Implementation Techniques For Fast Polynomial Arithmetic In A High-level Programming Environment

Akpodigha Filatei
ORCCA, University of Western Ontario (UWO)
London, Ontario, Canada
afilatei@orcca.on.ca

Xin Li
ORCCA, University of Western Ontario (UWO)
London, Ontario, Canada
xli96@orcca.on.ca

Marc Moreno Maza
ORCCA, University of Western Ontario (UWO)
London, Ontario, Canada
moreno@orcca.on.ca

Éric Schost
LIX, École polytechnique
91128 Palaiseau, France
schost@lix.polytechnique.fr

ABSTRACT

Though there is increased activity in the implementation of asymptotically fast polynomial arithmetic, little is reported on the details of such effort. In this paper, we discuss how we achieve high performance in implementing some well-studied fast algorithms for polynomial arithmetic in two high-level programming environments, AXIOM and ALDOR.

Two approaches are investigated. With ALDOR we rely only on high-level generic code, whereas with AXIOM we endeavor to mix high-level, middle-level and low-level specialized code. We show that our implementations are satisfactory compared with other known computer algebra systems or libraries such as MAGMA v2.11-2 and NTL v5.4.

Categories and Subject Descriptors:
I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation – *Algebraic Algorithms*

General Terms:
Algorithms, Experimentation, Performance, Theory

Keywords:
High-performance, polynomials, Axiom, Aldor.

1. INTRODUCTION

Asymptotically fast algorithms for exact polynomial and matrix arithmetic have been known for more than forty years. Among others, the work of Karatsuba [21], Cooley and Tukey [6], and Strassen [27] has initiated an intense activity in this area. Unfortunately, its impact on computer algebra systems has been reduced until recently. One reason was, probably, the belief that these algorithms were of very limited practical interest. In [13] p. 132, referring to [25], the authors state that the FFT-based univariate polynomial multiplication is "better than the classical method approximately when $n + m \geq 600$", where n and m are the degrees of the input polynomials. In [22] p. 501, quoting [3], Knuth writes "He (R. P. Brent) estimated that Strassen's scheme would not begin to excel over Winograd's until $n \approx 250$ and such enormous matrices rarely occur in practice unless they are very sparse, when other techniques apply."

The implementation of asymptotically fast arithmetic was not the primary concern of the early computer algebra systems, which had many other challenges to face. For instance, one of the main motivations for the development of the AXIOM computer algebra system [19] was the design of a language where mathematical properties and algorithms could be expressed in a natural and efficient manner. Nevertheless, successful implementations of the FFT-based univariate polynomial multiplication [25] and Strassen's matrix multiplication [2] have been reported for several decades.

In the last decade, several software for performing symbolic computations have put a great deal of effort in providing outstanding performances, including successful implementation of asymptotically fast arithmetic. As a result, the general-purpose computer algebra system MAGMA [5] and the Number Theory Library NTL [26, 18] have set world records for polynomial factorization and determining orders of elliptic curves. The book *Modern Computer Algebra* [12] has also contributed to increase the general interest of the computer algebra community for these algorithms.

As to linear algebra, in addition to MAGMA, let us mention the C++ template library LinBox [17] for exact linear algebra computation with dense, sparse, and structured matrices over the integers and over finite fields. A cornerstone of this library is the use of BLAS libraries such as ATLAS to provide high-speed routines for matrices over small finite fields, through floating-point computations [9].

Today, it is common practice to assume that a new algorithm, say for GCD computations over products of fields as in [8], can rely on asymptotically fast polynomial multiplication. Therefore, it is desirable not only to offer implementations of asymptotically fast arithmetic, but also programming environments for developing new such algorithms. In addition, it is also a demand to achieve this goal in the context of high-level programming languages, where

ISSAC'06, July 9–12, 2006, Genova, Italy.

new ideas can be tested quickly and where algorithms can be easily made generic. These are the goals of this paper, which reports on implementation techniques for asymptotically fast algorithms in two high-level programming environments, AXIOM and ALDOR. We focus on polynomial arithmetic and our test-operations are univariate and multivariate multiplication, and computations of power series inverses as well as GCDs for univariate polynomials.

Implementing asymptotically fast algorithms for these operations in a high-level programming environment presents several difficulties. First, compilation of high-level generic code to machine code through middle-level, say LISP, and low-level, say C, may lead to a running-time overhead, with respect to carefully hand-written C code. This may reduce the benefit of these algorithms, since they generally involve changes of data representation, whereas classical algorithms work usually in a straightforward manner. Minimizing this overhead is the motivation of our work in ALDOR where our entire code is written for univariate polynomials over an arbitrary field supporting the FFT. Second, compiled and optimized high-level code may not take advantage of some hardware features. If writing architecture-aware code can be done in C [20], this remains a challenge in a non-imperative language like LISP. Thus, in our second high-level programming environment, namely AXIOM, we take advantage of every component of the system, by mixing low-level code (C and assembly code), middle-level code (LISP) and high-level code in the AXIOM language. We develop specialized code for univariate and multivariate polynomials over $\mathbb{Z}/p\mathbb{Z}$ where p is a prime; we distinguish also the cases where p is machine word size and a big integer.

Section 2 contains an overview of the features of AXIOM and ALDOR systems. In Sections 3 and 4, we discuss our implementation techniques in the ALDOR and AXIOM environments. We compare our implementation of asymptotically fast algorithms with those of MAGMA and NTL. In Section 5 we report on our experiments. Our generic implementations in ALDOR are only, approximately, twice slower than those of NTL for comparable operations. Our specialized implementation in AXIOM leads to comparable performances and sometimes outperforms those of MAGMA and NTL. A review of the algorithms we implemented is given in appendix. All timings given in this article are obtained on a bi-Pentium 4, 2.80GHz machine, with 1 Gb of RAM.

