Karatsuba Long Multiplication Algorithm

Andrea Simonassi
11 Jan 2023CPOL9 min read
A simple yet powerful multiplication algorithm

As the number of digits grow, elementary school multiplication gets surpassed by better scaling algorithms, Karatsuba algorithm, the elder of this kind of algorithms.

- **Download source code - 73.9 KB**

Introduction

In previous article series, I wrote about “elementary school” multiplication algorithm and stated its importance.

As the number of digits grows, elementary school multiplication gets surpassed by better scaling algorithms.

The first algorithm discovered in early 1960s by Soviet mathematician, Anatoly Karatsuba, to be “faster” than elementary school algorithm (ES from now on) is Karatsuba algorithm.

ES Algorithm Recursive

Karatsuba algorithm can be studied departing from a variant of the standard ES algorithm made recursive.

Example:

A	9	3	2	8	2	2	5	x
B		3	9	9	1	0	3	=

First of all, split A and B into two parts, at about half the length of A (longest operand):

A	9	3	2	8	2	2	5	x
B		3	9	9	1	0	3	=

Store split parts into variables: ***a***=932; ***b***=8,225; ***c***=39; ***d***=9,103

A = 932*e*+8,225

B = 39*e*+9,103

Where *e* = 10,000 (I just need symbol *e* to remember we need shifts later).

Now since A=***a****e*+***b*** and B=***c****e*+***d*** then AxB =(***a****e*+***b***)(***c****e*+***d***) = ***ac****e*^{2}+ ***ad****e* + ***bc****e* + ***bd*** = ***ac****e*^{2}+ (***ad***+***bc***)*e* + ***bd***

Therefore, ES algorithm recursive, requires 4 long multiplications ***ac***, ***ad***, ***bc***, ***bd*** (to be computed by recursion):

ac = 932x39=36,348

ad = 932x9,103= 8,483,996

bc = 8,225x39=320,775

bd = 8,225x9,103= 74,872,175

after 4 long multiplications are computed reorder and sum partial results:

carry	0	1	1	1	1	2	2	1					
ac =	3	6	3	4	8	0	0	0	0	0	0	0	+
ad =			8	4	8	3	9	9	6	0	0	0	+
bc =				3	2	0	7	7	5	0	0	0	
bd =						7	4	8	7	2	1	7	5	=
	3	7	2	2	9	2	2	5	8	2	1	7	5	

Karatsuba Algorithm

The genius of Karatsuba was to note that, instead of running 2 long multiplications, ***ad*** and ***bc***, we can save one multiplication for some more sums and subtractions.

We had expression:

AxB= $\boldsymbol{ac}e^2 + (\boldsymbol{ad}+\boldsymbol{bc})e + \boldsymbol{bd}$

we may replace the grayed part with the following:

$\boldsymbol{ac}e^2 + ((\boldsymbol{a}+\boldsymbol{b})(\boldsymbol{c}+\boldsymbol{d})-\boldsymbol{ac}-\boldsymbol{bd})e + \boldsymbol{bd}$

"Wait a minute, that is 5 multiplications instead of 4!": yes; but we already have ***ac*** and ***bd***, only 3 long multiplications are required in the end:

Verify that grayed part are equals if you don't trust: (***a***+***b***)(***c***+***d***)-***ac***-***bd*** = ac+***ad***+***bc***+bd-ac-bd= ***ad***+***bc***+bd-bd+ac-ac = ***ad***+***bc***

Algorithm steps are:

