Karatsuba and Toom-Cook Multiplication Optimizations
By Shrey Srivastava

Outline:

1. Goals + Motivation
2. Explanation of the optimizations themselves
3. Mini Benchmarking Report to highlight speed benefits
4. Challenges
5. Commands to run code

Goals + Motivation:

Having enjoyed implementing the various numerical algorithms in SSH, I wanted to code an enhancement for one of the base functions.
After chatting with Ilana, it was decided that I could do either 2 multiplication optimizations or 1 division optimization. I chose to do 2 multiplication optimizations, namely Karatsuba and Toom-Cook.

Goals (taken verbatim from proposal (emailed to Ilana)):

1. The implementation of these 2 methods
2. Tests that cover edge cases and stress tests to ensure the code works
3. A mini report that highlights the speed benefits that the optimizations provide

In order to accomplish the above, I also implemented a custom method that does 1 particular division, a divide by 3. I will explain why this was necessary in the challenges section. I also commented all the methods, as well as provide code for my tests and 2 experiments. Commentary is done for justification of tests.

As my extension were primarily these optimizations, the best way to run them would be to simply run the test code I provide. You can also run the experiments which provide you with the time it takes for a given algorithm to perform the multiplication and customize them with the length of the hexadecimal strings representing the two numbers you would like to multiply!

Explanation of the optimizations

1. Karatsuba
    a. Karatsuba is a divide and conquer algorithm that splits the numbers you want to multiply together into "high" and "low" parts, effectively in half. Let's call these parts A1 and A0 for one number. What we want to do is calculate these 3 special values, Z0, Z1, and Z2. These values are equal to the following: Z0 = A0 * B0, Z2 = A1 * B1, and Z1 = (A0 + A1) * (B0 + B1) - Z0 - Z2. Let's examine why we need these values.

b. If I was to multiply two numbers A and B together such that A = A1*(2^(n/2)) + A0 and B = B1*(2^(n/2)) + B0, where for simplicity n is an even number of bits in the base, then A*B = 2^n*(A1*B1) + 2^(n/2)*(A1*B0 + A0*B1) + (A0*B0). For our representation of bigints, if we were dealing with a 1 limb multiplication, then A1/B1 would represent the top 32 bits each, and A0/B0 would represent the bottom 32 bits each. This expression has a total of 4 multiplications (disregarding the multiplication of 2^n, as it can be handled by bit/limb shifts). Note that Z2 and Z0 are the coefficients of the largest and smallest term respectively. Unfortunately, this split leads to a recurrence relation of T(n) = 4T(n/2) + O(n), which is still O(n^2).

c. What Karatsuba does is finds a trick to turn the middle term's coefficient from 2 multiplications to only 1. This is trick is the expression for Z1. (A1*B0 + A0*B1) = (A0 + A1) * (B0 + B1) - Z0 - Z2. Since we got rid of 1 multiplication, the recurrence now becomes T(n) = 3T(n/2) + O(n), which is approximately O(n^1.59).

d. After we have Z0, Z1, and Z2, all that's left is to shift them to their proper magnitudes and add them together. Since this step only requires bit/limb shifts, it doesn't incur the cost of any multiplicate step. The cost saved by not doing 1 multiplication earlier matters a lot when we get into bigints with several hundred limbs.

e. The recursive nature of the algorithm occurs during the calculation of Z0, Z1, and Z2 themselves, with calls such as big_karatsuba (&Z0, &A0, &A1) splitting up A0 and A1 even further if they themselves are too "big", past a certain threshold.

2. Toom-Cook

a. Toom-Cook is a generalization of Karatsuba, and is much more complicated. Instead of splitting the number into halves, Toom-Cook splits it into thirds. Toom-Cook represents every number as a polynomial and evaluates that number at some particular points. The most used points are 0, 1, -1, 2, and "inf". For example, let's say that we have two bigints A and B. Let's let each A and B for example's sake have 3 limbs, each limb being equal to 1. Then A and B are split into A0, A1, and A2, like Karatsuba, which all contain 1 each. Then treating A like A2*x^2 + A1*x + A0, where x is the size of the base (where x for bigints would be 2^64), A and B are evaluated at the aforementioned points. At x = "Inf", the result is approximated by A2. This provides 5 new bigints that store the results of these expressions for each number.