2. HIGH LEVEL PROGRAMMING ENVIRONMENT

AXIOM and ALDOR designers attempted to surmount the challenges of providing an environment for implementing the extremely rich relationships among mathematical structures. Hence, their design is of somewhat different direction than that of other contemporary programming languages. They have a two-level object model of *categories* and *domains* that is similar to *Interfaces* and *Classes* in Java. They provide a type system that allows the programmer the flexibility to extend or build on existing types or create new type categories as is usually required in algebra.

In AXIOM and ALDOR, types and functions can be constructed and manipulated within programs dynamically like the way values are manipulated. This makes it easy to create generic programs in which independently developed components are combined in many useful ways. For instance, for a given AXIOM or ALDOR ring `R`, the domains `SUP(R)` and `DUP(R)`, for sparse and dense univariate polynomials respectively, provide exactly the same operations; that is they have the same user interface, which is defined by the category `UnivariatePolynomialCategory(R)`. But, of course, the implementation of the operations of `SUP(R)` and `DUP(R)` is quite different. While `SUP(R)` implements polynomials with linked lists of terms, `DUP(R)` implements them with arrays of coefficients indexed by their degrees. This allows us to specify a package, `FFTPolynomialMultiplication(R, U)`, parameterized by `R`, an `FFTRing`, that is, a ring supporting the FFT; and by `U`, a domain of `UnivariatePolynomialCategory(R)`.

2.1 The ALDOR environment

ALDOR can be used both as a compiled and interpreted language. Code optimization is however only available when used in compiled mode. An ALDOR program can be compiled into: stand-alone executable programs; object libraries in native operating system formats (which can be linked with one another, or with C or Fortran code to form application programs); portable byte code libraries; and C or Lisp source [16]. Code improvements by techniques such as program specialization, cross-file procedural integration and data structure elimination, are performed at intermediate stages of compilation [28]. This produces code that is comparable to hand-optimized C.

2.2 The AXIOM environment

AXIOM has both an interactive mode for user interactions and a high level programming language, called SPAD, for building library modules. In the interactive mode, users can evaluate arithmetic expressions, declare and define variables, call library functions and define their own functions. Programmers can also add new functions to the local AXIOM library. To do so, they need to integrate their code in AXIOM type constructors.

SPAD code is translated into COMMON LISP code by a built-in compiler, then translated into C code by the GCL compiler. Finally, GCL makes use of a native C compiler, such as GCC, to generate machine code. Since these compilers can generate fairly efficient code, programmers can concentrate on their mathematical algorithms and write them in SPAD. However, to achieve higher performance, our implementation also involves LISP, C, and assembly level code. By modifying the AXIOM makefiles, new LISP functions can be compiled and made available at SPAD level. Moreover, by using the GCL system provided *make-function* macro, one can add new C functions into the GCL system, then use them at the GCL and SPAD level. Finally, assembly code can either be inlined in C code or compiled into LISP images, and so available for LISP and SPAD level as well.

2.3 Implementation Strategies

In the case of ALDOR, we write optimizer-friendly and garbage collector (GC)-friendly code without compromising the high-level nature of our implementations. Thus, we achieve completely generic code. In the case of AXIOM, we put additional efforts on investigating the efficiency of the compiled code. The reasons are as follows. First, we are curious about how exactly a compiler can optimize our code, and what it cannot do for us. Second, our work is largely motivated by the implementation of modular methods. High performance for these methods relies on appropriately utilizing machine arithmetic as well as carefully constructing underlying data structures. This leads us to look into machine-level questions, such as machine integer arithmetic, memory hierarchy, and processor architecture. At this level, C and assembly code is preferred. Third, we are interested in parallel programming, which is not available at SPAD level, but can be achieved in LISP and C. Another reason for our LISP is to avoid some potential overhead.

By integrating our assembly and C functions into GCL compiler and our LISP code into its libraries, we are able to extend the AXIOM system at the middle and low level. At SPAD level we directly use these extended functionalities in both interpreter and compiler mode.

3. IMPLEMENTATION TECHNIQUES: THE GENERIC CASE

Our goal here is to implement algorithms with quasi-linear time complexities in a high-level programming environment (ALDOR), without resorting to low-level techniques [11]. The primary focus is not to outperform other implementations of similar algorithms in other platforms, but rather to ensure that we achieve the best in terms of space and time complexities in our target environment. This work will form part of the ALDOR library.

3.1 Efficiency-critical operations in ALDOR

We first discuss the techniques and results of our ALDOR implementation of two efficiency-critical algorithms: Fast Fourier Transform (FFT) and power series inversion.

FFT. We specify a FFT multiplication package that accepts a generic polynomial type, but performs all operations on arrays of coefficients, which are pre-allocated and released when necessary, without using the compiler's garbage collector. For coefficient fields of the form $\mathbb{Z}/p\mathbb{Z}$, ALDOR's optimizer produces code comparable to hand-optimized C code.

Power series inversion. We implemented two versions of the power series inversion algorithm. The "naive" version implemented the algorithm as is; then we implemented a space-efficient version, using the following ideas:

• We pre-determine all array sizes and pre-allocate all needed buffers, so that there is no memory allocation in the loop.

• Even though we accept a generic polynomial type, we change the data representation to arrays of coefficients, work only with these arrays, and reuse DFT's as much as possible.

• As in NTL, we use wrapped convolution to compute the n middle coefficients of a $(2n-1) \times n$ full product (this is the middle-product operation of [14]).

