When is FFT multiplication of arbitrary-precision polynomials practical?

Richard J. Fateman
University of California
Berkeley, CA 94720-1776

March 14, 2006

Abstract

It is well recognized in the computer algebra theory and systems communities that the Fast Fourier Transform (FFT) can be used for multiplying polynomials. Theory predicts that it is fast for "large enough" polynomials. Even so, it appears that no major *general-purpose* computer algebra system uses the FFT as a general default technique for this purpose. Some provide (optional) access to FFT techniques, so it is possible that a rational decision was made against using FFT. By contrast, the implementors of some more restricted systems, typically specializing in univariate, dense polynomials over finite or infinite fields, seem to think the FFT is good, in practice, even for smallish problems, and so they have made the decision the other way.

We provide some benchmarks for polynomial multiplication that the FFT should be good at, and also suggest an alternative version of FFT polynomial multiplication using floating-point numbers that has some nicer properties than the typical finite-field "Number Theoretic Transform" approach. In particular, in spite of the fact that a floating-point FFT gives *approximate* answers, we can nevertheless use it to produce EXACT answers to a class of polynomial multiplication problems for arbitrary-precision coefficient polynomials, simply by noting that a sufficiently-accurate coefficient, namely error is less than 0.5, we have the exact integer answer. We also have some more data on what "large enough" to warrant the use of FFT, might be.

1 Introduction

We were recently prompted to re-examine the practical possibilities of using the FFT for polynomial multiplication. We described, in a recent paper, our experiments using a kind of shorthand method for packing coefficients of polynomials into long integers or "bignums" and using efficient integer library routines to do polynomial multiplication [4]. The algorithm consisted in scanning the coefficients and with a simple linear-time computation finding a bound on the coefficients in the answer. This bound provides a "spacing" for packing and unpacking coefficients. We then called the "bignum" multiply on the packed numbers, and unpacked the result. This technique is sometimes called Kronecker substitution. We used the GMP [2](GNU Multiprecision number package) library routines. This package uses FFTs for bignum arithmetic when it deems this to be appropriate. In our earlier paper, we found some cases in which such an apparently roundabout method was indeed faster than the most obvious classical method, a method which takes $O(n^2)$ coefficient operations for two polynomials of degree $n-1$. (The coefficient operations are not constant-time: they depend on the lengths of the coefficients.)

Just because a new program runs faster than one other program does not mean it is the program of choice. We went back to our problems and thought about other cleverer polynomial methods, making the same kinds of restrictions we were forced to make in order to implement the bignum-GMP-FFT version. (Univariate, dense, similar-size inputs). In fact, with a little programming effort we could substantially improve on the classical method, and push the crossover point where the FFT finally is advisable, out some distance. In particular, our results showed that a "Karatsuba" style multiplication was a much better competitor for any but rather small cases, and therefore it should be compared to the FFT.

Comments from reviewers of that paper suggested that we push just a bit further and implement the FFT ourselves on the coefficients of the polynomial, rather than packing them into bignums. Wouldn't this be better? [1].

The answer: In principle, yes. Yet the point of that earlier paper *was to avoid writing and tuning a bignum polynomial FFT, and to just use someone else's carefully tuned program.*

We've written some FFT programs; frankly it is tedious to get all the details right to run *as efficiently as possible*, and an endless task to continue to refine such programs in the face of changes in computer architecture and memory implementation, as well as compiler software [2]. Compounding this is the realization that the FFT-based program is, according to both theory and practice, *expected to work poorly for what is probably the common case in computer algebra systems: sparse polynomials of modest degree in many variables.* This is probably why, among system builders, there appears to be low priority to write FFT routines. Certainly it was our own view, so rather than implement such a program, we let the paper sit for a while.

2 Theoretically fast multiplication

This paper is not going to describe in detail the fast multiplication algorithms or FFT; we presume the reader has some familiarity with them. For descriptions, see for example, Gathen and Gerhard [5], or (especially) Knuth [3].

