Implementation of the DKSS Algorithm for Multiplication of Large Numbers

Christoph Lüders
Universität Bonn
Institut für Informatik
Bonn, Germany
chris@cfos.de

ABSTRACT

The Schönhage-Strassen algorithm (SSA) is the de-facto standard for multiplication of large integers. For N-bit numbers it has a time bound of $O(N \cdot \log N \cdot \log\log N)$. De, Kurur, Saha and Saptharishi (DKSS) presented an asymptotically faster algorithm with a better time bound of $N \cdot \log N \cdot 2^{O(\log^* N)}$. For this paper, a simplified DKSS multiplication was implemented. Assuming a sensible upper limit on the input size, some required constants could be precomputed. This allowed to simplify the algorithm to save some complexity and run-time. Still, run-time is about 30 times larger than SSA, while memory requirements are about 2.3 times higher than SSA. A possible crossover point is estimated to be out of reach even if we utilized the whole universe for computer memory.

Categories and Subject Descriptors

G.1.0 [**Numerical Analysis**]: General—*Multiple precision arithmetic*; G.4 [**Mathematical Software**]: Algorithm design and analysis, Efficiency; I.1.2 [**Symbolic And Algebraic Manipulation**]: Algorithms—*Algebraic algorithms, Analysis of algorithms*

General Terms

Algorithms, Performance.

Keywords

Integer multiplication, multiprecision arithmetic, fast Fourier transform.

1. INTRODUCTION

Multiplication of integers is one of the most basic arithmetic operations. The naive method to multiply two N-bit integers requires $O(N^2)$ bit-operations. As numbers get larger, soon this process becomes too slow and faster means are desirable.

ISSAC'15, July 6–9, 2015, Bath, United Kingdom.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3435-8/15/07 ...$15.00.
DOI: http://dx.doi.org/10.1145/2755996.2756643.

In the 1960s methods were discovered (cf. [14], [20], [3]) that lowered the number of bit-operations successively until in 1971 Schönhage and Strassen [19] presented their now well known algorithm (abbreviated as *SSA*). If $\log N$ denotes the logarithm to base 2, their algorithm has a time bound of

$$T(N) = O(N \cdot \log N \cdot \log\log N). \tag{1}$$

It uses a *fast Fourier transform* (FFT), a technique that was already known by Gauss, but rediscovered in 1965 by Cooley and Tukey [4].

SSA was the asymptotically fastest known method for multiplication until in 2007 Fürer [9] found an even faster way. Fürer's algorithm inspired De, Kurur, Saha and Saptharishi to their multiplication method [6] (see [7] for an expanded text), here called *DKSS multiplication* (or *DKSSA* for short). Both Fürer's and DKSS' algorithms require

$$T(N) = N \cdot \log N \cdot 2^{O(\log^* N)} \tag{2}$$

bit-operations, where $\log^* N$ is the number of times the logarithm function has to be applied to get a value ≤ 1.

However, Fürer conjectured that his method only becomes faster than SSA for "astronomically large numbers" [9, sec. 7]. Fürer's algorithm uses floating point operations, in contrast to SSA and DKSSA, which both use integer operations. To obtain a fair comparison, I implemented DKSSA and compared it to SSA.

The ability to multiply numbers with millions or billions of digits is not only academically interesting, but bears practical relevance. Number theoretical tasks like specialized primality tests require fast multiplication of potentially very large integers. Multiplication of dense polynomials with numerical coefficients can be reduced to one huge integer multiplication through Kronecker-Schönhage substitution [18, sec. 2]. Likewise, multiplication and factoring of multivariate polynomials [10] and calculation of π or e to billions of digits or computing billions of roots of Riemann's zeta function are other fields that require fast multiplication of large numbers [21, sec. 8.0]. Such calculations can be performed nowadays with general purpose computer algebra systems.

Fast multiplication is an important building block of a general library for arithmetic operations on large numbers like GMP, the GNU Multiple Precision Arithmetic Library [12] or MPIR, a popular GMP fork for Windows [11]. Many of the more complex tasks — like inversion, division, square root or greatest common divisor — revert back to multiplication, cf. [10].

The method of Schönhage and Strassen (we use the updated version from [18]) uses the ring $\mathbb{Z}/(2^K + 1)\mathbb{Z}$, where

2 is a primitive $2K$-th root of unity. It breaks numbers of N bits into pieces of $O(\sqrt{N})$ bits, which are in turn multiplied, maybe using the algorithm recursively. It is cleverly designed to take advantage of the binary nature of today's computers: multiplications by powers of 2 are particularly simple and fast to perform and this permits a crucial speedup for the FFT. This is why it has not only held the crown of the asymptotically fastest multiplication algorithm for over 35 years, but is also in widespread practical use today.

DKSS multiplication has a better asymptotic time bound, but is more complicated. Its elaborate structure allows input numbers to be broken into pieces only $O(\log^2 N)$ bits small. However, its arithmetic operations are more costly.

We are investigating here if or when an implementation of DKSS multiplication becomes faster than one of Schönhage-Strassen multiplication. An expanded account on the theory, implementation and analysis can be found in [16].

1.1 Note on Asymptotics

We know from theory that DKSSA is asymptotically faster than SSA. Unfortunately, that doesn't tell us if a practical implementation will be faster for the range of input lengths it is used for.

