Faster Integer Multiplication

Martin Fürer*
Department of Computer Science and Engineering
Pennsylvania State University
furer@cse.psu.edu

April 22, 2007

Abstract

For more than 35 years, the fastest known method for integer multiplication has been the Schönhage-Strassen algorithm running in time $O(n \log n \log\log n)$. Under certain restrictive conditions there is a corresponding $\Omega(n \log n)$ lower bound. The prevailing conjecture has always been that the complexity of an optimal algorithm is $\Theta(n \log n)$. We present a major step towards closing the gap from above by presenting an algorithm running in time $n \log n\, 2^{O(\log^* n)}$.

The main result is for boolean circuits as well as for multitape Turing machines, but it has consequences to other models of computation as well.

1 Introduction

All known methods for integer multiplication (except the trivial school method) are based on some version of the Chinese remainder theorem. Schönhage [Sch66] computes modulo numbers of the form $2^k + 1$. Most methods can be interpreted as schemes for the evaluation of polynomials, multiplication of the values, followed by interpolation. The classical method of Karatsuba and Ofman [KO62] can be viewed as selecting the values of linear forms at $(0, 1)$, $(1, 0)$, and $(1, 1)$ to achieve time $T(n) = O(n^{\lg 3})$. Toom [Too63] evaluates at small consecutive integer values to improve the time to $T(n) = O(n^{1+\epsilon})$. Finally Schönhage and Strassen [SS71] use the usual fast Fourier transform (FFT) (i.e., evaluation and interpolation at 2^mth roots of unity) to compute integer products in time $O(n \log n \log\log n)$. They conjecture the optimal upper bound (for a yet unknown algorithm) to be $O(n \log n)$, but their result has remained unchallenged.

Schönhage and Strassen [SS71] really propose two distinct methods. The first uses numerical approximation to complex arithmetic, and reduces multiplication of length n to that of length $O(\log n)$. The complexity of this method is slightly higher. It is only proposed as a one level approach. Even with the next level of multiplications done by a trivial algorithm, it is already very fast. The second method employs arithmetic in rings of integers modulo numbers of the form $F_m = 2^{2^m} + 1$ (Fermat numbers), and reduces the length of the factors from n to

*This work is supported in part by the Penn State Grove Award

$O(\sqrt{n})$. This second method is used recursively with $O(\log\log n)$ nested calls. In the ring $\mathbb{Z}_{F_m}$ of integers modulo F_m, the integer 2 is a particularly convenient root of unity for the FFT computation, because all multiplications with this root of unity are just modified cyclic shifts.

On the other hand, the first method has the advantage of the significant length reduction from n to $O(\log n)$. If this method is applied recursively, it results in a running time of order $n \log n \log\log n \ldots 2^{O(\log^* n)}$, because during the kth of the $O(\log^* n)$ recursion levels, the amount of work increases by a factor of $O(\log\log\ldots\log n)$ (with the log iterated k times). Note that, for their second method, Schönhage and Strassen have succeeded with the difficult task of keeping the work of each level basically constant, avoiding a factor of $\log^{O(1)} n = 2^{O(\log\log n)}$ instead of $O(\log\log n)$.

Our novel use of the FFT allows us to combine the main advantages of both methods. The reduction is from length n to length $O(\log n)$, and still most multiplications with roots of unity are just cyclic shifts. Unfortunately, we are not able to avoid the geometric increase over the $\log^* n$ levels.

Relative to the conjectured optimal time of $\Theta(n \log n)$, the first Schönhage and Strassen method had an overhead factor of $\log\log n \ldots 2^{O(\log^* n)}$, representing a doubly exponential decrease compared to previous methods. Their second method with an overhead of $O(\log\log n)$ constitutes another polynomial improvement. Our new method reduces the overhead to $2^{O(\log^* n)}$, and thus represents a more than multiple exponential improvement of the overhead factor.

We use a new divide-and-conquer approach to the N-point FFT, where N is a power of 2. It is well known and obvious that the JK-point FFT graph (butterfly graph) can be composed of two levels, one containing K copies of a J-point FFT graph, and the other containing J copies of a K-point FFT graph. Clearly $N = JK$ could be factored differently into $N = J'K'$ and the same N-point FFT graph could be viewed as being composed of J'-point and K'-point FFT graphs. The astonishing fact is that this is just true for the FFT graph and not for the FFT computation. Every way of (recursively) partitioning N produces another FFT algorithm. Multiplications with other powers of ω appear when another recursive decomposition is used.

It seems that this fact has been mainly unnoticed except for its use some time ago [Für89] in an earlier attempt to obtain a faster integer multiplication algorithm. In that paper, the following result has been shown. If there is an integer $k > 0$ such that for every m, there is a Fermat prime in the sequence $F_{m+1}, F_{m+2}, \ldots, F_{2^{m+k}}$, then multiplication of binary integers of length n can be done in time $n \log n\, 2^{O(\log^* n)}$. Hence, the Fermat primes could be extremely sparse and would still be sufficient for a fast integer multiplication algorithm. Nevertheless, this paper is not so exciting, because it is well known that the number of Fermat primes is conjectured to be finite.

It has long become standard to view the FFT as an iterative process (see e.g., [SS71, AHU74]). Even though the description of the algorithm gets more complicated, it results in less computational overhead. A vector of coefficients at level 0 is transformed level by level, until we reach the Fourier transformed vector at level $\lg N$. The operations at each level are additions, subtractions, and multiplications with powers of ω. They are done as if the N-point FFT were recursively decomposed into $N/2$-point FFT's followed by 2-point FFT's. The current author has

seen Schönhage present the other natural recursive decomposition into 2-point FFT's followed by $N/2$-point FFT's. It results in another distribution of the powers of ω, even though each power of ω appears as a coefficient in both iterative methods with the same frequency. But other decompositions produce completely different frequencies. The standard fast algorithm design principle, divide-and-conquer, calls for a balanced partition, but in this case it is not at all obvious that this will provide any benefit.

A balanced approach uses two stages of roughly $\sqrt{N}$-point FFT's. This allows an improvement, because it turns out that "odd" powers of ω are then very seldom. This key observation alone is not sufficiently powerful, to obtain a better asymptotic running time, because usually 1, -1, i, $-i$ and to a lesser extent $\pm(1\pm i)/\sqrt{2}$ are the only powers of ω that are easier to handle. We will achieve the desired speed-up by working over a ring with many "easy" powers of ω. Hence, the new faster integer multiplication algorithm is based on two key ideas.

- An unconventional FFT algorithm is used with the property that most occurring roots of unity are of low order.
- The computation is done over a ring with very simple multiplications with many low order roots of unity.

