Can you save time in multiplying polynomials by encoding them as integers?/revised 2010

Richard J. Fateman
University of California
Berkeley, CA 947220-1776

August 21, 2010

Abstract

Encoding a univariate polynomial as a long integer conceptually unifies some processing tasks on polynomials with those on integers, including multiplication. An evaluation of a polynomial at a single point is sufficient to encode and decode all its coefficients if the point is sufficiently larger than the coefficients. This is sometimes referred to as “Kronecker’s trick”. We show that this idea has practical implications for efficient programs.

1 Introduction

I was recently prompted[1] to re-examine the practical possibilities of packing coefficients of polynomials into long integers or “bignums” and using efficient integer library routines to do polynomial multiplication.

This kind of transformation, evaluating a polynomial to map it to an integer, is sometimes called “Kronecker’s trick” (see von zur Gathen and Gerhard [15] section 8 for some references and history suggesting this dates back to at least 1882). The essential equivalence of dense polynomials and long integers is fairly widely-known in the symbolic computation community, to the extent that in the first (1969) edition of Knuth’s *Art of Computer Programming,* volume 2 [12] we see “polynomial arithmetic modulo b is essentially identical to multiple-precision arithmetic with radix b, except that all carries are suppressed. ... In view of this strong analogy with multiple-precision arithmetic, it is unnecessary to discuss polynomial addition, subtraction, and multiplication any further in this section [4.6.1].”

Yet during the early development of computer algebra systems starting in the mid-1960s, these possibilities, if entertained at all, were generally dismissed as of theoretical interest only. Circumstances have changed. In particular we are provided with a competitive environment for the deployment of high-quality (free) software libraries that might relieve the computer algebra systems programmers from many concerns about implementation of arithmetic on multi-precision integers. Additionally, we are presented with far more powerful and complicated computers.

In fact, the main practical motivation here for using this technique is to provide fast implementation of a polynomial multiplication *with less programming* by (re-)using others’ efficient software. Additionally, when we use (others’) library programs, we can benefit from their continued refinement as well as adaptation to new and improved hardware. Somewhat paradoxically, we reduce the polynomial arithmetic problems to integer problems which are solved (conceptually at least) by viewing the integers as polynomials.

This still leaves us a choice as to which of the elaborate implementation libraries for bignum arithmetic to incorporate as a library, We currently are using GMP [9] but could probably use one of its forks (MPIR). In the past we have made use of a similar package, ARPREC [1], but this seems to have fallen into relative disuse. NTL [16] was another library, but its author seems to recommend linking it to GMP for best performance. GMP seems to have an enthusiastic and substantial effort mounted to optimize it for different hardware

[1]by a note from Daniel Lichtblau concerning the use of GMP for multiplying polynomials in Mathematica 5.0. DL’s note is in the appendix.

architectures, as well as subtle variations in the architectures. A major concern seems to be optimizing the choice of algorithm for the best long-integer multiplication variation (classical to FFT with variations between), and with different (e.g. unbalanced size) inputs. Using more than one "core" CPU may also be of interest[2].

Given such a standard library, we can view a system for computer algebra polynomial manipulation as one that is initially machine independent and stated at a high level in a common language, without concern for the underlying computer architecture. Starting from this perspective we can achieve efficiency on any problem *for each machine* because – once turned over to the corresponding bignum package – the principal operations are effectively *optimized for that architecture by the authors of GMP*.

Just to be clear, let us ask and then answer this obvious question, "Would it be better to directly write an FFT program for fast polynomial multiplication, instead of using this route?" The answer: In principle, yes. In practice, probably not. We've written such programs; frankly it is laborious to get all the details right, and an endless task to continue to refine such programs in the face of changes in computer architecture and memory implementation. We are also sure that our programs are not the best possible, even for the architecture we have used for our tests: certainly they are not targetted to specific *other* architectures. They are not as good as the GMP versions. Shoup in his NTL system [16] *does* do this hard work; our technique consequently is likely to be a retrograde step for NTL. (This is confirmed by our simple benchmark.) Perhaps we can find a library FFT routine that is not only right, but has been constructed with the same obssessive attention to detail and efficiency that is seen in GMP? We have in fact experimented somewhat successfully with FFTW [6] Such an FFT performed on fixed-size "floating-point" words is quite fast (and correct) in certain examples: when the result coefficients are somewhat less than the size of the floating-point fraction size. However, if the coefficients are much smaller, then the Kronecker trick, by packing several per word, may win. For example, Computing with 8 coefficients packed into a 64-bit word clearly has the potential to speed up calculations over 8 64-bit words. Intel's MMX architecture provides a view of the same potentiality with applications in signal processing and graphics. Vector or array super-scalar operations on some computer may make the argument plausible for larger assemblages, say if one can add two vectors of 128 words at once.

Yet another technology argument for this potentially more compact encoding: if the version of the polynomial as an integer keeps the total size within some cache limit, it may provide a substantial performance gain. Since the encodings of polynomials require an initial scan of the data to compute a bound on coefficient sizes prior to packing into an array, the behavior of this scan with respect to caches should be incorporated in any detailed analysis.

2 Multiplication

Take the task of multiplying two polynomials in a single variable with non-negative integer coefficients. For simplicity of exposition we will talk about multiplying a polynomial by itself, so as to require encoding only one input, but the method is not limited to squaring. Consider the problem of squaring $p = 34x^3 + 56x^2 + 78x + 90$. The result is this polynomial:

$$1156\,x^6 + 3808\,x^5 + 8440\,x^4 + 14856\,x^3 + 16164\,x^2 + 14040\,x + 8100$$

Now take p evaluated at $x = 10^5$ and square this integer to get 1156038080844014856161641404008100. One can just read off the coefficients in the answer, modulo 10^5, 10^{10} etc.