1. Split number A and B and store into temp vars as per ES recursive.
   a=932; ***b***=8,225; ***c***=39; ***d***=9,103
2. Compute recursively and store into other temp vars *until multiplication is hardware doable*.
   ac = 932x39=36,348
   bd = 8,225x9,103= 74,872,175
3. Now sum ***a+b*** and ***c+d***, store in temp variables
   a_plus_b=***a+b*** = 932+8225=9157
   c_plus_d=***c+d*** = 39+9103=9142
4. Multiply ***a_plus_b*** x ***c_plus_d***, by recursion, store in temp variables
   temp=***a_plus_b*** x ***c_plus_d***= 9157x9142= 83,713,294
5. Now subtract ***ac*** from ***temp***
   temp = ***temp*** – ***ac*** = 83,713,294 - 36,348 = 83,676,946
6. Subtract ***bd*** from ***temp***
   temp = ***temp*** – ***bd*** = 83,676,946 - 74,872,175 = 8,804,771
7. Now do shifts and sum:

carry	0	1	1	0	1	1	1	0						
ac =	3	6	3	4	8	0	0	0	0	0	0	0	0	+
temp =			8	8	0	4	7	7	1	0	0	0	0	+
bd =						7	4	8	7	2	1	7	5	=
	3	7	2	2	9	2	2	5	8	2	1	7	5	

We did 3 long multiplication on 4 digits numbers, 4 long sums, 2 subtractions, we needed temp space (allocation).

Let see it on C language *(comments, asserts, memory checks stripped: find original version on attached source code)*:

C++
Shrink ▲

```
/*
KaratsubaRecursive: beware:
    simpleSum and simpleSub are required to allow operation in place
    (i.e the result of A - B stored in A)
*/

numsize_t KaratsubaRecursive(
    reg_t* A /* first operand */,
    numsize_t m /* size of first operand */,
    reg_t* B /* second operand */,
    numsize_t n /* size of second operand */,
    reg_t* R /* pointer to pre-allocated area where result will be stored */,
    operation simpleMul, /* pointer to primitive multiplication function (ES) */
    operation simpleSum, /* pointer to a long sum function */
    operation simpleSub, /* pointer to a long subtraction function */
    numsize_t simpleMulThreshold /* if operand short than this, use primitive
long
                                    multiplication and stop recursion
                                    this is useful to create hybrid algorithm as
                                    discussed later in article
                                   */)
{

    numsize_t min = m;
    numsize_t max = n;
    if (m > n) {
        min = n;
        max = m;
    }

    if (min == 0)
        return 0;

    if (min <= simpleMulThreshold)
        return simpleMul(A, m, B, n, R);

    numsize_t split_point = (max + 1) >> 1;

    if (split_point > min)
        split_point = min;

    reg_t* a = A + split_point;
```

```
    reg_t* b = A;
    reg_t* c = B + split_point;
    reg_t* d = B;

    numsize_t len_a = m - split_point,
        len_c = n - split_point;

    numsize_t allocsize = ((max - split_point + 2)
        + (max - split_point + 2) +
        ((max - split_point + 2) << 1)
        )
        * sizeof(reg_t);

    reg_t* a_plus_b;

    a_plus_b = (reg_t*)ALLOC(allocsize);

    reg_t* c_plus_d = a_plus_b + (max - split_point + 2) ;
    reg_t* z1 = c_plus_d + (max - split_point + 2);

    numsize_t len_z0 = KaratsubaRecursive(a, len_a, c, len_c, R + (split_point <<
1),
                        simpleMul, simpleSum, simpleSub, simpleMulThreshold);
    numsize_t len_z2 = KaratsubaRecursive(b, split_point, d, split_point, R,
                        simpleMul, simpleSum, simpleSub, simpleMulThreshold);

    numsize_t len_a_plus_b = simpleSum(a, len_a, b, split_point, a_plus_b);
    numsize_t len_c_plus_d = simpleSum(c, len_c, d, split_point, c_plus_d);

    numsize_t z1_len = KaratsubaRecursive(a_plus_b, len_a_plus_b, c_plus_d,
           len_c_plus_d, z1, simpleMul, simpleSum, simpleSub,
simpleMulThreshold);

    z1_len = simpleSub(z1, z1_len, R, len_z2, z1);
    z1_len = simpleSub(z1, z1_len, R + (split_point << 1), len_z0, z1);
    z1_len = simpleSum(R + split_point, len_z0 + split_point,
                        z1, z1_len, R+split_point);

    DEALLOC(a_plus_b);

    return z1_len+split_point;
}
```

Karatsuba Cost

Note for the reader: The article ends here, the following is bonus track.

tl;dr: *why ES outperforms Karatsuba on small numbers? On this chapter, we will compute complexity and conclude that hybrid Karatsuba/ES algorithm is better than both.*

Asymptotic complexity for ES is $O(N^2)$, Karatsuba around $O(N^{1.58})$.