Software that multiplies large numbers usually chooses the fastest algorithm for a given input bit-length N. In our case, the implementation was done with my own `BIGNUM` library [15] that features grade-school (or ordinary), Karatsuba, Toom-Cook 3-way and Schönhage-Strassen multiplication. Except for grade-school, all other methods are recursive schemes, that means, they reduce the input to multiple smaller multiplications that in turn are handled by the fastest algorithm for their length until, at the lowest level, grade-school multiplication is used.

This requires to find the *crossover points* between different algorithms, i.e. the bit-length N, where the more complex, yet asymptotically faster algorithm becomes just as fast as the simpler algorithm. Since these crossover points depend on several factors (many of them implementation details or rooted in code optimization), they are usually found by benchmarking, that is, measuring and comparing the runtime of the different algorithms. Figure 2 lists their values for `BIGNUM` and MPIR.

Since a practical implementation is limited by the hardware it's run on, this implies an upper limit on the bit-length N. Our implementation runs on a 64-bit CPU, so the length of input numbers is limited to $8 \cdot 2^{64}/4 = 2^{65}$ bits. This is not a serious limitation. According to `top500.org`, even the fastest supercomputers in 2015 are equipped with around 1 PB $= 2^{53}$ bits of memory.

This is only to show that for a practical application today a limitation of $N < 2^{65}$ bits is not very limiting. As we will see, the picture doesn't change much even if we assume $N < 2^{128}$ bits. If Moore's law holds, that limit will hardly be exceeded within this century.

2. (SIMPLIFIED) DKSS ALGORITHM

DKSS multiplication as laid our in [7] uses the polynomial quotient ring $\mathcal{R} = \mathcal{P}[\alpha]/(\alpha^m + 1)$. Since $\alpha^m \equiv -1$, α is a primitive $2m$-th root of unity and multiplications by powers of α can be done as cyclic shifts (where coefficients only change place and possibly their sign, but not their absolute value; hence a cyclic shift can be done in time linear to the number of coefficients). Underlying $\mathcal{R}$ is the ring $\mathcal{P} = \mathbb{Z}/p^c\mathbb{Z}$, where p is a prime number and c is a constant. This "double structure" can be exploited in the FFT and allows to break down an N-bit input number into numbers of $O(\log^2 N)$ bits.

In their paper, De, Kurur, Saha and Saptharishi describe the algorithm without any assumptions about the hardware, but as described in §1.1, we assume here that the length of input numbers is limited to 2^{65} bits.

DKSS go to great lengths to show that suitable primes p can be found at run-time. To facilitate that, the modulus p^c with $c > 1$ is used and numbers are encoded as k-variate polynomials, $k > 1$. Both c and k are constants that depend on Linnik's constant. Polynomials have a degree less than M in each variable, i.e. they have M^k coefficients. To calculate the modulus p^c, Hensel lifting is used. All of this is done to keep the time to find the prime p bounded, cf. [7, §4.2].

Instead, in our simplified DKSS, the needed constants (the prime p and a generator ζ of $\mathbb{F}_p^*$) can be precomputed, since an upper limit of input numbers is known. Therefore, our M is chosen as large as M^k in the original paper and our p as large as p^c in the original paper. There is no more need for Hensel lifting. This allows us to simplify the algorithm and to skip c and k from now on.

The computational cost can of course only be lower, since we leave out some steps (no searching for prime p or generator ζ, no Hensel lifting, no recursion to reduce the number of variables in multivariate polynomials). Between the genuine DKSS and our simplified version, the number of coefficients is chosen in the same way, whereas in the simplified case the ring $\mathcal{P}$ is even a field.

2.1 Short Description

Let $[a : b]$ denote the set $\{x \in \mathbb{Z} \mid a \leq x \leq b\}$.

Furthermore, for a ring R, a primitive n-th root of unity $\omega \in R$ is called *principal*, if and only if n is coprime to the characteristic of R and $\sum_{i=0}^{n-1} \omega^{ij} = 0$ for $j \in [1 : n-1]$.

To multiply two nonnegative integers a, $b < 2^N$, $N \in \mathbb{N}$ to obtain their product $c := ab < 2^{2N}$, we convert the numbers into polynomials over a ring $\mathcal{R}$, use the fast Fourier transform to transform their coefficients, then multiply the sample values and transform backwards to gain the product polynomial. From there, we can easily recover the resulting integer product.

Define $\mathcal{R} := \mathcal{P}[\alpha]/(\alpha^m + 1)$ and $\mathcal{P} := \mathbb{Z}/p\mathbb{Z}$. Polynomial coefficients are in $\mathcal{P}$ and are henceforth called *inner* coefficients.

Input numbers a and b are encoded as polynomials $a(x)$ and $b(x) \in \mathcal{R}[x]$ with degree less than M (M will be defined shortly). The coefficients are called *outer* coefficients.

We can multiply a and b as follows:

1. Choose integers $m \geq 2$ and $M \geq m$ as powers of 2, such that $m \approx \log N$ and $M \approx N/\log^2 N$. $2M$ will be the length of the FFTs, while m is the degree of elements of $\mathcal{R}$. For simplicity of notation, define $\mu := M/m$.

2. Let $u := \lceil 2N/Mm \rceil$ denote the number of input bits per inner coefficient. Find a prime p with $2M \mid p - 1$, i.e. $p := h \cdot 2M + 1$ for some $h \in \mathbb{N}$.

