Arithmetic Software Libraries*

Victor Shoup
(New York University)

1 Introduction

This chapter discusses NTL, a library for doing number theory, as well as its relation to a few other libraries.

NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields. It is maintained and mostly written by the author, although a number of contributions have been made by others.

1.1 NTL, LIP, and GMP

Work on NTL started around 1990, when the author wanted to implement some new algorithms for factoring polynomials over finite fields and related problems. It seemed that none of the publicly available software was adequate for this task, mainly because the code for polynomial arithmetic offered by this software was too slow. More precisely, many papers by the author and others working in this field would give big-O estimates for algorithms under the assumption of fast (i.e., quasi-linear) polynomial multiplication algorithms; however, the publicly available software at the time did not implement any of these fast multiplication algorithms. Thus, to turn these theoretical results into practically useful implementations, the author began working on software for fast polynomial arithmetic over finite fields. This software eventually grew into a full-fledged library, and eventually led to the public release of NTL version 1.0 in 1997. NTL has evolved quite a bit since then, with a lot of new functionality and algorithmic improvements, as well as features like thread-safety, and exploitation of multi-core and SIMD hardware.

The starting point for NTL was Arjen Lenstra's LIP package for long (i.e., multi-precision) integer arithmetic, which was written in C. LIP was originally written by Arjen Lenstra in the late 1980s. Later, in 1995, a version of LIP called FreeLIP was released and maintained by Paul Leyland. It soon became clear that using C++ instead of C would be much more productive and less prone to errors, mainly because of C++'s constructors and destructors, which allowed memory management to be automated. Using C++ had other benefits as well, like function and operator overloading, which

*This material (written January 2020) will be published in revised form in *Computational Cryptography*, edited by Joppe W. Bos and Martijn Stam and published by Cambridge University Press. See https://www.cambridge.org/9781108795937.

makes for more readable code. So an essential part of NTL's original design was a lightweight C++ wrapper around LIP to manage the storage of long integers. For example, suppose one wanted to write a program that input integers a, b, c, d, and output $ab + cd$. This could be written in LIP as

```
verylong a=0, b=0, c=0, d=0, t1=0, t2=0;
zread(&a); zread(&b); zread(&c); zread(&d);
zmul(&t1, a, b); zmul(&t2, c, d);
zadd(&t1, t1, t2); zwriteln(t1);
zfree(&a); zfree(&b); zfree(&c); zfree(&d);
zfree(&t1); zfree(&t2);
```

while in NTL, this could be written as

```
ZZ a, b, c, d;
cin >> a >> b >> c >> d;
cout << (a*b + c*d) << "\n";
```

Among other things, LIP was designed to be used in widely distributed integer factorization projects, and so portability, without sacrificing too much performance, was a major design feature of LIP. Indeed, quoting from the LIP documentation itself: "This very long int package is supposed to be easy to use, portable, and not too slow." This design principle was adopted for NTL as well, for a number of reasons, not the least of which was that the author's computing environment kept changing whenever he changed jobs (which was unfortunately happening quite frequently in those days), and so portability was a real issue. Achieving portability eventually became easier as standards, like IEEE floating-point [9], got widely adopted, as the definition of and implementations of the C++ language became standardized and more stable, and as important language extensions, such as double-word integer arithmetic, became more widely available.

When work started on NTL, besides LIP, there were not that many good, portable long integer packages around. Shortly thereafter, Torbjörn Granlund released (in 1991) the first version of GMP (the GNU MP library) [10]. GMP itself was largely compatible with the earlier Berkeley MP library, but it was quite a bit more efficient. From the documentation of one of the first public releases of GMP (v1.1, Sept. 29, 1991): "The speed of GNU MP is about 5 to 100 times that of Berkeley MP for small operands. The speed-up increases with the operand sizes for certain operations, where GNU MP has asymptotically faster algorithms." At that time, it was not clear that GMP was going to be developed and maintained as well as it ultimately proved to be, so NTL continued to use LIP for its long integer arithmetic. However, as time went on, GMP became much faster relative to LIP, and was very well supported across many platforms, and so NTL was eventually (in 2000) restructured to use GMP instead of LIP. For the most part, this restructuring was invisible to NTL users, as these implementation details are all hidden behind a layer of abstraction. Moreover, the current version of NTL can still be built without GMP, in which case it reverts to logic that is derived from LIP (although by now, even that logic has been almost entirely restructured). LIP itself has apparently not been supported for many years.

1.2 A quick tour of NTL

As already mentioned above, the initial purpose for working on NTL was to implement some new algorithms for factoring polynomials over finite fields. The author's early work in this area was reported in [27]. Besides the class ZZ mentioned above, which represents multi-precision integers, NTL also provides a class ZZ_pX, which represents the ring of univariate polynomials $\mathbb{Z}_p[X]$ over the quotient ring $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$, where p is a multi-precision integer (not necessarily prime). NTL also provides a corresponding class zz_pX, optimized for single-precision p, and a class GF2X, optimized for the important special case of $p = 2$. In addition, NTL provides classes for the ring of univariate polynomials $E[X]$ over the quotient ring $E = \mathbb{Z}_p[T]/(m(T))$, where $m(T) \in \mathbb{Z}_p[T]$. These classes are called ZZ_pEX, zz_pEX, and GF2EX, respectively, depending on whether p is multi-precision, single-precision, or equal to 2. For each of these polynomial classes, NTL provides algorithms for all of the basic arithmetic operations, and when the coefficient domain is a field, for other operations, such as GCD, factoring, minimal polynomial calculation, and generating irreducible polynomials.

In addition to polynomials over finite rings and fields, NTL provides the class ZZX, which represents the ring $\mathbb{Z}[X]$ of univariate polynomials over the integers, where coefficients are of arbitrary size. Naturally, NTL provides algorithms for all of the basic arithmetic operations over $\mathbb{Z}[X]$, as well as GCD and factoring algorithms.

Unfortunately, at the present time, NTL does not provide any support for multivariate or even bivariate polynomials.

NTL provides support for matrices over finite fields and rings, as well as over the integers. For integer matrices, NTL provides implementations of various lattice basis reduction algorithms, including (fast, heuristic) algorithms that are based on floating-point arithmetic. In support of these floating-point lattice basis reduction routines, and for other applications as well, NTL also provides specialized floating-point classes.

NTL is licensed under the GNU LGPL license, and is available at `www.shoup.net/ntl`.

1.3 Outline

In Section 2, we describe some of the technical details of how LIP and NTL originally implemented long integer arithmetic, and compare the performance of these techniques to GMP's performance. In Section 3, we describe NTL's implementation techniques for the Number Theoretic Transforms (NTT), i.e., the Fast Fourier Transform (FFT) for small primes. As we shall see, the NTT plays a central role in a number of NTL's other arithmetic modules. In Section 4, we discuss how NTL implements arithmetic in $\mathbb{Z}_p[X]$ for multi-precision p. In Section 5, we discuss how NTL implements arithmetic in $\mathbb{Z}_p[X]$ for single-precision p. In Section 6, we discuss how NTL implements matrix arithmetic over $\mathbb{Z}_p$. In Section 7, we discuss how NTL implements polynomial and matrix arithmetic over other finite rings. Finally, in Section 8, we discuss how NTL implements polynomial and matrix arithmetic over the integers, including a brief discussion of NTL's lattice basis reduction algorithms. In Section 9 we conclude with a brief look into the future of NTL.

2 Long integer arithmetic

In this section, we describe some of the technical details of how LIP and NTL originally implemented long integer arithmetic. These details are mostly of only historical interest, as NTL now mainly relies on GMP for its long integer arithmetic.

As mentioned above, LIP was designed to be highly portable, but not too slow, and NTL adopted this philosophy as well. One way in which LIP initially achieved a good balance between portability and performance was to avoid the use of double-word integer arithmetic, and instead, to use double-precision floating-point to achieve similar functionality.

By "double-word integer arithmetic", we mean some mechanism that allows one to multiply two single-word integers to get the double-word product, as well as related operations such as adding a single- or double-word integer to a double-word integer. At the time NTL was initially developed, some hardware simply did not have any implementation at all of double-word integer arithmetic, and even if it did, its implementation could often be fairly inefficient compared to its floating-point implementation; moreover, programming languages and compilers often did not give the programmer access to this hardware, and so exploiting such hardware would require writing and maintaining a lot of assembly code (which, among other things, is exactly what GMP does).

It might seem surprising to use floating-point arithmetic, which is inherently subject to rounding errors, to implement long integer arithmetic, which does not tolerate any errors. Nevertheless, under reasonable assumptions, this can indeed be done. The assumptions we need are simply bounds on the relative error introduced when performing the basic arithmetic operations of addition, subtraction, multiplication, and division. These assumptions are certainly implied by the IEEE floating-point standard, which was already being quickly adopted at the time of NTL's initial development.

On 32-bit machines (which were ubiquitous in the early days of NTL development), LIP represents long integers in base R, where $R = 2^{30}$. Thus, a "base-R digit" is an integer in the interval $[0, R)$. A basic operation that is used in several algorithms, including long integer multiplication, is *addmulp*, which takes as input base-R digits a, b, d, and t, computes the two base-R digits representing $b \cdot d + a + t$, and stores the high-order digit back in t and the low-order digit back in a.