The question remains whether the optimal running time for integer multiplication is indeed of the form $n \log n\, 2^{O(\log^* n)}$. Already, Schönhage and Strassen [SS71] have conjectured that the more elegant expression $O(n \log n)$ is optimal as we mentioned before. It would indeed be strange if such a natural operation as integer multiplication had such a complicated expression for its running time. But even for $O(n \log n)$ there is no unconditional corresponding lower bound. Still, long ago there have been some remarkable attempts. In the algebraic model, Morgenstern [Mor73] has shown that every N-point Fourier transform done by just using linear combinations $\alpha a + \beta b$ with $|\alpha| + |\beta| \leq c$ for inputs or previously computed values a and b requires at least $(n \lg n)/(2 \lg c)$ operations. Under different assumptions on the computation graph, Papadimitriou [Pap79] and Pan [Pan86] have shown lower bounds of $\Omega(n \log n)$ for the FFT. Both are for the interesting case of n being a power of 2. Cook and Anderaa [CA69] have developed a method for proving non-linear lower bounds for on-line computations of integer products and related functions. Based on this method, Paterson, Fischer and Meyer [PFM74] have improved the lower bound for on-line integer multiplication to $\Omega(n \log n)$. Naturally, one would like to see unconditional lower bounds, as the on-line requirement is a very severe restriction. On-line means that starting with the least significant bit, the kth bit of the product is written before the $k+1$st bit of the factors are read.

Besides showing that our algorithm is more efficient in terms of circuit complexity and Turing machine time, one could reasonably ask how well it performs in terms of more practical complexity measures. Well, first of all, it is worthwhile pointing out that all currently competitive algorithms are nicely structured, and for such algorithms, a Turing machine model with an alphabet size of 2^w (where w is the computer word length) is actually a very realistic model as can be seen from the first implementation of fast integer multiplication (see [SGV94]).

Usually, the random access machine (RAM) is considered to be a better model for actual computing machines. But in the case of long arithmetic, such superiority is much in doubt. Indeed,

for random access machines, addition and multiplication have the same linear time complexity [Sch80] (see also Knuth [Knu98]). This complexity result is very impressive, as it even holds for Schönhage's [Sch80] pointer machines (initially called storage modification machines), which are equivalent to the rudimentary successor RAM.

Nevertheless, one could interpret this result as proving a deficiency of the random access machine model, as obviously in any practical sense long addition can be done much easier and faster than long multiplication. The linear time multiplication algorithm is based on the observation that during the multiplication of large numbers many products of small numbers are repeatedly needed and could be computed in an organized way and distributed to the places where they are needed. No practical implementation would do something like that, because multiplications of small numbers can be done fast on most computers.

A more reasonable, but also more complicated RAM-like complexity model might use an additional parameter, the word length w, with all reasonable operations (like input and multiplication) of length w numbers doable in one step. Then the complexity of addition would be $O(n/w)$, but we would expect multiplication to require more time. The complexities in terms of n and w could be reduced to single parameter complexities by focussing on a reasonable relation between n and w, which holds for a specific range of n and w of interest. For example, to have a good estimate for very large but still practical values of n, we could select $w = \Theta(\log n)$ or $w = \Theta(\log \log n)$, as the hidden constants would be small in this range. In any case, it should not be too hard to see that an implementation of our new algorithm with currently used methods could be quite competitive, even though this does not show up in the asymptotic complexity of the unit cost random access machine model.

In Section 2, we review the FFT in a way that shows which powers of ω are used for any recursive decomposition of the algorithm. In Section 3, we present a ring with many nice roots of unity allowing our faster FFT computation. In Section 4, we describe the new method of using the FFT for integer multiplication. It is helpful if the reader is familiar with the Schönhage-Strassen integer multiplication algorithm as described in the original, or e.g., in Aho, Hopcroft and Ullman [AHU74], but this is not a prerequisite. In Section 5, we study the precision requirements for the numerical approximations used in the Fourier transforms. In Section 6, we state the complexity results, followed by open problems in Section 7.

2 The Fast Fourier Transform

The N-point discrete Fourier transform (DFT) is the linear function, mapping the vector $\boldsymbol{a} = (a_0, \ldots, a_{N-1})^{\mathsf{T}}$ to $\boldsymbol{b} = (b_0, \ldots, b_{N-1})^{\mathsf{T}}$ by

$$\boldsymbol{b} = \Omega \boldsymbol{a}, \text{ where } \Omega = (\omega^{jk})_{0 \le j,\, k \le N-1}$$

with a principal Nth root of unity ω. Hence, the discrete Fourier transform maps the vector of coefficients $(a_0, \ldots, a_{N-1})^{\mathsf{T}}$ of a polynomial

$$p(x) = \sum_{j=0}^{N-1} a_j x^j$$

of degree $N-1$ to the vector of values

$$(p(1), p(\omega), p(\omega^2), \ldots, p(\omega^{N-1}))^{\mathsf{T}}$$

at the Nth roots of unity.

The inverse Ω^{-1} of the discrete Fourier transform is $1/N$ times the discrete Fourier transform

with the principal Nth root of unity ω^{-1}, because

$$\sum_{j=0}^{N-1} \omega^{-ij}\omega^{jk} = \sum_{j=0}^{N-1} \omega^{(k-i)j} = N\delta_{ik}$$

Here, we use the Kronecker δ defined by

$$\delta ik = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases}$$

If the Fourier transform is done over the field $\mathbb{C}$, then $\omega^{-1} = \bar{\omega}$, the complex conjugate of ω. Therefore, the discrete Fourier transform scaled by $1/\sqrt{N}$ is a unitary function, and $\frac{1}{\sqrt{N}}\Omega$ is a unitary matrix.

Now let $N = JK$. Then $\omega^{JK} = 1$. We want to represent the N-point DFT as a set of K parallel J-point DFT's (inner DFT's), followed by scalar multiplications and a set of J parallel K-point DFT's (outer DFT's). The inner DFT's employ the principal Jth root of unity ω^K, while the outer DFT's work with the principal Kth root of unity ω^J. Hence, most powers of ω used during the transformation are powers of ω^J or ω^K. Only the scalar multiplications in the middle are by "odd" powers of ω. This general recursive decomposition of the DFT has in fact been presented in the original paper of Cooley and Tukey [CT65], but might have been widely forgotten since. Any such recursive decomposition (even for $J = 2$ or $K = 2$) results in a fast algorithm for the DFT and is called "fast Fourier transform" (FFT). At one time, the FFT has been fully credited to Cooley and Tukey, but the FFT has appeared earlier. For the older history of the FFT back to Gauss, we refer the reader to [HJB84].

Here, we are only interested in the usual case of N being a power of 2. Instead of j and k with $0 \leq j, k \leq N-1$, we use $j'J+j$ and $k'K+k$ with $0 \leq j, k' \leq J-1$ and $0 \leq j', k \leq K-1$. Any textbook presenting the Fourier transformation recursively would use either $K = 2$ or $J = 2$.