Figure 1 shows the runtimes of our two implementations, together with the time for a single multiplication, in a field of the form $\mathbb{Z}/p\mathbb{Z}$. We measured the maximum Resident Set Size; Figure 2 shows that the naive version used a total of over 16000 Kb to invert a polynomial of degree 8000 while the space efficient version used less than 2500 Kb for the same polynomial. For higher degrees, the factor is larger. We first give the source code of the naive version:

```
modularInversion(f:U,n:Z):U == {
 assert(one?(trailingCoefficient(f)));
 local m,g0,g__old,g__new,mi:U;
 m: == monom;
 g0:U:=1; g__old:U:=1; g__new:U:=1;
 local r,mii:MI;
 if PowerOfTwo?(n) then r := length(n)-1;
```

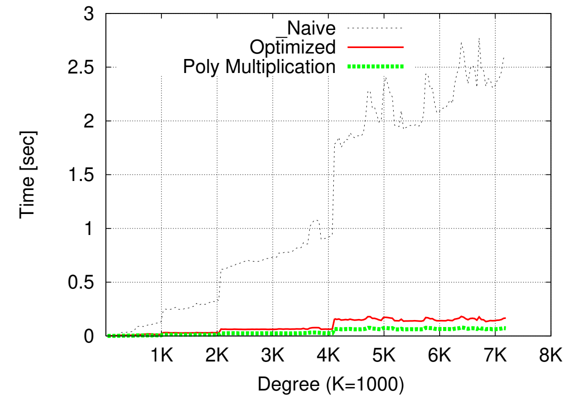

Figure 1: Power Series Inversion: naive vs. optimized implementation vs. multiplication, 27-bit prime.

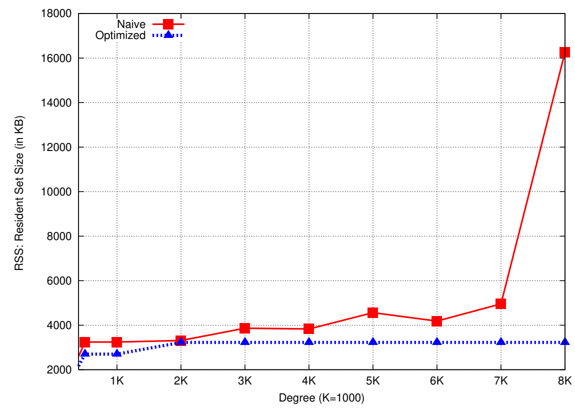

Figure 2: Power Series Inversion: Space usage of naive vs. optimized implementations, 27-bit prime.

```
 else  r := length(n);

 for i in 1..r repeat {
  mi := m^(2^i);
  g__new := (2*(g__old)-(f*((g__old)*(g__old)))) mod mi;
  g__old := g__new;
 }
 return (g__new);
}
```

Then follows the source code of the efficient version.

```
macro {
    U == DenseUnivariatePolynomial(K:Field);
    Z == AldorInteger;
}
fastModInverse(f:U,n:Z):U == {
  import from Z,MI;
  local dftf,dftg,Y,G,workspace,dftw,op,coeff:AK;
  local di__1,di,r,mii:MI; local res:U; local wi:K;

  if PowerOfTwo?(n) then r := length(n)-1;
    else  r := length(n);
  nn:MI := shift(1,r); -- 2^r
```

– **allocate storage**

```
  dftg := new(nn,0$K);
  Y := new(nn,0$K);
  G := new(nn,0$K);
  workspace := new(nn,0$K);
  op := new(nn,0$K);
```

– **stores** g_{i-1}

```
G.0 := 1$K;
dftg.0 := 1$K;
```

– **stores truncated f**

```
coeff := new(nn,0$K);
dftf := new(nn,0$K);
dftw := new(nn,0$K);
kk:MI := 0;
for k in coefficients(f) repeat {
 kk = nn => break;
 coeff.kk := k; kk := next(kk);
}
for i in 1..r repeat {
 mii := shift(1,i); -- 2^i
```

– **degree of** g_i

```
 di := mii - 1;
 w:Partial K := primitiveRootOfUnity(mii);
 wi := retract(w);
```

– **op stores OmegaPowers up to mii**

```
OmegaPowers!(op,wi,mii);
dftg := dft!(dftg,mii,i,op,workspace);
```

– $f \bmod X^{2^i}$**: truncates f**

```
for j in 0..di repeat dftf.j := coeff.j;
dftf := dft!(dftf,mii,i,op,workspace);
```

– **dftf*dftg pointwise**

```
  for j in 0..di repeat dftf.j := dftf.j*dftg.j;
  dftf := idft!(dftf,mii,i,op,workspace); -- invert dft
  di__1 := shift(1,i-1) - 1; --  degree of g_i_1
  ndi__1 := next di__1;
```

– **takes the end part**

```
  kk:=0;
  for j in ndi__1..di repeat {
    dftw.kk := dftf.j; kk:=next kk;
  }
  dftw := dft!(dftw,mii,i,op,workspace);
  for j in 0..di repeat dftg.j := dftg.j*dftw.j;
  dftg := idft!(dftg,mii,i,op,workspace);
```

– $X^{\mathrm{ndi\_\_1}} * Y$**: the middle product**

```
  for j in 0..di__1 repeat Y.(j+(ndi__1)) := dftg.j;
  for j in ndi__1..di repeat G.j := G.j - Y.j;
```

– **to allow dft! in-place of G, save G**

```
    for j in 0..di repeat dftg.j := G.j;
}
```

– **convert to polynomial**

```
 res := unvectorize(dftg,nn);
 free!(dftg); free!(dftf); free!(dftw); free!(workspace);
 free!(op); free!(coeff);
 return res;
}
```

3.2 Extended Euclidean Algorithm

We implemented the Half-GCD algorithms of [29] and [4], adapted to yield monic remainders. The timings in this paper are based on the adaptation of Yap's version. Though the algorithms are classic, we faced the difficulties on determining truncation degrees already experienced by others, see for instance [24] for the report on a variation of the integer Half-GCD in Mathematica. The algorithms given in appendix contain the changes we made.

Our implementation of Euclidean division uses power series inversion [12, Ch. 9], when the degree difference between two consecutive remainders is large enough. We used Strassen's algorithm [12, Ch. 13] for the 2×2 polynomial matrix multiplications; we plan to use it to perform the unbalanced matrix / vector multiplications as well. This implementation outperforms the standard Euclidean algorithm by a factor of 8 at degree 3000.