The original implementation of *addmulp* in LIP was essentially that presented in Fig. 1.[1] Here, we are assuming that the types `long` and `unsigned long` represent 32-bit integers. To see why this works, let *lo* and *hi* denote the low-order and high-order digit, respectively, of $b \cdot d + a + t$. Observe that the value of $t3$ equals *lo*, since unsigned arithmetic is guaranteed to be done correctly modulo 2^{32}. Also observe that

$$hi = (b \cdot d + a + t - lo)/R.$$

Now, the value of $d1$ is

$$bd(1 + \theta_1),$$

[1] Actually, the code in Fig. 1 is somewhat more robust and efficient than the original LIP implementation, but it is still in the same spirit.

```
typedef long L;
typedef unsigned long U;
typedef double D;
const int NBITS = 30;
const U R = U(1) << NBITS;

inline void addmulp(U& a, U b, U d, U& t)
{
   U t1 = b * d;
   U t2 = a + t;
   U t3 = (t1+t2) & (R-U(1));

   double d1 = D(L(b))* D(L(d));
   double d2 = d1 + D( L(t2) - L(t3) + L(R/2) );
   double d3 = d2 * (1.0 / D(L(R)) );

   t = L(d3);
   a = t3;
}
```

Figure 1: A floating-point implementation of *addmulp*

where $|\theta_1| \leq \epsilon := 2^{-53}$ on a machine that correctly implements the IEEE floating-point standard. The value of $d2$ is

$$(d1 + a + t - t3 + R/2)(1 + \theta_2) = bd(1 + \theta_1)(1 + \theta_2) + (a + t - lo + R/2)(1 + \theta_2),$$

where $|\theta_2| \leq \epsilon$. Finally, the value of $d3$ is $d2/R$. It follows that

$$d3 = hi + 1/2 + \theta_3,$$

where

$$|\theta_3| \leq R|\theta_1 + \theta_2 + \theta_1\theta_2| + \frac{2.5}{R}|\theta_2|.$$

Therefore, since $R = 2^{30}$ and $|\theta_1|$ and $|\theta_2|$ are both at most $\epsilon = 2^{-53}$, we see that $|\theta_3|$ is at most approximately $2R\epsilon = 2^{-22}$, and so the calculation of $L(d3)$, which drops the fractional part of $d3$, will certainly yield hi, as required.

On 64-bit machines (which eventually replaced 32-bit machines in their ubiquity), the same analysis allows us to prove that *addmulp* works correctly with $R = 2^{50}$. In that case, the value $|\theta_3|$ above is at most approximately $2R\epsilon = 1/4$, which is still small enough to ensure that $L(d3) = hi$.

Some implementation notes. Note that all conversions between integers and floating-point values in Fig. 1 are done from or to *signed* integers, the reason being that many machines implement such conversions much more efficiently than the corresponding

```
inline void addmulp(U& a, U b, U d, U& t)
{
   U t1 = b*d + a + t;
   U t2 = D(L(b))*D(L(d))*(1.0/D(L(R)));
   L t3 = L(t1 - (t2 << NBITS)) >> NBITS;
   t = t2 + t3;
   a = t1 & (R-U(1));
}
```

Figure 2: Another floating-point implementation of *addmulp*

```
inline void addmulp(U& a, U b, U d, U& t)
{
   UU t1 = UU(b)*UU(d) + UU(a+t);
   a = U(t1) & (R-U(1));
   t = U(t1 >> NBITS);
}
```

Figure 3: A double-word integer implementation of *addmulp*

conversions from or to *unsigned* integers. Also note that the above correctness analysis only depended on the relative error guarantees provided by the IEEE floating-point standard, and not on any "exact rounding" requirements. Thus, the analysis is still valid, even in the presence of "double rounding", which can occur on platforms which perform some calculations in extended double precision (such as old x86 machines that use the x87 floating-point instructions), or in the presence of "contractions", which can occur in implementations that provide "fused multiply and add" instructions (such as newer x86 machines).

Fig. 2 presents another implementation of *addmulp*, which was introduced in the original version of NTL, and which replaced LIP's implementation of *addmulp*. This implementation works on 32-bit machines (with $R = 2^{30}$) or on 64-bit machines (with $R = 2^{50}$). It also assumes that signed integer arithmetic is two's complement, conversion from unsigned to signed works as expected (i.e., the bit pattern is unchanged), and that right shifts of signed integers are arithmetic shifts.[2]

Under our relative-error assumptions for floating-point arithmetic, we have $\lfloor bd/R \rfloor = t2 + \delta$, where $\delta \in \{0, \pm 1\}$. It follows that $hi = t2 + \delta'$, where $\delta' \in \{-1, \ldots, 3\}$. The calculation of $t3$ uses the high-order bits of $t1$ to compute the value $t3 = \delta'$, which is then added to $t2$ to compute the correct value of hi.

Eventually, NTL was updated to exploit double-word integer arithmetic if possible (and desired). This is done by either using a special type or by using inline assembly.[3]

[2]Here, and elsewhere, we make some assumptions about signed arithmetic which technically leads to "implementation defined" behavior; however, the code can easily be adapted to use only unsigned arithmetic.

[3]Inline assembly is only implemented on x86 machines. On many 32-bit machines, the type unsigned

```
inline void addmul(U* a, U* b, U d, L n)
{
   U t = 0;
   for (L i = 0; i < n; i++) addmulp(a[i], b[i], d, t);
}
```

Figure 4: Computing $a \leftarrow a + bd$ using *addmulp*

method	time (ns)
Fig. 1	1426
Fig. 2	735
Fig. 3	548
GMP	176

Table 1: Time (in nanoseconds) to multiply two 1000-bit integers

Assuming an appropriate double-word unsigned integer type `UU`, the routine *addmulp* can be implemented as in Fig. 3.

Fig. 4 illustrates how the *addmulp* routine is typically used in LIP and NTL. The routine *addmul* computes $a \leftarrow a+bd$, where a and b point to arrays representing n-digit integers, and d is a single digit.

If GMP is not available, then either the implementation based on Fig. 3 (if a double-word integer type is available) or Fig. 2 (otherwise) is used. The implementation based on Fig. 1 is currently never used. Of course, if GMP is available, that is used instead.

Table 1 shows the time (in nanoseconds) needed to multiply two 1000-bit integers, based on several different implementations. Except where otherwise noted, all benchmarks in this chapter were carried out on an Intel Xeon CPU E5-2698 v3 (64-bit Haswell architecture) at 2.30GHz, using GCC version 7.3.1, and GMP version 6.1.0 (note that GMP was compiled with version 4.8.5 of GCC).

The takeaway is that for integers of this size, GMP is 3–4 times faster than NTL's native implementations, and 8 times faster than LIP's original implementation technique. It should also be noted that to multiply larger integers, NTL's native implementation uses Karatsuba, while GMP employs a number of different algorithms in addition to Karatsuba (including an FFT for very large integers). Thus, GMP's performance relative to NTL gets even better for larger integers.

We also note that LIP (and NTL, when GMP is not used) also exploits floating-point arithmetic in a number of other ways, for operations such as long integer division and GCDs. However, we shall not discuss these techniques in any detail here.

---

long long is a 64-bit unsigned integer, and on many 64-bit machines, the type `__uint128_t` is a 128-bit unsigned integer.

3 Number-Theoretic Transforms

Let p be a prime such that $p-1$ is divisible by $N = 2^k$. We will assume here that p is a "single precision" number, i.e., it fits in a single word. In fact, we will usually assume that the bit-length of p is a bit smaller than the width of a word (typically 2 or 4 bits less).

We know that $\mathbb{Z}_p^*$ contains an element ω of multiplicative order N. The N-point Discrete Fourier Transform (DFT) maps the coefficient vector of a polynomial $f \in \mathbb{Z}_p[X]$ to the *evaluation vector*

$$(f(\omega^0), f(\omega^1), \ldots, f(\omega^{N-1})).$$

The DFT over $\mathbb{Z}_p$ is sometimes called a *Number-Theoretic Transform* or *NTT*. As is well known, the inverse transform is also essentially just an NTT. Of course, using the Fast Fourier Transform (FFT) [6], we can compute the NTT and inverse NTT using $O(N \log N)$ additions, subtractions, and multiplications in $\mathbb{Z}_p$.

We shall sometimes call such a prime p an "FFT prime".

An important application of NTTs is to implement fast polynomial multiplication over $\mathbb{Z}_p$, using the following well known technique. Suppose $f, g \in \mathbb{Z}_p[X]$ with $\deg(f)+\deg(g) < N$. Then to compute $h = f \cdot g$, we proceed as follows:

- Compute the evaluation vectors u and v of f and g using the NTT.
- Compute the component-wise product vector $w := u \cdot v$.
- Compute h as the inverse NTT of w.

NTTs play a crucial role in NTL, especially in their use in the multimodular algorithms for polynomial multiplication, as we will discuss later in Sections 4, 5, and 8. Also, in Section 4.5, we shall discuss more recent applications of NTTs to a new type of encryption called *fully homomorphic encryption*.

The remainder of Section 3 will discuss the techniques for efficient implementations of NTTs that are used in NTL.

3.1 Single-precision modular arithmetic

Evidently, in order to get a high-performance implementation of a fast NTT algorithm modulo a single-precision prime, one needs high-performance implementations of addition, subtraction, and multiplication modulo such primes.

More generally, assume $n > 1$, and a and b are integers in the range $[0, n)$. We can compute $a + b \bmod n$ and $a - b \bmod n$ using the routines *AddMod* and *SubMod* in Fig. 5 (see the `typedef`s in Fig. 1). Correctness is assured provided $n < 2^{BPL-2}$, where *BPL* is the number of bits in the type `long`. In Fig. 5, the routine *CorrectExcess* subtracts n from r if r exceeds n. Similarly, the routine *CorrectDeficit* adds n to r if r is negative. These routines are implemented using one of two strategies, depending on a compile-time switch. The first strategy is branch free, and uses an appropriate sequence of addition, subtractions, shifts, and ANDs; as presented, this code assumes signed integer arithmetic is two's complement, and that right shifts of signed integers

are arithmetic shifts. The second strategy uses explicit branching. On modern architectures, which are typically highly pipelined, and which exhibit high branch penalties, the branch-free strategy can be much faster than the branching strategy. However, some modern architectures, such as the x86, provide so-called "conditional move" instructions, and the branching strategy can be slightly faster than the branch-free strategy on such architectures. NTL installation logic will usually make a good decision on which strategy is better.