Philosophy about Big O Notation

While studying Karatsuba algorithm, I re-discovered something I knew but underestimate; big O measures algorithm performance as N grows toward infinity, not the actual speed of the algorithm.

All N are equals, but some N are more equals than others.

In many practical use cases constants and smaller factors, *which gets negligible as N goes toward infinity* are not so negligible.

Karatsuba costs $N^{1.58}$ vs N^2, means Karatsuba better than ES as N goes toward infinity.

Why is my Karatsuba implementation slower than ES algorithm then? I assumed that for 500-word sized operands, Karatsuba should be faster, but measured times shown I underestimated lower exponent costs (*and using malloc to allocate very small chunks of memory which killed performance but that's not the point*).

Karatsuba does less steps to complete its duties but each step costs more than ES.

That means Karatsuba begins to be faster after N surpasses some threshold that was higher than I expected.

Compute Karatsuba Algorithm Complexity

We start computing the cost of a single recursive step, ignore nested calls for a second:

- 6 long sums of size N, their total cost is 6Ns, where s the cost of a CPU single sum
- 3 recursive calls of size N/2
- some constant cost K (*stack preparation, allocation, boilerplate)*

Therefore, the cost of a single recursive step having operand length N is:

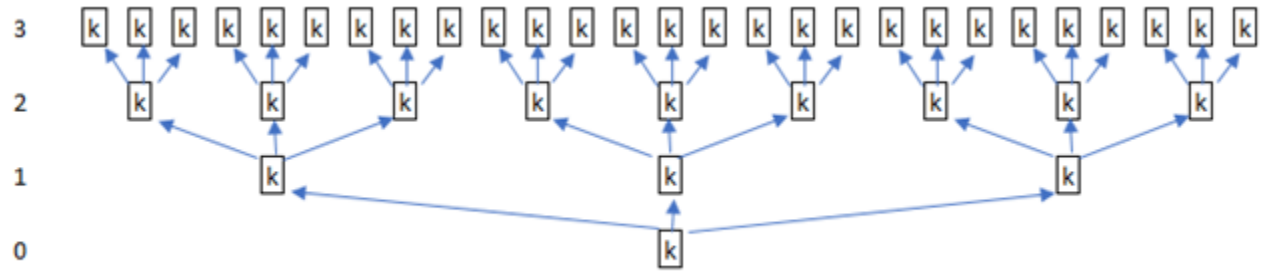
$$Cost(N) = 6Ns + 3Cost(N/2) + K$$

Try to achieve some non-recursive formula

The above formula does not help since it is recursive, we must sum costs for each recursive call.

Start with an example, let say constant k cost is 100, that a single sum costs 1, that a single mul costs 4, N is 8.

See the below image, our recursion tree will have depth log2(8) that is 3 levels + root level (level 0):



at level 3: 27 calls, operand size 1: paying 27*100 + 27*(6 sums of size 1) + 27*4(multiplication cost) = 27*(100+6+4)=2970
at level 2: 9 calls, operand size 2: paying 9*100 + 9*(6 * 2 sums) = 1008
at level 1: 3 calls, operand size 4: paying 3*100 + 3*(6 * 4) = 372
at level 0: 1 call, operand size 8: paying 100 + (6 * 8) = 148

total cost = 2970+1008+372+148=4498 (unit of measure are imaginary clock cycles, nothing real)

Generalizing

At level i:

$ 3^i k + 6 \frac{N}{2^i} 3^i $

But we have to do the sum of the above for each recursive level (and remember to add cost of the multiplications at leaf level, let call the cost of a cpu MUL \(\mu\)).
Since we have log2(N) levels, to simplify notation let \(log_2(N) = M \) therefore