For $0 \leq j \leq J-1$ and $0 \leq j' \leq K-1$, we define

$$\begin{aligned} b_{j'J+j} &= \sum_{k=0}^{K-1}\sum_{k'=0}^{J-1} \omega^{(j'J+j)(k'K+k)} a_{k'K+k} \\ &= \underbrace{\sum_{k=0}^{K-1} \omega^{Jj'k} \quad \omega^{jk} \quad \underbrace{\sum_{k'=0}^{J-1} \omega^{Kjk'} a_{k'K+k}}_{\text{inner (first) DFT's}}}_{\text{coefficients of outer DFT's}} \end{aligned}$$

$$\text{outer (second) DFT's}$$

For N being a power of 2, the fast Fourier transforms (FFT's) are obtained by recursive application of this method until $J = K = 2$.

We could apply a *balanced* FFT with $J = K = \sqrt{N}$ or $J = 2K = \sqrt{2N}$ depending on N being an even or odd power of 2. But actually, we just require that the partition is not extremely unbalanced.

3 The Ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$

An element ζ in a ring $\mathcal{R}$ is a *principal* mth root of unity if it has the following properties.

1. $\zeta \neq 1$
2. $\zeta^m = 1$
3. $\sum_{j=0}^{m-1} \zeta^{jk} = 0$, for $1 \leq k < m$

We consider the set of polynomials $\mathcal{R}[y]$ over the ring $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$. In all applications,

we will assume P to be a power of 2. For a primitive $2P$th root of unity ζ in $\mathbb{C}$, e.g., $\zeta = e^{i\pi/P}$, we have

$$\begin{aligned}\mathcal{R} &= \mathbb{C}[x]/(x^P+1) \\ &= \mathbb{C}[x]/\prod_{j=0}^{P-1}(x-\zeta^{2j+1}) \\ &\cong \bigoplus_{j=0}^{P-1}\mathbb{C}[x]/(x-\zeta^{2j+1})\end{aligned}$$

$\mathcal{R}$ contains an interesting root of unity, namely x. It is a very desirable root of unity, because multiplication by x can be done very efficiently.

Lemma 1 *For P a power of 2, the variable x is a principal $2P$th root of unity in the ring $\mathcal{R} = \mathbb{C}[x]/(x^P+1)$.*

Proof Property 1 is trivial. Property 2 follows, because in $\mathcal{R}$ the equation $x^P = -1$ holds. Property 3 is shown as follows.

Let $0 \le k = (2u+1)2^v < 2P = m$. This implies $2^v \le P$, and $k/2^v$ is an odd integer.

$$\begin{aligned}\sum_{j=0}^{2P-1} x^{jk} &= \sum_{i=0}^{2^{v+1}-1}\sum_{j=0}^{P/2^v-1} x^{(iP/2^v+j)k} \\ &= \sum_{j=0}^{P/2^v-1} x^{jk} \underbrace{\sum_{i=0}^{2^{v+1}-1} \underbrace{x^{iPk/2^v}}_{(-1)^i}}_{0} \\ &= 0\end{aligned}$$

Alternatively, because $\mathcal{R}$ is isomorphic to $\mathbb{C}^P$, one can argue that x is a principal root of unity in $\mathcal{R}$, if and only if it is a principal root of unity in every factor $\mathbb{C}[x]/(x-\zeta^{2j+1})$. But $x \bmod (x-\zeta^{2j+1})$ is just ζ^{2j+1}, which is a principal root of unity in $\mathbb{C}$. ∎

There are many principal Nth roots of unity in

$$\mathcal{R} = \mathbb{C}[x]/(x^P+1)$$

One can choose an arbitrary primitive Nth root of unity in every factor

$$\mathbb{C}[x]/(x-\zeta^{2j+1})$$

independently. We want to pick one such Nth root of unity

$$\rho(x) \in \mathcal{R} = \mathbb{C}[x]/(x^P+1)$$

with the convenient property

$$\rho(x)^{N/2P} = x$$

Let σ be a primitive Nth root of unity in $\mathbb{C}$, with

$$\sigma^{N/2P} = \zeta$$

e.g.,

$$\sigma = e^{2\pi i/N}$$

Now we select the polynomial

$$\rho(x) = \sum_{j=0}^{P-1} r_j x^j$$

such that

$$\rho(x) \equiv \sigma^{2k+1} \pmod{x-\zeta^{2k+1}}$$

for $k = 0, 1, \ldots, P-1$, i.e., σ^{2k+1} is the value of the polynomial $\rho(x)$ at ζ^{2k+1}. Then

$$\rho(x)^{N/2P} \equiv \sigma^{(2k+1)N/2P} = \zeta^{2k+1} \equiv x \pmod{x-\zeta^{2k+1}}$$

for $k = 0, 1, \ldots, P-1$, implying

$$\rho(x)^{N/2P} \equiv x \pmod{x^P+1}$$

For the FFT algorithm, the coefficients of $\rho(x)$ could be computed from Lagrange's interpolation formula without affecting the asymptotic running time, because we will select $P = O(\log N)$.

$$\rho(x) = \sum_{k=0}^{P-1} \sigma^{2k+1} \frac{\prod_{j \neq k}(x - \zeta^{2j+1})}{\prod_{j \neq k}(\zeta^{2k+1} - \zeta^{2j+1})}$$

In both products, j ranges over $\{0, \ldots, P-1\} - \{k\}$. The numerator in the previous expression is

$$\frac{x^P + 1}{x - \zeta^{2k+1}} = -\sum_{j=0}^{P-1} \zeta^{-(j+1)(2k+1)} x^j$$

for P being even. This implies that in our case, all coefficients of each of the additive terms in Lagrange's formula have the same absolute value. We show a little more, namely that all coefficients of $\rho(x)$ have an absolute value of at most 1.

Definition: The *l_2-norm* of a polynomial $p(x) = \sum a_k x^k$ is $||p(x)|| = \sqrt{\sum |a_k|^2}$.

Our FFT will be done with the principal root of unity $\rho(x)$ defined above. In order to control the required numerical accuracy of our computations, we need a bound on the absolute value of the coefficients of $\rho(x)$. Such a bound is provided by the l_2-norm $||\rho(x)||$ of $\rho(x)$.

Lemma 2 *The l_2-norm of $\rho(x)$ is $||\rho(x)|| = 1$.*

Proof Note that the values of the polynomial $\rho(x)$ at all the Pth roots of unity are also roots of unity, in particular complex numbers with absolute value 1. Thus the vector v of these values has l_2 norm $\sqrt{P}$. The coefficients of $\rho(x)$ are obtained by the inverse discrete Fourier transform $\Omega^{-1}v$. As $\sqrt{P}\Omega^{-1}$ is a unitary matrix, the vector of coefficients has norm 1. ∎

Corollary 1 *The absolute value of every coefficient of $\rho(x)$ is at most 1.* ∎

4 The Algorithm

In order to multiply two non-negative integers of length $n/2$ each, we encode them as polynomials of $\mathcal{R}[y]$, where $\mathcal{R} = \mathbb{C}[x]/(x^P + 1)$. We multiply these polynomials with the help of the Fourier transform as follows. Let $P = \Theta(\log n)$ be rounded to a power of 2. The binary integers to be multiplied are decomposed into (large) pieces of length $P^2/2$. Again, each such piece is decomposed into small pieces of length P. If $a_{i\,P/2-1}, \ldots, a_{i0}$ are the small pieces belonging to a common big piece a_i, then they are encoded as

$$\sum_{j=0}^{P-1} a_{ij} x^j \in \mathcal{R} = \mathbb{C}[x]/(x^P + 1)$$

with $a_{i\,P-1} = a_{i\,P-2} = \cdots = a_{i\,P/2} = 0$. Thus each large piece is encoded as an element of $\mathcal{R}$, which is a coefficient of a polynomial in y.

