was still capped at 8000 digits. But, there is a significant improvement in time. Further tuning can be made; as on 8 bits we store values up to 99 instead of 256.

Naive-encoded performs better than naive multiplication. Therefore, we also compare naive-encoded with the Karatsuba algorithm. Karatsuba algorithm uses the naive approach and stores the one digit in a *char*. It performs better as it uses less multiplications.

5. Measures

1. CPU running time:
CPU running time is too small even for multiplication of large numbers. Because the system clock's granularity cannot be chosen arbitrarily, we get distorted results. Therefore, we do several runs with the same input data,i.e., we do several runs of multiplying same numbers and report the total time taken by them.
2. Space consumption:
We make use of as much memory as possible.
3. Representative operation counts:
The number of times helper functions(like, addition and subtraction) are called is proportional to the number of digits. For, a single call their time complexity don't make any significant change.
`Naive-encoded`: The real bottleneck is the number of multiplications of type $uint16\_t$ digits.

Scalability: Handling large input numbers. The number of digits in the result are capped at 8000. This is the extent of the algorithms for our machines.

6. Factors and Sampling Points

Number of digits is the only factor that we change in our experiment.

7. Generation of input numbers (test data)

We pass number of digits as a parameter to the function which computes a random number. We use `uniform_int_distribution` from std library to generate every digit independently. The random number engine is used from `mt199337` from std library.
Time taken to generate the numbers is not taken into consideration.