To compute $ab \bmod n$, NTL originally employed floating-point arithmetic, essentially as in Fig. 6. This was very similar to the strategy employed by LIP, as well. Correctness is assured provided $n\epsilon \le 1/8$, where $\epsilon := 2^{-53}$ on a machine that correctly implements the IEEE floating-point standard. This ensures that the computed quotient q is equal to $\lfloor ab/n \rfloor + \delta$, where $\delta \in \{0, \pm 1\}$. The code also assumes that signed integer arithmetic is two's complement, conversion from unsigned to signed work as expected (i.e., the bit pattern is unchanged), and that right shifts of signed integers are arithmetic shifts.

In a 32-bit machine, the assumption that $n\epsilon \le 1/8$ is not a real restriction; however, on a 64-bit machine, it essentially restricts the modulus n to be a 50-bit number at most.

Typically, the modulus n will remain fixed for many modular multiplication operations (for example, in NTTs). In this situation, a good optimizing compiler may generate code to compute the quantity *ninv* just once. However, an alternative interface is provided in which the value *ninv* can be explicitly precomputed by the programmer.

Often, in addition to the modulus n, one of the multiplicands, say b, may also remain fixed for many modular multiplication operations (again, for example, in NTTs, where b is a root of unity). While an optimizing compiler may sometimes be able to generate code to compute the quantity *bninv* just once, an alternative interface is provided in which the value *bninv* can be explicitly precomputed by the programmer. In fact, if the machine provides double-word integer arithmetic, a much faster code sequence is used, which is shown in Fig. 7. The code shown here uses a double-word unsigned integer type `UU`. Inline assembly code may also be used.

In this code, the value *bninv* should be precomputed as

$$\lfloor 2^{SPBITS} b/n \rfloor \cdot 2^{BPL-SPBITS}.$$

Here, *SPBITS* is a bound on the bit length of the modulus n, i.e., $n < 2^{SPBITS}$, and it is assumed that $SPBITS \le BPL - 2$. The value *bninv* can be computed using floating-point logic similar to that in Fig. 6 (and a function is provided to the programmer to do so).

One can prove that with `bninv` computed in this way, the quotient q computed in Fig. 7 is equal to either $\lfloor ab/n \rfloor$ or $\lfloor ab/n \rfloor - 1$.

A good optimizing compiler targeting the x86 instruction set will generate code that consists of three integer multiplication instructions, two subtractions, one "conditional move" instruction, and several register-to-register moves (which nowadays are almost completely cost free). Moreover, two of the multiplications are "single word" multiplications (i.e., only the low-order word need be computed), and these two multiplications may run concurrently (there are no data dependencies between them).

```
const int BPL = 64;
#ifdef AVOID_BRANCHING
   inline long CorrectExcess(L r, L n)
   {
      return (r-n) + (((r-n) >> (BPL-1)) & n);
   }

   inline long CorrectDeficit(L r, L n)
   {
      return r + ((r >> (BPL-1)) & n);
   }
#else
   inline long CorrectExcess(L r, L n)
   {
      return r-n >= 0 ? r-n : r;
   }

   inline long CorrectDeficit(L r, L n)
   {
      return r >= 0 ? r : r+n;
   }
#endif
   inline long AddMod(L a, L b, L n)
   {
      return CorrectExcess(a + b, n);
   }

   inline long SubMod(L a, L b, L n)
   {
      return CorrectDeficit(a - b, n);
   }
```

Figure 5: Single precision modular addition and subtraction

```
inline long MulMod(L a, L b, L n)
{
   D ninv = 1/D(n);
   D bninv = D(b) * ninv;
   L q = L( D(a) * bninv );
   L r = L( U(a)*U(b) - U(q)*U(n) );
   r = CorrectDeficit(r, n);
   r = CorrectExcess(r, n);
   return r;
}
```

Figure 6: Single precision modular multiplication based on floating-point arithmetic

```
inline long MulModPrecon(L a, L b, L n, U bninv)
{
   U q = U( (UU(U(a)) * UU(bninv)) >> BPL );
   L r = L( U(a)*U(b) - q*U(n) );
   return CorrectExcess(r, n);
}
```

Figure 7: Single precision modular multiplication with pre-conditioning

3.2 A floating-point-free implementation

While the above floating-point implementations for single-precision modular multiplication are simple and quite effective, they have the disadvantage of restricting the modulus to only 50 bits on a 64-bit machine. In this section, we describe the floating-point-free implementation that NTL currently deploys on machines that provide double-word integer arithmetic. With this technique, one can easily support a 62-bit modulus on a 64-bit machine; however, for a number of reasons, NTL usually restricts the modulus to just 60 bits. Moreover, on modern hardware, this floating-point-free implementation typically outperforms the floating-point implementations.

We assume the modulus n has w bits, so

$$2^{w-1} \le n < 2^w. \tag{1}$$

We also assume that $2 \le w \le BPL - 2$, although we may also make the stronger assumption that $w \le BPL - 4$.

As a precomputation step, we compute

$$X := \lfloor (2^v - 1)/n \rfloor. \tag{2}$$

Here, $v > w$ is a constant to be discussed below. Note that

$$2^{v-w} \le X < 2^v/n \le 2^{v-w+1}. \tag{3}$$

We can also write

$$2^v - 1 = Xn + Y, \quad \text{where } 0 \le Y < n. \tag{4}$$

Now suppose we are give an integer C satisfying

$$0 \le C < 2^w n \tag{5}$$

that we want to reduce mod n. We split C into high-order and low-order bits:

$$C = A2^s + B, \quad \text{where } 0 \le B < 2^s, \tag{6}$$

and then compute the product of A and X, splitting that into high-order and low-order bits:

$$AX = Q2^t + R, \quad \text{where } 0 \le R < 2^t. \tag{7}$$

Here, s and t are constants satisfying $s + t = v$, to be discussed below.

We have

$$AX \le \frac{C}{2^s}X < \frac{C}{2^s}\frac{2^v}{n} = \frac{2^t C}{n}.$$

These calculations follow from (3) and (6). In particular, $AX/2^t < C/n$, and by (7), we have

$$Q = \lfloor AX/2^t \rfloor \le \lfloor C/n \rfloor.$$

We will show, with an appropriate choices of parameters, that

$$Q \ge \lfloor C/n \rfloor - 1. \tag{8}$$

To this end, we claim that

$$C - Qn < nC/2^v + 2^s + n. \tag{9}$$

To see this, observe that

$$\begin{aligned} Qn &= \frac{AX - R}{2^t}n = \frac{AXn}{2^t} - \frac{R}{2^t}n > \frac{AXn}{2^t} - n \\ &= \frac{A}{2^t}(2^v - 1 - Y) - n \ge \frac{A}{2^t}(2^v - n) - n \\ &= A2^s - A\frac{n}{2^t} - n = C - B - A\frac{n}{2^t} - n > C - 2^s - A\frac{n}{2^t} - n \\ &\ge C - 2^s - C\frac{n}{2^v} - n. \end{aligned}$$

These calculations follow from (4), (6), and (7)

Suppose $BPL \ge w + 4$ (which is the default on 64-bit machines, where $w = 60$). In this case, we set

$$v := 2w + 2, \quad s := w - 2, \quad t := w + 4.$$

Then we have

$$C - Qn < nC/2^v + 2^s + n < n/4 + n/2 + n < 2n.$$

```
inline long NormalizedMulMod(L a, L b, L n, U X)
{
   UU C = UU(U(a)) * UU(U(b));
   U A = U( C >> 58 );
   U Q = (UU(A)*UU(X)) >> 64;
   L r = L( U(C) - Q*U(n) );
   return CorrectExcess(r, n);
}
```

Figure 8: Single precision modular multiplication without floating-point (on a 64-bit machine with $w = 60$)

These calculations follow from (9) and the fact that $nC/2^v \le n2^{2w-v} = n/4$ (based on (5) and (1)) and $2^s = 2^{w-2} \le n/2$ (based on (1)). This establishes (8). From (3), we also have

$$X < 2^{v-w+1} = 2^{w+3} < 2^{BPL},$$

so X fits in one machine word. From (6), (5), and (1), we also have

$$A \le \frac{C}{2^s} < \frac{2^w n}{2^s} < 2^{2w-s} = 2^{w+2} < 2^{BPL},$$

so A also fits in one machine word.

Suppose $BPL \ge w + 2$ (which is the default on 32-bit machines, where $w = 30$). In this case, we set

$$v := 2w + 1, \quad s := w - 2, \quad t := w + 3.$$

Then we have

$$C - Qn < nC/2^v + 2^s + n < n/2 + n/2 + n \le 2n.$$

This again establishes (8). We also have

$$X < 2^{v-w+1} = 2^{w+2} \le 2^{BPL},$$

so X fits in one machine word. (This is the only place where it is essential that X is computed as in (2), rather than as $\lfloor 2^v/n \rfloor$). We also have

$$A \le \frac{C}{2^s} < \frac{2^w n}{2^s} < 2^{2w-s} = 2^{w+2} \le 2^{BPL},$$

so A also fits in one machine word.