These elements of $\mathcal{R}$ are themselves polynomials in x. Their coefficients are integers at the beginning and at the end of the algorithm. The intermediate results, as well as the roots of unity are polynomials with complex coefficients, which themselves are represented by pairs of reals that have to be approximated numerically. In Section 5, we will show that it is sufficient to use fixed-point arithmetic with $O(P) = O(\log n)$ bits in the integer and fraction part.

Now every factor is represented by a polynomial of $\mathcal{R}[y]$. An FFT computes the values of such a polynomial at those roots of unity which are powers of the Nth root of unity $\rho(x) \in \mathcal{R}$. The values are multiplied and an inverse FFT produces another polynomial of $\mathcal{R}[y]$. From this

polynomial the resulting integer product can be recovered by just doing some additions. The relevant parts of the coefficients have now grown to a length of $O(P)$ from the initial length of P. (The constant factor growth could actually be decreased to a factor $2 + o(1)$ by increasing the length P of the small pieces from $O(\log n)$ to $O(\log^2 n)$, but this would only affect the constant factor in front of $\log^*$ in the exponent of the running time.)

Thus the algorithm runs pretty much like that of Schönhage and Strassen [SS71] except that the field $\mathbb{C}$ has been replaced by the ring $\mathcal{R} = \mathbb{C}[x]/(x^P+1)$, and the FFT is decomposed more evenly. The standard decomposition of the N-point FFT into two $N/2$-point FFT's and many 2-point FFT's would not allow an improvement. Nevertheless, there is no need for balancing completely. Instead of recursively decomposing the $N = \Theta(\frac{n}{\log^2 n})$-point FFT in the middle (in a divide-and-conquer fashion), we decompose into $2P$-point FFT's and $N/(2P)$-point FFT's. This is not important, as in either of the two cases, only about every $\log P$th level of the overall FFT requires complicated multiplications with difficult roots of unity. At all the other levels, multiplications are with roots of unity which are powers of x. Multiplication with these roots of unity are just cyclic rotations of coefficients of elements of $\mathcal{R}$ (with sign change on wrap around).

We use the auxiliary functions Decompose (Figure 1) and Compose (Figure 2). "Decompose" takes a binary number a of length $NP^2/2$ and decomposes it into $NP/2$ pieces a_{ij} ($0 \le i < N$ and $0 \le j < P/2$) of length P each, to serve as coefficients of polynomials from $\mathcal{R}$. More precisely, first a is decomposed into N pieces $a_{N-1}, a_{N-2}, \ldots, a_0$ of length $P^2/2$. Then each a_i is decomposed into $P/2$ pieces $a_{i\,P/2-1}, a_{i,\,P/2-2}, \ldots, a_{i\,0}$ of length P each. The remaining a_{ij} (for $0 \le i < N$ and $P/2 \le j < P$) are defined to be 0. This padding allows to properly recover the integer product from the product of the polynomials. In other words, we have

$$a_i = \sum_{j=0}^{P-1} a_{ij} 2^{jP} \tag{1}$$

and

$$a = \sum_{i=0}^{N-1} a_i 2^{iP^2/2} = \sum_{i=0}^{N-1} \sum_{j=0}^{P-1} a_{ij} 2^{i(P^2/2)+jP} \tag{2}$$

with

$$0 \le a_{ij} < 2^P \quad \text{for all } i, j \tag{3}$$

and

$$a_{ij} = 0 \quad \text{for } N/2 \le i < N \text{ or } P/2 \le j < P \tag{4}$$

Finally, a_i defines the polynomial $\alpha_i \in \mathcal{R}$ by

$$\alpha_i = \sum_{j=0}^{P-1} a_{ij} x^j = a_{i0} + a_{i1}x + a_{i2}x^2 + \cdots + a_{i\,P-1}x^{P-1} \tag{5}$$

Thus "Decompose" produces a normal form where the padding defined by Equation (4) is designed to avoid any wrap around modulo $2^P + 1$ when doing multiplication in $\mathcal{R}$. "Compose" not only reverses the effect of "Decompose", but it works just as well for representations not in normal form, as they occur at the end of the computation.

The procedure Select(N) (Figure 3) determines how the FFT is recursively broken down, corresponding to a factorization $N = JK$. Schönhage has used Select(N) = 2, Aho, Hopcroft and Ullman use Select(N) = $N/2$, a balanced approach is obtained by Select(N) = $2^{\lfloor (\lg N)/2 \rfloor}$. We choose Select($N$) = $2P$, which

Procedure Decompose:

Input: Integer a of length at most $NP^2/4$ in binary, N, P (powers of 2)

Output: $\boldsymbol{a} \in \mathcal{R}^N$ encoding the integer a

Comment: The integer a is the concatenation of the a_{ij} for $0 \le i < N$ and $0 \le j < P/2$ as binary integers of length $P/2$ defined by Equations (1), (2) and (4), and Inequalities (3). $a_{i0}, a_{i1}, \ldots, a_{i\,P-1}$ are the coefficients of $a_i \in \mathcal{R}$. $a_0, a_1, \ldots, a_{N-1}$ are the components of $\boldsymbol{a} \in \mathcal{R}^N$ as defined by Equation (5).

for $i = 0$ **to** $N/2 - 1$ **do**
  for $j = 0$ **to** $P/2 - 1$ **do**
    $a_{ij} = a \bmod 2^P$
    $a = \lfloor a/2^P \rfloor$
  for $j = P/2$ **to** $P - 1$ **do**
    $a_{ij} = 0$
for $i = N/2$ **to** $N - 1$ **do**
  for $j = 0$ **to** $P - 1$ **do**
    $a_{ij} = 0$
  $a_i = a_{i0} + a_{i1}x + a_{i2}x^2 + \cdots + a_{i\,P-1}x^{P-1}$
Return $\boldsymbol{a}$

Figure 1: The procedure Decompose

Procedure Compose:

Input: $\boldsymbol{a} \in \mathcal{R}^N$, N, P (powers of 2)

Output: Integer a encoded by $\boldsymbol{a}$

Comment: $a_0, a_1, \ldots, a_{N-1}$ are the components of a vector $\boldsymbol{a} \in \mathcal{R}^N$. For all i, j, a_{ij} is the coefficient of x^j in a_i. The integer a is obtained from the rounded a_{ij} as defined in Equation (2).

round all a_{ij} to the nearest integer
$a = 0$
for $j = P - 1$ **downto** $P/2$ **do**
  $a = a \cdot 2^P + a_{N-1\,j}$
for $i = N - 1$ **downto** 1 **do**
  for $j = P/2 - 1$ **downto** 0 **do**
    $a = a \cdot 2^P + a_{ij} + a_{i-1\,j+P/2}$
for $j = P/2 - 1$ **downto** 0 **do**
  $a = a \cdot 2^P + a_{0j}$
Return a

Figure 2: The procedure Compose

is slightly better than a balanced solution (by a constant factor only), because only every $2P$th operation (instead of every ith for some $P < i \le 2P$) requires expensive multiplications with powers of ω.