4. IMPLEMENTATION TECHNIQUES: THE NON-GENERIC CASE

Obtaining fast implementations of algorithms over fields of the form $\mathbb{Z}/p\mathbb{Z}$ requires low-level considerations of data structures, machine arithmetic, memory traffic, compiler optimization, etc [23]. In this section we discuss such techniques, applied to univariate polynomial algorithms, for our AXIOM implementation. This work will be integrated into the AXIOM CVS repository.

4.1 Data representation

We use *dense* polynomials: we have in mind to implement algorithms for solving polynomial systems, and experience shows that the univariate polynomials appeared in such applications tend to become dense, due to the use of the Euclidean algorithm, Hensel lifting techniques, etc.

Elements of the prime field $\mathbb{Z}/p\mathbb{Z}$ are encoded by integers in the range $0, \ldots, p-1$, thus using a fixed number of machine words to store each number. This allows us to use C-like arrays such as `fixnum-array` in Lisp to encode polynomials in $\mathbb{Z}/p\mathbb{Z}[X]$. If p is small, we tell the compiler to use machine integer arithmetic; for large p, we use the Gnu Multiple Precision library (GMP), adapting it to handle the arithmetic of polynomial coefficients.

Then, we accomplish such tasks as univariate polynomial addition or multiplication in C or assembly code for higher efficiency: we pass the arrays' references to our low-level code and return the result array to AXIOM.

We performed comparisons between the `SUP` constructor (from the SPAD level), and `UMA`, our dense univariate polynomials written in Lisp, C and assembly. Over a 64-bit prime field, `UMA` addition of polynomials is up to 20 times faster than `SUP` addition, in degree 30000; the quadratic `UMA` implementation of polynomial multiplication is up to 10 times faster than `SUP` multiplication, in degree 5000. FFT multiplication is discussed below.

With this data representation, we created a specialized AXIOM univariate polynomial domain for $\mathbb{Z}/p\mathbb{Z}$. It can be integrated into AXIOM library and used in a user-transparent way, since AXIOM supports *conditional implementation*.

Similarly, we have implemented a specialized multivariate polynomial domain over $\mathbb{Z}/p\mathbb{Z}$. The operations in this domain are mostly implemented at the Lisp level which offers us more flexibility (less type checking, better support from the machine arithmetic) than at the SPAD level, where objects are strongly-typed. We follow the *vector-based approach* proposed by Fateman [10] where a polynomial is either a number or a vector: If a coefficient is a polynomial, then the corresponding slot keeps a pointer to that polynomial or, say, another vector; otherwise, if the coefficient is a number, the slot keeps the pointer to this number.

4.2 FFT

Our implementation of FFT-based univariate polynomial multiplication in $\mathbb{Z}/p\mathbb{Z}[X]$ distinguishes the cases of small (single-precision) primes and big (multiple-precision) primes. For both cases, we used the algorithm of [7] and techniques discussed in Subsection 4.3 below. However, the big prime case requires extra efforts, since two strategies are available. One can directly implement the DFT algorithm on big integers, by adapting the code of the small prime case to the big prime case. Alternatively, one can use the Chinese Remainder Theorem (CRT) based approach, which reduces the big prime problem into 2 or more small prime problems [26, 12].

Figure 3 shows a comparison between these approaches. We put special effort on the big prime case, rewriting some GMP low-level functions. Figure 3 shows that the specialized double precision big prime functions and CRT approaches are faster than the generic GMP functions. The CRT recombination part spends a negligible 0.06% to 0.07% percent of the time in the whole FFT algorithm.

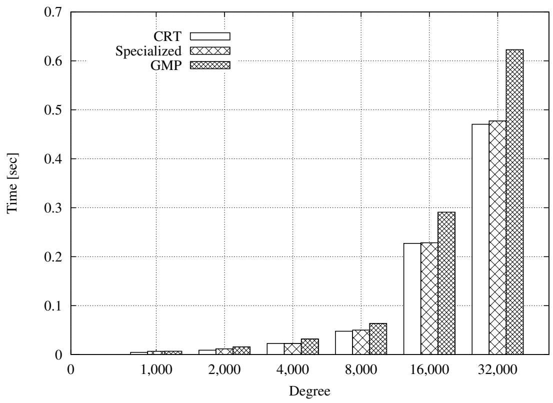

Figure 3: FFT multiplication: GMP functions vs. double precision integer functions vs. CRT, 64 bit prime.

4.3 SSE2, loop unrolling, parallelism

Modern compilers can generate highly efficient code, but sometimes do not provide the highest efficiency. We show three examples of hand-tuned improvements from our FFT implementation; timings are reported for small primes.

Single precision integer division with SSE2. The single precision modular reduction uses floating point arithmetic, based on the formula $a \equiv a - \lfloor a * \ 1/p \rfloor * p$ [26]. We implemented this idea in assembly for the Pentium IA-32 architecture with SSE2 support. This set of instructions is of Single Instruction Multiple Data style, since they make use of XMM registers which pack 2 double floats or 4 single floats/integers in one single register. The following sample code computes $(a * b) \mod p$ with SSE2 instructions.

1 movl RPTR, %edx	11 movups (%eax), %xmm0
2 movl WD1, %eax	12 cvttpd2pi %xmm2, %mm2
3 movl WPD1, %ecx	13 cvtpi2pd %mm2, %xmm2
4 movq (%edx), %mm0	14 mulpd %xmm2, %xmm0
5 movups (%eax), %xmm1	15 subpd %xmm0, %xmm1
6 cvtpi2pd %mm0, %xmm0	16 cvttpd2pi %xmm1, %mm1
7 movups (%ecx), %xmm2	17 movq %mm1, (%edx)
8 movl PD, %eax	18 emms
9 mulpd %xmm0, %xmm1	19 ret
10 mulpd %xmm0, %xmm2	

Figure 4 shows that our SSE2-based FFT implementation is significantly faster than our generic assembly version.

