An implementation of Schönhage's multiplication algorithm (or how to compute the square of a number with one million digits on your workstation in less than one minute)

Paul Zimmermann

May 13, 1992

Abstract

This report describes an implementation of a fast multiplication algorithm proposed by A. Schönhage [5]. The algorithm performs the multiplication of two integers modulo a number of the form $2^N + 1$ in $O(N \log N \log\log N)$ operations. Using the BigNum package [2], we wrote a C program of less than 350 lines that performs both the modular and integer multiplication. We give detailed timings and comparisons with the naive method and Karatsuba's algorithm on two particular machines, a DecStation 5000 and an IBM RS 6000.

1 Introduction

The Fast Fourier Transform (FFT) is a well-known tool to reduce the theoretical complexity of algorithms, typically to transform a $O(n^2)$ algorithm into a $O(n \log n)$ one. But so far, only a few people published an implementation of an algorithm using the FFT, and studied values of n for which the "fast" algorithm was really better.

In the field of integer multiplication, two papers describe such implementations. In 1986, A. Schönhage encoded his algorithm on a virtual multitape Turing processor called TP, which was itself implemented on a Motorola MC68000 [6]. In 1990 [7], M. Scott described an implementation of an other algorithm due to Pollard [4]. Schönhage finds that the break even point with Karatsuba's method is close to 720 digits for the modular multiplication, but does not give any break even point for integer multiplication. Scott concludes that the FFT method is faster for multiplying numbers of at least 1152 decimal digits long; but in his implementation, the number of words must be an exact power of two. Another implementation was made by Finkel, Guibas and Simonyi around 1974 in 987 lines of SAIL, a dialect of Algol 60, but no results were published [1].

We consider here an algorithm designed by Arnold Schönhage for the fast multiplication of large integers. The main idea is to use the Fast Fourier Transform, but unlike other algorithms [3, pages 290–295] that work with complex floating point numbers, here all operations are done in a finite field, namely the residues modulo $2^N + 1$:

Algorithm FastMultMod(k):
Input: two residues n and m modulo $2^N + 1$
Output: the product nm modulo $2^N + 1$
Necessary condition. $K = 2^k$ must divide N: let $N = KM$.

Initialization. Choose $N' \geq 2M + k + 3$ such that K divides N': $N' = KM'$.
Decomposition. Cut n and m in K parts of M bits, and store these parts, considered as residues modulo $2^{N'} + 1$, in two arrays $A_0 \ldots A_{K-1}$ and $B_0 \ldots B_{K-1}$.

Preparation. In both arrays, multiply the i-th element by $2^{iM'}$.
Direct FFTs. Compute the FFT of order K of each array, with $2^{2M'}$ as K-th root of unity.
Term to term multiplication. Multiply A_i by B_i modulo $2^{N'}+1$, and store the result in A_i.
Inverse FFT. Compute the inverse FFT of order K of A, with $2^{2M'}$ as K-th root of unity.
Normalization. Define $C_0 = A_0$, $C_{K-1} = A_1$, $C_{K-2} = A_2$, ..., $C_1 = A_{K-1}$; divide C_j by $2^{k+jM'}$.
Adjust signs. If $C_j > (j+1)2^{2M}$, subtract $2^{N'}+1$ to it (consider it now as an integer).
Additions. The product nm equals $\sum C_j 2^{jM}$ modulo 2^N+1.

We have first encoded this algorithm in the Maple computer algebra language, where all multiprecision operations are easy to do. The resulting Maple program is the following:

```
\input{fft.ltg}
```

2 Results and conclusion

In this section, we present our results and a real example where our implementation enables a large saving of computer time: verify the primality of the Mersenne number $2^{756839}-1$. First we compare the algorithm FastMultMod with other methods for the multiplication modulo 2^N+1. Then we say a few words about the recursive use of the algorithm (multiplication modulo $2^N+1 \longrightarrow$ multiplication modulo $2^{N'}+1 \longrightarrow$ multiplication modulo $2^{N''}+1 \longrightarrow \cdots$). Finally we consider integer multiplication and give some figures for verifying the primality of the Mersenne number $2^{756839}-1$.

2.1 Multiplication modulo 2^N+1

Like Arnold Schönhage, to obtain a better efficiency of the algorithm, we restrict ourselves to integers N that are multiples of the number L of bits in a machine word (usually $L=32$). In this way, residues modulo 2^N+1 are encoded by $n = N/L$ words and an extra word that contains only one bit. In fact, the operations on residues (addition, subtraction, multiplication by a power of two) do not keep them completely normalized, that is $0 \le r \le 2^N$, but only *semi-normalized*, that is $0 \le r < 2^{N+1}$, or in other words the most significant word is at most 1. The fact that the residues are semi-normalized explains the additional term of 3 in the initialization step of the algorithm (N' must be greater than $2M+k+3$). If the residues were completely normalized, $2M+k+1$ would be enough.