Fig. 8 shows how to use the above strategy to implement modular multiplication on a 64-bit machine with $w = 60$. As usual, `L` is a synonym for `long`, `U` is a synonym for `unsigned long`, and `UU` is a synonym for a double-word unsigned integer type. A good optimizing compiler targeting the x86 instruction set will generate code that consists of three integer multiplication instructions, one "shrd" (or "shld") instruction (which performs the 58-bit double-word right shift), two subtractions, one "conditional

```
inline long
GeneralMulMod(L a, L b, L n, U X, int shamt)
{
   return
   NormalizedMulMod(a, b << shamt, n << shamt, X)
      >> shamt;
}
```

Figure 9: Floating point free modular multiplication — the general case

move" instruction, and several register-to-register moves. Some compilers may choose to replace the "shrd" instruction with two single-word shifts and an addition or logical-or instruction. Note that in contrast to the logic in Fig. 7, only one of the three is a single-word multiplication, and moreover, none of them can run concurrently with the others.

It is also possible to adapt the logic to work with $w = 62$, but for a number of reasons, choosing $w = 60$ is advantageous.

Note that the logic in Fig. 8 requires n satisfies (1). For certain applications, this will be true, but in general, one may have to employ the logic in Fig. 9, which only assumes $n < 2^w$. The amount *shamt* is precomputed so that $2^{w-1} \le n2^{shamt} < 2^w$.

3.3 Lazy butterflies and truncated FFTs

Consider again the problem of computing NTTs, that is, an FFT modulo a single-precision prime p. NTL's current NTT implementation incorporates two important optimizations.

3.3.1 Lazy butterflies

The first optimization is a "lazy butterfly" technique. A "forward FFT" (a.k.a., decimation in frequency) maps naturally ordered inputs to outputs that are in bit-reversed order. The basic operation performed in this algorithm is a "forward butterfly":

$$\begin{bmatrix} x \\ y \end{bmatrix} \longmapsto \begin{bmatrix} x + y \\ \omega(x - y) \end{bmatrix},$$

where ω is a root of unity. We can assume that the value ω is precomputed and stored in table. Thus, we can implement one forward butterfly using one multiplication, one addition, and one subtraction mod p. For the modular multiplication, we can use the pre-conditioned modular multiplication logic discussed above (see Fig. 7). The modular addition and subtraction steps can be implemented as in Fig. 5. This implementation requires a total of three "correction steps" (specifically, two invocations of *CorrectExcess* and one of *CorrectDeficit*).

David Harvey [12] has shown how to reduce the number of correction steps in the forward butterfly from three to one. This is achieved by keeping intermediate results

reduced only mod $2p$ rather than mod p (one might call this a "lazy reduction" technique). It turns out that this leads to a significant performance improvement in practice. In the same paper, Harvey also gives a similar improvement to the implementation of the "inverse butterfly" step,

$$\begin{bmatrix} x \\ y \end{bmatrix} \longmapsto \begin{bmatrix} x + \omega y \\ x - \omega y \end{bmatrix},$$

which is used in the "inverse FFT" (a.k.a., decimation in time) transform that maps bit-reverse-ordered inputs to naturally ordered outputs. This is achieved by keeping intermediate results reduced only mod $4p$ rather than mod p.

3.3.2 The truncated FFT

In the application of the FFT to polynomial multiplication, one has to perform N-point FFTs, where N is a power of two that is larger than the degree d of the product polynomial. If d is at or just above a power of two, this leads to an unfortunate inefficiency. Indeed, if one graphs the running time of such a polynomial multiplication algorithm as a function of d, the graph looks like a step function, doubling at each power of two.

Joris van der Hoeven [31] introduced a "truncated FFT" technique that effectively smooths out these jumps. This "truncated FFT" technique can be combined with Harvey's "lazy butterfly" technique, and NTL now incorporates both of these techniques. In fact, NTL's current NTT implementation is derived from code originally developed by Harvey (although it has been extensively rewritten to conform to NTL's internal software conventions).

3.4 Comparing implementation techniques

Table 2 shows the time (in nanoseconds) to perform a modular multiplication using the various techniques outlined above.

The first column measures the time for a pre-conditioned modular multiplication (where both one multiplicand and the modulus are fixed). The second column measures the time for a modular multiplication where the modulus is fixed and normalized (i.e., satisfies (1)). The third column measures the time for a modular multiplication where the modulus is fixed but need not be normalized. The first three rows all employ the floating-point-free implementations discussed above (which work with 60-bit primes), where the only difference is how the *CorrectExcess* logic is implemented: using a conditional move instruction, using shifts and masks, and using actual jumps.[4] The next three rows are based on floating-point implementations that do not rely at all on a double-word integer type (and which work with 50-bit primes).

For comparison, the last row in the table is based on assembly code that uses a single hardware instruction for multiplication and a single hardware instruction for division. This method is by far the *least* efficient method. This is not surprising, as hardware support for integer division is typically very poor: even though it is just a single instruction, it has a very high latency.

[4] The GCC compiler flags `-fno-if-conversion -fno-if-conversion2 -fno-tree-loop-if-convert` were used to force actual jumps.

method	precon time (ns)	normal time (ns)	nonnorm time (ns)
no float / cmov	3.3	4.7	5.4
no float / mask	3.6	5.0	5.7
no float / jump	4.0	5.2	5.9
float / cmov	6.5	7.9	7.9
float / mask	6.7	8.1	8.1
float / jump	4.9	6.4	6.4
hardware div	26.6	26.6	26.6

Table 2: Time (in nanoseconds) for single-precision modular multiplication

The floating-point-free methods are generally the fastest, and have the advantage in that they allow for larger moduli on 64-bit machines (60, or even 62 bits, rather than 50 bits). Notice that among the floating-point-free methods, the implementations based on jumps are the slowest, whereas among the floating-point methods, the implementations based on jumps are the *fastest*. A reasonable hypothesis to explain this behavior is that the hardware branch predictor is doing much better in the floating-point methods than in the floating-point-free methods. We tested this hypothesis by empirically measuring the percentage of time the correction logic actually triggered a correction. For the floating-point methods, this was just 1–2%, while for the floating-point-free methods, this was 15-20% of the time.

Using the various techniques described above, Table 3 shows the time (in microseconds) to multiply two polynomials of degree 1023 modulo a single-precision "FFT prime", i.e., a prime p such that $p - 1$ is divisible by a suitable power of 2. The first three rows all employ the floating-point-free implementations discussed above (which work with 60-bit primes), where, again, the only difference is how the *CorrectExcess* logic is implemented. The last three rows are based on floating-point implementations that do not rely at all on a double-word integer type (and which work with 50-bit primes). The first column measures the time using a "non-lazy butterfly" implementation, and the second column measure a "lazy butterfly" implementation. Note that the "lazy butterfly" implementation is not available in the floating-point-only implementation.

Observe that in all cases, the implementation of the correction logic based on jumps is always dramatically slower. This is in contrast to what we saw in Table 2. The likely explanation for this is that the butterfly steps involve modular additions and/or subtractions, and for these, the branch misprediction rate is likely much higher than for multiplications.

Fig. 10 shows some timing data comparing the current truncated FFT implementation to an earlier FFT implementation without truncation. The x-axis is the degree bound, y-axis is time (in seconds), shown on a log/log scale. This is the time to multiply two polynomials modulo a single-precision "FFT" prime (60 bits). This figure nicely illustrates how the truncated FFT smooths out the step-function behavior exhibited in the non-truncated version.

The integer-only `MulModPrecon` routine (see Fig. 7) was introduced in NTL ver-

method	non-lazy time (μs)	lazy time (μs)
no float / cmov	74	56
no float / mask	80	59
no float / jump	300	130
float / cmov	110	n/a
float / mask	110	n/a
float / jump	290	n/a

Table 3: Time (in microseconds) for multiplying two degree 1023 polynomials modulo a single-precision "FFT" prime

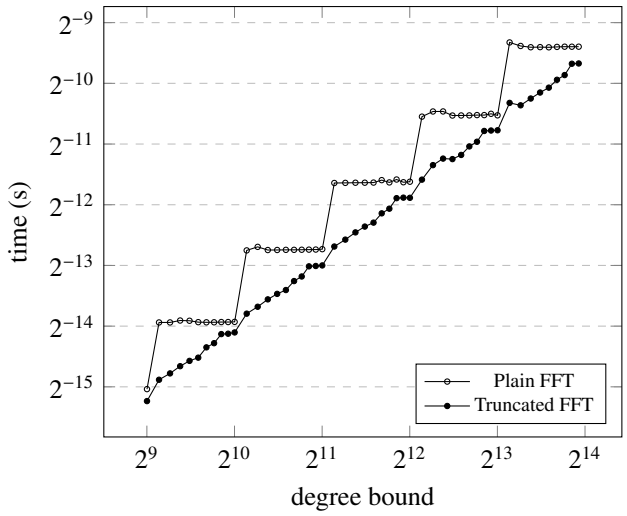

Figure 10: Truncated vs Plain FFT

sion 5.4 in 2005. Harvey's "lazy butterfly" technique was first implemented in NTL version 6.0 in 2013. The integer-only non-preconditioned `MulMod` routines (see Section 3.2) were introduced in NTL version 9.2 in 2015.[5] Van der Hoeven's "truncated FFT" was first implemented in NTL version 11.1 in 2018.

4 Arithmetic in $\mathbb{Z}_p[X]$ for multi-precision p

We next discuss how NTL implements arithmetic in $\mathbb{Z}_p[X]$ for multi-precision p.

NTL implements a class `ZZ` that represents multi-precision integers. In its default configuration, this is a thin wrapper around GMP. Specifically, an object of class `ZZ` is a pointer to an array of words representing a multi-precision integer, and GMP routines are used to do the arithmetic on such integers. In fact, NTL itself manages the memory allocation and de-allocation (using `malloc` and `free`), and (for the most part) relies only on the lower-level `mpn` routines in GMP that do not themselves manage memory.