If we choose a power of two instead of a power of 10, and if the bignum representation is binary or some power of two (say 2^{32}), then evaluation into an integer can be done by shifting and masking: packing the coefficients padded by zeros. Unpacking the coefficients is similarly a matter of shifting and masking.

Doing this efficiently requires attention[3] *to nasty details of the representation of bignums!* Nevertheless, ignoring efficiency, it is possible to state the process very simply as arithmetic, as shown below.

[2]In our experience with GMP we found it is not always possible to install the latest, most-specific and efficient version on a target machine, but we suspect that a slightly more generic version, pre-compiled for a class of similar machines and available with less effort, may be nearly as efficient.

[3]But not necessarily attention by US!

We seem to be done with the task, except for figuring out how much padding is necessary.

An exact bound could be computed by multiplying the two polynomials, but that is exactly the work we hope to avoid. If we define max-norm $N(p)$ to be the maximum of the absolute values of the coefficients in a polynomial p, and d_p to be the degree of p in the variable x, then the largest coefficient in $p \times q$ is bounded by $(1 + \min(d_p, d_q)) \times N(p) \times N(q)$.

[Sketch of proof: if $p = \sum a_i x^i$ and $q = \sum b_k x^k$ then the largest coefficients in $r = p \times q$ for fixed $N(p)$ and $N(q)$ occur when $a_0 = a_1 = \cdots = N(p)$ and $b_0 = b_1 = \cdots = N(q)$. We can factor out $N(p)$ and $N(q)$ simplifying the question then to what is the largest coefficient in a product of two polynomials with coefficients all equal to 1? The coefficient of x^k in r is$\sum_{i+j=k} a_i b_j$. There are at most $(1+\min(d_p, d_q))$ terms in such a sum, since you run out of a_i or b_j terms beyond that point.]

This bound is tight (i.e. achievable in the case of all equal coefficients), but we expect that for some applications of polynomials over the integers the coefficients will often be much smaller than the computed bound, and there will be substantial over-padding. For applications of polynomials with coefficients in a finite field the padding may be just adequate.

After computing this bound B, assuming we have an underlying binary representation, we choose the smallest n such that $2^n > B$, pack, multiply, and unpack.

3 Extensions

3.1 Negative Coefficients

Assume the leading coefficient is positive. (This is not a restriction because we can convert the problem to the negation of the polynomial and then change its sign before returning the result.) We will compute using a kind of complement notation. Consider again our decimal example, but with some negative signs: $34x^3 - 56x^2 + 78x - 90$. To evaluate at 10^5 we rewrite this polynomial as $33 * (10^5)^3 + (10^5 - 56) * (10^5)^2 + 77(10^5) + (10^5 - 90)$ where we construct each of the negative numbers by "borrowing" 10^5 from the next higher digit. If that digit is itself negative, we borrow again. There is no change in converting to the integer representation; converting back simply follows the rewriting shown above. Assuming we use a binary base, we should choose an n such that $2^n > 2B$ so that we can get both + and - coefficients reconstituted. That is, we change the padding– bumping it up by a factor of two (one bit more), and if the value (remainder) extracted is more than half the padding, it is a negative number and should be adjusted. It is sometimes suggested that negative coefficients could be dealt with by splitting polynomials into positive and negative parts [15],8.4. Using complement notation is much simpler.

The description as a program to do all this can be quite short, and perhaps easier to understand than our description above.

```
 /* this is Macsyma Code */
/* convert the polynomial p(x) to an integer, with padding v */
tobig(p, x, v) := subst(v, x, p);

/* notation:
  [q,r]:divide(i,v)  sets q to quotient, r to remainder of i by v */

frombig (i,x,v):=   if (i=0) then 0
               else if (i<0) then -frombig(-i,x,v)
               else block([q,r],
                     [q,r]:divide(i,v),
                     if r>v/2 then r-v+x*frombig(q+1,x,v)
                              else r +x*frombig(q,x,v));
```

e.g. if $p = 34 * x^3 + 56 * x^2 + 78 * x + 90$, $h = \text{tobig}(p, x, 10^6)$ gives 34000056000078000090 and frombig$(h, x, 10^6)$ gives $x * (x * (34 * x + 56) + 78) + 90$ which is equivalent to p.

Actually, creating this polynomial may not be as useful as creating a list of the coefficients, which is what this next program does. The first coefficient in the list is highest degree. We've also rewritten this to use iteration rather than recursion.

```
frombigL(i,v):=block([ans:[],q,r,signum:if (i<0) then -1 else 1],
                     i:i*signum,
                     [q,r]:divide(i,v),
                     while (i>0) do
                      (if r>v/2 then (push(signum*(r-v),ans),
                                      i:q+1)
                                else (push(signum*r,ans),
                                      i:q),
                      [q,r]:divide(i,v)),
    ans);
```

3.2 More Than One Variable

It is possible to use the same trick recursively. That is, consider a polynomial p in main variable y whose coefficients are polynomials in variable x. That is, $p = \sum_{0\le i\le n} q_i(x)y^i$. Then p can be encoded by finding some sufficient uniform padding for *ALL* the coefficients within the $\{q_i\}$ to maintain their separation, as well as another padding to maintain the separation between these $\{q_i\}$. This is not a particularly appealing encoding unless the polynomials $\{q_i\}$ are dense and of uniform degree.