$ 3^M\mu + \sum_{i=0}^{M} 3^i k + 6 \frac{N}{2^i} 3^i $

equals to:

$ 3^M\mu + k\sum_{i=0}^{M} 3^i + 6N\sum_{i=0}^{M} \left(\frac{3}{2} \right)^i $

and since \(3^i \) and \(\left(\frac{3}{2} \right)^i\) are geometric progressions, we know how to compute the sum of their first M values:

$ \sum_{i=0}^{\\M} 3^i = \frac{3^{M+1}-1}{3-1}; \sum_{i=0}^{\\M} \left(\frac{3}{2}\right)^i = \frac{1.5^{M+1}-1}{1.5-1} $

simplify a bit:

$ \sum_{i=0}^{\\M} 3^i = \frac{3^{M+1}-1}{2}; \sum_{i=0}^{\\M} \left(\frac{3}{2}\right)^i = \frac{1.5^{M+1}-1}{0.5} $

Now we can rewrite the cost formula as:

$ k \frac{3^{M+1}-1}{2}+ 6N \frac{1.5^{M+1}-1}{0.5} + 3^M\mu $

simplify a bit:

$ \frac{k}{2} (3^{M+1}-1) + 12N (1.5^{M+1}-1) + 3^M\mu $

plugin some numbers to test that the formula works indeed:

$k = 100 \\ N=8\\ \mu=4\\ M=log_2(N) = 3\\ . \\ \frac{k}{2} (3^{M+1}-1) + 12N (1.5^{M+1}-1) + 3^M\mu $ replacing variables $ \frac{100}{2} \cdot(3^{4}-1) + 12 \cdot 8\cdot (1.5^{4}-1) + 3^3\cdot4\\ 50\cdot80 + 12 \cdot 8 \cdot(\frac{3^4}{2^4}-1) + 3^3\cdot4\\ 4000 + 96\cdot (\frac{3^4}{2^4}-1) + 108\\ 4108 + 96\cdot (\frac{81}{16}-\frac{16}{16}) \\ 4108 + 6\cdot (81-16) \\ 4108 + 6\cdot 65 \\ 4108 + 390 \\ 4498 $

Same numbers as doing the manual process.

Remember, we must decide constant k and μ values depending on tests, but we have enough information to simulate when Karatsuba is convenient to use versus ES.

Compare Karatsuba to ES Cost

We know how to define Karatsuba cost function if we know k and μ, I will use some reasonable number (unit of measure clock cycles).

let $k = 200 \\ \mu=4\\ M=log_2(N) \\ . \\ $define Karatsuba cost function as $ 100 (3^{M+1}-1) + 12N (1.5^{M+1}-1) + 3^M\cdot4 $

ES cost function (this is derived from actual experiments):

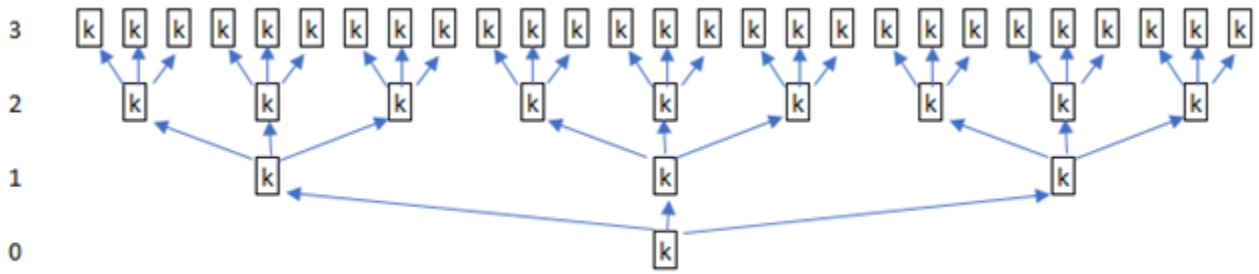
$N^2\cdot16 $

k	N	μ	M	karatsuba cost	ES Cost
200	200	4	7,643856	1.426.072	**640.000**
200	400	4	8,643856	4.280.816	**2.560.000**
200	600	4	9,228819	8.142.021	**5.760.000**
200	800	4	9,643856	12.847.448	**10.240.000**
200	1000	4	9,965784	18.300.220	**16.000.000**
200	1200	4	10,22882	24.433.463	**23.040.000**
200	1400	4	10,45121	**31.197.201**	31.360.000
200	1600	4	10,64386	**38.552.143**	40.960.000
200	1800	4	10,81378	**46.466.285**	51.840.000
200	2000	4	10,96578	**54.912.861**	64.000.000