NTL also implements a class `ZZ_p` that represents the ring $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ of integers mod p, where p is a multi-precision modulus. Note that despite the notation, p is not necessarily a prime. Elements of $\mathbb{Z}_p$ are naturally represented as integers in the range $[0, p)$, using the class `ZZ`. The modulus p is recorded in a global variable containing the value p itself (along with various precomputed values to make certain computations more efficient). Note that in multi-threaded applications, this global variable is implemented using "thread local storage". Using a global variable in this way surely offends some programming purists' sensibilities, but it also is the only way to allow one to use overload arithmetic operators to express computations in the most natural way.

NTL implements a class `ZZ_pX` that represents the ring of polynomials $\mathbb{Z}_p[X]$. An object of class `ZZ_pX` is a vector of `ZZ_p` objects. The memory for elements of this type of vector are specially managed.[6] The `ZZ_p` objects in such a vector are represented by `ZZ` objects, which themselves are represented by word-vectors, each of the same size; moreover, several such word-vectors are packed contiguously into larger blocks of memory. This packing has two benefits: first, when allocating such a vector of `ZZ_p` objects, the number of calls to `malloc` is significantly reduced; second, when accessing elements of such a vector in order, the cache behavior is improved, due to better locality of reference.

NTL implements several polynomial multiplication algorithms for $\mathbb{Z}_p[X]$: plain, Karatsuba, Schönhage-Strassen, and multimodular FFT.

For the most part, each of these algorithms reduces the problem of multiplying two polynomials in $\mathbb{Z}_p[X]$ to that of multiplying two polynomials in $\mathbb{Z}[X]$, and then reducing the coefficients of the product polynomial mod p.

[5] The design of these routines were initially inspired by the logic in [18], but the final form, which is optimized based on assumptions specific to NTL's software conventions, is quite a bit different. Note also that NTL does not implement Montgomery modular multiplication [19], as it is a bit less convenient to use: it involves a more expensive precomputation, a non-standard representation, and does not work with even moduli; moreover, it does not offer a significant performance benefit over the algorithms discussed above.

[6] For this, and other reasons, NTL implements its own vector template class, and does not rely on the vector class defined in the C++ standard template library (STL). Indeed, NTL was initially implemented long before templates or the STL existed.

The plain algorithm. If the input polynomials have degree less than n, the plain algorithm performs $O(n^2)$ multiplications of integers of bit length $\approx \log_2(p)$, as well as $O(n^2)$ additions of integers of bit length $\approx 2\log_2(p) + \log_2(n)$.

The Karutsuba algorithm. The Karutsuba algorithm [14] performs $O(n^{\log_2 3})$ multiplications of integers of bit length $\approx \log_2(p) + \log_2(n)$, as well as $O(n^{\log_2 3})$ additions and subtractions of integers of bit length $\approx 2\log_2(p) + \log_2(n)$.

The Schönhage-Strassen algorithm. The Schönhage-Strassen algorithm [26] is based on the same high-level FFT strategy as outlined at the beginning of Section 3: using two FFTs to compute the evaluation vectors of the input polynomials, multiplying these two vectors component-wise to obtain the evaluation vector of the product polynomial, and using one inverse FFT to obtain the product polynomial from its evaluation vector. The difference is that now the FFTs (and inverse FFTs) are performed over the ring $\mathbb{Z}_q$, where $q = 2^{mr} + 1$, m is a power of two, r is odd, and $\log_2(q)$ is at least $\approx \ell := 2\log_2(p) + \log_2(n)$. The element $\omega := 2^r \in \mathbb{Z}_q^*$ is a primitive $2m$th root of unity, which enables the use of the FFT to multiply polynomials of degree less than n, provided $m \geq n$. The requirement that $\log_2(q)$ is at least $\approx \ell$ ensures that computing the coefficients of the product polynomial mod q allows us to recover the coefficients over the integers.

Multiplication in $\mathbb{Z}_q$ by a power of ω can be efficiently implemented in terms of shifts and additions/subtractions. Also, a general multiplication in $\mathbb{Z}_q$ can be implemented using one integer multiplication (of integers of bit length $\approx \ell$), as well as some shifts and additions/subtractions. The overall cost of this polynomial multiplication algorithm is $O(n \log n)$ additions and multiplications by powers of ω in $\mathbb{Z}_q$, and $O(n)$ multiplications in $\mathbb{Z}_q$. This polynomial multiplication algorithm works best when $\log_2(p)$ is not too small relative to n, as otherwise we have to work modulo number q whose bit length (which is at least n) is significantly larger than that of p. One can improve performance for somewhat smaller values of p by making use of an optimization known as the "square root of 2 trick". This optimization is based on the observation that $\omega_1 := 2^{3mr/4} - 2^{mr/4}$ is a square root of 2 in the ring $\mathbb{Z}_q$, which means that ω_1 is a primitive $4m$th root of unity.

The multimodular algorithm. The multimodular algorithm reduces the coefficients of the input polynomials modulo several small primes $p_1, \ldots, p_k$. We must have $P := \prod_i p_i > B$, where B is a bound on the magnitude of the coefficients of the product polynomial, so $\sum_i \log_2(p_i)$ is at least $\approx 2\log_2(p) + \log_2(n)$. Each p_i is a single-precision "FFT prime", so that each $\mathbb{Z}_{p_i}$ contains appropriate roots of unity to enable the use of the FFT to compute the product of the input polynomials mod p_i. After computing the product polynomial mod each p_i, the Chinese remainder algorithm is used to reconstruct the coefficients of the product polynomial over the integers. We have already discussed above in Section 3 the techniques to multiply polynomials modulo such single-precision "FFT primes". To achieve good performance, some care must be taken in implementing the coefficient reduction modulo the small primes, and in the inverse Chinese remaindering step.

Assuming the hardware supports double-word integer arithmetic, the following technique is used for the reduction step. Suppose R is the radix used to represent long integers. Further, assume we are given a vector $(a_0, \ldots, a_{\ell-1})$ of base-R digits representing a long integer $a = \sum_j a_j R^j$ that we want to reduce mod each p_i. To do this, we assume we have precomputed a table of values $r_{ij} := R^j \bmod p_i$. To compute $a \bmod p_i$, we compute the inner product $s_i := \sum_j a_j r_j$, and then reduce $s_i \bmod p_i$. On a typical 64-bit machine, and assuming GMP is used to implement long-integer arithmetic, we have $R = 2^{64}$ and each p_i is a 60-bit integer. Each term $a_j r_j$ in the inner product can be computed using a single multiplication instruction. We can accumulate up to 16 such terms using a double-word addition (which on an x86 machine can be implemented using one addition and one addition-with-carry instruction). If there are more than 16 terms, then this double-word accumulator needs to be accumulated in a larger triple-word accumulator every 16 terms. After computing s_i, which will be represented as either a two-word or three-word integer, it is reduced mod p_i (using techniques similar to those discussed in Section 3.2).

Reducing such an ℓ-digit number a modulo $p_1, \ldots, p_k$, takes time $O(k\ell)$, using a precomputed table of size $O(k\ell)$. The implied big-O constant in the running time is quite small, and although NTL implements an asymptotically fast algorithm for multi-reduction (relying on GMP's asymptotically fast algorithms), the crossover is extremely high (for $\log_2 p$ in the tens of thousands).

As for the Chinese remaindering step, we are given integers s_i for $i = 1, \ldots, k$, where each s_i is in the range $[0, p_i)$, and we want to compute the integer

$$c = \Big(\sum_i P_i s_i \bmod P \Big) \bmod p,$$

where $P_i := P/p_i$. A relatively straightforward implementation takes time $O(k\ell)$, but we get an easy speedup by a factor of two with the following observation. Assume that $P > 4B$. Let $A := \sum_i P_i s_i$. Suppose we write $A = PQ + R$, where Q is the integer nearest A/P and $|R| \leq P/2$. We want to compute $R \bmod p$. In fact, by the assumption that $P > 4B$, we have $|R| < P/4$, and so $A/P = Q + \delta$, where $|\delta| < 1/4$. We also have $A/P = \sum_i s_i/p_i$. Therefore, under reasonable assumptions, we can compute the quotient Q by computing the sum $\sum_i s_i/p_i$ using double-precision floating-point, and rounding to the nearest integer. Assuming we have precomputed the values $\bar{P}_i := P_i \bmod p$ (each of which roughly half the size of P_i), as well as $\bar{P} := P \bmod p$, we can compute c above as

$$c = \Big(\sum_i \bar{P}_i s_i - \bar{P} Q \Big) \bmod p.$$

The sum $\sum_i \bar{P}_i s_i$ can be computed fairly quickly in the obvious way (GMP provides relatively good support for this); however, when double-word integer arithmetic is available, and especially when $\log_2(p)$ is not too huge, a technique similar to that used above in the reduction step for accumulating double-word products can be somewhat more efficient. Also as above, although NTL implements an asymptotically fast algorithm for Chinese remaindering, the crossover is extremely high.

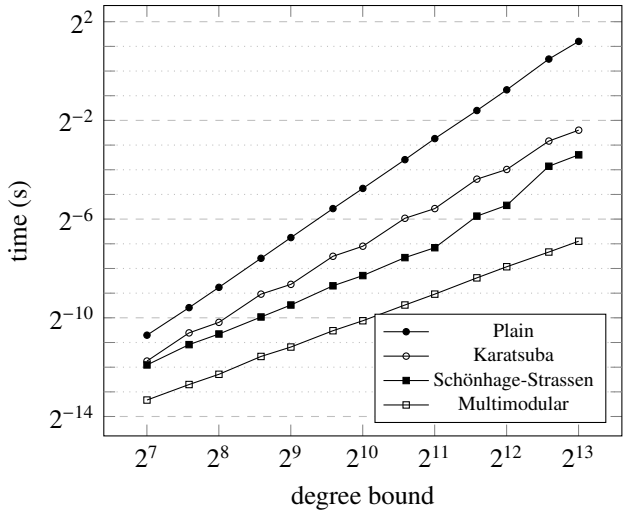

Figure 11: Polynomial multiplication modulo a 256-bit prime

4.1 Comparing algorithms for multiplication in $\mathbb{Z}_p[X]$

Figs. 11–13 show the running time (in seconds) for each of the algorithms described above for primes p of bit-length 256, 1024, and 4096, respectively. Each figure shows the time to compute the product of two randomly chosen polynomials over $\mathbb{Z}_p$ of degree $< n$ for values of n ranging between 128 and 8192 (at values of n that are powers of two and midway between powers of two). The graphs are on a log-log scale.