In the list below, the algorithm we call Toom is the second of a possible sequence of variations of polynomial multiplication that can be devised by dividing the multiplication into subproblems [3]. Our Toom divides each input in three, and uses five multiplications of one-third the size. Its asymptotic growth rate for multiplying polynomials of degree n is about $O(n^{1.4649})$, where the exponent is really $\log_3(5)$. The better-known Karatsuba, which is Toom with "$r = 1$" [3], divides each input in two, and uses 3 multiplications of half the size. Its asymptotic growth rate for multiplying polynomials of degree n is about $O(n^{1.5849})$, where the exponent is $\log_2(3)$.

Exact polynomial multiplication, one variable, small "fixnum" integer coefficients in the input and output. *Asymptotically* they can be arranged from least efficient to most efficient this way:
1. Classical ($O(n^2)$)
2. Karatsuba ($O(n^{1.58})$), dropping to Classical for small problems
3. Toom ($O(n^{1.46})$), dropping to Karatsuba for small problems
4. FFT ($O(n \log n)$).

3 A new spark

E. Kaltofen, in a talk at ECCAD 2005, Ashland, Ohio, suggested that the FFT for polynomial multiplication was **quite practical even for modest size polynomials** and cited a paper by Victor Shoup, [9], in his assertion that even polynomials of degree less than 100 could be multiplied faster with FFT. Quoting that paper:

> "In all of the known factoring methods, a key operation is multiplication of polynomials over $\mathbf{F_p}$. We shall assume that multiplication of two degree d polynomials uses $O(M(d))$ scalar

[1](This kind of comment is sometimes called "Sure, your program works in practice, but does it work in theory?")
[2]http://www.fftw.org/benchfft/

operations, where $M(d) = d \log d \log \log d$. This running-time bound is attained using the Fast Fourier Transform or FFT (Aho *et al.* 1974), and (as we shall later see) is quite realistic in practice."

As we see, in Shoup's paper[3], the task being considered in this paragraph is polynomial multiplication *in a finite field,* not over the integers. This changes the ground-rules substantially. In Shoup's careful tests, his FFT becomes better than a traditional multiplication at degree 30. At degree 511, with 332-bit coefficients, (table 4 [8]) shows that it is some 24 times faster than Shoup's own implementation of classical arithmetic.

This is impressive. But as we have said, for the FFT to be "practical" at a given size and with a given coefficient bound it should be the fastest of all methods, not merely faster than the slowest[4].

In our own implementations in Common Lisp, We find that at size 512 (i.e. degree 511), the FFT does not win versus using a Karatsuba or Toom ([3] 4.3) algorithm or even a carefully-coded classical version, *optimized for use with fixed-precision arithmetic.* How could it be that Shoup sees a factor of 24X in favor of FFT, and for us it is behind!

To see if we are just not using a good enough FFT, we have also tried to interface NTL to Lisp[5]. If we use NTL, we cannot be accused of using a poor implementation of the FFT: we are using Shoup's, the same one cited previously. That is, we are timing calls directly to NTL's polynomial multiplication routines. It also matters substantially if the data must be translated to a different format for the FFT programs to work. This comes into play in two contexts below: if we must translate (small) integers to complex double-float machine numbers (cheap), or if we must translate (arbitrary precision) integers into high-precision quad-double (QD) software-implemented floating-point.

The times look like this for multiplying polynomials of size 512 with small coefficients, namely all 1's:

Method	Time (ms)	Notes
FFT	3.1	over double-floats, including conversion from ints
FFT	267	over QD, including conversion from ints
Karatsuba	2.3	
Classical	2.6	
NTL (in C)	7.8	(jumping to 12.1 ms if size is 513)

Now with 332 bit initial coefficients, as suggested by Shoup, the computer must do more work.

Method	Time (ms)	Notes
FFT using double	15	(but answer has wrong coefficients!)
FFT using double	9	(if 332-bit integers pre-converted to double)
FFT with 212 bits	485	(that is, using QD arithmetic []
FFT with 212 bits	284	(if 332-bit integers pre-converted to QD)
Karatsuba	438	[65% in bignum multiply] lim=100
Classical	985	[67% in bignum mult, 12% in add]
NTL (in C)	100	
NTL (in C)	183	(size is 513 instead of 512)

The floating-point double FFT gets the answer wrong for three reasons:

1. The coefficients in the inputs were substantially truncated to fit them into double-floats.

[3]Incidentally, a very nice blending of theory and implementation

[4]We have already given the FFT an enormous advantage by asking it only to do single-variable dense cases; sparse or multivariate programs are probably inefficient, viewed as a function of the number of non-zero terms

[5]these programs are posted on the author's web site "papers/lisp2ntl"directory.