3.3 Other Operations

It is possible to do more operations on the bignum form as long as the number remains a faithful representation of the polynomial with respect to the operation. Clearly we must avoid any "carry" from one coefficient to another. We can try guarding these coefficents more carefully by some heuristic computation for extra padding in the case where we do not have a convenient bound, and test to see if our result is consistent. One approach is to replace x by y^2 and thus our true polynomials have only even powers of the variable; any time there is a non-zero odd coefficient, there has been some "leakage".

Addition is easily supported. If, after doing an add, there are any non-zero coefficients of the odd powers, we have detected an invalid operation. (This idea of having guard space between fields packed into a single word was used in the ALTRAN system of the 1960's. In their design only one bit between fields was allocated, since that is all that is necessary.) For other operations we must also make sure that the absence of odd powers guarantees a valid operation, as is the case with addition of two such polynomials.

There are other operations that can be reduced to integer operations *at least sometimes* or whose results can be computed with high probability, and then tested. The most prominent is the polynomial greatest common divisor (GCD) calculation which is notoriously expensive and frequently needed in simplifying rational expressions. There is a literature on the technique here, called the Heuristic GCD (GCDHeu); it is used in the Maple and MuPAD computer algebra systems [2, 14]. The checking to make sure the (heuristic) integer answer really maps to the correct polynomial is non-trivial, and a complete GCDHeu algorithm must consider that the procedure may require more padding than suggested by the initial heuristic, and that it may still fail and that another GCD algorithm entirely should be used. (Testing of the result is included in the appendix; fallback algorithms are not.)

There are other suggestions for using this kind of Kronecker evaluation in the reduction of multivariable polynomial factoring to eliminate one or more variables (e.g. to bivariate [11]).

An important difference between our proposal here and that in the GCDHeu or factoring approach is that we advocate rapid encoding of coefficients via shifting and masking of binary quantities, not evaluation of the polynomial at some "randomized" point[4].

[4]A starting evaluation point for at least one version of GCDHeu is 29 + twice the minimum of the maximum absolute values of the coefficients in the input, hardly likely to be a power of two.

In some earlier experiments [13], we found areas where GCDHeu was superior to other tested algorithms, even though these tests were using the built-in Lisp (Allegro CL) bignum package (classical arithmetic), and was therefore not especially speedy compared to what we know we can do with GMP. Using a basic arithmetic suite that includes *much* speedier code for much larger sizes should make GCDHeu competitive for somewhat larger problems than shown in [13], since the integer GCD calculation can be a major portion of the expense.

More recently, von zur Gathen and Gerhard [15] (in figure 6.4) show tests that using a power-of-two evaluation point for GCDHeu is (slightly) faster on their randomly generated examples compared to other points. They do not discuss the reasons for this, or the probability of the GCDHeu heuristic failing in this case.

It is beyond the scope of this paper to exhaustively test a power-of-two GCDHeu algorithm, but worth asking a feasibility question concerning how fast we can calculate the GCDHeu. Here is one data point: consider two polynomials p and q each of degree 2000 with random coefficients bounded by 10,000. To compute the correct GCD of $p \times q$ and p (The GCD is p) took 78ms including encoding and decoding. To compute the product of p and q took 31 ms using GMP encoding. To compute the product $p \times q$ by "regular" arithmetic took 343 ms. The full cost of the GCDHeu would have to include additional confirmation steps which might dominate the time quoted, so this is a "most favorable" comparison. Yet the conventional polynomial subresultant sequence GCD algorithm took 245 seconds, or over 3,000 times longer, using programs from an earlier comparison [13], leaving considerable room for testing or re-performing the heuristic GCD with different settings. (Arguably this instance of GCD calculation is implausibly large for most uses of a CAS, although much emphasis appears to be made of far larger examples in von zur Gathen and Gerhard [15] section 6.13.)

Further analysis of the GCDHeu idea and extensions may be found, for example, in a paper by Geddes, Gonnet and Smedley [7]. Although the cited paper is primarily concerned with GCD, it includes an algorithm for polynomial trial division. Also see Davenport/Padgett [3], and Geddes *etal*[8] section 7.7, theorem 7.7 where single-point evaluation and interpolation are discussed and the bound on an evaluation point is defended; the emphasis is, however on using a much smaller evaluation point as a heuristic.

Other possible compound operations can be performed, with suitable checks for validity, on point-evaluations and single-point "interpolation" to integers. For some operations, validity is not guaranteed by the mere absence of carries. In some cases it appears that the cost of the checks may exceed the cost of doing the job in the first place. We include in the category of "possible, but be careful" the evaluation of determinants of polynomials, because algorithms of an iterative nature with many sequential additions and multiplications can be translated into integer operations. It remains to determine if we can succeed by finding a suitable "tight" padding for the whole sequence of operations, or a probable testable padding.

Another notion we have explored briefly is an application to factorization of an integer I. Consider "guessing" the padding p, and convert I to a polynomial $q(x)$. Using any of the popular techniques for polynomial factorization, try to factor q. If there are factors, these can be immediately mapped to integers. Thus $I = 19624459 = 4409 \times 4451$ can be factored by guessing some padding, say 4403, which produces a polynomial which factors as $(x-6)(x+48)$. Any guess between 4379 and 4481 will produce a useful factorization. Note that 4429, a good guess, is near $\sqrt{I}$. There are many reasons for this method to not work, generally. If there are two large prime factors which are substantially different, the heuristic of choosing $\sqrt{(I)}$ does not work. We have not explored the relationship of this heuristic to current integer factorization algorithms.