   Furthermore, the condition
   $$p \geq \frac{1}{2} M m 2^{2u} \qquad (3)$$
   must be met to ensure that the inner coefficients don't overflow.

3. From parameters M, m and p compute a principal $2M$-th root of unity $\rho \in \mathcal{R}$ with the additional property $\rho^\mu = \alpha$:

   A generator ζ of $\mathbb{F}_p^*$ has order $p - 1 = h \cdot 2M$ and is a principal $(p-1)$-th root of unity, making $\omega := \zeta^h$ a principal $2M$-th root of unity.

   Denote $\gamma := \omega^\mu$, a principal $2m$-th root of unity. Furthermore, let $i \in [1 : 2m-1]$ be odd. Observe that γ^i is a root of $\alpha^m + 1 = 0$, since $(\gamma^i)^m = (\gamma^m)^i = (-1)^i = -1$. Now use Lagrange interpolation to find $\rho(\alpha)$ with $\rho(\gamma^i) = \omega^i$ for all i.

4. Encode a and b as polynomials $a(x)$, $b(x) \in \mathcal{R}[x]$ with degree less than M by breaking them into M blocks with $um/2$ bits in each block. Each such block describes an outer coefficient. Furthermore, split each outer coefficient block into $m/2$ blocks of u bits each, where each block forms an inner coefficient in the lower-degree half of a polynomial. Set the upper $m/2$ inner coefficients to zero. Finally, set the upper M outer coefficients to zero.

5. Use root ρ to perform a length-$2M$ fast Fourier transform of $a(x)$ and $b(x)$ to obtain $\widehat{a}_i := a(\rho^i) \in \mathcal{R}$, likewise $\widehat{b}_i$. Use the special structure of $\mathcal{R}$ to speed up the FFT, see the next section.

6. Multiply pointwise $\widehat{a}_i \widehat{b}_i =: \widehat{c}_i$. Note that $\widehat{a}_i$, $\widehat{b}_i \in \mathcal{R}$ are themselves polynomials. Reduce their multiplication to integer multiplication by Kronecker-Schönhage substitution and multiply them recursively (with the fastest algorithm for their length). See below for details.

7. Perform a backwards transform of length $2M$ to obtain the product polynomial $c(x) := a(x)b(x)$. This includes the usual reordering of the resulting coefficients and dividing them by $2M$.

8. Evaluate the inner polynomials of the product polynomial $c(x)$ at $\alpha = 2^u$ and the outer polynomial $c(x)$ at $x = 2^{um/2}$ to recover the integer result $c = ab$.

To multiply two elements of $\mathcal{R}$ (which are themselves polynomials), we use Kronecker-Schönhage substitution, cf. [18, sec. 2] and [2, sec. 1.3 & 1.9]. This reduces polynomial multiplication to integer multiplication with $m(2\lceil \log p \rceil + \log m) = O(\log^2 N)$ bits. To do so, we use the fastest multiplication for that length, possibly recursing into DKSSA.

After that we still have to perform both modulo operations: mod $(\alpha^m + 1)$ on the product polynomial and mod p on its coefficients.

2.2 Performing the FFT

A Cooley-Tukey FFT [4] works for any length that is a power of 2. Here the length is $2M$ and it can be split as $2M = 2m \cdot \mu$, with $\mu = M/m$. The input vector can be organized as a matrix with $2m$ rows of μ columns each.

The DKSS algorithm uses a radix-μ decimation in time Cooley-Tukey FFT, cf. [8, sec. 4.1]. That is, it first does μ FFTs of length $2m$ on the columns of the matrix, then multiplies the results by twiddle factors (called *bad multiplications* by DKSS) and finally performs $2m$ FFTs of length μ on the rows of the matrix.

The length-$2m$ column FFTs use α as root of unity and since multiplications with powers of α can be performed as cyclic shifts, they can be done in linear time.

Let's see how this works in detail. The length-$2M$ DFT of $a(x)$ with ρ as root of unity can be computed in three steps:

1. Perform *inner DFTs.*

   Rewrite the input vector a as a matrix of $2m$ rows of μ columns (called e_ℓ) and perform FFTs on the columns.

   Let $v \in [0 : 2m-1]$ and define polynomials $\bar{a}_v(x) \in \mathcal{R}[x]$ with degree less than μ as

   $$\bar{a}_v(x) := a(x) \bmod (x^\mu - \alpha^v). \tag{4}$$

   Denote $\bar{a}_{v,\ell}$ the ℓ-th coefficient of $\bar{a}_v(x)$. Then it holds that

   $$\bar{a}_{v,\ell} = e_\ell(\alpha^v).$$

   So to find the ℓ-th coefficient of each $\bar{a}_v(x)$ perform a length-$2m$ DFT of $e_\ell(y)$, using α as root of unity. Call these the *inner DFTs.*

   Perform multiplications by powers of α as cyclic shifts. Since $\alpha^m \equiv -1$, coefficients of powers $\geq m$ wrap around with changed sign.