2. Even if there were 332 available bits for input, the arithmetic would have to be done to higher precision than available with hardware floating point.

3. We would have to find sufficient space to represent the up-to 673-bit numbers in the answer.

The QD FFT also gets the wrong answer because even 212 bits is not enough here. Consequently, neither of the the first two versions of FFT are contenders.

Yet, if the FFT is done (say) 23 times over with 23 different inputs, modulo "single precision" primes and stitched together with the Chinese Remainder Algorithm (CRA), *we could get the right answer.* This would cost at least $15 \cdot 23 = 345$ ms (plus time for CRA). But if we wanted to do that, it seems that we would be better off using Karatsuba or even classical multiplication with small coefficients 23 times over, plus the CRA.

But then, how can we figure that the NTL polynomial multiplier program, which is *relatively slower for degree 512 with small coefficients*, is so much faster when the only change is that coefficients are larger? By our own measurements, Shoup's FFT is about 10 times faster (maybe really 5 times faster if you choose a "pessimal" rather than an "optimal" problem size) compared to classical, and about 4.4 times faster than Karatsuba (2.4 discounting the exact power of two).

What if we could beat down the time for integer multiplication by a factor of 3 or more for arithmetic multiplying two numbers of about size 332? This is entirely feasible by using better big arithmetic rather than our native Lisp's[6]. At a factor of 3, we get down to about 270 ms. After corresponding with NTL's author, Victor Shoup, we learned that native NTL arithmetic is not as fast as GMP arithmetic, and for best performance we should use GMP. We encountered (so far insurmountable) difficulties compiling NTL with GMP under Windows, but we compiled the NTL-5.4 + GMP 4.1.4 system using a version of Cygwin with GCC 3.4[7] We used the default configurations computed for a 933Mhz Pentium III and found that, over repeated trials of multiplying degree 1023 polynomials modulo a 1024-bit number, the version using GMP was on average about 40 percent faster than the NTL version. We can also use GMP arithmetic in our Lisp: we could also get about a 40 percent speedup using 332-bit coefficients for classical or Karatsuba multiplication.[8]

Our basic finding is that NTL is slower than Lisp for sufficiently small coefficients (fixnums), even if we spend time to discover that the coefficients are "fixnums". For (considerably) larger coefficients, our tests *eventually* agree with the predictions of algorithm complexity theory that the FFT should be faster. But even at degree 10,000, if the routines are compiled to know the types of input, the FFT advantage is only about a factor of 3 in speed versus the best of the others.

Turing to a standard reference in computer algebra algorithms, we looked at the figures 9.10 and 9.11 of Gathen/Gerhard [5] based on benchmarks of Shoup's NTL. They seem to suggest that even at degree 63, if coefficients are of 1024 bits, the "Fermat number FFT" wins; and if coefficients are of 64 bits, then the "modular FFT" wins over all other methods tested, at some rather low degree. It is not possible to be sure from their graph but it appears to be something less than degree 500. The alternatives include a Karatsuba-style multiplication routine as well as a "classical" $O(n^2)$ version for multiplying degree-n polynomials. As we have already remarked, any such comparison is strongly influenced by the cleverness of the implementations of the algorithms being compared, the details of the compiler and memory management used, the setting of parameters for optimization, and especially for large polynomials, the size of the cache relative to the size of the polynomials. We have seen that a number of the comparisons between runs on a Pentium 2 and Pentium

[6]The bignum package in Allegro CL 6.2 uses only 16 bit "bigits" in its arithmetic. Multiplication can be expected to be at least 4 times slower than a comparable system which uses 32 bit units, assuming that a 32×32 multiplier is available; even higher efficiency may be obtained with 53-bit fraction double-floats. This is without using any clever algorithms.

[7]The timing of some of these systems is very ticklish: setting different optimization parameters—and there are large numbers of them in NTL, GMP, and the compiler— some of which may affect times substantially. We compiled the system as suggested by the NTL documentation.