Similar to Karatsuba, we then recursively multiply each respective subcomponent of the two numbers (A0' * B0', A1' * B1', etc) through calls to big-toom-cook, and store the product in 5 final values (R0, R1, R_1 (-1), R2, R_inf). There is slight difference here to Karatsuba, Karatsuba split the numbers recursively before any form of evaluation, here we evaluated the numbers before, to allow for the last step: interpolation. This is the complicated part. Through some algebraic manipulation, you are able to establish 5 equations given the 5 separate points that allow you to solve for the 5 coefficients of the product. This is possible because each number was represented as A2*x^2 + A1*x + A0, the product must have 5 separate terms: 1 for x^4, 1 for x^3, 1 for x^2, 1 for x, and a constant. In here lies the optimization, due to the earlier evaluations of the R values, we can write (R0, R1, R_1 (-1), R2, R_inf) as some combination of a, b, c, d, e (the coefficients of the final product), and then solve the system of linear equations to solve for the final coefficients! Once we have these coefficients, then similar to Karatsuba, we can use limb_shifts to combine them the separate terms properly, and we have our product. This optimization afford Toom-Cook a time complexity of O(1.47)!

Mini Benchmarking Report

After having implemented the above two optimizations, I wanted to design two separate experiments to explore different facets of large number multiplication.

In the first experiment, I wanted to directly test how the three different strategies (Regular, Karatsuba, Toom-Cook) perform on the same two numbers (A and B) at vastly different sizes. Here size was determined by the number of hexadecimal characters that represented the large number. I formatted the experiment to take in bounds, low and high, such that if someone passed in 50 and 100, then A and B would be random hexadecimal numbers represented by some number between 50 and 100 hexadecimal characters long. I also wanted to test on what would happen if these two numbers did not share the same number of limbs, how each algorithm would fare in terms of time. This was motivated by the fact that both Karatsuba and Toom-Cook require some form of "padding" to ensure that their split subcomponents are equal in size. As I was aware of the potential variation even within 1 "size" (a hexadecimal string of size 5 could be "fffff" or "11111" which is either 1048575 or 69905) which would grow exponentially as the size increased, I had the experiment time 100 of these multiplications within the range provided to get a more wholistic performance of the 3 algorithms.