2. Perform *bad multiplications.*

   Let $[0 : 2M-1] \ni i = 2m \cdot f + v$ with $f \in [0 : \mu - 1]$ and $v \in [0 : 2m-1]$. Then it follows from (4) that

   $$a(\rho^i) = a(\rho^{2m \cdot f + v}) = \bar{a}_v(\rho^{2m \cdot f + v}). \tag{5}$$

   In order to efficiently compute $\bar{a}_v(\rho^{2m \cdot f + v})$, define

   $$\widetilde{a}_v(x) := \bar{a}_v(x \cdot \rho^v). \tag{6}$$

   Compute $\widetilde{a}_v(x)$ by computing its coefficients $\widetilde{a}_{v,\ell} = \bar{a}_{v,\ell} \cdot \rho^{v\ell}$, with $\ell \in [0 : \mu - 1]$.

3. Perform *outer DFTs.*

   Now all that is left is to evaluate the $\widetilde{a}_v(x)$, $v \in [0 : 2m-1]$, at $x = \rho^{2m \cdot f}$, for $f \in [0 : \mu - 1]$. The $\widetilde{a}_v(x)$ are arranged in such a way that these evaluations are nothing but length-μ DFTs of $\widetilde{a}_v(x)$ with ρ^{2m} as root of unity, performed on the rows of the matrix. Call these the *outer DFTs.*

   If $M \geq m$, this is done by a *recursive* call to the FFT routine. According to (5) and (6) it computes $\widetilde{a}_v(\rho^{2m \cdot f}) = \bar{a}_v(\rho^{2m \cdot f + v}) = a(\rho^{2m \cdot f + v}) = a(\rho^i)$.

   If called recursively, $M < m$ might hold. Then, just computing an inner DFT with $\alpha^{m/M}$ as (m/M)-th root of unity is sufficient.

The source code of the whole FFT can be found as `dkss_fft()` in `BIGNUM`.

3. IMPLEMENTATION

3.1 Parameter Selection

There is some freedom on how exactly to select parameters M, m, u and p. It follows from (3) that

$$p \geq \frac{1}{2} M m 2^{2u} \approx \frac{1}{2} N^5 / \log N. \tag{7}$$

Both allocated memory and cost of division (for modular reductions) depend on the length of p or, more precisely, on its length in processor words. But the larger the value of p (which is the modulus of $\mathcal{P}$), the more bits we can encode in each coefficient. This is turn can lead to fewer coefficients and thus maybe to a shorter FFT length, which is desirable.

So we select a value for p that satisfies condition (7), but makes the most of the memory it occupies. That is, $\log p$ should be slightly less than a multiple of the processor word size in bits. Benchmarking showed that this leads to faster run-time than using (7) without rounding up, see [16, fig. 19].

The prime p is selected from a list of precomputed primes. I precomputed suitable primes for bit-lengths from 2 to 1704. According to (7) this allows DKSSA up to a bit-length of $N = 2^{342}$ bits. Since the implementation is running on a 64-bit machine, a shorter table would have done. And since we're rounding the length of p to multiples of 64 bit, the table of possible primes for p contains only 6 entries, listed in Figure 1.

Next, the largest u is selected that permits the polynomial to hold the whole $2N$ bits of the result. It follows from (3) that $\log p \geq \log(Mm) + 2u - 1$. Since $\log p$ is already fixed, maximize u. The larger u is, the less coefficients are needed.

After finding an u that fits, minimize the product Mm, because the smaller Mm is, the smaller the FFT length and memory requirements are.

Lastly, set M and m. Factors can be moved around between M and m, since until now only the product Mm was needed. To prove the time-bound for DKSSA it is only required that $M = O(N/\log^2 N)$ and $m = O(\log N)$, cf. [7, §4.2]. This means that their quotient $M/m \approx k \cdot N/\log^3 N$ can contain some arbitrary constant k.

Some short tests on selecting k indicated that $k = 1$ is overall a good choice, but more research should confirm this. This selection leads to the same values that DKSS use and that were already described in §2.1, Step 1.

This parameter selection was implemented in `BIGNUM` as `dkss_set_mmu()`.

3.2 Test Environment

The implementation was done with my own `BIGNUM` library [15] that is written in C++ with a few inner subroutines in assembly language. `BIGNUM` allows different multiplication algorithms and selects the fastest one, depending on the size of the operands. An extensive explanation of the algorithms and their implementation can be found in [16, ch. 2].

Prime p	Bit-length of p
$27 \cdot 2^{59} + 1$	64
$81 \cdot 2^{121} + 1$	128
$13 \cdot 2^{188} + 1$	192
$207 \cdot 2^{248} + 1$	256
$13 \cdot 2^{316} + 1$	320
$285 \cdot 2^{375} + 1$	384

Figure 1: Primes p used by `BIGNUM`

Algorithm	`BIGNUM`	MPIR
Karatsuba	≥ 28	≥ 14
Toom-Cook 3-way	≥ 152	≥ 98
Toom-Cook 4-way	—	≥ 154
Toom-Cook 8.5-way	—	≥ 270
Schönhage-Strassen	> 2464	≥ 2880

Figure 2: Crossover points in 64-bit words for `BIGNUM` and MPIR

Implementations of SSA and DKSSA are called `SMUL` and `DKSS_MUL`, respectively. The source code is available under LGPL.

In `BIGNUM`, I chose to implement everything from scratch, that is, I did not use any other large integer library or special hardware, so I would not be limited by someone else's design decisions. This implies that `BIGNUM` did not benefit from the large codebase that e.g. GMP [12] or MPIR [11] already offer.