Reducing loops overhead. Many algorithms operating on dense polynomials have an iterative structure. One major overhead for such algorithms is loop indexing and loop condition testing. We can reduce this by unrolling loops. This feature is provided by some compilers, for instance by setting GCC's `funroll-loops` flag.

However, optimally setting the number of iterations a compiler will unroll is subtle. There is a trade-off: unrolled loops require less loop indexing, but they suffer from code size growth, which will aggravate the burden of instruction

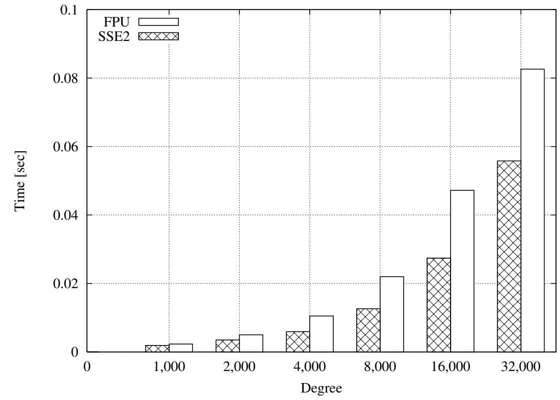

Figure 4: FFT multiplication: Generic assembly vs. SSE2 **assembly, 27-bit prime.**

caching. If the loop body contains branching statements, increased number of branches in each iteration will have a negative impact on branch prediction. Hence, compilers and interpreters usually do static or run-time analysis to decide how much to unroll a loop. However, the analysis may not be precise when loops become complex and nested. Moreover, compilers are very cautious when unrolling loops, since this may change the original program's data dependency. In addition, optimizing compilers usually do not check if there is a possibility to combine unrolled statements together for better performance. Hence, we have unrolled some loop structures by hand, and recombined the related statements into small assembly functions. This allows us to keep some values in registers or evict those unwanted ones. The following is a fragment of our implementation of the FFT-based univariate polynomial multiplication.

```
#include "fftdfttab_4.h"
typedef void (* F) (long int *, long int,  long int,
                    long int *, long int, int);
typedef void (* G) (long int *, long int *,
                    long int *, long int, int);
inline void
fftdftTAB_4( long int * a, long int * b, long int * w,
                    long int p, F f, G g1, G g2 ){
long int w0=1, w4=w[4], * w8=w+8;
f(a, w0, w4, a+2, p, 8); g2(a+4, w8, a+8, p, 4);
g2(a+12, w8, a+16, p, 4); g1(a+8, w8, a+16, p, 8);
f(b, w0, w4, b+2, p, 8); g2(b+4, w8, b+8, p, 4);
g2(b+12, w8, b+16, p, 4); g1(b+8, w8, b+16, p, 8); return;}
```

This function is dedicated to compute the case where $n = 4$ in the FFT algorithm. The functions `f, g1, g2` are small assembly functions which recombine related statements for higher efficiency. We also developed similar functions for the cases $n = 5$ to 8. However, for $n \geq 6$, these straight-line functions are less efficient than the ones using nested loops, for the reasons discussed above. Figure 5 shows that for small degrees, the inlined version may gain about 10% running time. This is significant, since our experiments show that 50% is already spent in performing integer divisions.

Parallelism. Parallelism is a fundamental technique used to achieve high performance. In the FFT-based polynomial multiplication, the DFT of the input polynomials are independent, hence, they can be computed simultaneously. Another example is the (standard) Chinese remaindering algorithm, where the computations w.r.t. each modulo can be

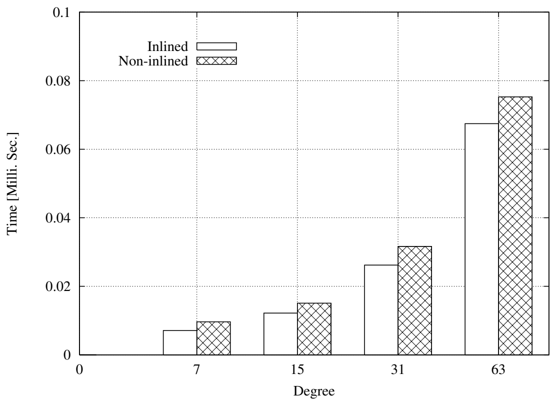

Figure 5: FFT multiplication: Inlined vs. non-inlined, 27-bit prime.

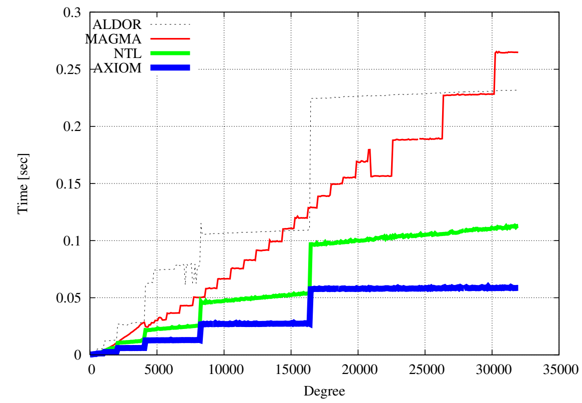

Figure 6: Multiplication modulo a 27-bit prime.

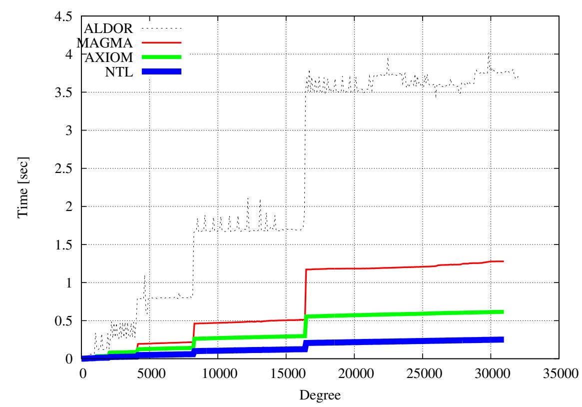

Figure 7: Multiplication modulo a 64-bit prime.

performed simultaneously. This can be achieved by thread-level parallelism. Under the Linux environment, we directly use the native Posix Thread Library to conduct parallel programming, since AXIOM's compiler is not able to provide this kind of optimization. The parallelized version of the FFT-based multiplication is 7% to 10% faster than the non-parallelized one on a dual CPU machine. This part of work is still work in process; the performance is still not satisfying, since we expect a 20–30 percent speed up.

