(Extended Abstract)

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. The critical algorithm in all these applications is the integer multiplication algorithm. The fastest known algorithm here is due to Schönhage and Strassen (1971) [1], 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. Simplified versions are known and 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`) and Bruno Haible's `CLN` (written in `C++`). 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 which 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. 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 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 these basic steps. Assume we are given two N-bit integers U and V, and we want to compute their produce $W = UV$. After

[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.

breaking up U and V into a vector of smaller numbers (viewed as elements of ZZ_M), we compute their discrete Fourier transforms. Then, in the transform domain, we compute their component-wise product. This product is inverse transformed, and the final result W is assembled from the inverse transform. In general, the component-wise product step requires recursive calls to the multiplication algorithm. But rule (2) forbids this. This implies a finite limit on N. What is this limit?

Computation Modulo $M = 2013265921$. We choose $M = 2,013,265,921$, a prime number with only 31 bits. 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. Setting

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

we may verify that 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 $\mathbb{Z}_M$. We break up an N-bit integer U into K integers each with L-bits, where $N = KL$. The largest possible N is achieved when $L = 1$ and $N = K$. Let N_{max} be this maximum value: a little calculation shows $N_{max} = 2^{28}$. So our method works for integers up to 256 megabits or 32 MB long.

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 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.

How well has QMUL stood up in this experiment? We note that QMUL is written in `C++` only, and in a straightforward manner. In contrast, both CLN and GMP exploit level-level optimized assembly code for critical parts, and perform specialized memory management. 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, the experience of Serpette et al [2] in their implementation of another big number package, BigNum, suggests that low-level optimization can improve speed by 10 fold.

References

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

[2] 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.