Since DKSSA is compared to SSA, some work on `BIGNUM` went into fine-tuning SSA to make it faster, like "Mersenne transforms" and extensive "tuning" (to use the same names as in [10]). Missing is Bailey's 4-step transform (cf. [10, §2.2.3]), which explains most of why `BIGNUM` is slower than MPIR (see below).

To the best of my knowledge, GMP is the leading and fastest open-source large integer library. Its port for the Windows operating system is MPIR. Comparing SSA multiplication speed for `BIGNUM` and MPIR 2.6.0, `BIGNUM` is on average slower by a factor of about 1.30. As we will see, this factor is negligible when compared to the factor of slowness that `DKSS_MUL` vs. `SMUL` exhibits.

Tests were run on an Intel Core i7-3770 processor (Ivy Bridge microarchitecture) with 3.40 GHz clock rate and 32 GB memory. This CPU has level 1 caches per core of both 32 KB for data and 32 KB for instructions, unified level 2 caches of 256 KB per core and a unified level 3 cache of 8 MB for all cores.

The operating system was Windows 7 64-bit and the compiler was Visual Studio 2012 (C++ compiler v17). To improve cache performance, the process affinity was fixed to one processor. The same testing conditions were used both for `BIGNUM` and MPIR.

Correctness of the code was verified with Lucas-Lehmer tests of hundreds of Mersenne numbers (both prime and composite), including all Mersenne primes up to $2^{1398269} - 1$.

To measure speed, two operands of the same bit-length were multiplied. Both were properly aligned, filled with random words (generated with the C++ `mt19937_64` Mersenne Twister) and the average run-time of several tests was used.

Timings were taken by use of Windows' `QueryThreadCycleTime()` function that counts only CPU cycles spent by the thread in question. It queries the CPU's Time Stamp Counter and has clock-cycle resolution, i.e. 3.4 GHz, which is very accurate.

The benchmarks were done with input lengths of 237 to 169,869,312 words. The largest input number requires already temporary memory of about 26 GB. Because of memory limitations, I could not test larger inputs.

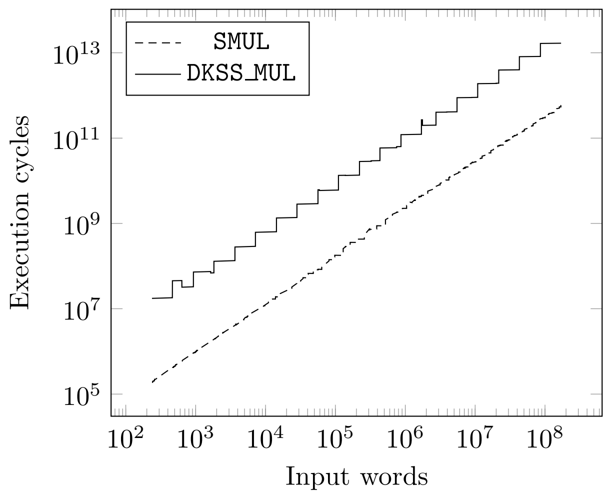

Figure 3: Execution time of DKSS_MUL

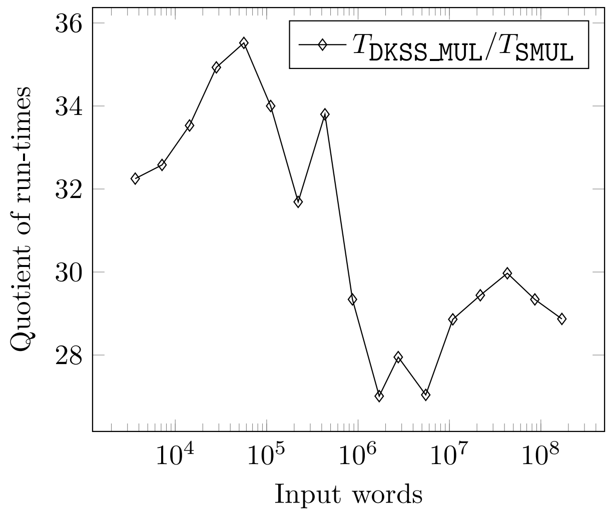

Figure 4: Quotient of DKSS_MUL and SMUL run-times vs. input length

3.3 Improvements

The implementation was improved over the simplified algorithm described in §2 in two respects:

In §2.2, Step 2 every coefficient is multiplied by some power of ρ. To calculate ρ^i, $i \in [0 : 2M-1]$, set $r = \lfloor i/\mu \rfloor$ and $s = i \bmod \mu$. Since $\rho^\mu = \alpha$, it holds that $\rho^i = \rho^{\mu r + s} = \rho^{\mu r}\rho^s = \alpha^r \rho^s$. Because multiplications by powers of α can be done as cyclic shifts, we can save almost half of the bad multiplications by precomputing ρ^s, for $s \in [0 : \mu-1]$. Doing so costs negligible memory (cf. §3.5) and saves almost half of execution time.

The FFT from §2.2 requires temporary memory of the size of a full input vector to store the $\bar{a}_v(x)$. By using an in-place matrix transposition the $\bar{a}_{v,\ell}$ can be reordered in such a way that the FFTs can be done in-place as well. This saves about one third of temporary memory. The FFT then works like Bailey's "six step" FFT algorithm [1], but with faster inner DFTs. Cache efficiency of the matrix transposition is not a concern here, since profiling showed that about 85 % of total run-time is spent with *bad multiplications*, cf. [16, sec. 4.7].