5. PERFORMANCES

5.1 FFT multiplication

We compared our implementations with their counterparts in NTL and MAGMA. For NTL-v5.4, we used the functions `FFTMul` in the classes `zz_p` and `ZZ_p`, respectively for small and big primes. For MAGMA-v2.11-2, we used the general multiplication function "*" over GF(p), the prime field of order p. The input polynomials are randomly generated, with no zero term. Figures 6 and 7 give our timings. Our AXIOM implementation is faster than NTL over small primes, but slower than NTL over big primes; it is faster than MAGMA and other known computer algebra systems in both cases. One possible reason is that NTL re-arranges the computations in a "cache-friendly" way. Our generic ALDOR implementation is comparable to MAGMA's one, though generally slower in our range of degrees.

5.2 Multivariate multiplication

We compute the product of multivariate polynomials via the Kronecker substitution (see the appendix). Recall that we use vector-based recursive representation for multivariate polynomials, and one-dimensional arrays for univariate ones. So, the forward substitution simply copies coefficients from the coefficient tree of a multivariate polynomial to the coefficient array of a univariate polynomial. We use a recursive depth first tree walk to compute all the univariate polynomial exponents from the corresponding multivariate monomials' exponents; at the same time, according to this correspondence we conduct the forward substitution. We use the same idea for the backward substitution The comparisons between MAGMA and our AXIOM code are given in Figures 8 to 10, where "degree" denotes the degree of the univariate polynomials obtained through Kronecker's substitution. We used random inputs, with no zero term.

Our FFT-based multivariate polynomial multiplication over $\mathbb{Z}/p\mathbb{Z}$ outperforms MAGMA's in these cases. From Figure 8, we may infer that MAGMA is in the "classical multiplication" stage; our FFT-based implementation is already faster. From Figures 9, 10 we observe that both our and MAGMA's FFT's show the usual FFT staircase-like curves.

5.3 Power series inversion

We compare here our power series inversion, in the optimized ALDOR version, with NTL and MAGMA implementations. MAGMA offers a built-in `InverseMod` function (called "builtin" in the figure), but the behavior of this generic function is that of an extended GCD computation. We also tested the MAGMA `PowerSeriesRing` domain inversion (called "powerseries" in the figure), and our own implementation of the Newton iteration. Figure 11 shows the relative performances: NTL is the fastest in this case, and ALDOR is second, within a factor of 2.

5.4 Fast Extended Euclidean Algorithm

Section 3.2 reported the relative performance between the existing standard Euclidean algorithm in ALDOR and our implementation of the fast algorithm. We also compared our generic fast algorithm with the existing implementations in

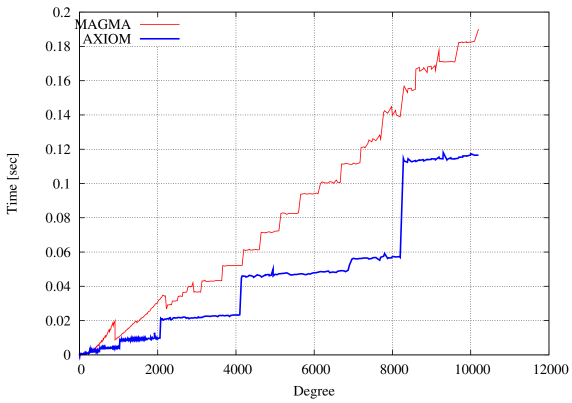

Figure 8: Bivariate multiplication, 27-bit prime.

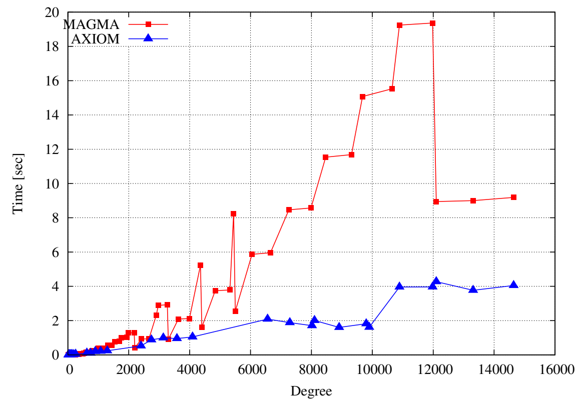

Figure 10: 4-variable multiplication, 64-bit prime.

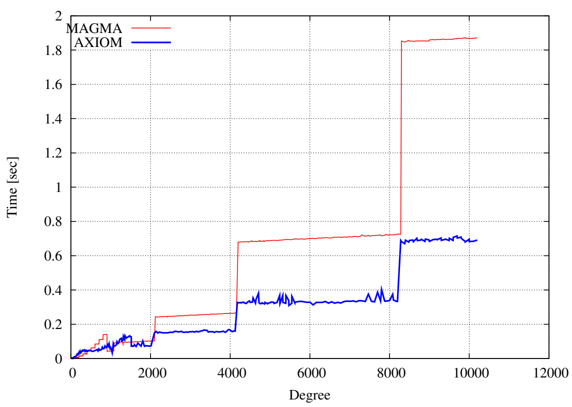

Figure 9: Bivariate multiplication, 64-bit prime.