These graphs speak for themselves. For these ranges of p and n, plain and Karatsuba are always slower than the FFT-based Schönhage-Strassen and multimodular algorithms. One also sees, not surprisingly, that the multimodular algorithm is significantly faster than Schönhage-Strassen when $\log_2(p)$ is not too huge, but eventually, as $\log_2(p)$ gets large, Schönhage-Strassen is somewhat faster.

For the multimodular algorithm, we also measured the percentage of the total time spent on coefficient reduction modulo the small primes plus the inverse Chinese remaindering step. For $n = 4096$, this percentage is 35% for 256-bit primes, 51% for 1024-bit primes, and 73% for 4096-bit primes.

4.2 Multi-core implementation of polynomial multiplication algorithms

When available, NTL can exploit multi-core machines to speed up polynomial multiplication, for both the Schönhage-Strassen and multimodular algorithms.

For Schönhage-Strassen, the FFT and inverse FFT algorithms are recursive divide-and-conquer algorithms, which divide a given problem into two subproblems of half the size. One thread is assigned to the top-level recursive invocation, two threads to the two second level recursive invocations, four threads to the four third level recursive

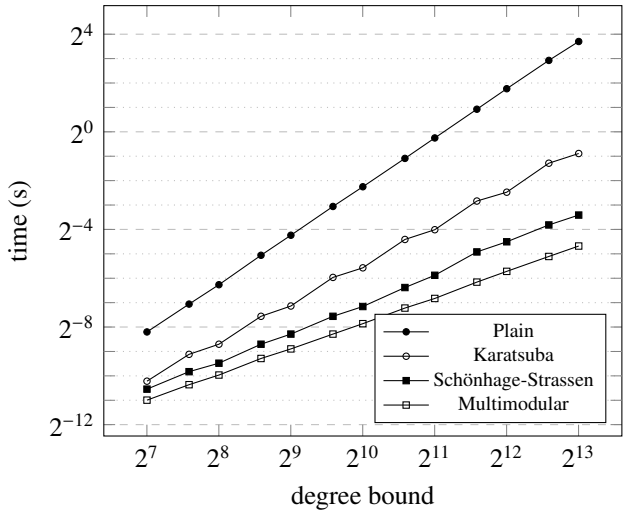

Figure 12: Polynomial multiplication modulo a 1024-bit prime

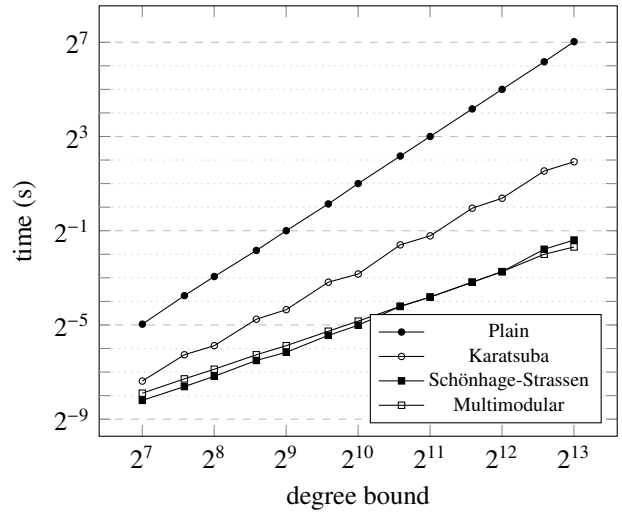

Figure 13: Polynomial multiplication modulo a 4096-bit prime

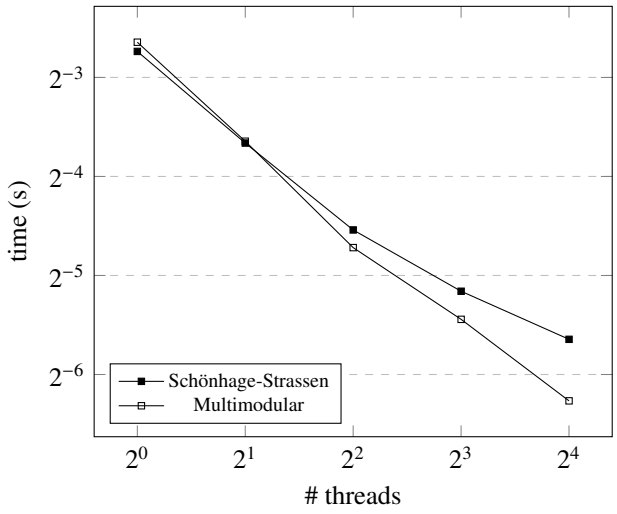

Figure 14: Multi-core performance: multiplication of polynomials of degree < 4096 modulo a 4096-bit prime

invocations, and so on. The component-wise multiplication of the evaluation vectors is trivially parallelizable.

The multimodular algorithm is even more straightforward to parallelize. The modular reductions and Chinese remaindering steps are trivially parallelizable across the coefficients, and the FFTs are trivially parallelizable across the primes. Fig. 14 shows the running time (in seconds) to multiply two polynomials of degree less than 4096 modulo a 4096-bit prime, using 1,2,4,8, and 16 threads, with both the Schönhage-Strassen and multimodular algorithms. This is a log-log plot. A straight line with slope -1 would indicate perfect scaling with respect to the number of threads. As one can see, the scaling is not quite perfect for either algorithm.

Note that NTL implements its own "thread pool mechanism" on top of standard `C++11` threading features, and does not rely on non-standard mechanisms, such as *OpenMP*. This thread pool mechanism is utilized in a number of places throughout NTL to boost performance when multiple cores are available, and may also be used directly by NTL clients.

4.3 Other operations on polynomials

Of course, NTL implements other operations over $\mathbb{Z}_p[X]$, such as division and GCDs. Asymptotically fast algorithms are used wherever possible. In theory, one can reduce many of these problems to polynomial multiplication, and simply rely on an asymptotically fast algorithm for polynomial multiplication to get asymptotically fast algorithms for these other problems. In practice, one can do better in certain situations. For example, if $f \in \mathbb{Z}_p[X]$ is a polynomial of degree n, and one wants to perform many polynomial multiplications mod f, then one can precompute an appropriate polyno-

mial "inverse" h of f, so that reducing a polynomial of degree less than $2n$ modulo f can be done using just two multiplications of polynomials (of degree less than n). Moreover, if a multimodular FFT algorithm is used for the polynomial multiplications, appropriate transforms of f and h can also be precomputed (specifically, the evaluation vectors of f and h modulo each small "FFT prime"), which speeds things up considerably. In particular, such reduction mod f can be performed at the cost of a *single* multiplication of polynomials (of degree less than n): a speedup by a factor of two. As another example, a single squaring of polynomials mod f can be computed at the cost of between 1.5 and 1.67 multiplications of polynomials (of degree less than n).

For more details on these and other optimizations, see the paper [28]. That paper also presented an algorithm for factoring polynomials over $\mathbb{Z}_p$, and reported on an experiment, conducted in 1995, that involved factoring a degree 2048 polynomial modulo a 2048-bit prime. Factoring that polynomial took just over 272 hours (over 11 days) on a SPARC-10 workstation. Using the current version of NTL, factoring the same polynomial using a single core on our Haswell machine took 188 seconds, a speedup of over 5,000×. Moreover, using 16 cores on our Haswell machine took just under 26 seconds, a speedup of over 37,000× (which represents a 45% utilization of these cores). The high-level algorithms have really not changed that much. The SPARC-10 and Haswell release dates differ by 20 years, so Moore's law by itself would predict about a 1,000× speedup. In addition to making NTL's polynomial arithmetic exploit multiple cores (when possible), other improvements to the running time include better implementations of:

- single-precision modular arithmetic (most notably, the pre-conditioned modular multiplication code in Fig. 7),
- the NTT itself (most notably, Harvey's lazy butterflies, discussed in Section 3.3),
- modular reduction and Chinese remaindering as deployed in the multimodular polynomial multiplication algorithm (see Section 4),
- modular composition, i.e., computing $g(h) \bmod f$ for $f, g, h \in \mathbb{Z}_p[X]$ — although the high-level algorithm is still the same baby-step/giant-step approach of Brent and Kung [3], much better algorithms for matrix multiplication over $\mathbb{Z}_p$ are used, which are discussed below in Section 6.2.[7]

4.4 NTL vs Flint

FLINT is a `C` library started in 2007 by Bill Hart and David Harvey, ostensibly as a "successor" to NTL, as some people were apparently under the (mistaken) impression that NTL was no longer actively maintained. Since 2007, both NTL and FLINT have evolved considerably, but they still share some common functionality whose performance we can compare.

[7] NTL does not implement any of the newer, asymptotically faster algorithms for modular composition by Kedlaya and Umans [15]. Indeed, as far as we are aware, these algorithms remain of only theoretical interest (see [32]).