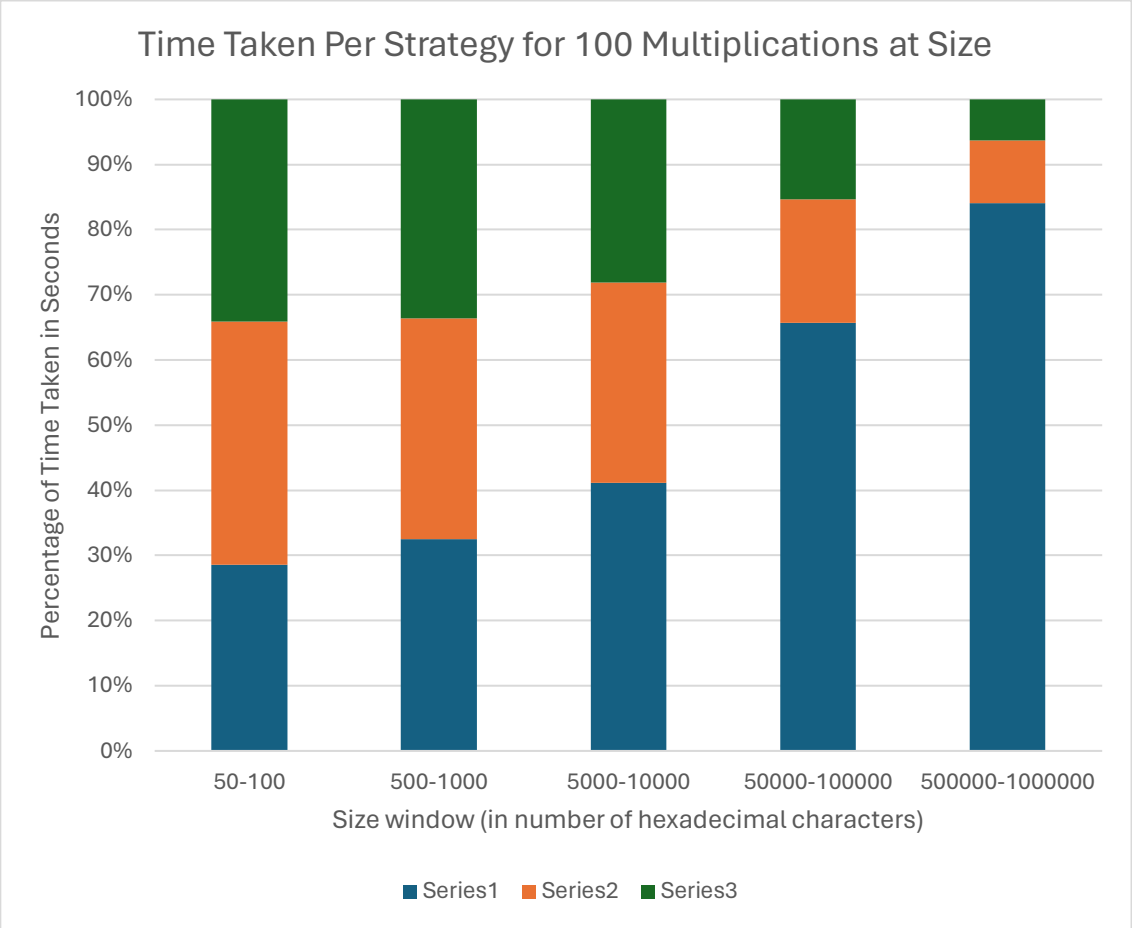
Additionally, for each multiplication, I would write out to 3 separate buffers the results of the 3 strategies, using this experiment as a large stress test at various size profiles to ensure that my algorithms were correct. As such, in 1 experiment, I was able to test and explore many different things: performance based on size, different number of limbs, accuracy, etc.

The results were as follows:

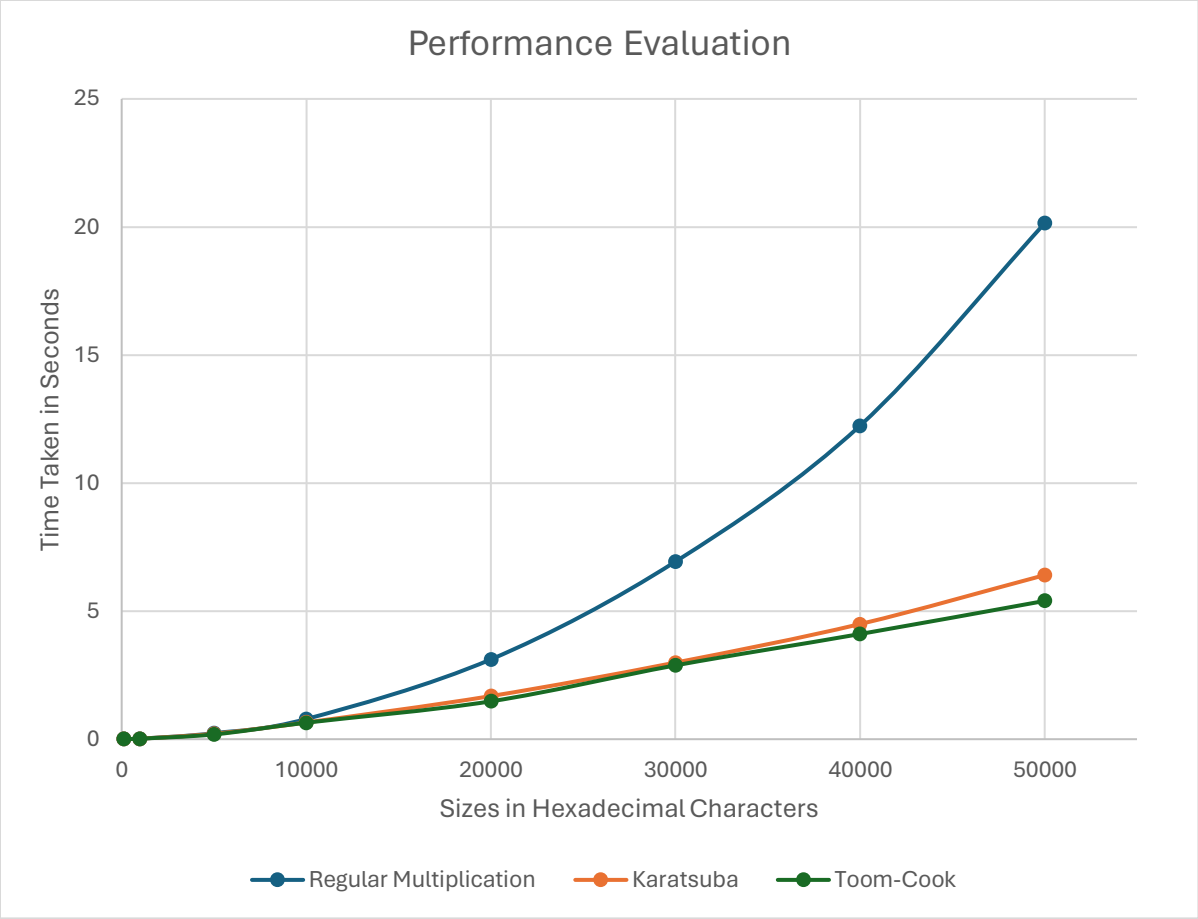
Sizes	Regular	Karatsuba	Toom-Cook
50-100	0.000083	0.000108	0.000099
500-1000	0.003779	0.003948	0.00391
5000-10000	0.279035	0.208699	0.190316
50000-100000	21.376942	6.187661	4.990822
*500000-1000000	2007.2515	231.13084	149.6651