Figure 1 shows the time to compute the square of a residue modulo $2^{32w}+1$ (the residue is stored in $w+1$ words of 32 bits each) on a DecStation 5000/200. The graph corresponds to $1 \le w \le 400$, that is numbers up to 3854 decimal digits. The six curves represent the naive algorithm, Karatsuba's algorithm, and Schönhage's algorithm for $k=3$, 4, 5 and 6. It is worth to note that for $k>5$, the condition that 2^k divides $N=32w$ implies that the algorithm FastMultMod can only be used for w being a multiple of 2^{k-5}, for example w even for $k=6$, a multiple of 4 for $k=7$, and so on. We see on the graph that for $w>200$, FastMultMod(3) and FastMultMod(4) are faster than the naive algorithm, like FastMultMod(5) for $w>250$ and FastMultMod(6) for $w>330$. We remark that FastMultMod(3) is nowhere faster than all other methods, and thus should never be used on this machine.

Several interesting facts appear from figure 1. First the curve corresponding to Karatsuba's algorithm is not smooth, nor monotonous. This is because this algorithm behaves differently if the length (number of words) of the integers to be multiplied is odd. It is faster for even length. For example, for $w=254$ (Karatsuba is called with length $w+1=255$), the squaring time is .0127 second, whereas for $w=255$ ($w+1=256$), it is only .0114 second.

Secondly we see that each version of Schönhage's algorithm has several stages. For example, for $k=5$, there is a first stage from $w=1$ to $w=12$, then a second stage from $w=13$ to $w=28$, a third from $w=29$ to $w=44$, and so on. The reason for these stages is the choice of N', which must be a multiple of $K=2^k$ greater or equal than $2M+k+3$, and a multiple of $L=32$ for efficiency reasons. Thus for $k \le 5$, the first stage corresponds to $N'=L=32$, the second to $N'=2L$, the third to $N'=3L$, ..., and the length of each

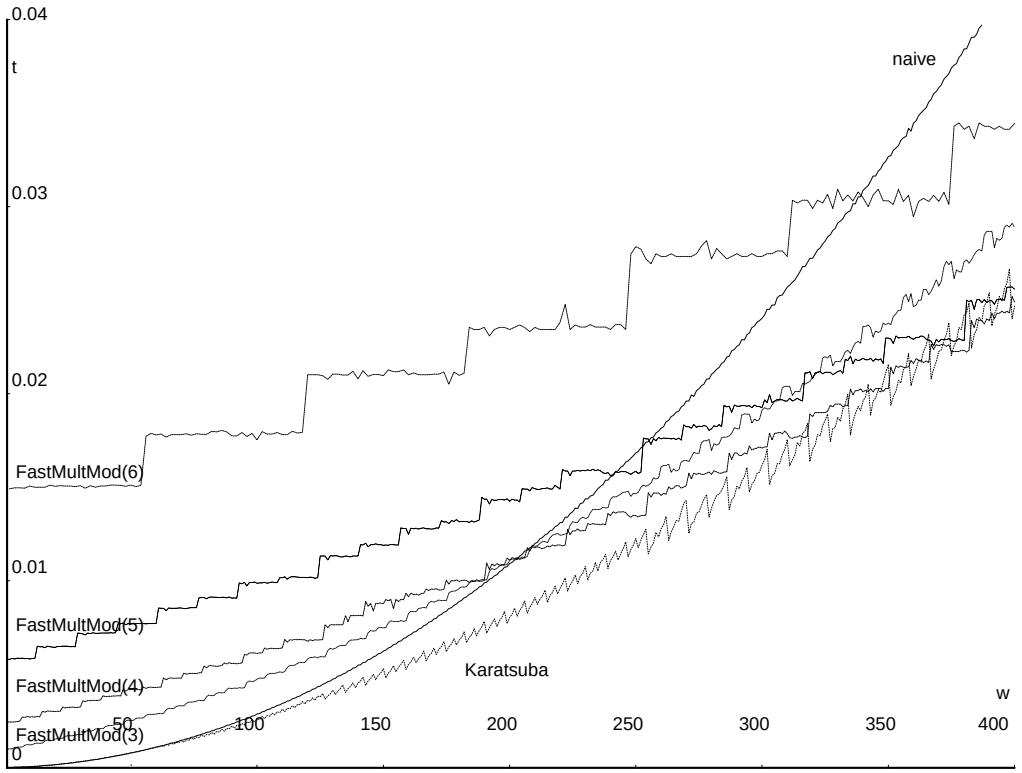

Figure 1: Time in seconds to square a number of w words modulo $2^{32w} + 1$ on a DS 5000/200.

Dec 5000	
number of words	fastest method
1–62	naive
63–369	Karatsuba
370–449	FastMultMod(4)
450–1039	FastMultMod(5)
1040–2166	FastMultMod(6)
2167–5612	FastMultMod(7)
5613–11216	FastMultMod(8)
11217–12192	FastMultMod(9)
12193–14288	FastMultMod(8)
14289–16288	FastMultMod(9)
16289–18384	FastMultMod(8)
18385–28576	FastMultMod(9)
28577–32544	FastMultMod(10)
32545–40864	FastMultMod(9)
40865–48928	FastMultMod(10)
48929–53152	FastMultMod(9)
53153–98080	FastMultMod(10)
98081–130624	FastMultMod(11)

Figure 2: Which method choose for squaring a residue modulo $2^{32w}+1$?

stage is 2^{k-1} words. For $k > 5$, the first stage corresponds to $N' = K$, the second to $N' = 2K$, the third to $N' = 3K$, ..., and the length of each stage is 2^{2k-6} words.