$k/1024$	$n/1024$												
	$^1/_4$		$^1/_2$		1		2		4		8		16
$^1/_4$	**1.89**	**1.98**	**2.20**	**2.31**	**2.49**	**2.40**	**2.60**	**2.51**	**2.68**	**2.68**	**2.80**	**2.75**	**2.96**
$^1/_2$	**1.50**	**1.65**	**1.60**	**1.84**	**1.81**	**2.21**	**2.18**	**2.77**	**3.09**	**2.90**	**3.19**	**3.23**	**3.37**
1	1.11	**1.20**	1.17	**1.30**	**1.27**	**1.49**	**1.46**	**1.96**	**1.94**	**2.93**	**3.11**	**2.89**	**3.07**
2	0.87	0.89	0.87	0.92	0.93	1.02	1.01	**1.25**	**1.24**	**1.81**	**1.59**	**2.23**	**2.35**
4	1.00	1.01	0.99	1.03	1.01	0.99	1.00	1.08	1.06	**1.25**	**1.22**	**1.52**	**1.40**
8	1.05	1.05	1.03	1.01	1.00	0.97	0.96	0.93	0.89	0.98	0.96	0.94	0.90
16	0.96	0.96	0.94	0.95	0.94	0.95	0.93	0.89	0.90	0.87	***0.83***	0.90	0.88

Table 4: Multiplication in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

4.4.1 Multiplication in $\mathbb{Z}_p[X]/(f)$

Table 4 compares the relative speed of NTL's `ZZ_pX mul` routine with the corresponding FLINT routine.[8] The polynomials were generated at random to have degree less than n, and the modulus p was chosen to be a random, odd k-bit number.[9] The unlabeled columns correspond to n-values half-way between the adjacent labeled columns. For example, just to be clear: the entry in the 3rd row and 7th column corresponds to $k = 1024$ and $n = 2048$; the entry in the 3rd row and 8th column corresponds to $k = 1024$ and $n = 2048 + 1024 = 3072$.

The numbers in the table shown are ratios:

$$\frac{\text{FLINT time}}{\text{NTL time}}.$$

So ratios greater than 1 mean NTL is faster, and ratios less than 1 mean FLINT is faster. Ratios outside of the range $(1/1.2, 1.2)$ are boldfaced (the others are essentially a tie).

The ratios in the upper right-hand corner of the table essentially compare NTL's multimodular FFT algorithm with FLINT's Kronecker-substitution algorithm, which reduces polynomial multiplication to integer multiplication, which is performed by GMP. The ratios in the lower left-hand corner of the table essentially compare NTL's Schönhage-Strassen algorithm with FLINT's Schönhage-Strassen algorithm.

As one can see, NTL's multimodular-FFT approach can be significantly faster than FLINT's Kronecker-substitution, being more than 3 times faster in some cases. One can also see that FLINT's Schönhage-Strassen implementation seems slightly better engineered than NTL's, being up to 20% faster than NTL's implementation in some cases.

4.4.2 Squaring in $\mathbb{Z}_p[X]/(f)$

Squaring in $\mathbb{Z}_p[X]/(f)$ is a critical operation that deserves special attention, as it is the bottleneck in many exponentiation algorithms in $\mathbb{Z}_p[X]/(f)$.

[8] We compared NTL v11.4.3 with FLINT v2.5.2, which is the most recent public release of FLINT at the time of this writing. These were both built using GMP v6.1.0. The compiler was GCC v4.8.5. All packages were configured using their default configuration flags. All of the programs used for benchmarking can be obtained at `www.shoup.net/ntl`.

[9] NTL's behavior is somewhat sensitive to whether p is even or odd, and since odd numbers correspond to the case where p is prime, we stuck with those.

k/1024	n/1024												
	1/4		1/2		1		2		4		8		16
1/4	**3.03**	**3.10**	**3.55**	**3.54**	**3.95**	**3.74**	**4.08**	**3.85**	**4.33**	**4.06**	**4.50**	**4.23**	**4.90**
1/2	**2.45**	**2.52**	**2.60**	**2.88**	**2.97**	**3.43**	**3.57**	**4.18**	**4.68**	**4.47**	**5.16**	**5.06**	**5.46**
1	**1.86**	**1.89**	**1.96**	**2.07**	**2.12**	**2.40**	**2.46**	**3.19**	**3.26**	**4.63**	**5.02**	**4.42**	**5.09**
2	**1.48**	**1.46**	**1.49**	**1.54**	**1.60**	**1.72**	**1.74**	**2.10**	**2.12**	**2.82**	**2.96**	**3.51**	**3.71**
4	**1.65**	**1.67**	**1.67**	**1.66**	**1.69**	**1.64**	**1.65**	**1.77**	**1.73**	**1.93**	**2.04**	**2.31**	**2.36**
8	0.86	0.91	0.87	0.91	0.88	0.93	0.89	0.95	1.04	0.98	0.96	1.11	1.11
16	***0.65***	***0.66***	***0.66***	***0.67***	***0.67***	***0.69***	***0.68***	***0.71***	***0.68***	***0.69***	***0.67***	***0.73***	***0.73***

Table 5: Squaring in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

k/1024	n/1024												
	1/4		1/2		1		2		4		8		16
1/4	**1.62**	**1.73**	**1.81**	**1.90**	**2.00**	**2.00**	**2.22**	**2.12**	**2.41**	**2.28**	**2.59**	**2.41**	**2.78**
1/2	**1.43**	**1.52**	**1.59**	**1.63**	**1.73**	**1.76**	**1.89**	**1.88**	**2.04**	**2.05**	**2.24**	**2.23**	**2.46**
1	**1.25**	**1.35**	**1.32**	**1.41**	**1.43**	**1.47**	**1.52**	**1.54**	**1.59**	**1.65**	**1.73**	**1.77**	**1.89**
2	**1.22**	**1.36**	**1.27**	**1.34**	**1.30**	**1.38**	**1.36**	**1.41**	**1.38**	**1.43**	**1.43**	**1.48**	**1.50**
4	**1.24**	**1.37**	**1.34**	**1.43**	**1.40**	**1.44**	**1.46**	**1.50**	**1.51**	**1.51**	**1.54**	**1.57**	**1.58**
8	1.04	1.16	1.03	1.15	1.03	1.11	1.03	1.11	1.02	1.07	1.01	1.09	1.02
16	0.98	1.11	0.93	1.04	0.93	1.01	0.89	0.98	0.87	0.95	0.87	0.94	0.87

Table 6: Computing GCDs in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

Table 5 compares the relative performance of NTL's `ZZ_pX SqrMod` routine with FLINT's corresponding routine. The NTL routine takes as input precomputations based on f, specifically, a `ZZ_pXModulus` object. The modulus p was chosen to be a random, odd k-bit number. The polynomial f was a random monic polynomial of degree n, while the polynomial to be squared was a random polynomial of degree less than n.

NTL is using a multimodular-FFT strategy throughout (combined with the precomputation techniques briefly discussed in 4.3), while FLINT is using Kronecker-substitution in the upper right region and Schönhage-Strassen in the lower left region. As one can see, NTL's strategy can be over 5 times faster in some cases, while FLINT's can be over 1.5 times faster in others.

4.4.3 Computing GCDs in $\mathbb{Z}_p[X]$

Table 6 compares the relative performance of NTL's `ZZ_pX GCD` routine with FLINT's corresponding routine. The modulus p was chosen to be a random k-bit prime, and the GCD was computed on two random polynomials of degree less than n. Both libraries use a fast "Half GCD" algorithm.

4.4.4 Modular composition in $\mathbb{Z}_p[X]$

Table 7 compares the relative performance of NTL's `ZZ_pX CompMod` routine and the corresponding FLINT routine. These routines compute $g(h)$ mod f for polynomials $f, g, h \in \mathbb{Z}_p[X]$ using Brent and Kung's [3] modular composition algorithm. The modulus p was chosen to be a random k-bit prime. The polynomial f was chosen to be a

$k/1024$	$n/1024$								
	$1/4$		$1/2$		1		2		4
$1/4$	**7.24**	**7.32**	**8.91**	**9.08**	**10.87**	**10.86**	**13.27**	**12.75**	**14.07**
$1/2$	**6.15**	**6.53**	**7.32**	**7.87**	**8.78**	**9.56**	**10.38**	**12.12**	**14.64**
1	**5.24**	**5.50**	**6.16**	**6.47**	**7.26**	**7.57**	**8.38**	**9.51**	**10.93**
2	**4.64**	**4.79**	**5.35**	**5.08**	**6.22**	**6.52**	**7.20**	**7.60**	**8.37**
4	**5.00**	**5.12**	**5.78**	**5.94**	**6.79**	**6.81**	**7.72**	**7.37**	**8.39**

Table 7: Composition modulo a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

$k/1024$	$n/1024$								
	$1/4$		$1/2$		1		2		4
$1/4$	**3.59**	**3.62**	**4.17**	**4.48**	**5.73**	**5.06**	**6.17**	**5.28**	**6.25**
$1/2$	**2.79**	**2.81**	**14.06**	**4.22**	**4.35**	**4.99**	**5.08**	**5.67**	**8.01**
1	**2.07**	**2.19**	**2.38**	**2.68**	**3.29**	**3.45**	**3.46**	**4.39**	**5.52**
2	**2.20**	**1.70**	**1.76**	**1.92**	**2.29**	**2.56**	**2.74**	**3.19**	**4.04**
4	**1.72**	**11.21**	**7.16**	**1.92**	**2.12**	**2.09**	**2.50**	**2.79**	**2.50**

Table 8: Factoring a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

random monic polynomial of degree n, and the polynomials g and h were chosen to be random polynomials of degree less than n.

4.4.5 Factoring in $\mathbb{Z}_p[X]$

Table 8 compares the relative performance of NTL's `ZZ_pX CanZass` factoring routine and the corresponding FLINT routine. Both routines implement the Kaltofen/Shoup algorithm [13], and for the range of parameters that were benchmarked, this is the fastest algorithm that each library has to offer. The modulus p was chosen to be a random k-bit prime. The polynomial to be factored was a random monic polynomial of degree n.