3.4 Execution Time

Figure 3 shows a double-logarithmic plot of the execution time of `DKSS_MUL` in comparison to `SMUL`. The stair-like graph of `DKSS_MUL` execution time stems from the fact that execution time almost totally depends on the FFT length $2M$ and the size of elements of $\mathcal{R}$. Since both M and m are powers of 2, many different input lengths lead to the same set of parameters and hence to the same FFT length.

In contrast, the `SMUL` execution time graph is much smoother. The reason for this is that SSA uses the ring $\mathbb{Z}/(2^K+1)\mathbb{Z}$, where K does not have to be a power of 2, thus allowing a finer granularity.

The `DKSS_MUL` graph shows that execution time is almost the same for the beginning and the end of each step. The only part that depends directly on N is the encoding of the input numbers and decoding into the resulting product. The time needed to do the FFT clearly dominates overall execution time.

To compare run-times and memory consumption numerically, we pick the rightmost point for each step of the `DKSS_MUL` graph, because only then are all outer coefficients filled with bits from the input and not zero-padded.

As can be seen clearly, `DKSS_MUL` is much slower over the whole range of tested input lengths. From this graph it is hard to see if `DKSS_MUL` is gaining on `SMUL`. Figure 4 shows the quotient of run-times that is between 27 to 36 times slower at best. The location of a crossover point is discussed in §4.

In §2.1, Step 6 it is mentioned that DKSSA might be called recursively. In the tests, it never came to that. Even with maximal long inputs, the inner multiplications were just 195 words long and are thus still in the range for Toom 3-way multiplication. In fact, for recursion into `DKSS_MUL` to happen, `DKSS_MUL` would have to be faster than `SMUL` on the top level first.

3.5 Memory Requirements

`DKSS_MUL` memory requirements are dominated by two times the size of the polynomials: input $a(x)$ and $b(x) \in \mathcal{R}[x]$. The result $c(x)$ requires no further memory, since storage of one of the input polynomials can be reused.

Each polynomial has $2M$ coefficients and with (7) we estimate its memory requirement as $2Mm\lceil \log p \rceil \approx 10N$ bits.

To be exact, more memory, namely another $m \cdot \lceil \log p \rceil \approx 5 \log^2 N$ bits of temporary memory, is allocated in `dkss_fft()`, but that is of no big consequence compared to $10N$ bits for each polynomial. The same applies to the M/m precomputed powers of ρ, each with a length of $m\lceil \log p \rceil$ bits. Together, they only need $M\lceil \log p \rceil$ bits, that is, a $2m$-th part of the memory of one polynomial.

If we assume fully utilized outer coefficients and both input numbers have N bits, total memory needed by `DKSS_MUL` is

$$M_{\texttt{DKSS\_MUL}}(N) \approx 20N \text{ bits.}$$

The above memory requirements are a direct consequence of the algorithm, since they stem from the memory needed to store the encoded input numbers.

Compare this to the memory requirements of `SMUL`. According to [16, eq. (2.32)], the approximate amount of tem-

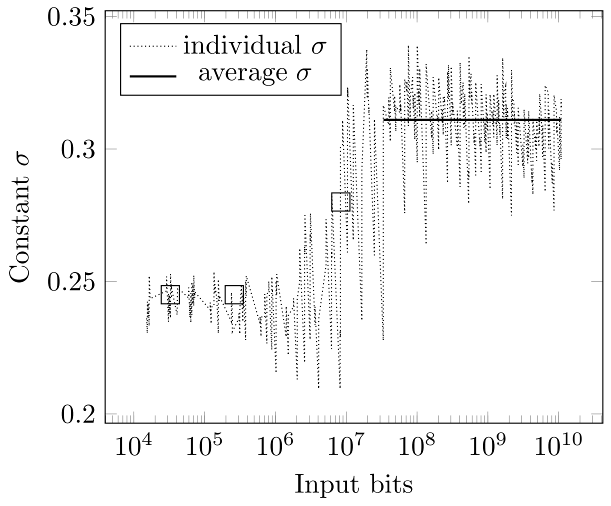

Figure 5: SMUL **run-time constant** σ

porary memory for SMUL is

$$M_{\text{SMUL}}(N) \approx 8N \text{ bits}.$$

Again, this is the memory needed to hold the encoded polynomials.

Measured memory consumption fitted theory nicely with an average $M_{\text{DKSS\_MUL}} \approx 18.88$ and $M_{\text{SMUL}} \approx 8.26$. The average quotient was $M_{\text{DKSS\_MUL}}/M_{\text{SMUL}} \approx 2.30$, which fits the theoretical $20/8 = 2.5$ well.

4. EXTRAPOLATION

Now that we have seen that in the ranges our tests have covered DKSS_MUL is still much slower than SMUL, we estimate the input length where DKSS_MUL starts to outperform SMUL.

To do that, we model the run-times for the algorithms, i.e. express them with explicit constants, then try to determine the constants from our measurements.

The formulas for run-times both contain terms that express the level of recursion: $\log\log N$ for SSA and $\log^* N$ for DKSSA, respectively. When it comes to recursion, the fastest algorithm for that length is used. For example, SSA calls itself only then for a second time if that would be faster than Toom-Cook. That leads to a relatively smooth run-time graph without large jumps.