With the help of these auxiliary procedures, we can now describe the main algorithm in Figure 4. In order to do Integer-Multiplication we employ various simple functions (Figure 5), as well as the previously presented procedures Decompose and Compose. Integer-Multiplication (Figure 6) is based on the three major parts: FFT (Figure 7) for both factors, Componentwise-Multiplication (Figure 8), and Inverse-FFT (Figure 9). The crucial part, FFT, is presented as a recursive algorithm for simplicity and clarity. It uses the auxiliary procedure Select (Figure 3). FFT, Componentwise-Multiplication, and Inverse-FFT, all use the operation $*$ (Figure 10), which is the multiplication in the ring $\mathcal{R}$. This operation is implemented by integer multiplications, which are executed by recursive calls to the algorithm Integer-Multiplication (Figure 6).

We compute the product of two complex polynomials by first writing each as a sum of a real and an imaginary polynomial, and then computing four products of real polynomials. Alternatively, we could achieve the same result with three real polynomial multiplications based on the basic idea of [KO62]. Real polynomials are multiplied by multiplying their values at a good

Procedure Select:

Input: $N \geq 4$ (a power of 2), P (a power of 2)
Output: $J \geq 2$ (a power of 2 dividing $N/2$)
Comment: The procedure selects J such that the N-point FFT is decomposed into J point FFT's followed by $K = N/J$-point FFT's.
if $N \leq 2P$ **then** Return 2 **else** Return $2P$

Figure 3: The procedure Select determining the recursive decomposition

power of 2 as proposed by Schönhage [Sch82]. A good power of 2 makes the values not too big, but still pads the space between the coefficients nicely such that the coefficients of the product polynomial can easily be recovered from the binary representation of the integer product. Schönhage [Sch82] has shown that this can easily be achieved with a constant factor blow-up. Actually, he proposes an even better method for handling complex polynomials. He does integer multiplication modulo $2^N + 1$ and notices that $2^{N/2}$ can serve as the imaginary unit i. We don't further elaborate on this method, as it only affects the constant factor in front of $\log^*$ in the exponent of the running time.

5 Precision for the FFT over $\mathcal{R}$

We will compute Fourier transforms over the ring $\mathcal{R}$. Elements of $\mathcal{R}$ are polynomials over $\mathbb{C}$ modulo $x^P + 1$. The coefficients are represented by pairs of reals with fixed precision for the real and imaginary part. We want to know the numerical precision needed for the coefficients of these polynomials. We start with integer coefficients. After doing two FFT's in parallel, and multiplying corresponding values followed by an inverse FFT, we know that the result has again integer

The Structure of the Complete Algorithm:

Comment: Lower level (indented) algorithms are called from the higher level algorithms. In addition, the Operation $*$ recursively calls Integer-Multiplication.

Integer-Multiplication
  Decompose
  FFT
    Select
  Componentwise-Multiplication
  Inverse-FFT
    Operation $*$ (= Multiplication in $\mathcal{R}$)
  Compose

Figure 4: Overall structure of the multiplication algorithm

Various functions:
lg: the log to the base 2
length: the length in binary
round: rounded up to the next power of 2

Figure 5: Various functions

coefficients. Therefore, the precision has to be such that at the end the absolute errors are less than $\frac{1}{2}$. Hence, a set of final rounding operations provably produces the correct result.

The N-point FFT's and the subsequent N-point inverse FFT are consisting of $2 \lg N + 2$ levels (Figure 11). The bottom level $-\lg N - 1$ is the input level. The FFT's proceed from level $-\lg N - 1$ to level -1. They are followed by the inverse FFT form level 0 to level $\lg N$.

Let $\mathcal{R}_\ell$ be the set of elements of $\mathcal{R}$ occurring at level ℓ. At every level ℓ from $-\lg N$ to $\lg N$, except for level 0, each entry $\gamma \in \mathcal{R}_\ell$ is obtained as a linear combination $\gamma = \tau(\alpha \pm \beta)$ with $\alpha, \beta \in \mathcal{R}_{\ell-1}$ being two entries of the level below, and

Algorithm Integer-Multiplication:

Input: Integers a and b in binary

Output: Product $d = ab$

Comment: n is defined as twice the maximal length of a and b rounded up to the next power of 2. The product $d = ab$ is computed with Fourier transforms over $\mathcal{R}$. Let ω be the Nth root of unity in $\mathcal{R}$ with value $e^{2\pi ik/N}$ at $e^{2\pi ik/P}$ (for $k = 0, \ldots, P-1$). Let n_0 be some constant. For $n \leq n_0$, a trivial algorithm is used.

$n = \mathsf{round}(2\max(\mathsf{length}(a), \mathsf{length}(b)))$
if $n \leq n_0$ **then** Return ab
$P = \mathsf{round}(\lg n)$
$N = \mathsf{round}(2n/P^2)$
$\boldsymbol{f} = \mathsf{FFT}(Decompose(a), \omega, N, P)$
$\boldsymbol{g} = \mathsf{FFT}(Decompose(b), \omega, N, P)$
$\boldsymbol{h} = \mathsf{Componentwise\text{-}Multiplication}(\boldsymbol{f}, \boldsymbol{g}, N, P)$
$\boldsymbol{d} = \mathsf{Inverse\text{-}FFT}(\boldsymbol{h}, \omega, N, P)$
Return $Compose(\boldsymbol{d})$

Figure 6: The Algorithm Integer-Multiplication

the multiplier $\tau \in \mathcal{R}$ being a root of unity in $\mathcal{R}$. Level 0 is special. Corresponding output entries of the two FFT's from $\mathcal{R}_{-1}$ are multiplied to form the inputs to the inverse FFT.

The relatively weak Lemma 3 is sufficient for our theorems. We could obtain slightly tighter results based on the fact that Fourier transforms over $\mathbb{C}$ (and their inverses) are unitary transformations up to simple scaling factors. This would produce a better constant factor in front of $\log^* n$ in the exponent.