4 More Compact Encodings

David Harvey [10] has explored an alternative, of knowingly choosing insufficient padding and performing a modified multiplication two or four different ways. Padding is essentially evaluation: padding with n, is evaluation at2^n. Choose a smaller n but re-do the calculation with 2^{-n}, shifted to allow for integer results. This gives us the same coefficients but in reverse order. Or for another orthogonal choice, use $(-2)^n$ which alternates signs. If we can unpack and disentangle possible overlaps efficiently (Harvey shows that we can) there are regions (ranges of input-sizes) in which there an advantage: when the inputs are neither too small or very large. This essentially takes advantage of the observation that, with the guaranteed padding bound, one may be wasting time since many of the digits (whole words of them!) will consist of zeros. Harvey's alternative gives us a chance to pack them tighter and use a clever unpacking to disentangle the overlaps. Because the numbers are smaller, cache overflows may be mitigated. Note that for very large examples where GMP uses FFTs, Harvey's benchmarks (see figures 1,2 [10]) illustrate there is not much advantage. For small examples the overhead of packing and unpacking swamps any improvements. In a "middle" range a factor of about 2 is achieved in a range of polynomial degrees between 100 and 5000 for coefficients in a 48-bit finite field.

5 Finite Field Operations

A particularly simple way of taking advantage of the mapping to bignums is available if the arithmetic is to be performed in a finite computation structure (usually a finite field, but weaker conditions could be supported– no inverses are needed, for example.)

In the useful case of univariate polynomials over $GF[p]$, the input coefficients might as well be integers k with $0 \leq k < P$ for some odd prime P. This is particularly appealing if P is less, perhaps much less, than the maximum single-word integer, as suggest previously.

Some caution is needed: Even though we know the coefficients can all be reduced modulo P, the bignum representation needs padding. In particular, for multiplication we must allow for a maximum coefficient size of $(1 + \min(d_p, d_q)) \times P^2$, and the translation back to conventional polynomial form must do a modulo P reduction for each coefficient. We know of no especially quick method here; it seems to require unpacking and computing a remainder, then re-packing coefficients.

However, knowing the domain of computation, with a finite field we do not have to look at the coefficients to find their maximum. This saves a linear-time scan through the memory storing the polynomial.

In such finite field or modular operations it is possible to take one or several such operations and through an interpolation or lifting operation, obtain results in a larger computation structure [15].

6 Benchmarks

It is not our intention in this paper to exhaustively explore all possible ranges of coefficients and degrees and to compare to all possible other implementations. **This would be a task that**[5] **is simply not worthwhile.** It would require comparison to systems that are obscure and special-purpose, that may or may not be available for convenient use by the author or by the reader, and whose characteristics may change without notice. Indeed some will certainly continue to improve, and others may cease to function as particular computers and operating systems go out of fashion. We have considerable experience with attempting to publish such figures in a reproducible fashion, and collect comments such as "Your comparison was done with version X, but you must use version X+1 which is faster; also what about system XYZ (unknown except to the referee!)"

Instead we wish to establish a feasibility criterion. Does this technique work for a sufficiently interesting example such that implementations should be considered for other systems? That is, based on the information here, should the authors of existing systems X, Y, or Z, look at the idea here?

We choose as our first point of comparison for polynomial multiplication speed, a fixed large polynomial, $p = (x + 1)^{1000}$ and translate it to a single bignum k with suitable padding. Then we compute (and time) the multiplication r=$k \times k$. Compare that to the time to directly multiply p by itself. Another comparison would be to multiply the polynomial given by polynomials of substantially different size, say $(p^3 \times p)$.

This test is unfair to the conventional representation (CR) because the CR is far more versatile. CR allows

1. more than one variable, stored sparsely.
2. wide variation in the size of the coefficients, without much loss in efficiency. That is, some may be individually bignums, others zero.

[5]In spite of some referees' suggestions to the contrary

3. the result of the operation (typically multiplication) to be immediately presented in the "normal" form which does not have to be decoded or re-converted to a representation with more (or possibly less) padding for a subsequent operation.

In fact, the notion of having to change representation depending upon what you are going to do subsequently with the object is discomforting. How would you feel about a representation for a linked list which had to be entirely recomputed depending upon how many new links would be appended to it? (We are willing to manage some discomfort for a major speed improvement, and if we can encode/decode fast. To be more precise, the encoding time is linear in the input size and the decoding time is linear in the output size. If we can make a major improvement in the computation time, say from an $O(N^2)$ algorithm to $O(N \log N)$ in theory this presents the prospect for an improvement. Of course benchmarks may show that this is not actually worthwhile in practice.)

Thus we propose to eliminate the last of these objections in a second version where we include in the bignum algorithm the cost of encoding and decoding operations. This requires looking for the maximum coefficient in each input and finding the appropriate bound for the product.

Yet this version is somewhat unfair to the bignum program performance, since we can do multiple operations in the encoding. It would be grossly unreasonable if we used the simple-minded programs given above in Macsyma's top-level lnguage. Even if the conversions are compiled into Lisp, possible from the Macsyma language, they are far from as efficient as possible. As indicated earlier, we have written a better program for encoding (`tobig`) that can use bignum shifting or masking and not really "evaluate" a polynomial. A better program for decoding (`frombig`) can use bignum operations equivalent to division with remainder, but really just multi-word shift and logical mask operations. As a result, the coefficients would presumably be laid out in a convenient data structure of an array or list.