Again it must be understand that these conclusions heavily depend on the particular processor we consider (figure 2). The reason is simple: fast methods like Karatsuba or FastMultMod(k) replace N^2 multiplications by a lower number of operations, including multiplications of course, but additions and other operations too. Thus the break even point between these methods depends on the time ratio between word multiplication and other word operations.

2.2 Recursive version of FastMultMod

For $w = 212768$, the fastest method according to figure 2 is FastMultMod(10). This leads to 1024 squares modulo $2^{32\cdot 416}+1$. For $w = 416$, the fastest method is FastMultMod(4), thus each square modulo $2^{32\cdot 416}+1$ is computed by the means of 16 squares modulo $2^{32\cdot 53}+1$, which are done with the naive algorithm. This example is a two-step recursive use of Schönhage's algorithm, which is done in less than one minute on a DecStation 5000/200. Thus we can compute the square of a 1024792-digit number in less than one minute on such a computer.

2.3 Integer multiplication

To compute the *exact* product p of two integers n and m, we determine their product p' modulo 2^N+1. If N is large enough and p' is normalized ($0 \le p' \le 2^N$), then $p = p'$. In practice, if n is l_n words long and m l_m words long, we choose $N \ge \lceil \frac{l_n+l_m}{L} \rceil$, where L is the number of bits in one word, usually $L = 32$.

Unfortunately, this gives a big overhead for integer multiplication using Schönhage's algorithm. For example, if n and m have same length N, a modular multiplication of length $2N$ must be done, whereas other methods (naive, Karatsuba) make only an integer multiplication of length N. This explains that the break even points for integer multiplication are higher than those for modular multiplication.

As an example, on a DecStation 5000, the break even point between Karatsuba's and Schönhage's algorithm is at 2000 words, that is about 20000 decimal digits.

2.4 The primality of $2^{756839} - 1$

As shown in the previous section, in what concerns the *integer* multiplication, our implementation of Schönhage's algorithm on a Dec5000 is faster than Karatsuba's one for numbers of 10000 digits and more. Such large integers are not very common: they are mainly encountered by people that try to compute more and more digits of π. In the field of number theory, one deals with small integers: at most 200 digits for integer factorization, and at most 2000 digits for (general purpose) primality testing. One exception is the primality testing of special kinds of integers, for example Mersenne numbers. For these integers of the form $P = 2^p - 1$, there exists a very simple test of primality. First define the sequence

$$u_0 = 4, u_{n+1} = u_n^2 - 2 \bmod P;$$

then P is prime if and only if $u_{p-2} = 0 \bmod P$. Thus the test requires only $p - 2$ squarings modulo P.

The last discovered Mersenne prime is $2^{756839} - 1$. To check its primality, we therefore have to compute 756837 squares modulo $2^{756839} - 1$, which is a number of 227832 digits, or 23652 words of 32 bits. With FastMultMod(10), each iteration is reduced to a square modulo $2^{47328} + 1$, that takes less than 6 seconds of cpu time on a DS5000, instead of 182 seconds with the naive algorithm.

	Dec 5000		IBM RS 6000	
method	one iteration	total estimated time	one iteration	total estimated time
naive	181 seconds	4 years 4 months	140 seconds	3 years 4 months
Karatsuba	17.5 seconds	5 months	13.3 seconds	4 months
FastMultMod(10)	5.9 seconds	52 days	3.2 seconds	28 days

As shown by the above table, the primality test would thus take about two months, instead of more than four years!

Implementation in C

```
\input{schon.bignum.ltg}
```

References

[1] G. Brassard, S. Monet, and D. Zuffellato. Algorithmes pour l'arithmétique des très grands entiers. *Technique et Science Informatiques*, 5(2):89–102, 1986.

[2] J-C. Hervé, François Morain, David Salesin, Bernard-P. Serpette, J. Vuillemin, and P. Zimmermann. BigNum : un module portable et efficace pour une arithmétique à précision arbitraire. Rapport de recherche 1016, Inria, avril 1989. 23 pages. Disponible aussi en rapport Dec PRL (2).

[3] D. E. Knuth. *The Art of Computer Programming*, volume 2 : Seminumerical Algorithms. Addison-Wesley, deuxième edition, 1981.

[4] J. M. Pollard. The Fast Fourier Transform in a Finite Field. *Mathematics of Computation*, 25(114):365–374, April 1971.

[5] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In *Computer Algebra, EUROCAM'82*, Lecture Notes in Computer Science, pages 3–15, 1982.

[6] Arnold Schönhage. Tapes versus pointers, a study in implementing fast algorithms. *Bulletin of the EATCS*, (30):23–32, 1986.

[7] M. Scott. An Implementation of the Fast-Fourier Multiplication Algorithm. Technical Report CA-0790, Dublin City University, 1990.