QuickMul: Practical FFT-based Integer Multiplication

Chee Yap and Chen Li
Department of Computer Science
Courant Institute, New York University
email: {yap,chenli}@cs.nyu.edu

October 6, 2000

1 Introduction

The use of arithmetic packages for arbitrarily large integers is growing in many areas of application, beyond their traditional applications which is mainly in computer algebra. For instance, the BigInteger class library that is considered a standard part of the popular Java programming language. One recent area of application is in robust geometric algorithms (e.g., [4, 2]). The critical algorithm in all these applications is the integer multiplication algorithm. The fastest known algorithm here is due to Schönhage and Strassen (1971) [5], achieving the time bound

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

for multiplying two N-bit integers. In this note, we are interested in exploring integer multiplication algorithms which, like the Schönhage-Strassen algorithm, are based on the Fast Fourier Transform (FFT). These algorithms may not achieve the record bound (1), but has order $T(N) = O(N \log^{O(1)} N)$ with small implicit constants. The hope is that the small implicit constants may make such algorithms more efficient for practical values of N. The original Schönhage-Strassen algorithm is relatively complex; a simplified algorithm is given in [8, chap. 1] with time bound $O(N \log^{1+\epsilon} N)$ for any ϵ. Such simplified algorithms are quite easy to implement (as this paper will show). Because such simplifications are not well-known, many important big number packages continue to avoid FFT-multiplications algorithms. Two notable exceptions among the freely available packages are David Bailey's `MPFUN` package (written in `Fortran`) [1] and Bruno Haible's `CLN` (written in `C++`) [3]. For instance, only recently (August 2000) did the widely used `Gnu gmp` package implement FFT-multiplication.

2 The One-Prime FFT Multiplication

The present paper implements an FFT-multiplication algorithm described in [7]. The basic idea is to perform the FFT in the ring $\mathbb{Z}_M = \{0, 1, \ldots, M-1\}$ of numbers modulo M. Here M is specially chosen prime number. It was suggested in [7] that such an algorithm should have "small" constants, but input numbers must be less than some finite but very large limit. This finite limitation plus small constants is what we mean[1] by "practical" in the title. Typically, small constants imply that the algorithm is simple. In particular, some optimization steps in the Schönhage-Strassen algorithm will be discarded. We specify choose two "game rules" help to further ensure small constants: (1) assume a 32-bit machine and (2) insist that the multiplication algorithm is non-recursive.

Rule (1) means that we want M at most 32-bits so that arithmetic in $\mathbb{Z}_M$ can be performed in $O(1)$ time. Rule (2) does not imply that we do not use recursion at all – indeed, the FFT algorithm which we will use is inherently recursive. However, the recursion in FFT algorithms is relatively simple and do not incur large overhead. So, what does rule (2) really exclude? A simplified FFT-based multiplication algorithm has

[1]For inputs larger than this limit, one can proceed in any number of ways. But we will not discuss this possibility as our limit seems quite adequate for most applications.

the following basic steps. Assume we are given two N-bit integers U and V, and we want to compute their produce $W = UV$.

I. Preparation Break up U and V into two K-vectors $\overline{U}$ and $\overline{V}$ in $(\mathbb{Z}_M)^K$ (for suitable natural numbers K and M).

II. DFT Computation Compute their discrete Fourier transforms, $DFT(\overline{U}), DFT(\overline{V}) \in (\mathbb{Z}_M)^K$.

III. Component-wise Product Compute the component-wise product of $DFT(\overline{U})$ and $DFT(\overline{V})$. This results in a vector in $(\mathbb{Z}_M)^K$ which we may denote as $DFT(\overline{W})$.

IV. Inverse DFT Compute the inverse discrete Fourier transform (IDFT) of $DFT(\overline{W})$ to obtain $\overline{W}$. Basically, $\overline{W}$ is the convolution of $\overline{U}$ and $\overline{V}$.

V. Re-assembly It is relatively straightforward to re-assemble $W = U \cdot V$ from $\overline{W}$.

Normally, the Component-wise Product step requires recursive calls to the multiplication algorithm. But rule (2) forbids this. In combination with rule (1), this says that M is at most 32-bits so that the component-wise product can be done in $O(1)$ machine operations. In a certain sense, such an algorithm is only "$O(N \log N)$" since the most complex of the above steps takes $O(N \log N)$ time. Moreover, the implicit constants in this big-Oh is small, as noted before. On the other hand, there is no real asymptotics here: our insistence on no recursion implies a finite limit on N. What is this limit?