To model that, we use the smooth function $\log\log N$ instead of $\lceil\log\log N\rceil$ for SSA and the super-logarithm $\operatorname{slog} N$ instead of $\log^* N$ for DKSSA, respectively.

4.1 Modeling SMUL Run-Time

Following (1) we model SMUL run-time, that is, rewrite it with an explicit constant as

$$T_\sigma(N) \le \sigma \cdot N \cdot \log N \cdot \log\log N. \tag{8}$$

Dividing measured execution cycles by $N \cdot \log N \cdot \log\log N$ to calculate σ leads to the graph depicted in Figure 5. Interestingly, this graph seems to have two plateau-like sections.

The plateaus correspond quite nicely with the cache sizes of the test machine, see §3.2. The three boxes indicate maximum input sizes that could still be calculated in the respective cache memory.

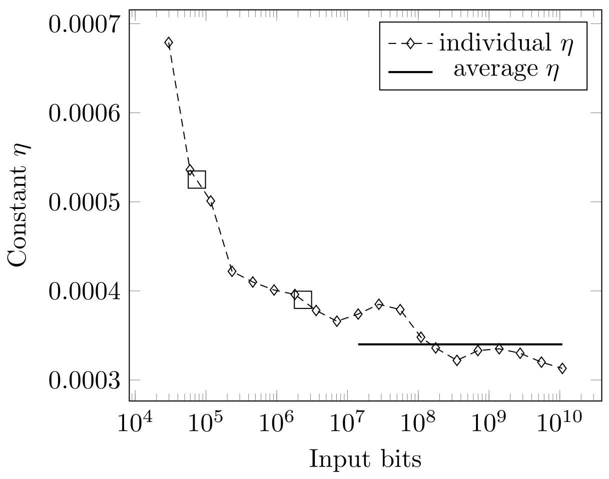

Figure 6: DKSS_MUL **run-time constant** η

After level 3 there is no further caching, hence when the required temporary memory is some orders larger there is no further visible influence on the run-time constant σ. Averaging from input sizes of 32 Mbits (this uses about 32 MB temporary memory) onwards leads to an average $\sigma \approx 0.311$.

4.2 Modeling DKSS_MUL Run-Time

Likewise, we use the measured run-times to model the run-time of DKSS_MUL. We write (2) with explicit constants and the smooth super-logarithm and get:

$$T_\eta(N) \le \eta \cdot N \cdot \log N \cdot 2^{\delta \cdot \operatorname{slog} N}. \tag{9}$$

Substitute $K := 2^\delta$ and we get

$$T_\eta(N) \le \eta \cdot N \cdot \log N \cdot K^{\operatorname{slog} N}.$$

The question is now: what is the value of K? The latest result on integer multiplication from 2014 by Harvey, van der Hoeven and Lecerf [13] suggests that DKSSA implies $K = 16$.

Unfortunately, with the few data points we have, any model of run-time is not very robust. As explained in §3.4, we get only one data point per step of the input bit-length N and since there is approximately one step per power of 2, we have only 20 measured run-time values to base our model on.

Figure 6 shows a graph of the constant η for each data point. The two boxes shown mark inputs that fit in level 2 and 3 caches, respectively. Like in Figure 5, we use only values of η where DKSS_MUL temporary memory has surely exceeded the level 3 cache size, that is, where is exceeds 12.8 Mbits (again, using 32 MB temporary memory).

To calculate the super-logarithm numerically, I used the linear approximation approach [17]. For $K = 16$ this leads to $\eta \approx 0.00034$. As we will see now, it leads to a crossover point that is extremely large.

4.3 When Will DKSS_MUL Trump SMUL?

Based on (8) and (9) we can solve

$$\begin{aligned} T_\eta(N) &\le T_\sigma(N) \\ \eta \cdot N \cdot \log N \cdot K^{\operatorname{slog} N} &\le \sigma \cdot N \cdot \log N \cdot \log\log N \\ K^{\operatorname{slog} N} &\le (\sigma/\eta) \cdot \log\log N \\ \log K \cdot \operatorname{slog} N &\le \log(\sigma/\eta) + \log\log\log N. \end{aligned}$$

For large N substitute $\nu := \log\log N$ and get

$$\log K \cdot (\operatorname{slog} \nu + 2) \le \log(\sigma/\eta) + \log \nu. \qquad (10)$$

Solving (10) numerically yields the enormous solution of $\nu \ge 15934$ and hence $N \ge 10^{10^{4796}}$! An optimistic estimation of the number of bits for computer memory available in this universe is 10^{100}. So the estimations of the crossover point are *orders of orders* of magnitude higher than the largest machine we can build.

It should be mentioned here that SMUL is better optimized than DKSS_MUL. The reason for that is that SSA is now studied and in wide use for many years and its details are well understood. In contrast, DKSSA is still quite young and to my knowledge this is the first implementation of it. Still, in my appraisal none of the possible improvements to DKSS_MUL of §5.1 have the potential to speed it up so much that it becomes faster than SMUL.

Even if DKSS_MUL was only 1.5 times slower than SMUL, the crossover point would still be at least at $N \approx 10^{10^{99}}$ bits and thus unreachable.

5. CONCLUSION

De, Kurur, Saha and Saptharishi describe a new procedure to multiply large integers efficiently that was implemented as DKSS_MUL. The currently widely used algorithm by Schönhage and Strassen was implemented as SMUL. Both algorithms were compared for their run-time and memory consumption.