[8]In our tests of using Lisp + NTL+GMP, we got incorrect answers except on numbers with more than 250 bits, and so we are cautious about trying to make a direct comparison in exactly the same environment.

4 computer do not reflect the speedup of the ratio (2.5GHz / .933GHz) but rather the ratio of data cache sizes.

4 Review of FFT implementations

The usual way of running an FFT for a computer algebra algorithm is to use a different field from the (approximate, floating-point) complex numbers. Typically the computations use a finite field, e.g. the integers modulo a prime P. To perform an FFT on a sequence of length $n = 2^k$, a primitive nth root of unity ω_n must be available in the field. P is either chosen sufficiently large so as to bound the size of the coefficients in the answer, or a more complicated arrangement is set up, based on doing the FFT computation in some small number[9] of finite fields, that is, mod P_1, P_2, ... and then pasting the results together with the Chinese Remainder Algorithm.

These FFTs all produce exact integers along the way, rather than floats (in particular, ω_n looks like an integer between $-(P-1)/2$ and $(P-1)/2$ in a so-called "balanced representation" of the field), and arithmetic is integer arithmetic *where each operation requires a remainder computation* (remainder modulo P). This represents a substantial overhead. Of course for the complex float FFT, arithmetic is "complex float" requiring four "real float" operations (and slightly more work if unnecessary overflow is to be avoided). For computers where integers are 32 bits and double-floats are 64 bits, we are probably getting more work done faster from the floating-point processor, replacing the integer remainder calculation[10].

The FFT is (eventually) going to be fast. But it doesn't look like it is necessarily faster even for thousands of coefficients.

Are there ways of making it more practical for smaller sequences? It could be made better in several ways: making it more accurate in computation of complex roots of unity, making it faster by numerous strategies including memory allocation tricks as well as fitting the FFT to other (non power-of-two) sequence lengths. We could also use another version of the source code and use (for example) a Fortran compiler to try to optimize the machine code. (See the FFTW reference).

Given that the usual floating-point FFT that is the object of so much numerical interest gives approximate answers, can we use it to produce EXACT answers to multiplication problems for arbitrary-precision coefficient polynomials?

Can we use the information from the floating-point FFT to

- Inform us when the answer is right exactly.
- In case it is inaccurate, allow us to get the correct answer (with more work, but still fast).
- Do so with a guaranteed modest amount of work.

Consider that the IEEE double-precision fraction has 53 bits (52 bits plus sign, 11 bit exponent), compared to the 32 bits for an integer (on most machines), and given the possibility of several arithmetic units, we may have access to a better encoding of numbers, even integers, as parts of floats. Not all numbers can be represented by floats, e.g. numbers exceeding about 1.8×10^{308} exceed the range, and the encoding of the fraction gives something less than 17 decimal digits.

Some polynomial problems never need more than what can be comfortably represented in the fraction of double-floats. The exact integer answer can be obtained by coercing to the nearest integer. We must be able to guarantee that the number we are rounding has error less than half in the last place!

Sometimes though, the answer cannot be represented, e.g. r=123,456,789,123,456,789 requires 56 bits to represent to full accuracy, so it seems we cannot use the floating FFT for a polynomial with this coefficient.

NOT SO!

[9]T. Granlund suggests this strategy doesn't make sense for more than about 5 primes, in his experience.
[10]See the GMP Manual for a brief discussion of exact remainder by word-sized P.

Consider the primes $m_0 = 33554393$, $m_1 = 33444383$, and $m_2 = 33554371$, all less than 2^{25}, so their pairwise products fit comfortably[11] in 53 bits. We can represent $r \bmod m_0 = 27733531$ exactly as a double-float, and similarly for $r \bmod m_1 = 7390453$ and $r \bmod m_2 = 5709040$. We can use these numbers in three separate *floating-point FFTs*, and combine them using the Chinese Remainder Algorithm. For example, if we are squaring $(1 + 123456789123456789x)$, we would instead square $(1 + 27733531x)$, $(1 + 5709040x)$, etc. As long as these computations can be done exactly in a floating-point FFT, we are safe.