We distinguish two fast Fourier transforms. Besides the main N-point FFT over $\mathcal{R}$, we also consider a Fourier transform operating on the coefficients of a single element of $\mathcal{R}$.

Algorithm FFT:

Input: $\boldsymbol{a} \in \mathcal{R}^N$, $\omega \in \mathcal{R}$ (an Nth root of unity), N, P (powers of 2),

Output: $\boldsymbol{b} \in \mathcal{R}^N$ the N-point DFT of the input

Comment: The N-point FFT is the composition of J-point inner FFT's and K-point outer FFT's. We use the vectors $\boldsymbol{a}$, $\boldsymbol{b} \in \mathcal{R}^N$, $\boldsymbol{c}^k \in \mathcal{R}^J$ $(k = 0, \ldots, K-1)$, and $\boldsymbol{d}^j \in \mathcal{R}^K$ $(j = 0, \ldots, J-1)$.

if $N = 1$ **then** Return $\boldsymbol{a}$
if $N = 2$ **then** $\{b_0 = a_0 + a_1; b_1 = a_0 - a_1$; Return $\boldsymbol{b}\}$
$J = \mathsf{Select}(N, P); K = N/J$
for $k = 0$ **to** $K-1$ **do**
  for $k' = 0$ **to** $J-1$ **do**
    $c^k_{k'} = a_{k'K+k}$
  $\boldsymbol{c}^k = \mathsf{FFT}(\boldsymbol{c}^k, \omega^K, J)$ //inner FFT's
for $j = 0$ **to** $J-1$ **do**
  for $k = 0$ **to** $K-1$ **do**
    $d^j_k = c^k_j * \omega^{jk}$
  $\boldsymbol{d}^j = \mathsf{FFT}(\boldsymbol{d}^j, \omega^J, K)$ //outer FFT's
  for $j' = 0$ **to** $K-1$ **do**
    $b_{j'J+j} = d^j_{j'}$
Return $\boldsymbol{b}$

Figure 7: The algorithm FFT

Algorithm Componentwise-Multiplication:

Input: $\boldsymbol{f}, \boldsymbol{g} \in \mathcal{R}^N$, N, P (powers of 2)

Output: $\boldsymbol{h} \in \mathcal{R}^N$ (the componentwise product of $\boldsymbol{f}$ and $\boldsymbol{g}$)

for $j = 0$ **to** $N-1$ **do**
  $h_j = f_j * g_j$

Figure 8: The algorithm Componentwise-Multiplication

Definition: *HDFT (half DFT)* is the linear function mapping the vector of coefficients $\boldsymbol{a}$ to

Algorithm Inverse-FFT:

Input: $\boldsymbol{a} \in \mathcal{R}^N$, $\omega \in \mathcal{R}$ (an Nth root of unity), N (a power of 2)

Output: $\boldsymbol{b} \in \mathcal{R}^N$ (the inverse N-point DFT of the input)

Return $\frac{1}{N}\,\mathsf{FFT}(\boldsymbol{a}, \omega^{-1}, N)$

Figure 9: The algorithm Inverse-FFT

Procedure Operation * (= Multiplication in $\mathcal{R}$):

Input: $\alpha, \beta \in \mathcal{R}$, P (a power of 2)

Output: γ (the product $\alpha \cdot \beta \in \mathcal{R}$)

Comment: Compute the product of the two polynomials first by writing each as a sum of a real and an imaginary polynomial. Then compute the 4 products of polynomials by multiplying their values at a good power of 2, which pads the space between the coefficients nicely such that the coefficients of the product polynomial can easily be recovered from the binary representation of the integer product.

details omitted

Figure 10: The multiplication in $\mathcal{R}$ (= operation *)

the vector of values of the polynomial $\sum_{i=0}^{P-1} a_i x^i$ at the P primitive $2P$th roots of unity.

HDFT could be computed by extending $\boldsymbol{a}$ with $a_P = \cdots = a_{2P-1} = 0$ and doing a DFT, but actually only about half the work is needed.

During the N-point FFT over $\mathcal{R}$ the elements of $\mathcal{R}$ are represented by their vector of coefficients $\boldsymbol{a}$. Nevertheless, to bound the size of these coefficients, we argue about the size of the components of HDFT($\boldsymbol{a}$). Initially, for every element of $\mathcal{R}$ involved in the input to the N-point FFT,

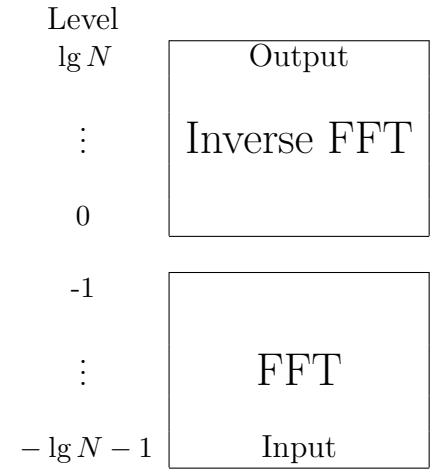

Figure 11: The computation proceeds from the bottom level $-\lg N - 1$ to the top level $\lg n$

the coefficients form a vector $\boldsymbol{a}$ consisting of P non-negative integers less than 2^P. As the P-point Fourier transform is $\sqrt{P}$ times a unitary mapping, the l_2-norm of HDFT($\boldsymbol{a}$) is at most $P(2^P - 1)$.

We proceed with a bound on the precision without aiming at the best constant factor.

Lemma 3 *$O(\log^2 n)$ precision is sufficient for the recursive call to Integer-Multiplication.*

Proof We start by estimating the absolute values of the coefficients of all the ring elements involved in the FFT. Our goal is a simple estimate of the absolute values of any coefficient. Still, the most obvious estimates would be quite imprecise. They would only be sufficient for our purpose if we changed the algorithms slightly by selecting a larger $P = \Omega(\log N \log\log N)$ instead of $P = O(\log N)$. This would not be a problem, as we would still get the same type of bound on the complexity.

Level	Value bound	Absolute error bound
$-\lg N - 1$	$P(2^P - 1)$	0
$-\lg N - 1 + \ell$	$P(2^P - 1)2^\ell$	$P^2 2^P 2^{2\ell} 2^{-S}$ for $\ell = 1, \ldots, \lg N$
-1	$P(2^P - 1)N$	$P^2 2^P N^2 2^{-S}$
0	$P^2(2^P - 1)^2 N^2$	$2P^4 2^{2P} N^3 2^{-S}$
ℓ	$P^2(2^P - 1)^2 N^2 2^\ell$	$2P^4 2^{2\ell} 2^{2P} N^3 2^{-S}$ for $\ell = 0, \ldots, \lg N - 1$
$\lg N$	$P^2(2^P - 1)^2 N^2$	$2P^4 2^{2P} N^4 2^{-S}$

Table 1: Bounds on absolute values and errors

Instead, we use a simple trick. We first estimate not the absolute values of the coefficients of $\boldsymbol{a} \in \mathcal{R}$, but the absolute values of the components of HDFT($\boldsymbol{a}$) for all elements $\boldsymbol{a}$ occurring in the FFT. For all levels, these bounds are given in Column 2 of Table 1. The bounds are easily computed row by row. At level $-\lg N - 1$, we have the obvious bound. At the negative levels, we notice that during an addition or subtraction not only the coefficients, but also the Half Fourier transformed values, i.e., the values at primitive $2P$th roots of unity, are just added or subtracted. Thus all these entries can at most double in absolute value. The advantage of focussing on the Fourier transformed values is that during the subsequent multiplication with a root of unity from $\mathcal{R}$, the absolute values remain fixed.