As expected, at the lower size magnitudes, 50-100 and 500-1000 hexadecimal character numbers, there is no perceivable difference between the three strategies, with the grade school multiplication being faster due not needing to do any splitting of the bigints like Karatsuba and Toom-Cook. This influenced the decision of the final number of limbs that I restricted both Karatsuba and Toom-Cook's base case switch to big_mul to (explained at the end of this section). However, as the size of the number grows, we can see that between 5000-10000 characters, Karatsuba and Toom-Cook begin to pull ahead, still relatively tied to each other. However, for the larger sizes, it becomes evident that even only to do 100 calculations, Toom-Cook is a whole second faster than Karatsuba at the 50000-100000 profile. Note the asterisk next to the last size, at this point, we were dealing with excessively LARGE numbers, at minimum tens of thousands of limbs. As such, I was unable to get a ready for 100 calculations directly, but I got a reading for 10 calculations, and chose to multiply those numbers by 10 to get some sense of proportion for that last size class. The regular approach would have taken an estimated 30 minutes to do 100 calculations! This is in sharp contrast to Karatsuba and Toom-Cook that took somewhere between 3.85 and 2.48 minutes respectively – a massive improvement!

The following bar-chart allows for some sense of comparison to understand that at smaller sizes, the three strategies take about the same proportion of time, but as the size increases, we can see the grade-school multiplication rising exponentially, while both Karatsuba and Toom-Cook take proportionally less time. Note this doesn't mean that Karatsuba and Toom-Cook somehow did larger multiplications faster, just that they took less time in comparison.



The second experiment was much more straightforward when compared to the first one. For the previous experiment, all 3 approaches were multiplying the same 2 random numbers together at different limb sizes along with different size profiles. However, one downside for comparison was that due to the ranges increasing in such an exponential manner, it was difficult to plot a line-plot that would help visibly notice the trend in amount of time took to calculate. So, for the second experiment, I chose to allow the user to pass in 1 size parameter, ensured that both bigints A and B were generated from random hexadecimal strings from that length. This allowed me to compare to see how Karatsuba and Toom-Cook perform when they don't need to pad or do other less unnecessary calculations that could eat at their performance. After learning from the first experiment, I also ensure to keep the sizes themselves a bit smaller than the extreme of 1000000, to ensure I could get multiple size profiles evaluated. As before, the total time for 100 calculations at each size profile was calculated.