How did we know that three FFTs were needed? We compute a bound for the coefficients in the product of two polynomials. Based on this, as well as the comfort bound (see below) we can tell how many images we need. We can then modularly reduce the inputs so that the answer's coefficients can be represented correctly in the floating-point fraction. One coefficient bound for the answer is easy to compute. For polynomials p_1 and p_2, the bound is $\mathrm{maxcoef}(p_1)\cdot\mathrm{maxcoef}(p_2)\cdot(1+\min(\deg(p_1),\deg(p_2)))$. (Proof left as an exercise: consider squaring a polynomial with coefficients that are all 1).

How comfortable do we need to be? That is, how much head-room do we need to make sure the answer, rounded to the nearest integer, is correct? A substantial discussion can be found in Percival [6]. A simpler characterization can be found, however. According to the FFTW website[12], we can expect that accuracy is proportional to $\sqrt{\log(N)}$ for a length N transpose. Since the computed bounds tend to be somewhat pessimistic, we tried some experiments. We can choose a particular sequence size, and try to find the largest k such that an input polynomial p whose coefficient bound is 2^k, can be squared and the exact answer obtained, using only double-precision floats. This is what we found:

size	k
32	23
64	23
128	21
256	20
512	20
1024	19
2048	19
4096	18
8000	17
16000	17

That is, if you have a polynomial p of length 2048, whose coefficients are random integers less than 2^{19}, you can multiply p by p and get the *exact answer* by running the floating-point FFT procedure we tested, and rounding the coefficients in the answer to the nearest integer. We believe these numbers may be somewhat optimistic, and we would like to run more tests and analysis. The details will depend on the particular FFT implementation.

If the problem can't meet these bounds, there are several common approaches, and two more which we suggest below.

1. Don't believe the bound. Compute a better one. *We think this idea is new.*If the input coefficients fit in the double-float format, compute the product with the double-float FFT, or for that matter, using classical multiplication, with double-floats. Or even single-floats if possible. That is, the output coefficients must have magnitude less than the overflow threshold (about 1.8×10^{308}). This operation takes very little time, and it might not get the right answer, but except for the possibility of overflow, it will get a much closer bound on the magnitude of the coefficients in the answer. For example, if the largest coefficient in this approximate answer is a floating point 3.0×10^{80} we know that we need

[11]How comfortably they must fit is an important question.
[12]http://www.fftw.org/accuracy/method.html

to have at least 81 decimal digits of accuracy in our FFT, and based on the other components of the error bound, we can determine a precision width W for the FFT.

2. If W can be met by doubled-double (about 32 decimal digits) or quad-double arithmetic (about 64 decimal digits), redo the FFT in that precision. The truly arbitrary-precision (e.g. MPFR using GMP) software is more flexible but substantially slower.

   (Example: Consider that the number of bits B needed in the largest coefficient is computed by a double-float FFT and found to be 59. That is, the magnitude of the coefficients in p^2 is bounded by 2^{59}. If you wish to compute exactly $p \times p$ of length 2048, our table suggests 19 more bits, so quad-double should be fine.)

3. Find a bound B for the largest coefficient in the answer, as above. Identify a single finite field of size at least $2B$ which also contains a value ω which is a primitive 2^nth root of unity, where $d = 2^n$ exceeds the degree of the answer. Computing two forward transforms, d multiplications, and one reverse transform gives us the product.

4. Do the arithmetic *not* in a single finite field of size greater than $2B$, but in a number of finite fields whose moduli, when multiplied together, exceed $2B$. The Chinese Remainder Algorithm (CRA) allows these pieces to be put together, in a final step. (There are numerous minor variations [5], some of which can make substantial timing differences.)

5. *We think this idea is new.* We do the arithmetic not in a single finite field, or a collection of finite fields each with an ω or over a high-precision floating-point domain. Rather, we use the existing, presumably highly-tuned floating-point computations familiar in numerical computation, *repeatedly.* The novelty is that the *inputs only* have been reduced modulo selected primes q_1, $q_2, \cdots$, (e.g. each less than 2^{19} if the inputs are 1024 term polynomials) The arithmetic (as we have observed) done rapidly *in floating-point* and then rounded to integers. We know these integers are exactly correct, and so we can reduce them modulo those same primes q_1, $q_2, \cdots$. The resulting exact images can be collected via the Chinese Remainder Algorithm (Garner's algorithm) in a final step.