In our earlier draft of this paper, we discussed and presented what we believed to be an efficient version of these transformations; a challenge from Bill Hart, who is currently (2010) maintaining FLINT [5], showed that we were not really so efficient! In fact both FLINT and our programs were essentially encoding polynomials to present to the same library (GMP) to multiply as bignums, and decoding them. If we were using the same version of GMP, the "middle" of the algorithm would be the same and thus a comparison would be in the programming skills (and choice of language!) for packing and unpacking. One consideration is that Hart is using C and our programs were in Lisp [6]

An observation: our original programs for interfacing Lisp with GMP were adequate for most of our experiments because in all cases we converted numbers to GMP form and did not convert them back to ordinary numbers in Lisp until they were to be printed. This meant that any inefficiency in conversion was masked by the output costs. This Kronecker encoding/decoding must be done repeatedly within the computation – to extract coefficients, to present the results in conventional form. This presents a much higher demand on efficiency. The old programs could be lackadaisical about (for example) using GMP programs to produce character-string versions of decimal numbers. The new programs access the machine-representation of GMP integers extracting "limbs" (the 32-bit sections of multi-precision numbers). This requires an implementation-dependent (non-ANSI Lisp) program. Even so, our new Lisp programs have some inefficiencies because the version we use has a different integer representation. Other Lisps that have adopted GMP directly may be more efficient in some regards, though just using GMP through a "functional programming" veneer may not be as efficient as using the raw GMP, as we do.

A note for implementors: If it is to be as general as the Macsyma program given here, the polynomial encoding/decoding program must be prepared to work on coefficients *which are themselves bignums*, and so the masking and shifting can't be written trivially in a language that has only fixed-length arithmetic (such as C or Fortran). This subtlety can be avoided if the polynomials are multiplied in a finite field whose modulus is substantially less than a word-size. There are such applications, particularly in factoring of polynomials, where a simple C program might perform the task and such a speedup is potentially worthwhile.

A final consideration is the issue of multiplying polynomials in the "normal" $O(nm)$ way, where n and m are the polynomial degrees of each input. Is this just too easy a target for improvement? Is it easy to improve this without going into this proposed encoding? In fact, for a large enough polynomial multiplication problem we can run faster with a modicum of additional coding using a "divide and conquer" or "Karatsuba"

[6] An alternative though is our Lisp program to call FLINT to do the packing and unpacking.

style multiplication. This has asymptotic growth $O(n^{1.585})$, where n is the maximum degree of the inputs, but bookkeeping costs make it more expensive for smaller polynomials. (The asymptotically faster FFT requires more than a modicum of coding.)

(2005) Consider again the problem of multiplying $p \times p$. On our current desktop computer[7], we can perform this calculation in 23.6 seconds. Using a Karatsuba version, it takes as little as 3.7 seconds. Converting this problem to a GMP number takes 63 ms. For each of two similar-sized inputs, this would be 126 ms[8]. The multiplication itself takes only 485ms. to produce a number with 31,251 words. The reverse transformation, also phrased in compiled Lisp but consisting mostly of calls to special GMP *division and remainder by powers of 2*, (implemented as shifts) takes an additional 186ms, for a total time of 0.8 seconds. This total then can be compared to 3.7 seconds using Karatsuba.

Consider multiplication of $p \times p$. On our current (2010) desktop computer, 2.99GHz Pentium 4, GMP 4.1.2, Allegro CL 8.0, better packing/unpacking programs than previously, we can encode the polynomial p as a GMP number K with padding 2002 in 53 ms. (we would do this twice, since we are oblivious to the fact that the two inputs are the same). Squaring K takes 109 ms, resulting in a GMP integer requiring 125,126 32-bit words. Unpacking K^2 takes 200 ms. Total 0.415 sec. Using a Karatsuba version of polynomial multiplication but using the Lisp bignum arithmetic for coefficients, takes 3.28 sec., and a conventional naive multiplication takes 12.9 seconds. Kronecker beats the next best by about 8X.

One alteration to the technique here is to use GMP bignum arithmetic on coefficients and use either the naive multiplication or Karatsuba. Note that the largest coefficient in $(x + 1)^{100}$ is 995 bits long, and so a relatively slow bignum package hampers the native Lisp program. In fact, rewriting the Lisp programs to always use GMP coefficients reduces the time from 12.9 seconds cited earlier, to 2.19 seconds. For the Karatsuba version, the time is reduced from 3.28 seconds to 0.72 seconds, switching to naive multiplication at size 50.

Kronecker still wins, but only by a factor of 1.7X over Karatsuba.

Let us modify the problem slightly to eliminate most of the cost related to the coefficient size. It shows Kronecker to considerable advantage. Indeed, all the programs become much faster (up to an astounding 300X !)

Here is the experiment: We change p to be a polynomial with all coefficients being 1. How long does it take to square p (including all packing, unpacking, in the case of Kronecker?).

Note that the necessary padding is not 2001, but 11 bits, so several coefficients can be packed in one word (word sizes might be 32 or 64 bits, or perhaps a few bits less if there is some memory sacrificed for tags).

Squaring p can now be done in 5.9 ms rather than 415 ms with Kronecker.

It can be done in 43 ms rather than 12,900 ms using naive multiplication.

It can be done in 33 ms rather than 3,280 ms using Karatsuba multiplication.

Kronecker beats the next best by about 5.6X for small coefficients.

Note that if all of the coefficients are ones, but they are GMP ones, then the times are considerably *longer* in our tests. That is, the native Lisp integer representation has much less overhead than the GMP representation and operations *for small integers*. For example the Karatsuba version with GMP ones takes 160 ms rather than 33 ms. The naive version takes 516 ms rather than 43 ms.