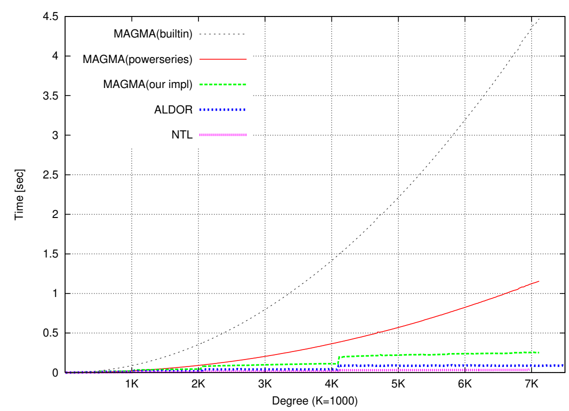

Figure 11: Power series inversion: Aldor vs. NTL vs. MAGMA, 27-bit prime.

NTL and MAGMA. Unlike ours, the NTL implementation is not over a generic field but over a finite field, and uses improvements like FFT-based polynomial matrix multiplication. MAGMA's performance differ, according to whether we use the `GCD` or `XGCD` commands: we report on both. Figure 12 shows the relative performances; our input were degree d polynomials, with a GCD of degree $d/2$. Again, NTL is the fastest and ALDOR is second, within a factor of 2.

6. CONCLUSION AND FUTURE WORK

The work reported in here is the beginning of a larger scale effort; it has raised several new objectives. Regarding the low-level development, we are implementing the Truncated Fourier Transform [15] and developing cache-friendly code for this algorithm, using the strategies of [20]. The GCD computation deserves low-level work as well: we wish to develop a version making the best use of FFT, and of techniques such as the middle product. Having in mind to implement algorithms such as the *coprime factorization* algorithm of [8], we still need several basic algorithms on univariate and multivariate polynomials: Chinese remaindering techniques and (sub)resultant algorithms. These tools form the basic algorithms for a further goal, polynomial systems solving algorithms.

7. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *The Journal of Supercomputing*, 4(4):357–371, 1990.

[3] R. P. Brent. Algorithms for matrix multiplication. Master's thesis, Stanford University, 1970. `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/`.

[4] R. P. Brent, F. G. Gustavson, and D. Y. Y. Yun. Fast solution of Toeplitz systems of equations and computations of Padé approximants. *Journal of Algorithms*, 1:259–295, 1980.

[5] The Computational Algebra Group in the School of Mathematics and Statistics at the University of Sydney. *The MAGMA Computational Algebra System*. `http://magma.maths.usyd.edu.au/magma/`.

[6] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2002.

[8] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *TC'06*, 2006.

[9] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC 02*, pages 63–74. ACM, 2002.

[10] R. J. Fateman. Vector-based polynomial recursive representation arithmetic. 1990. `http://www.norvig.com/ltd/test/poly.dylan`.

[11] A. Filatei. Implementation of fast polynomial arithmetic in Aldor, 2006. University of Western Ontario.

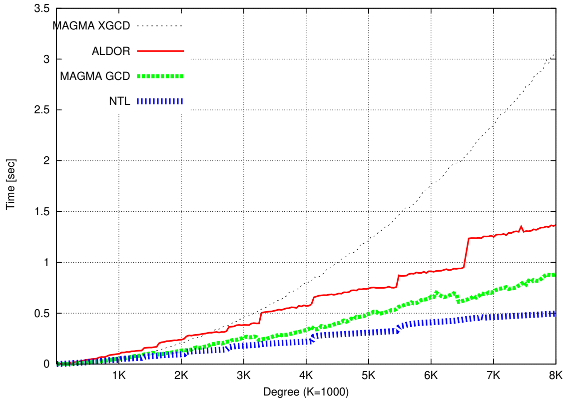

Figure 12: EEA: ALDOR vs. NTL vs. MAGMA, 27-bit prime.

[12] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

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

[14] G. Hanrot, M. Quercia, and P. Zimmermann. The middle product algorithm, I. *Appl. Algebra Engrg. Comm. Comput.*, 14(6):415–438, 2004.

[15] J. van der Hoeven. The Truncated Fourier Transform and applications. In *ISSAC 04*, pages 290–296. ACM, 2004.

[16] http://www.aldor.org. *The* Aldor *compiler web site*. The University of Western Ontario, 2002.

[17] http://www.linalg.org/. *LinBox*. The LinBox group, 2005.

[18] http://www.shoup.net/ntl. *The Number Theory Library*. V. Shoup, 1996–2006.

[19] R. D. Jenks and R. S. Sutor. *AXIOM the Scientific Computation System*. Springer-Verlag, 1992.

[20] J. R. Johnson, W. Krandick, and A. D. Ruslanov. Architecture-aware classical Taylor shift by 1. In *ISSAC 05*, pages 200–207. ACM, 2005.

[21] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys. Dokl.*, (7):595–596, 1963.

[22] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1999.

[23] X. Li. Efficient management of symbolic computations with polynomials, 2005. University of Western Ontario.

[24] D. Lichtblau. Half-gcd and fast rational recovery. In *ISSAC 05*, pages 231–236. ACM, 2005.

[25] R. T. Moenck. Practical fast polynomial multiplication. In *SYMSAC 76*, pages 136–148, ACM, 1976.

[26] V. Shoup. A new polynomial factorization algorithm and its implementation. *J. Symb. Comp.*, 20(4):363–397, 1995.

[27] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik.*, 13:354–356, 1969.

[28] S. M. Watt. The $A^{\#}$ programming language and its compiler. Technical report, IBM Research, 1993.

[29] C. K. Yap. *Fundamental Problems in Algorithmic Algebra*. Princeton University Press, 1993.

8. APPENDIX

In this section, we describe, or give references to, the basic algorithms we implemented. All rings and fields are commutative with 1; we denote by M a multiplication time function [12, Ch. 8].