Karatsuba starts to be faster somewhere after N > 1200. Now it is clear why my first tests were showing Karatsuba slower than ES, because I needed to use longer input size (*and needed to reduce K costs... I was allocating like 4 ints using malloc at leaf levels*).

Constant costs per call K makes the algorithm slower than ES on small numbers.

Visualize again, where you pay recursion cost the most, using visual intuition: do you see at which depth you are going to pay more the constant costs?



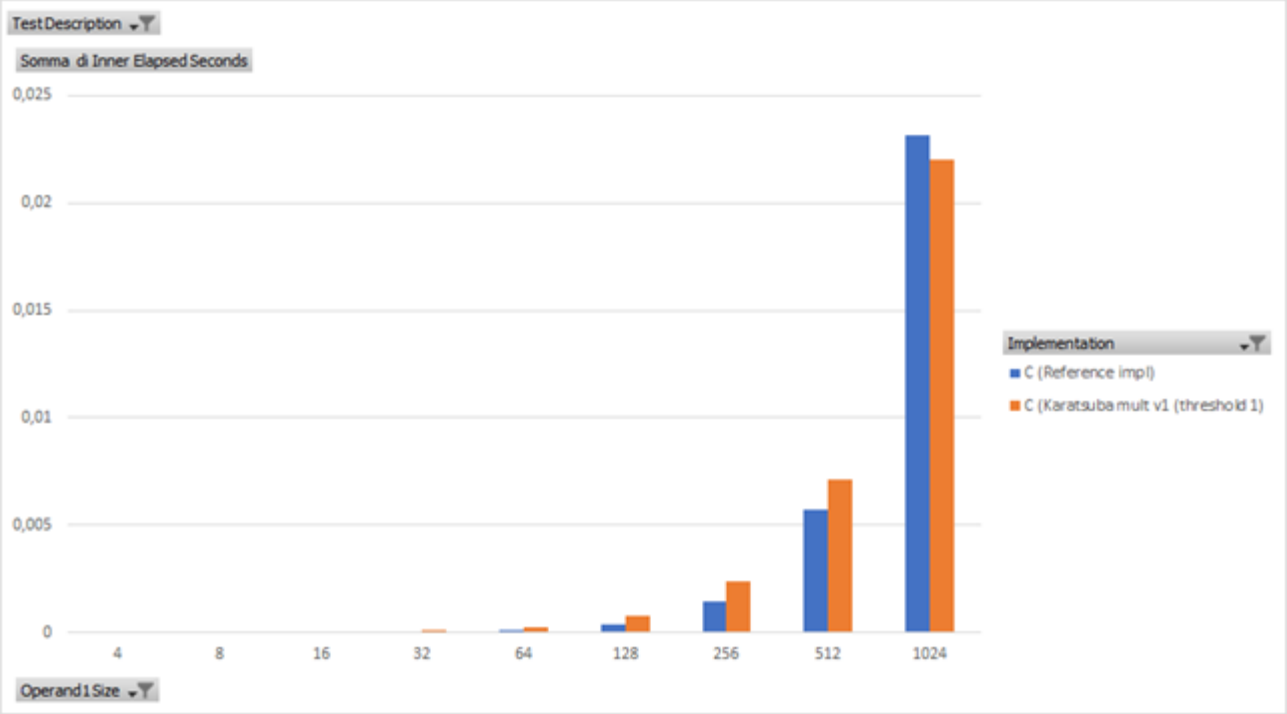
As you can see, about 1/2 of the fixed costs are at leaf level, regardless of tree size, by eliminating the leaf level, we get rid of 1/2 of total K costs.