As the coefficients of elements $\boldsymbol{a} \in \mathcal{R}$ are obtained by an inverse HDFT (half Fourier transform), they actually satisfy the same upper bounds as the components of HDFT($\boldsymbol{a}$). So the entries in Column 2 of Table 1 can be viewed as bounds on the coefficients of $\mathcal{R}_\ell$. Just the simple argument to produce the bound for level $\ell+1$ from the bound of level ℓ does no longer work for the coefficients. At level $-\lg N - 1$, the trivial upper bound on the coefficient is actually better, but we don't use that fact.

In order to obtain error bounds, we have a closer look at the operations performed at each level. On most levels, every entry is obtained by an addition or subtraction, a multiplication with a root of unity from $\mathcal{R}$, and in the last level also a division by N. But most of the multiplications with roots of unity from $\mathcal{R}$ are nothing else but cyclic permutations of the coefficients. Thus, regarding coefficients of $\mathcal{R}$, we usually do just additions and subtractions, which produce no new errors.

The error bounds for the absolute values of the coefficients from each level are presented in Column 3 of Table 1. The factor 2^{-S} comes from assuming we store the coefficients of elements of $\mathcal{R}$ with a precision of S bits after the binary point. It is easy to choose a precise value for $S = O(\log N)$ such that the last entry in Column 3 is less than $1/2$. Then all entries are less than $1/2$.

Each entry in Column 3 is obtained from the error bound of the previous level and the absolute value bound of the current level. Additions and subtractions are straightforward. The multiplication is somewhat tricky if one wants to be exact. We handle it according to the following scheme.

Let u and v be the values whose product $w = uv$ we want to approximate. Instead of u and

v, we know $u + e_u$ and $v + e_v$ where e_u and e_v are small errors. In addition we have real upper bounds U, V, E_U, E_V with $|u| \leq U - 2$, $|v| \leq V$, $|e_u| \leq E_U \leq 1$, and $|e_v| \leq E_V$. Assume the multiplication is done with a rounding error e_{mult} with $|e_{\text{mult}}| \leq E_V$ due to the use of fixed-point arithmetic. Then the error e_w is defined by the equation

$$w + e_w = e_{\text{mult}} + (u + e_u)(v + e_v)$$

implying

$$e_w = e_{\text{mult}} + (u + e_u)e_v + ve_u$$

Hence, we obtain the following bound on $|e_w|$

$$|e_w| \leq UE_V + VE_U$$

We apply this method as follows. Usually, $U - 2$ is the bound on the absolute value of the previous level and E_u is its error bound, while $V = 1$ is the bound on the coefficients of roots of unity of $\mathcal{R}$. We select $E_V = E_U/U$. The additions and subtractions cause a doubling of the error bound per level in the FFT and Inverse-FFT. Multiplications with roots of unity from $\mathcal{R}$ occur every $\lg P + 1$ level. Each new coefficient in $\mathcal{R}$ is the sum of P products of previous such coefficients. This results in an error bound of

$$(UE_V + VE_U)P \leq (UE_U/U + E_U)P = 2PE_U$$

Instead of increasing the error bound E_U by a factor of $2P$ at once, we just increase it by a factor of 2 per level. Combined with the factor of 2 increase due to additions and subtractions, we obtain a factor 4 increase per level. Level 0 is special as multiplication is with two elements of $\mathcal{R}$, neither of them being a root of unity. Here $U = V$ is the value bound, and $U_U = E_V$ is the error bound at level -1. The new error bound is then

$$(UE_V + VE_U)P \leq 2UE_UP$$

Finally, at level $\lg N$ the bounds on the absolute value and the error decrease by a factor of N due to the $1/N$ factor in front of the Inverse-FFT.

Aiming at the somewhat strange bound of $U - 2$ on $|u|$ explains why we insist on writing $(2^P - 1)$ in the Value bound column. A nice bound U is then obtained easily from the generous inequality $(2^P - 1)^2 \leq 2^{2P} - 2$. ∎

6 Complexity

Independently of how an N-point Fourier transform is recursively decomposed, the computation can always be visualized by the well known butterfly graph with $\lg N + 1$ rows. Every row represents N elements of the ring $\mathcal{R}$. Row 0 represents the input, row N represents the output, and every entry of row $j + 1$ is obtained from row j $(0 \leq j < N)$ by an addition or subtraction and possibly a multiplication with a power of ω. When investigating the complexity of performing the multiplications in $\mathcal{R}$ recursively, it is best to still think in terms of the same $\lg N + 1$ rows. At the next level of recursion, N multiplications are done per row. It is important to observe that the sum of the lengths of the representations of all entries in one row grows just by a constant factor from each level of recursion to the next. The blow-up by a constant factor is due to the padding with 0's, and due to the precision needed to represent numerical approximations of complex roots of unity. Padding with 0's occurs when reducing multiplication in $\mathcal{R}$ to integer multiplication and during the procedure Decompose.

We do $O(\log^* n)$ levels of recursive calls to Integer-Multiplication. As the total length of a row grows by a constant factor from level to level, we obtain the factor $2^{O(\log^* n)}$ in the running time. For all practical purposes, $\log^* n$ in the exponent of the running time actually represents something like $\log^* n - 3$ which could be thought of as being 2 or 1, because at a low level one would switch to school multiplication.

The crucial advantage of our new FFT algorithm is the fact that most multiplications with powers of ω can be done in linear time, as each of them only involves a cyclic shift (with sign change on wrap around) of a vector of coefficient representing an element of $\mathcal{R}$. Indeed, only every $O(\log\log N)$th row of the FFT requires recursive calls for non-trivial multiplications with roots of unity. We recall that our Fourier transform is over the ring $\mathcal{R}$, whose elements are represented by polynomials of degree $P-1$ with coefficients of length $O(P) = O(\log N)$. Therefore, we get the main result on the running time of our algorithms Integer-Multiplication and FFT.