The results indicate that Schönhage and Strassen's algorithm is still the better choice for a variety of reasons:

1. SMUL is faster than DKSS_MUL.

   Benchmarks show that SMUL is at least between 27 to 36 times faster than DKSS_MUL, see Figure 3. The estimated input length where DKSS_MUL could become faster than SMUL is $N \ge 10^{10^{4796}}$ bits (which is larger than googolplex). But even if SMUL was only 1.5 times faster than DKSS_MUL, the crossover point would be so large that it could never be reached.

2. SMUL requires less memory than DKSS_MUL.

   If both input numbers are N bits long, DKSS_MUL requires about $20N$ bits of temporary memory, while SMUL requires only about $8N$ bits. In practice, the quotient is about 2.30.

3. SMUL is simpler to implement than DKSS_MUL.

   A simple implementation of SMUL needs about 550 lines of C++ code, where DKSS_MUL requires about 1000 lines plus more supporting routines, see [16, sec. 4.6] with more complex program code that is harder to test.

5.1 Future Work

Some possible improvements to DKSS_MUL are listed below. For more on profiling DKSS_MUL, see [16, sec. 4.7].

- Find optimum values of parameters M, m, u and p for any given N.

  Figure 3 shows some areas where longer input numbers lead to shorter execution times. Furthermore, profiling showed developments in percentages of run-times that suggest that a better choice of parameters is possible.

- Add support for "sparse integers" in the underlying multiplication.

  DKSSA reduces multiplication of large integers to multiplications in $\mathcal{R}$, a polynomial ring. To multiply two elements of $\mathcal{R}$, each is converted to one huge integer. About half of the words of it are zero and a future multiplication routine could exploit that. Profiling showed that up to 85 % of execution time is spent with multiplication of elements of $\mathcal{R}$ and a rising percentage of that is used by the underlying integer multiplication. I optimistically estimate the potential for speed up to be almost a factor of 2.

  Any other means of improving the speed of bad multiplications would be very beneficial, like speeding up Kronecker-Schönhage substitution.

- Exploit the structure of prime numbers p.

  The modulus of $\mathcal{P}$ is a prime number of the form $h \cdot 2M + 1$, where h is a small positive odd integer and M is a power of 2. Maybe modular reductions can be sped up by the technique listed in [5, p. 457]. This has the potential to save a great part of the cost of modular reductions, which showed to cost about 22 % of run-time in profiling.

If the potential savings listed above could be achieved, this would speed up DKSS_MUL by a factor of about 2.5. Not included in this factor is a better parameter selection. But even if that and other, yet unthought-of, improvements lead to another speed-up by a factor of 2, DKSS_MUL would still be at least about 5.4 times slower than SMUL.

Source Code

The full source code of of the implementation is available for evaluation as the BIGNUM library [15]. It is licensed under LGPL. Requirements are listed in §3.2.

Acknowledgments

I thank Andreas Weber for his support and advice with this paper. Furthermore, I am grateful to Michael Clausen for his help and advice with my diploma thesis which contains an earlier and expanded account on this topic. Last, but not least, I am thankful for the detailed feedback of the reviewers.

6. REFERENCES

[1] D. H. Bailey. FFTs in external or hierarchical memory. *Journal of Supercomputing*, 4:23–35, Dec. 1990.

[2] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic.* Cambridge University Press, 2011.

[3] S. A. Cook. *On the Minimum Computation Time of Functions.* PhD thesis, Harvard University, 1966.

[4] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[5] R. Crandall and C. Pomerance. *Prime numbers: A Computational Perspective.* Springer, 2nd edition, 2005.

[6] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 499–506, 2008.

[7] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM Journal on Compution*, 42(2):685–699, 2013.

[8] P. Duhamel and M. Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19:250 – 299, 1990.

[9] M. Fürer. Faster integer multiplication. In *Proceedings of the 39th ACM Symposium on Theory of Computing*, pages 57–66, June 2007.

[10] P. Gaudry, A. Kruppa, and P. Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, pages 167–174, 2007.

[11] T. Granlund, W. Hart, and the GMP and MPIR teams. The multiple precision integers and rationals library. `http://www.mpir.org/mpir-2.6.0.pdf`, Nov. 2012.

[12] T. Granlund and the GMP development team. The GNU multiple precision arithmetic library manual. `https://gmplib.org/gmp-man-6.0.0a.pdf`, Mar. 2014.

[13] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. `http://arxiv.org/abs/1407.3360`, July 2014.

[14] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics — Doklady*, 7:595–596, 1963.

[15] C. Lüders. BIGNUM library for multiple precision arithmetic. `http://www.wrogn.com/bignum`.

[16] C. Lüders. Fast multiplication of large integers: Implementation and analysis of the DKSS algorithm. Diploma thesis, Universität Bonn, Apr. 2014. `http://arxiv.org/abs/1503.04955`.

[17] A. Robbins. Solving for the analytic piecewise extension of tetration and the super-logarithm. 2005. `http://iteror.org/big/Source/articles/TetrationSuperlog_Robbins.pdf`.

[18] A. Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In J. Calmet, editor, *European Computer Algebra Conference*, volume 144, pages 3–15, 1982.

[19] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[20] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics — Doklady*, 3:714–716, 1963.

[21] J. von zur Gathen and J. Gerhard. *Modern Computer* Algebra. Cambridge University Press, 3rd edition, 2013.