What I noticed was that Karatsuba and Toom-Cook separated at roughly 40,000 hexadecimal character input, and they diverged from regular multiplication at an order of 10,000. In this graph it becomes exceedingly apparent why Karatsuba's O(1.59) and Toom-Cook's O(1.47) become so valuable for larger numbers. The raw data for the graph is written out below:

Sizes	Regular	Karatsuba	Toom-Cook
100	0.001606	0.002302	0.002472
1000	0.021615	0.022407	0.022185
5000	0.235342	0.21298	0.186786
10000	0.788806	0.653776	0.644642
20000	3.11635	1.684037	1.478207
30000	6.934095	2.991823	2.882202
40000	12.242723	4.496403	4.106197
50000	20.14292	6.406042	5.39822

From both experiment 1 and experiment 2, I chose the threshold for when Karatsuba becomes preferred to big_mul to be at about when the numbers are 65 limbs or higher (order of ~1000 characters), and for Toom-Cook to be at when there are at least 225 limbs or higher (order of ~3600 characters). Wanted to choose values that were under 10,000 characters, because grade-school multiplication becomes noticeably worse then.

Challenges

There were a lot of challenges associated with implementing these algorithms, I wanted to discuss the top 3 to offer more insight into the difficulty of this mini project.

1. The first was just the sheer conceptual difficulty of Toom-Cook. Karatsuba was much simpler/straightforward to understand and implement, with simple equations, and simple split of limbs. Toom-Cook took days to comprehend, understanding the various phases of the algorithm was challenging, and due to the number of calculations, it wasn't easy to find bugs even when testing while coding.
2. The second is tied into the first, which is due to the recursive nature of the algorithm, debugging was very hard, especially as the bigints increased in size. There were small edge cases or logic gaps missing that were not evident at smaller test cases, and especially for Toom-Cook, there were multiple pages of arithmetic that I checked by hand to isolate bugs/logical issues.
3. Lastly, and perhaps the most series challenge was one specific step within Toom-Cook, due to the nature of how we are representing big numbers. There was one calculation, specifically regarding the calculation of b, the coefficient of x^3, where in order to solve the algebraic expression, a division by 6 became necessary. This puzzled me for a long time, as how was I supposed to/allowed to use division in order to develop Toom-Cook? At first, I

paid it no heed, and told myself it was more important to get the algorithm's logic as a whole working first. I did so, using big_div, and got it to work. However, then when I came to the performance testing, I was horrified. It made sense that as the numbers got very large, during this final step of interpolation, dividing by 6 became **very** expensive operation. This was causing Toom-Cook to become even slower than regular big_mul for large numbers. That step was necessary as well, as there was no other way around it to solve for that coefficient. I had even pinpointed the division as the problem using benchmarking, as the algorithm was spending 80% of its time in the division step! I then realized that I would need a separate division method that was built purposefully to handle only divisions by 3. I realized that when dividing by 3, depending on the current number's carry, I could figure out what number (either 1/3 or 2/3 of UINTMAX) to the following limb, to get the right dividend. This method is defined in the code as divide_by_3. After using this method instead of big_div in big_toom_cook, performance shot up tenfold, and I was able to get results that were indicative of toom-cook's true potential. This was perhaps my greatest challenge apart from implementing the algorithm itself.

Commands to run code:

I wrote code for various tests (documented in the code), along with experiments that can be run by uncommenting them in the provided main function. As such, to run everything, you can uncomment all the code inside main and then just compile and run to run the extension. As provided, the code will run the various tests I wrote along with the experiments doing 10 calculations in 1 run (to allow for larger sizes to be run!). It is default set to run experiment 1 between 50,000 to 100,000, and experiment 2 at 5000!. Note experiment 2 runs 1 multiplication type at a time, so be sure to change that to whichever strategy you want if you want to run others! It is default set to Karatsuba. Experiment 1 runs all 3! * Note that for larger numbers, if the time estimated seems a little less than expected, not that freeing up, mallocing, and copying larger numbers where appropriate does also take time! (this was properly not counted in the time).

Compile: gcc bigint.c -o bigint -fsanitize=address,undefined,leak -static-libasan -g
To run: ./bigint

*Note: I wrote a comment called EXTENSION STARTS HERE, to indicate where new code was started being added for the extension, stuff before it already existed from keygen.