So... we could modify algorithm and when at some point on recursion tree, if size of operand shorter than X, use elementary school algorithm and stop recursion.

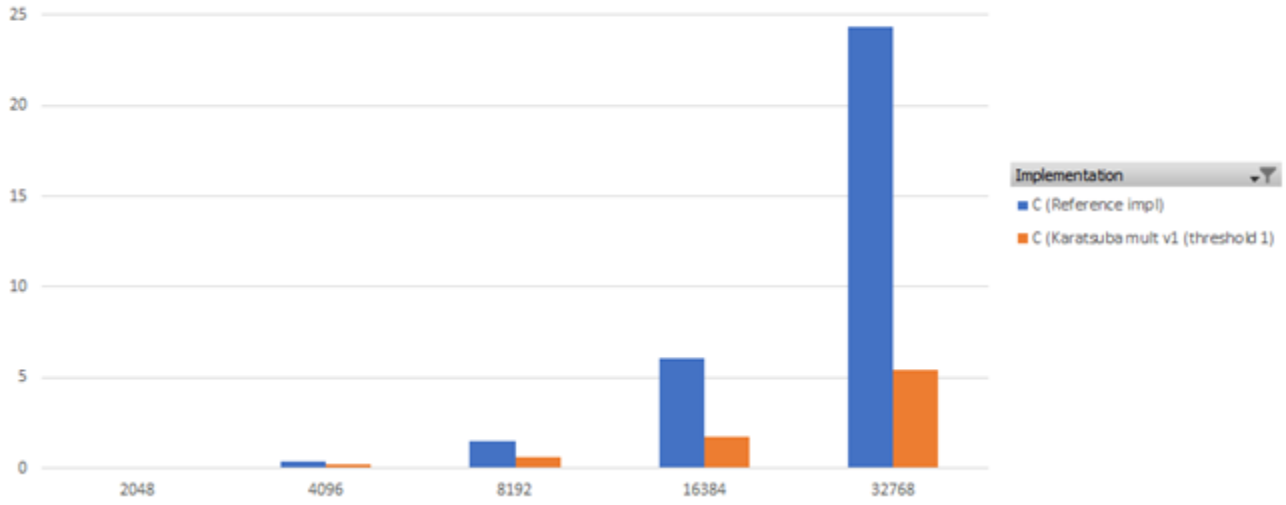
Speed Comparison Between the Actual Algorithms

Karatsuba vs ES

Real data says Karatsuba constant time is way less than I estimated, I was trying to estimate the average cost of malloc, but while implementing, since I used the C language, I dynamically allocated memory on stack by using the "`alloca`" function, using malloc only when you need more than 4KB of memory (but I guess I can trigger malloc near 1KB instead), if you just need small amounts of memory like 16 bytes calling malloc would be overkill, so we have a cost for allocation way lower than I expected. It turns out that Karatsuba starts to win against ES between 512 and 1024 words instead of after 1200 as per the simulated cost.