From these examples we can see there is sometimes a speed advantage in mapping whole (univariate, dense) polynomial multiplication problems into large integer multiplications. This mapping also may be useful for some other operations previously noted. One might ask – why not just keep the polynomials like this, never unpacking? While this may work for finite-field representations, unfortunately for exact integer coefficients, this is generally not acceptable: the padding may have to be renegotiated periodically, in fact at each multiplication operation!

For our example problem, the Allegro CL bignum package falls short, and the Kronecker trick gets about a factor of 1.7 over what we would get by simply using GMP and a Karatsuba multiplication as the preferred library routines for our Lisp system's bignums.

[7]2.6GHz Pentium 4, Allegro CL 6.2

[8]This conversion is written in Lisp using GMP arithmetic. It is possible it might be done even faster by direct coding in the GMP number representation closer to an assembler level.

The prospective user of such programs should be aware that converting a very small problem such as $p = (x + 1)^{10}$ to a GMP number and squaring takes about 4.5 times longer than the naive calculation in our program trials. In a small example the encoding and decoding dominates possible savings, at least in our programming framework.

More benchmarks, only in a particular lisp on a particular computer, may provide some "relative" timings of interest. See Appendix 3.

How important is it to use a power-of-two pad? Intuitively it seems that shifting should be much preferable to evaluation of a polynomial.

Earlier experiments suggest that using some arbitrary number as a pad might be some 4 times slower in packing and 14 times slower unpacking. Since our current hacked-up packing and unpacking is substantially faster, and could apparently be even faster if written in a lower-level language, it seems that there is quite a handicap for a non-binary-power.

If an algorithm truly requires that the padding must be done with (say) powers of some large prime instead of powers of two, our testing suggests that such slower encoding may severely impact the efficiency of encoding and decoding. On the other hand, for carefully chosen numbers there might be compensating tricks that could speed them up.

A final observation: one might propose to replace *all* integer arithmetic in a Lisp implementation by GMP numbers. We think that this would likely be a mistake. Most arithmetic is likely to be on small (single-word) numbers. In the system we used here, Allegro Common Lisp (version 6.2) on a typical 32-bit computer, the fixnum data type provides 30 bits: the largest positive fixnum is 536,870,911. Counting down from this number to zero using compiled Lisp takes 0.3 seconds, but using GMP arithmetic takes 267 seconds. This factor of 890 on common small-number arithmetic is unacceptable since it is used for array indexing and almost every other integer arithmetic application. (compare this to (say) C, where there is not really an alternative in the language for fixed-length arithmetic). Given this reality, most Lisp systems, as well as computer algebra systems, have at least two *underlying* integer types. Such systems tend to cover over this distinction for most operations (e.g. arithmetic has a uniform higher-level model that covers the data type "integer") but at least in compiled Lisp, advisory implementation declarations can specify that some integers will always be small, and that compilation can produce tight code [4].

7 Conclusions

We have illustrated a case in which we can get a factor of 26 speedup by mapping our naive polynomial arithmetic into clever arithmetic on GMP bignums. This speedup is real, but we can also improve the naive (but commonly used) old algorithm. That is, we can improve the standard polynomial multiplication program which is brief and reasonably straightforward, but perhaps not the most appropriate for very large examples.

Hacking on the naive algorithm, the improvement factor is reduced to about 6 or even less: we replace the naive polynomial arithmetic by a better polynomial multiplication program (using Karatsuba divide and conquer).

A further improvement on the naive versions is possible: Given this particular example in which a polynomial $p = (x + 1)^{1000}$ is multiplied by itself, we observe that the polynomial p has coefficients as large as 10^{299}. If we replace the Lisp bignum arithmetic with GMP arithmetic, then the factor is reduced further, to about a factor of 2 improvement. Still, there is considerable merit in being "only" twice as fast.

The Kronecker version really shines, however, when the original coefficients are small. In this case (e.g. coefficients all single digits), it can be 100X faster than the next best, even for substantial degree (1000).

Although we have not presented our benchmarks here to support it, in tests, the Kronecker method may be relatively less advantageous for drastically unequal-sized input polynomials (depending on the qualities of the underlying bignum multiplication program) and furthermore will ordinarily fall behind (along with other dense-oriented algorithms) when compared to algorithms specialized for sparse polynomials, given sufficiently sparse inputs. Separately we have written about sparse polynomial experiments; here we note that for some apparently sparse inputs, the answer may be rather dense, and dense algorithms may be the fastest prospects.

Daniel Lichtblau of Wolfram Research has suggested that a suitable implementation in Mathematica 5.0 specifically to use GMP provides advantages in certain internal algorithms even for relatively small polynomials *over a finite field*. This may be, in part, because Mathematica does not have a particularly efficient encoding for ordinary polynomials, lacks speedups like the Karatsuba multiplication, or like the Lisp we were using, has a relatively inefficient bignum library compared to GMP, once the coefficients are more than a few words long.

On the suggestion of a referee, and against our better judgment (see comments above) we include some comparison timings: In Maple 7, on the same machine: let `q:=expand((x+1)^1000). expand(q*q)` takes 11.20 seconds.

Mathematica 4.1 on the same machine: let `q=Expand[(x+1)^1000]. Expand(q*q)` takes 13.156 seconds.

GP/PARI version 2.2.7(alpha) takes 1.391 seconds. Macsyma 2.4 89.2 seconds (43 in garbage collection).

This can be contrasted with a multiplication time of about 0.485 seconds using GMP bignums, called from Lisp (Allegro Common Lisp).