2.1 Computation Modulo $M = 2013265921$

Our goal is to compute in $\mathbb{Z}_M = \{0, 1, \ldots, M-1\}$. We choose $M = 2,013,265,921$, a prime number with only 31 bits. Let us now indicate how large N can be with this choice of M. In the field $\mathbb{Z}_M$, we need primitive K-th roots of unity. Moreover, for the recursion to proceed, we need K to be a power of 2. We can express M as

$$M = 2^{27}m + 1$$

where $m = 15$. In the following, we write $a \equiv b$ to mean $a - b \equiv 0 (\bmod M)$ and write $(a)_M$ for the value $(a \bmod M) \in \mathbb{Z}_M$. The number 31 turns out to be a primitive element of $\mathbb{Z}_M$ in the sense that $\{(31^i)_M : i = 0, 1, \ldots, M-2\} = \{1, 2, \ldots, M-1\}$. Setting

$$\omega := (31^m)_M = 440564289,$$

we have $(\omega^i)_M = (31^{mi})_M \neq 1$ for $i = 1, 2, 3, \ldots, 2^{27} - 1$. But $\omega^{2^{27}} = 31^{M-1} \equiv 1$, by Fermat's little theorem. This proves ω is a 2^{27}-th primitive root of unity.

Using ω, we can implement the FFT algorithm on vectors of length $K = 2^{27}$ in $\mathbb{Z}_M$. To implement the FFT, we compute everything mod M. At the level of machine instructions, we can multiply $U, V \in \mathbb{Z}_M$ as to normal integers at double precision, and then reduce the result mod M. Alternatively, to avoid double precision machine arithmetic, we can split a number $U \bmod M$ into two parts (U_1, U_0), each part with at most 16 bits. Then we can multiply two such numbers with 3 single-precision multiplications (as in Karatsuba's method). The implementation of FFT is relatively standard, and we will not discuss this further.

2.2 Fast Multiplication

We break up an N-bit integer U into K integers each with L-bits, where

$$N = KL.$$

We view this as a $(2K)$-vector, after padding with K additional zeros. Let

$$\overline{U} = (0, 0, \ldots, 0, U_{K-1}, U_{K-2}, \ldots, U_0).$$

Similarly, let

$$\overline{V} = (0, 0, \ldots, 0, V_{K-1}, V_{K-2}, \ldots, V_0).$$

Let us estimate the largest value of N which can be achieved under rules (1) and (2).

Suppose we compute the $2K$-vector $\overline{W} = (W_{2K-1}, \ldots, W_0)$ as outlined above. Since $\overline{W}$ is the convolution of $\overline{U}$ and $\overline{V}$, it is easy to see that each W_i is the sum of $\leq K$ numbers, each with $\leq L$ bits. Thus W_i has $\leq 2L + \lg(K)$ bits. [Note: lg is logarithm to base 2.] So we need

$$2L + \lg(K) \leq \lg M = 30.9\ldots.$$

Since L and $\lg(K)$ are integers, we have have $2L + \lg(K) \leq 30$. Clearly $N = KL$ is maximized by making K as large as possible. Let N_{max} be this maximum value. Setting $L = 1$, we have $\lg(K) \leq (\lg M) - 2$ or $K \leq M/4 = 503,316,480$. Since K is a power of 2, we obtain $\lg(K) = \lfloor \lg(503,316,480) \rfloor = 28$ or $N_{max} = 2^{28}$. So this method works for integers up to 256 megabits or 32 MB long (counting "1024" rather than "1000" as one kilo).

In general, given two numbers each with $N \leq N_{max}$ bits, we choose positive K, L so as to minimize K (and hence maximize L) and subject to

$$KL \geq N, \qquad 2L + \lg(K) \leq \lg M.$$

The following table gives the upper bound on N for any $L = 1, 2, \ldots, 8$.

L	N upper bound		$\lg(K)$
8	131,072	$= 2^{17}$	14
7	458,752	$= 7 \cdot 2^{16}$	16
6	1,572,864	$= 3 \cdot 2^{19}$	18
5	5,242,880	$= 5 \cdot 2^{20}$	20
4	16,777,216	$= 2^{24}$	22
3	50,331,648	$= 3 \cdot 2^{24}$	24
2	134,217,728	$= 2^{27}$	26
1	268,435,456	$= 2^{28}$	28

For instance, for $N = 1,000,000$, we choose $L = 6$ and $K = 2^{18} = 262,144$. How fast can we multiply two such numbers? Well, FFT on $2K$-vectors takes $1.5(2K)\lg(2K)$ (properly implemented) arithmetic operations. This gives $3 \cdot 2^{18} \cdot 19$ or about 15 million floating point operations (flops). Counting a factor of f per flops, we have about $15f$ seconds on a megaflop machine. This factor f can be estimated for various platforms. For instance, if $f = 20$, this is 300 seconds.