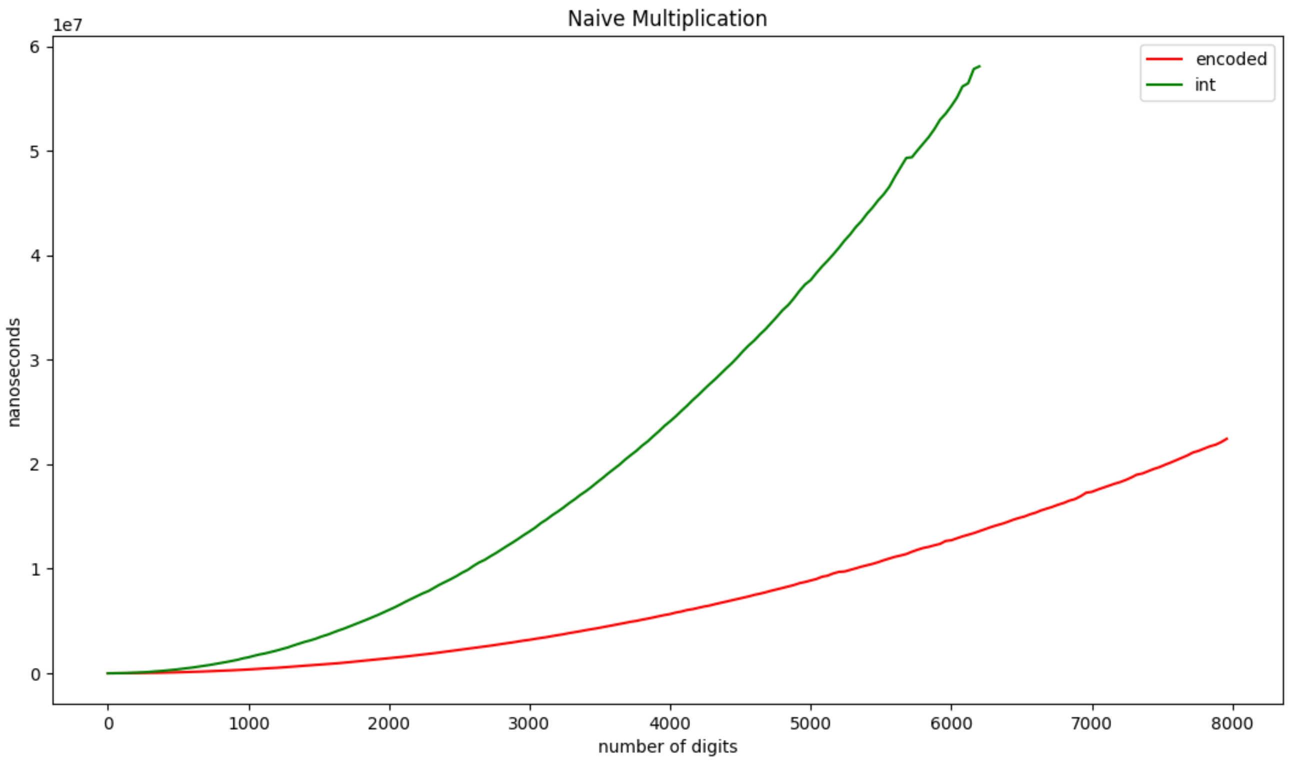
8. System Info

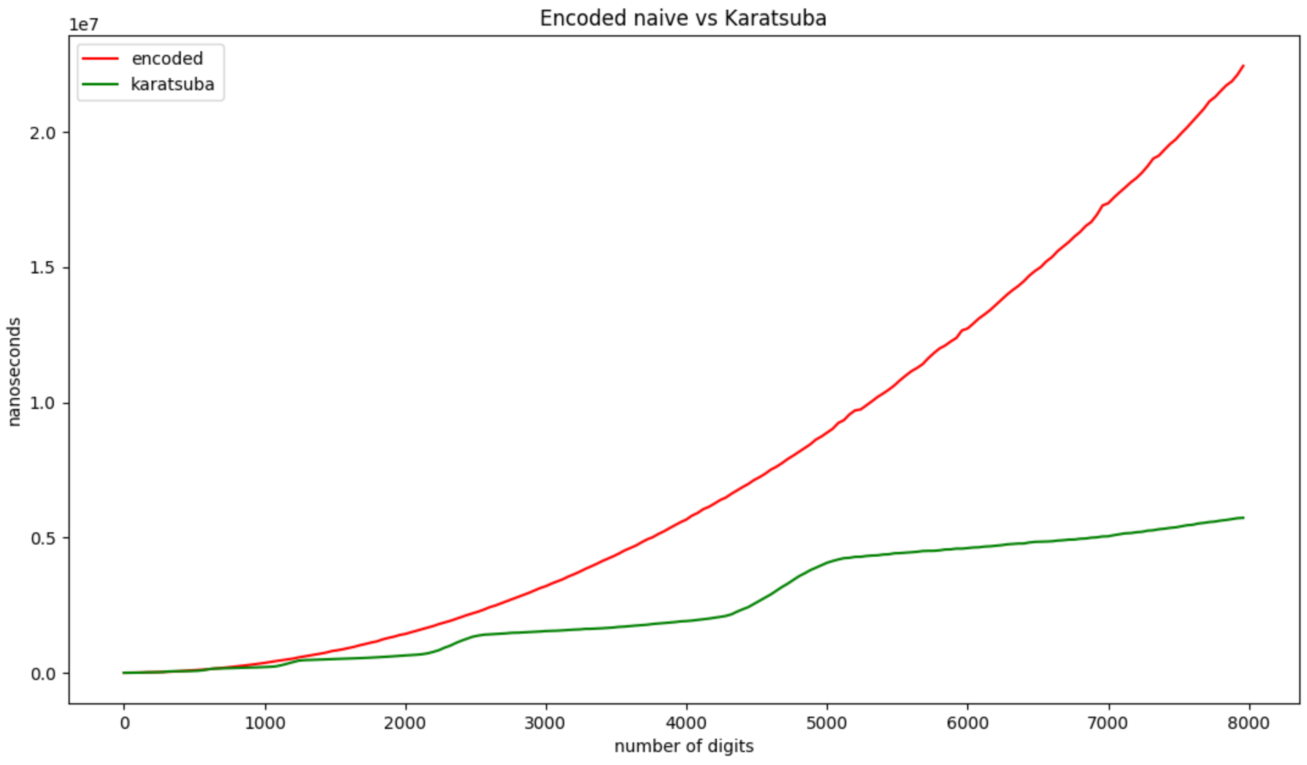
Listing 1. Machine 1

```
# dmidecode 3.0
Getting SMBIOS data from sysfs.
SMBIOS 2.7 present.

Handle 0x0004, DMI type 4, 42 bytes
Processor Information
  Socket Designation: CPU 1
  Type: Central Processor
  Family: Atom
  Version: Intel(R) Pentium(R) CPU  N3540  @ 2.16GHz
  External Clock: 83 MHz
  Max Speed: 2656 MHz
  L1 Cache Handle: 0x0007
  L2 Cache Handle: 0x0008
  L3 Cache Handle: Not Provided
  Core Count: 4
  Core Enabled: 4
  Thread Count: 1
Linux 4.4.0-97-generic
#120-Ubuntu 16.04 LTS x86_64 GNU/Linux
```

9. Results


Naive Multiplication
1e7
6
5
4
3
2
1
0
nanoseconds
0
1000
2000
3000
4000
5000
6000
7000
8000
number of digits
encoded
int



10. Conclusion

As expected Karatsuba is performing better; But we are still capped at the number of digits. The encoded version does not help in increasing the number of digits, but improves the time complexity.

11. Future Work

1. Karatsuba: Implementation of Karatsuba using the encoded format. 2. Instead of having cell size of $uint16\_t$ we can use *long long*. Will this result in an improvement on the amount of digits we can use?
3. Why not have a lookup table for cell multiplication values, and do it in O(1); this table will take up memory space, it might make it faster for instances up to a number of digits, but the trade off will be in the length of the result.
4. x^z is computed much faster than $x_1x_2...x_z$ (exponentiation by squaring); having a lookup table to break the cell in power of primes, will this yield in better performance?