A 1: Fast Fourier Transform. Let $\mathbb{A}$ be a ring, and $n \in \mathbb{N}$. The Discrete Fourier Transform (DFT) of $a \in \mathbb{A}[X]$ is the evaluation of a at the powers $1, \omega, \dots, \omega^{n-1}$, where ω is a primitive nth root of unity in $\mathbb{A}$ [12, Ch. 8]. When n is a power of 2, and when $\deg a \le n$, the *Fast Fourier Transform*, or FFT [6], performs this operation in complexity $O(n \log(n))$. We implemented the iterative version of the FFT given in [7], assuming that ω is known. Hence, for rings supporting FFT, with known primitive roots, we can take $\mathsf{M}(n) \in O(n \log(n))$.

A 2: Power series inversion. We used the algorithm for modular inversion using Newton iteration [12, Ch. 9]. This algorithm takes as input a polynomial (or power series) f with coefficients in a ring $\mathbb{A}$, with $f(0) = 1$, and an integer $\ell \in \mathbb{N}$, and outputs the polynomial $g \in \mathbb{A}[X]$ such that $fg \equiv 1 \bmod x^\ell$. The complexity is $3\mathsf{M}(\ell) + O(\ell)$ operations in $\mathbb{A}$.

A 3: Half-GCD. We now discuss (extended) GCD over a field. Let $\mathbb{K}$ be our base field, let a, b be in $\mathbb{K}[X]$, with $\deg b \le \deg a$, and write $d = \deg a$. The half-GCD algorithm [1] returns a matrix $M = \mathrm{M}_{\mathrm{hgcd}}(a, b)$ such that if (t, s) are defined by $(t, s)^{\mathrm{T}} = M(a, b)^{\mathrm{T}}$, then we have $\deg t \ge d/2 > \deg s$, and t and s are consecutive polynomials of degrees straddling d in the Euclidean remainder sequence associated to a and b. We implemented the following adaptation of Yap's [29] version of the half-GCD algorithm that yields monic remainders. The complexity, and that of the subsequent GCD algorithm, are in $O(\mathsf{M}(d) \log(d))$ operations in $\mathbb{K}$.

$\mathrm{M}_{\mathrm{hgcd}}$(a,b) ==
$d := \deg(a)$; $m := \lceil \frac{d}{2} \rceil$;
if $\deg(b) < m$ **then return** $\left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)$
$a^{\uparrow} := a$ quo x^m
$b^{\uparrow} := b$ quo x^m
$M_1 := \mathrm{M}_{\mathrm{hgcd}}(a^{\uparrow}, b^{\uparrow})$
$\left(\begin{smallmatrix}t\\ s\end{smallmatrix}\right) := M_1 \left(\begin{smallmatrix}a\\ b\end{smallmatrix}\right)$
if $s = 0$ **then return** M_1
$(q, r) :=$ QuotientRemainder(t, s)
if $r = 0$ **then**
    $M_2 := \left(\begin{smallmatrix}0 & 1\\ 1 & -q\end{smallmatrix}\right)$
    return $M_2 M_1$
$v :=$ LeadingCoefficient$(r)^{-1}$
$\overline{r} := rv$
$M_2 := \left(\begin{smallmatrix}0 & 1\\ v & -vq\end{smallmatrix}\right)$
$\ell := 2m - \deg(s)$
$s^{\uparrow} := s$ quo x^ℓ
$r^{\uparrow} := \overline{r}$ quo x^ℓ
$M_3 := \mathrm{M}_{\mathrm{hgcd}}(s^{\uparrow}, r^{\uparrow})$
return $M_3 M_2 M_1$

Using the half-GCD algorithm, one deduces the GCD algorithm itself. Taking as input a and b, with $\deg b \le \deg a$, it outputs the matrix M of cofactors such that $M(a, b)^{\mathrm{T}}$ equals $(g, 0)^{\mathrm{T}}$, where g is the monic GCD of a and b.

$\mathrm{M}_{\mathrm{gcd}}$(a,b) ==
$M_1 := \mathrm{M}_{\mathrm{hgcd}}(a, b)$;
$\left(\begin{smallmatrix}t\\ s\end{smallmatrix}\right) := M_1 \left(\begin{smallmatrix}a\\ b\end{smallmatrix}\right)$
if $s = 0$ **then return** M_1
$(q, r) :=$ QuotientRemainder(t, s)
if $r = 0$ **then**
    $M_2 := \left(\begin{smallmatrix}0 & 1\\ 1 & -q\end{smallmatrix}\right)$
    return $M_2 \times M_1$
$v :=$ LeadingCoefficient$(r)^{-1}$
$M_2 := \left(\begin{smallmatrix}0 & 1\\ v & -vq\end{smallmatrix}\right)$
$\overline{r} := rv$
$M_3 := \mathrm{M}_{\mathrm{gcd}}(s, \overline{r})$
return $M_3 M_2 M1$

A 4: Kronecker's substitution. Let $\mathbb{A}$ be a ring and let $X_1, \dots, X_n$ be indeterminates over $\mathbb{A}$. Given positive integers $\alpha = (\alpha_1 = 1, \alpha_2, \dots, \alpha_n)$, we define a ring homomorphism $\Psi_\alpha : \mathbb{A}[X_1, \dots, X_n] \to \mathbb{A}[X_1]$, by letting $\Psi_\alpha(X_i) = X_1^{\alpha_i}$. This homomorphism is used to reduce multivariate to univariate multiplication, as follows. Let $f, g \in \mathbb{A}[X_1, X_2, \dots, X_n]$ and let $p = fg$. For all $1 \le i \le n$ we let $d_i = \deg(f, X_i) + \deg(g, X_i)$, and we define $\delta_0 = 0$ and $\delta_i = \sum_{j=1}^{i} \alpha_j d_j$, with $\alpha_j = \delta_{j-1} + 1$. We can then compute p using the following simple algorithm:

MultivariateProduct(f,g) ==
$u_f := \Psi_\alpha(f)$
$u_g := \Psi_\alpha(g)$
$u_{fg} := u_f u_g$
$p := \Psi_\alpha^{-1}(u_{fg})$
return p

This algorithm runs in $\mathsf{M}((d_1 + 1) \cdots (d_n + 1))$ operations in $\mathbb{A}$.