Lemma 4 *The boolean circuit complexity $T(n)$ of the algorithm Integer-Multiplication and the running time $T'(N)$ of the algorithm FFT fulfill the following recurrence equations for some constants c_1, c_2, c_3, c_4.*

$$T(n) \leq \begin{cases} c_1 & \text{if } n < c_2 \\ O(T'(n/\log^2 n)) & \text{otherwise} \end{cases}$$

$$T'(N) \leq \begin{cases} c_3 & \text{if } N < c_4 \\ O\left(N\log^3 N + \frac{N\log N}{\log\log N} T(O(\log^2 N))\right) & \\ & \text{otherwise} \end{cases}$$

Proof The factor $\log^2 n$ in the first recurrence equation is due to the fact that the FFT is done over $\mathcal{R}$, and elements of $\mathcal{R}$ encode $P^2/2 = \Theta(\log^2 n)$ bits. In the first term of the second recurrence equation, one factor $\log N$ is coming from the $\log N + 1$ levels of the FFT, while the remaining factor $\log^2 N$ represents the lengths of the entries being elements of the ring $\mathcal{R}$. There are $O((\log N)/\log\log N)$ levels with non-trivial multiplications implemented by recursive calls to integer multiplications of length $O(\log^2 N)$. This explains the second term of second recurrence equation. ∎

Lemma 5 *The previous recurrence equations have solutions of the form*

$$T(n) = n\log n\, 2^{O(\log^* n)}$$

and

$$T'(N) = N\log^3 N\, 2^{O(\log^* N)}$$

Proof Combining both recurrence equations, we obtain

$$\begin{aligned} T(n) &= O(T'(n/\log^2 n)) \\ &= O\left(n\log n + \frac{n}{\log n\log\log n}\, T(O(\log^2 n))\right) \end{aligned}$$

The iteration method produces a geometrically increasing sum of $\log^* n - O(1)$ terms, the first being $O(n\log n)$. ∎

The lemma implies our main results for circuit complexity and (except for the organizational details) for multitape Turing machines.

Theorem 1 *Multiplication of binary integers of length n can be done by a boolean circuit of size $n\log n\, 2^{O(\log^* n)}$.*

Theorem 2 *Multiplication of binary integers of length n can be done in time $n\log n\, 2^{O(\log^* n)}$ on a 2-tape Turing machine.*

Detailed proofs of Theorem 2 would be quite tedious. Nevertheless, it should be obvious that due to the relatively simple structure of the algorithms, there is no principle problem to implement them on Turing machines.

As an important application of integer multiplication, we obtain corresponding bounds for polynomial multiplication by boolean circuits or Turing machines. We are looking at bit complexity, not assuming that products of coefficients can be obtained in one step.

Corollary 2 *Products of polynomials of degree less than n, with an $O(m)$ upper bound on the absolute values of their real or complex coefficients, can be approximated in time $mn \log mn\, 2^{O(\log^* mn)}$ with an absolute error bound of 2^{-m}, for a given $m = \Omega(\log n)$.*

Proof Schönhage [Sch82] has shown how to reduce the multiplication of polynomials with complex coefficients to integer multiplication with only a constant factor in time increase. ∎

Indeed, multiplying polynomials with real or complex coefficients is a major area where long integer multiplication is very useful. Long integer multiplication is extensively used in some areas of number theory like testing the Riemann hypothesis for millions of roots of the zeta function and in particular for finding large prime numbers.

7 Open Problem

Besides the obvious question whether integer multiplication is in $O(n \log n)$, a multiplication algorithm running in time $O(n \log n \log^* n)$ would also be very desirable. Furthermore, it would be nice to have an implementation that compares favorably with current implementations of the algorithm of Schönhage and Strassen. The asymptotic improvement form $O(n \log n \log \log n)$ to $n \log n 2^{O(\log^* n)}$ might suggest that an actual speed-up only shows up for astronomically large numbers. Indeed, the expressions are not very helpful to judge the performance, because the constants hidden in the O-notation might well be comparable with $\log \log n$ as well as with $2^{\log^* n}$ for reasonable values of n. Looking at the algorithms themselves, we conjecture that a noticeable practical improvement is possible for numbers n in the tens of millions. Such multiplications are routinely done in primality testing algorithms.

References

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[CA69] S. A. Cook and S. O. Aanderaa, *On the minimum computation time of functions*, Transactions of the AMS **142** (1969), 291–314.

[CT65] J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex fourier series*, Math. of Comput. **19** (1965), 297–301.

[Für89] Martin Fürer, *On the complexity of integer multiplication, (extended abstract)*, Tech. Report Technical report CS-89-17, Department of Computer Science, The Pennsylvania State University, 1989.

[HJB84] M. T. Heideman, D. H. Johnson, and C. S. Burrus, *Gauss and the history of the FFT*, IEEE Acoustics, Speech, and Signal Processing **1** (1984), 14–21.

[Knu98] Donald E. Knuth, *The art of computer programming, Volume 2, Seminumerical algorithms*, third ed., Addison-Wesley, Reading, MA, USA, 1998.

[KO62] Anatolii Karatsuba and Yu Ofman, *Multiplication of multidigit numbers on automata*, Doklady Akademii Nauk SSSR **145** (1962), no. 2, 293–294, (in Russian). English translation in Soviet Physics-Doklady 7, 595-596,1963.

[Mor73] Jacques Morgenstern, *Note on a lower bound on the linear complexity of the fast Fourier transform*, Journal of the ACM **20** (1973), no. 2, 305–306.

[Pan86] Victor Ya. Pan, *The trade-off between the additive complexity and the asynchronicity of linear and bilinear algorithms*, Information Processing Letters **22** (1986), no. 1, 11–14.

[Pap79] Christos H. Papadimitriou, *Optimality of the fast Fourier transform*, Journal of the ACM **26** (1979), no. 1, 95–102.

[PFM74] Michael S. Paterson, Michael J. Fischer, and Albert R. Meyer, *An improved overlap argument for on-line multiplication*, Tech. Report 40, Project MAC, MIT, January 1974.

[Sch66] Arnold Schönhage, *Multiplikation großer Zahlen*, Computing **1** (1966), no. 3, 182–196 (German).

[Sch80] ———, *Storage modification machines*, SIAM J. Comput. **9** (1980), no. 3, 490–508.

[Sch82] ———, *Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coeficients*, Computer Algebra, EUROCAM '82, European Computer Algebra Conference, Marseille, France, 5-7 April, 1982, Proceedings (Jacques Calmet, ed.), Lecture Notes in Computer Science, vol. 144, Springer, 1982, pp. 3–15.

[SGV94] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter, *Fast algorithms: A Turing machine implementation*, B.I. Wissenschaftsverlag, Mannheim-Leipzig-Wien-Zürich, 1994.

[SS71] Arnold Schönhage and Volker Strassen, *Schnelle Multiplikation grosser Zahlen*, Computing **7** (1971), 281–292.

[Too63] Andre L. Toom, *The complexity of a scheme of functional elements simulating the multiplication of integers*, Dokl. Akad. Nauk SSSR **150** (1963), 496–498, (in Russian). English translation in Soviet Mathematics 3, 714-716, 1963.