6. *We think this idea is new.* Use one finite field image, (perhaps even computed using traditional multiplication!) to check the TRAILING bits of the coefficients, to make sure the rounding is done to the right integer. As a simple example, reduce the inputs modulo 3, multiply them. This preserves the information of which of the coefficients are even and which are odd.

The big win here seems to be the observation that any time a better numerical FFT is available on whatever computer we are using, we can use it. The FFTW website shows how intense this development activity has been.

5 Conclusion

Benchmarking is always prone to misinterpretation. In comparing two or more algorithms there is always the temptation to continually embroider your program's implementation until it is faster than the competition, which, grabbed at a particular point in time, tends to be stationary. Or one can run the programs on datasets that especially show your ideas in the best light. Benchmarks are also subject to variations in compiler optimization, memory speed and cache size, or in our case whether GMP or other underlying arithmetic is used.

We find rather little support in our own experiments for the idea that the FFT should be used for *typical* polynomial multiplication problems which, we contend, are unlikely to be univariate and dense. Even in the FFT's favored realm (univariate, dense) we see advantages to a version of the Karatsuba algorithm in our implementation, but only if it switches to the classical method for inputs of degree below 12 or so. Our

Toom program has advantages, but benefits if it switches to Karatsuba at about 300. (Choosing these cutoff numbers for a particular environment can be done by testing and tuning; it would be preferable to have a more principled approach to this art.). If you know that your inputs have small coefficients and a single floating-point FFT can either get you an answer, or a good distance toward the answer, FFT starts looking promising. But if you have small coefficients and tell this to the classical method, it beats the FFT for polynomials of degree even 1,024 or more in our spot check of polynomials with all-one coefficients. For this set of cases, only beyond degree 2048 does the FFT become faster. If you have extremely large degree and large coefficient *univariate* problems, at some point the cost of conversion of the data from Lisp into NTL is negligible compared to the time in polynomial multiplication, and, since it takes almost no effort to load NTL into Lisp as we have done, it is plausible to consider just using NTL or similar programs from Lisp-based CAS. This idea of using multiple representations (lists, arrays, NTL encoding) for efficiency or convenience was used in the Macsyma CAS since the late 1960's, and in Matlab '68 before that.

(Incidentally, anticipating some readers' questions, we have not included times for Maple and Mathematica because they are slower.)

6 Acknowledgments

I would like to thank Dr. Kamal Abdali, of the US National Science Foundation who was visiting UC Berkeley from the NSF in 2005, for numerous useful discussions on the topics in this paper and much more. Student Yozo Hida provided advice on accuracy of the FFT. Additionally Charles Cox provided advice on the C++ NTL interface to Lisp. Numerous emails exchanges with Victor Shoup were helpful in understanding NTL. Thanks to the MPFR team helped in getting pieces their software running in Windows, which will enable us to run FFTs to higher precision. Also thanks for comments by P. Zimmermann and T. Granlund. Special thanks to Brian Gladman and Mikel Bancroft in supplying critical help to overcome my ignorance of producing and using Windows dlls.

References

[1] D. Bailey, ARPREC. http://crd.lbl.gov/ dhbailey/mpdist/

[2] The GNU MP Home page. http://swox.com/gmp/

[3] D. E. Knuth, *The Art of Computer Programming*, volume 2. (1969, 1981, 1998). Addison-Wesley.

[4] R. Fateman, "Can you save time in multiplying polynomials by encoding them as integers?", draft, 2004

[5] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge Univ. Press 1999, 2003.

[6] Percival, Colin. "Rapid multiplication modulo the sum and difference of highly composite numbers," *Mathematics of Computation, 72:241* (Jan 2003) pp. 387–395.

[7] William H. Press, Brian P. Flannery, Saul A. Teukolsky, William T. Vetterling. *Numerical Recipes in Fortran* (various editions), Lisp version online at www.library.cornell.edu/nr/cornell_only/othlangs/lisp.1ed/kdo/readme.htm.

[8] V. Shoup. NTL, (Number Theory Library) `http://shoup.net/ntl/`.

[9] Victor Shoup. " A New Polynomial Factorization Algorithm and its Implementation." *J. Symbolic Comp., 20(4)* pp. 363–397, 1995.

Program Appendices omitted, all source programs available from the author.