3 Implementation of QuickMul

We have implemented our algorithm in `C++`, in a relatively straightforward implementation. This code and experimental results can be obtained from our website `http://cs.nyu.edu/exact/`. The following is a table comparing our algorithm `QMUL` (for "quickMul") to the latest Gnu's `gmp` (version 3.1), Bruno Haible's `CLN` (as incorporated into `LiDIA`). For historical[2] reasons, we also include a comparison with the original Gnu's big Integer. The platform for these tests is a "Sun UltraSPARC-IIi" 440 MHz CPU machine with 512MB main memory.

N	10000	50000	100000	200000	500000	1M	2M	5M	10M
QMUL	0.087	0.409	0.865	1.86	4.076	8.9	18.45	92.11	185.38
CLN	0.007	0.054	0.105	0.27	0.746	1.012	2.079	11.171	24.646
GMP	0.002	0.024	0.05	0.106	0.365	1.061	3.048	11.832	31.628
old GNU	0.033	0.801	3.244	13.012	82.134	331.51	1334.40	> 3600	–

Here, N is in bits and time is in seconds. This table shows that in the range $N = 50,000$ to 10M, our implemention is about 8 to 9 times slower than CLN and GMP 3.1. Also, CLN and GMP are comparable in the tested range, with CLN slowly gaining on and surpassing GMP after 1 million bits.

[2]It was used by our Core Library before version 1.3.

How well has QMUL stood up in this experiment? We note that QMUL is written in `C++` only, and in a relatively straightforward manner (basically, as outlined above). In contrast, both CLN and GMP exploit level-level optimized assembly code for critical parts, and perform specialized memory management.

Another difference between QMUL and both CLN and GMP is this: QMUL uses a single algorithm (as outlined above). CLN and GMP are hybrid algorithms in that they use some various multiplication algorithms for different ranges of N, invoking some FFT-based algorithms for the final range. For instance, CLN starts with an $O(N^2)$ method, then switches to Karatsuba's $O(N^{\log_2(3)}) = O(N^{1.58})$ algorithm for larger N. At $12,000$ digits, it switches over to Schonhage-Strassen's algorithm. These implies a fairly complicated code. Our program has less than 400 lines of `C++` code.

Considering all these factors, we may have reason to hope that a low-level tuning of our algorithm may compete directly with CLN and GMP. In particular, we note the experience of Serpette et al [6] in their implementation of another big number package, BigNum. Low-level optimization improved their speed about 10 fold.

Remarks: Our implementation above reflects an improvement of about 15% after we switched to a "non-recursive variant" of FFT. FFT is inherently recursive, so this just mean that we manage our own recursion in FFT calls. There are other "high-level" ideas that may further speedup our code. It may be interesting to exploit the increasingly available 64-bit architecture by choosing a larger M with similar properties, thus increasing the range of QuickMul.

References

[1] D. H. Bailey. Multiprecision translation and execution of Fortran programs. *ACM Trans. on Math. Software*, 19(3):288–319, 1993. URL for MPFUN software `http://science.nas.nasa.gov/Groups/AAA/db.webpage/mpdist/mpdist.html`.

[2] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. Exact geometric computation made easy. In *Proc. 15th ACM Symp. Comp. Geom.*, pages 341–450, 1999.

[3] CLN Homepage, 2000. CLN is a C++ library for arbitrary precision arithmetic and other elementary functions. In implements that Schönhage-Strassen multiplication algorithm. URL `http://clisp.cons.org/ haible/packages-cln.html/`.

[4] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A Core library for robust numerical and geometric libraries. In *15th ACM Symp. Computational Geometry*, pages 351–359, 1999. Download, `ftp://cs.nyu.edu/pub/local/yap/exact/core.ps.gz`.

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

[6] B. Serpette, J. Vuillemin, and J. Hervé. BigNum: a portable and efficient package for arbitrary-precision arithmetic. Research Report 2, Digital Paris Research Laboratory, May, 1989.

[7] C. Yap. Quickmul: Practical fast integer multiplication, July 1993. Preprint: `ftp://cs.nyu.edu/pub/local/yap/exact/quickMul.ps.gz`.

[8] C. K. Yap. *Fundamental Problems in Algorithmic Algebra.* Oxford University Press, 2000. A version is available at URL `ftp:/Preliminary/cs.nyu.edu/pub/local/yap/algebra-bk`.