In some sense the “purest” comparison may be a system like NTL, assuming you are willing to write and compile a C++ program to construct the inputs and call the arithmetic routines. Using NTL’s default “LIP” arithmetic takes 0.344 seconds, faster by a factor of 1.4. Recompiling NTL to use GMP 4.2.1 arithmetic (on a fresh system) takes about 0.047 seconds This is remarkably fast, about 10.3 times faster than encoding as a single GMP number. NTL’s ZZX module uses one of four multiplication methods (classical, Karatsuba, and two FFT variations); in this case an FFT. This method could obvious also be programmed in Lisp, or since it is written in C, called from Lisp. That is, even faster might be to translate the whole problem’s data into NTL format, link to NTL for the polynomial arithmetic, and translate back if necessary. Or performing the calculation entirely in NTL. Or perhaps there is another system even faster than NTL, in which case that could be linked to Lisp.

Our major point remains: it appears that we obtained a substantial speedup *without having to write an FFT program ourselves.* When the GMP multiplication program is improved or tuned for a new architecture (or merely debugged!), we get that benefit without recoding our own programs.

Conclusion in brief: this Kronecker technique has promise for speedups of multiplication for carefully chosen domains of large dense univariate polynomials, and requires basically access to a fast arbitrary-precision integer arithmetic library such as GMP. Systems in which GMP arithmetic is easily available may achieve substantial speedups. There is one catch: efficient packing and unpacking must be available.

Previously (2005) we speculated that one could beneficially devote extensive time and effort to specifically code an equivalently clever polynomial multiplication routine, with appropriate cut-off heuristics between methods, including an FFT routine (as in NTL), then mapping to long integers will not likely run faster. It appears that this was done by FLINT. In 2010 we learned from Bill Hart that this Kronecker trick has a more substantial range of usefulness than we had expected. Hart demonstrated by benchmarks in FLINT 1.5, and by more careful programming we were able, to some extent, reproduce the relative rankings as compared with a Karatsuba method, or even naive multiplication. We had previously discounted Kronecker because our packing and unpacking routines were not as efficient in our earlier code. (We expect that they are still considerably less efficient than FLINT!) Consequently, the choice of algorithm depends more on these “linear time” overheads than we realized.

All Lisp programs used are included in an on-line directory linked from the author’s home page, `lisp/mul/` along with additional documentation. The only other programs needed are those available (free) from the GMP web site [9], and Allegro Common Lisp (the Express/free version).

Recommendation to builders of computer algebra systems: We could spend far more time on more benchmarks and vary parameters such as denseness and size of coefficients, as well as degree. We could use implementations in C, C++, GMP, ARPREC, Lisp, Maple, Mathematica etc. On different machine architectures and even different versions of compilers under different optimizations. Rather than attempt such an inevitably fruitless comparison in the hope of providing CAS system-building advice[9], we hope that this paper provides the essential impetus to try something that is likely to be simple to implement and that will possibly be quite fast if you have a good arbitrary-precision library, and pack/unpack routines. (Another recommendation [10], if you are considering implementing a CAS from scratch or implementing

[9]Fruitless because it will be unlikely to tell you how much *your* precise circumstances will benefit from this technique.

[10]Which, since 2005, seems to have caught on

bignum arithmetic for a language like Lisp, and you intend to build very fast polynomial arithmetic yet again: have you considered just using one or more of the free libraries?)

8 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. Thanks to D. Lichtblau for spurring this exercise and subsequent corrections and comments. Thanks also to (anonymous) reviewers of an earlier draft for their comments. In 2010, I was prompted me to revise this paper thanks to extensive email exchanges with Bill Hart regarding FLINT, Kronecker substitution, and alternatives.

References

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

[2] B. Char, K. O. Geddes, and G. H. Gonnet, Heuristic GCD B. Char, K. Geddes, and G. Gonnet. Gcdheu: Heuristic polynomial gcd algorithm based on integer gcd computation. *J. of Symbolic Computation, 7*:31-48, 1989. The algorithm has been modified for implementation, and the analysis cleaned up. See for example http://arxiv.org/PS_cache/cs/pdf/0206/0206032.pdf

[3] J. H. Davenport, J. A. Padget, "HEUGCD: How Elementary Upperbounds Generated Cheaper Data," European Conference on Computer Algebra (2) 1985: 18-28

[4] R. Fateman,"Importing the Gnu Multiple Precision Package (GMP) into Lisp, and implications for Functional Programming", draft. 2003.

[5] FLINT: Fast Library for Number Theory. www.flintlib.org

[6] The FFTW Home Page, www.fftw.org.

[7] K. O. Geddes, G. H. Gonnet, T. J. Smedley, "Heuristic Methods for Operations With Algebraic Numbers," (Extended Abstract). ISSAC 98 475-480

[8] K. O. Geddes, S. R. Czapor, G. Labahn: *Algorithms for Computer Algebra*, Kluwer Academic, 1992.

[9] The Gnu MP Home page. http://swox.com/gmp/

[10] David Harvey. "Faster Polynomial Multiplication via Kronecker Substitution," arXiv:0712.4046v1 [cs.SC] 25 Dec 2007

[11] E. Kaltofen, "A polynomial reduction from multivariate to bivariate integral polynomial factorization," 14th ACM Symposium on Theory of Computing 1982, p 261 − 266. ISBN:0-89791-070-2

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

[13] P. Liao, R. Fateman. "Evaluation of the heuristic polynomial GCD" *Proc ISSAC 1995* 240–247. http://doi.acm.org/10.1145/220346.220376

[14] A. Schonhage, "Probabilistic Computation of Integer Polynomial GCDs" *J. Algorithms, 1988* 9 365–371.