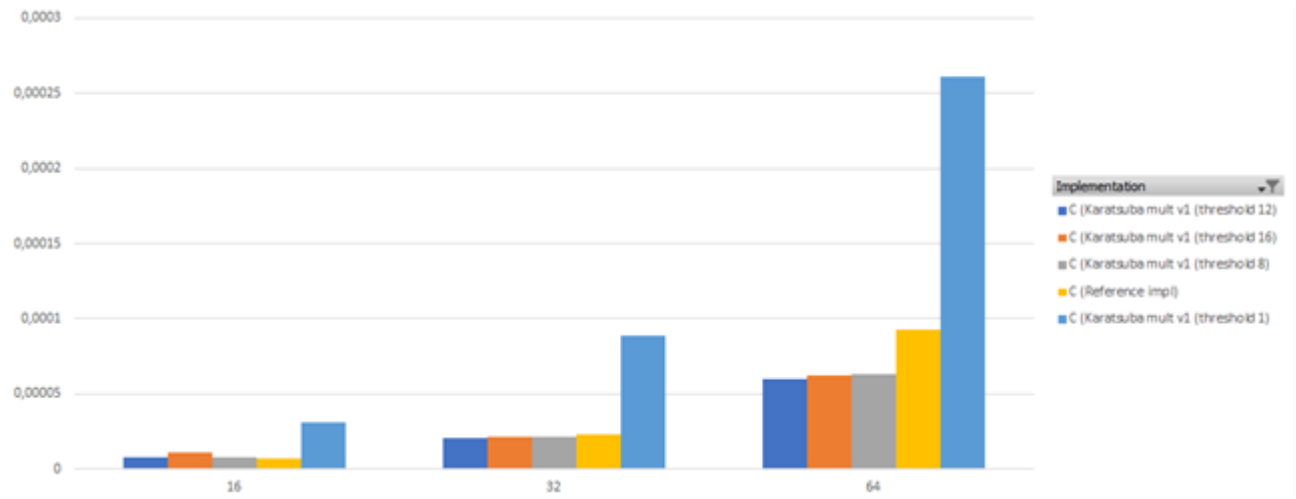


And after that 1024 words point, it gets better and better, at 32Kwords being 5x faster.

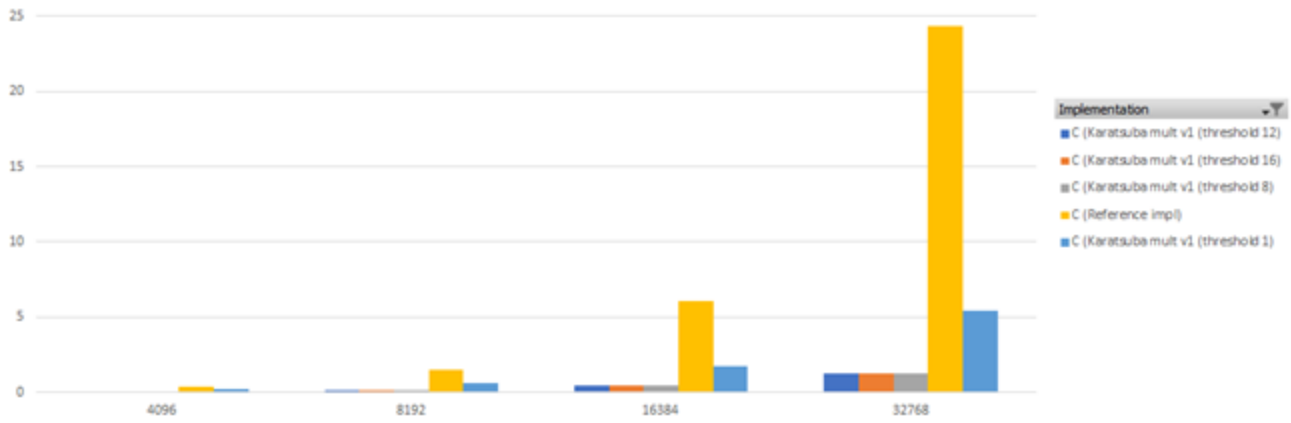


Karatsuba vs Karatsuba/ES Hybrid

Now we apply a threshold > 1 to Karatsuba algorithm, so that during recursion, operand size is less than or equals to threshold switches to ES algorithm.



As you can see after N being bigger than 16 karatsuba(threshold 12) starts being better than ES, cannot even compare ES to Karatsuba with threshold for bigger N.



We are 30 times faster at 32K Words, also note that ES used in conjunction with Karatsuba after some threshold makes it run 4 times faster than pure Karatsuba on 32KWords.

After having discussed the fact that Karatsuba (+Elementary School for a threshold of N=12) is good for numbers bigger than 16 WORDS, let me anticipate that this is not the end, there are better algorithms than Karatsuba for bigger numbers... coming sooner or later.

Thank you for following me until here.

History

- 12th January, 2023: Initial version

License

Written By

Andrea Simonassi

Software Developer

Italy

This member has not yet provided a Biography. Assume it's interesting and varied, and probably something to do with programming.