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

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

9 Appendix 1

DL's email note (excerpt)

"For univariate polynomials mod p we do what is called (I believe) binary segmentation. Based on polynomial length and size of p we create large integers appropriately zero-packed, then multiply them and recover the polynomial product coefficients. An analysis will show that this compares reasonably well with direct FFT convolution methods, especially when the modulus is large compared to polynomial length. I'm pretty sure this is discussed in von zur Gathen and Gerhard[11].

As we use GMP for integer multiplication, this becomes a case where the library software decides whether to use schoolbook, Karatsuba, Toom-Cook, or NTT. My experience is this general approach is quite efficient in practice, especially as GMP is a fairly robust library. Not perfect, but fairly robust."

10 Macsyma Programs

```
/*padpower2 computes the power k such that 2^k is an adequate
pad for the multiplication of polynomials p1 and p2. Double it
if you expect positive and negative numbers. */

(pad(p1,p2,x):=(1+min(hipow(p1,x),hipow(p2,x)))*maxcoef(p1,x)*maxcoef(p2,x),
padpower2(p1,p2,x):= ceiling(log(bfloat(pad(p1,p2,x)))/log(2)))$

/* Maxcoef is a kludge to compute max abs value of coefficients in a
 polynomial.  A simple lisp program does this much faster */

maxcoef(p,x):=max (map (abs, subst(1,x,
                      substinpart("[", ratdisrep(p),0))))$

/* For comparison with the result of frombigL, here is
a program to make a list of the coefficients in a polynomial, zeros too.
They should be the same values if the multiplication worked. */

allcoeflist(p,x):=block([h:hipow(p,x),ans:[],head,rest],
         for k:h thru 1 step -1
                do ([head,rest]: bothcoef(p,x^k),
                                 push(head,ans),
                             p: rest),
          push(p,ans), /* last piece is the constant */
          reverse(ans))$

tobig(p, x, v) := subst(v, x, p);
/* [q,r]:divide(i,v)  sets q to quotient, r to remainder of i by v */
frombigL(i,v):=block([ans:[],q,r,signum:if (i<0) then -1 else 1],
                     i:i*signum,
                     [q,r]:divide(i,v),
                     while (i>0) do
                      (if r>v/2 then (push(signum*(r-v),ans),
                                      i:q+1)
                                else (push(signum*r,ans),
                                      i:q),
                      [q,r]:divide(i,v)),
                    ans);
```

[11]in section 8.4

11 Appendix 2: Some Common Lisp Code

```
;;; SEE files http://www.cs.berkeley.edu/~fateman/lisp/mul/
;;; and especially the file just3GMP.lisp
```

12 Appendix 3

Since we have considerably more timing data, and readers seem to want more, here's some of it, taken in August, 2010 on a Pentium D CPU, 2.99GHz, 3.0GB of RAM, Microsoft Windows XP Professional SP3 Allegro Common Lisp 8.0

We compare the following polynomial multiplication programs:

fft, which is based on FFTW3, a double-float FFT program. This provides only a double-float approximation to the coefficients in the answer, but this is in some cases (provably) enough to provide exact integer answers. We discuss elsewhere means for modifying it to be more robust.

m-ks, based on the Karatsuba idea, but adapted so that it can work adequately on sparse inputs. It switches to mul-dense, the next method, when given either input of degree less than 50, since that is faster.

mul-dense, based on the obvious n^2 algorithm, but carefully coded.

mul, which is the Kronecker program using GMP 4.1.4 multiplication, the primary subject of this paper.

```
ones is a polynomial of all 1's coefficients, dense to degree 999.
ones2 is ones X ones, and is of degree 1,998


for ones X ones2.

fft:           12.5 ms
m-ks:          25.0 ms
mul-dense      75.0 ms
mul:           12.5 ms


for ones2 X ones2   i.e. degree 1998 X 1998

fft:           15.6 ms
m-ks:          79.7 ms
mul-dense     429.7 ms
mul:           25.0 ms



for ones X ones  i.e. degree 999 X 999

fft:            6.0 ms
m-ks:          12.8 ms
mul-dense      37.5 ms
mul:            6.0 ms

For coefficients that are larger, We start with L= degree 9,
all coefficients=1, and use powers of L.
```

```
Let L32 = L^32. degree 288 each
for L32 X L32, max coefficient 2443553412892220489195278947230,

fft:          12.2 ms ;; float approx only
m-ks:         23.0 ms
mul-dense     44.5 ms
mul:           9.0 ms

Some unbalanced examples

Let L8= L^8

for L32 X L8   degrees 288 and 72.  max coeff of L8 is 4816030

fft:          12.2 ms ;; float approx only
m-ks:         16.6 ms
mul-dense     21.1 ms
mul:           3.9 ms

for L4 x L32

fft:            6.7 ms ;; float approx only
m-ks:           9.8 ms
mul-dense      10.2 ms
mul:            3.6 ms


SMALLER SIZES

for L X L where L is all ones, of degree 9.  degree 9 X degree 9

fft:          .1641 ms
m-ks:         .0172 ms
mul-dense     .0156 ms
mul:          .0750 ms


for L8 X L8  degree 72 X 72

fft:          .1250 ms
m-ks:         .1125 ms
mul-dense     .1329 ms
mul:          .0656 ms


for L4 X L4  degree 36 X 36

fft:           .453 ms
m-ks:          .062 ms  ; just calls mul-dense
mul-dense      .062 ms
mul:           .219 ms
```

We encourage others to try these different programs, or similar programs in their own favorite languages and on their own computers before concluding which is preferable in their own situations. We do wish to note that the naive mul-dense version appears to be quite advantageous if many of the multiplication problems are small.