4.5 Application to fully homomorphic encryption

Interest in NTTs and corresponding multimodular techniques for polynomial arithmetic has intensified recently, because of the critical role they play in the design and implementation of several so-called *fully homomorphic encryption* schemes.

In such a scheme, one party, Alice, can encrypt a value, x, under the public key of a second party, Bob, obtaining a ciphertext c that encrypts x. Next, Alice can send c to a third party Charlie, who can efficiently transform c into another ciphertext c' that encrypts the value $f(x)$, where f is some specific function. Surprisingly, Charlie can do this only knowing c (and an appropriate description of f), without Bob's secret decryption key, and indeed, without learning anything at all about the values x or $f(x)$.

We cannot go into the details of these schemes here. However, for many of these schemes, the bulk of the computation time is spent operating on objects in the ring $R := \mathbb{Z}_q[X]/(\Phi_m(X))$, where q is typically a composite number a few hundred bits in length, and $\Phi_m(X)$ is the mth cyclotomic polynomial, with m typically in the range 16,000–64,000. Multimodular techniques are especially applicable in this range of parameters. Moreover, there is typically a lot of flexibility in the choice of q.

For example, the library HElib [11], which is built on top of NTL, and which was mainly developed by Shai Halevi and this author, the modulus q can be chosen to be the product of single-precision primes p, where $p - 1$ is divisible by both m and a large power of two. In this setting, in addition to the traditional "coefficient vector" representation, in which elements of R are represented as the vector of coefficients of a polynomial $f(X) \in \mathbb{Z}_q[X]$ representing a residue class mod $\Phi_m(X)$, one can also advantageously work with a "Double CRT" representation, in which for each prime p dividing q, we store the values $f_p(\omega_p^j)$, for $j \in \mathbb{Z}_m^*$, where ω_p is a primitive mth root of unity in $\mathbb{Z}_p$, and $f_p(X)$ is the image of $f(X)$ in $\mathbb{Z}_p[X]$. In this "Double CRT" representation, elements of R can be added and multiplied in linear time. However, there are still situations where we need to convert back and forth between "coefficient vector" and "Double CRT" representations. Since each single-precision prime p is also an "FFT prime", we can use fast NTTs to efficiently implement these conversions. Ignoring the modular reduction or Chinese remaindering, and focusing on the computations mod each small prime p, if $m = 2^r$ is a power of two, so that $\Phi_m(X) = X^{2^{r-1}} + 1$, such a conversion can be done almost directly, using a 2^{r-1}-point NTT (this is sometimes called a nega-cyclic transformation). For general m, HElib uses Bluestein's FFT algorithm [2]. In fact, HElib uses a variation of Bluestein's algorithm that exploits the truncated FFT discussed above in Section 3.3.

5 Arithmetic in $\mathbb{Z}_p[X]$ for single-precision p

We briefly discuss how NTL implements arithmetic in $\mathbb{Z}_p[X]$ for single-precision p.

NTL implements a class `zz_p` that represents the ring $\mathbb{Z}_p = \mathbb{Z}/p\mathbb{Z}$ of integers mod p, where p is a single-precision modulus. (Again, despite the notation, p is not necessarily a prime.) Elements of $\mathbb{Z}_p$ are naturally represented as integers in the range $[0, p)$ using the built-in type `long`.

NTL implements a class `zz_pX` that represents the ring of polynomials $\mathbb{Z}_p[X]$. An object of class `zz_pX` is a vector of `zz_p` objects.

NTL implements several polynomial multiplication algorithms for $\mathbb{Z}_p[X]$: plain, Karatsuba, and multimodular FFT.

For the multimodular FFT, if p is a prime such that $\mathbb{Z}_p$ contains appropriate roots of unity, then just a single application of the FFT-based polynomial multiplication algorithm (see Section 3) is required. Otherwise, just as for the case of multi-precision p, several "FFT primes" are used together with Chinese remaindering. Depending on the size of p and the degrees of the polynomials, either one, two, or three such "FFT primes" are used.

Just as for arithmetic in $\mathbb{Z}_p[X]$ for multi-precision p, for single-precision p, when the degrees of the polynomials are large enough, many other operations (such as division and GCD) are reduced to fast polynomial multiplication.

5.1 NTL vs Flint

Just as we did for the multi-precision case, in Section 4.4, we compare the performance of NTL's polynomial arithmetic over $\mathbb{Z}_p[X]$, for single-precision p, to that of FLINT.

k	$n/1024$												
	1		2		4		8		16		32		64
5	***0.47***	***0.55***	***0.59***	***0.63***	***0.68***	***0.70***	***0.75***	***0.72***	0.86	0.99	1.17	1.14	**1.27**
10	***0.68***	***0.72***	***0.79***	***0.83***	0.90	1.05	1.16	**1.22**	**1.37**	**1.41**	**1.60**	**1.60**	**1.58**
15	0.85	1.00	1.09	1.14	**1.24**	**1.33**	**1.46**	**1.55**	**1.68**	**2.04**	**2.14**	**2.04**	**2.28**
20	***0.52***	***0.55***	***0.60***	***0.63***	***0.67***	***0.78***	0.87	0.95	1.01	1.15	1.13	1.14	1.11
25	***0.61***	***0.71***	***0.77***	***0.82***	0.89	0.95	1.06	1.11	**1.26**	**1.35**	**1.54**	**1.36**	**1.48**
30	***0.83***	0.86	0.93	0.97	1.08	**1.23**	**1.40**	**1.40**	**1.67**	**1.50**	**1.66**	**1.51**	**1.66**
35	0.93	1.01	1.11	1.17	**1.25**	**1.39**	**1.55**	**1.59**	**1.72**	**1.86**	**1.77**	**1.91**	**1.82**
40	1.08	1.14	**1.26**	**1.27**	**1.44**	**1.64**	**1.81**	**2.02**	**2.07**	**1.95**	**2.13**	**2.13**	**2.24**
45	1.19	**1.31**	**1.47**	**1.49**	**1.72**	**1.87**	**2.01**	**2.19**	**2.20**	**2.29**	**2.31**	**2.29**	**2.35**
50	0.97	0.96	1.06	1.11	**1.25**	**1.40**	**1.54**	**1.57**	**1.56**	**1.56**	**1.56**	**1.57**	**1.61**
55	1.00	1.09	**1.22**	**1.31**	**1.40**	**1.50**	**1.64**	**1.82**	**1.85**	**1.87**	**1.99**	**1.98**	**2.14**
60	1.17	**1.23**	**1.36**	**1.46**	**1.62**	**1.79**	**2.03**	**1.96**	**2.08**	**1.98**	**2.06**	**2.03**	**2.17**

Table 9: Single precision: Multiplication in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

k	$n/1024$												
	1		2		4		8		16		32		64
5	***0.81***	***0.83***	0.98	1.03	1.19	**1.23**	**1.43**	**1.52**	**1.65**	**1.57**	**1.94**	**1.76**	**2.01**
10	**1.21**	**1.24**	**1.48**	**1.52**	**1.78**	**1.87**	**2.23**	**2.15**	**2.37**	**2.29**	**2.50**	**2.49**	**2.71**
15	**1.64**	**1.76**	**2.06**	**2.14**	**2.51**	**2.53**	**2.92**	**2.79**	**2.99**	**3.00**	**3.13**	**3.30**	**3.48**
20	0.99	1.03	**1.28**	**1.28**	**1.47**	**1.54**	**1.69**	**1.67**	**1.75**	**1.77**	**1.93**	**1.94**	**1.96**
25	**1.26**	**1.32**	**1.51**	**1.64**	**1.84**	**1.88**	**2.09**	**1.95**	**2.28**	**2.21**	**2.58**	**2.40**	**2.69**
30	**1.61**	**1.67**	**1.90**	**1.92**	**2.33**	**2.28**	**2.51**	**2.31**	**2.73**	**2.38**	**2.64**	**2.56**	**2.93**
35	**1.91**	**1.92**	**2.31**	**2.29**	**2.66**	**2.70**	**2.88**	**2.82**	**2.93**	**3.04**	**3.14**	**3.38**	**3.37**
40	**2.22**	**2.28**	**2.68**	**2.77**	**2.91**	**2.93**	**3.38**	**3.24**	**3.42**	**3.22**	**3.96**	**3.65**	**3.99**
45	**2.56**	**2.61**	**3.09**	**3.19**	**3.52**	**3.42**	**3.65**	**3.40**	**3.71**	**3.97**	**4.03**	**4.03**	**4.06**
50	**1.93**	**1.96**	**2.24**	**2.26**	**2.53**	**2.44**	**2.68**	**2.47**	**2.57**	**2.58**	**2.72**	**2.69**	**2.86**
55	**2.17**	**2.21**	**2.53**	**2.49**	**2.81**	**2.80**	**2.89**	**2.88**	**2.88**	**2.84**	**3.32**	**3.49**	**3.45**
60	**2.49**	**2.49**	**2.86**	**2.81**	**3.15**	**3.02**	**3.35**	**3.20**	**3.68**	**3.30**	**3.51**	**3.42**	**4.11**

Table 10: Single precision: Squaring in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

Table 9 compares the relative speed of NTL's `zz_pX mul` routine with FLINT's corresponding routine. As in Section 4.4, the numbers in the table are the ratio of FLINT time to NTL, so ratios greater than 1 mean NTL is faster. NTL is using a multimodular FFT throughout, while FLINT is using Kronecker substitution throughout. Table 10 compares the relative performance of NTL's `zz_pX SqrMod` routine with FLINT's corresponding routine. Table 11 compares the relative performance of NTL's `ZZ_pX GCD` routine with FLINT's corresponding routine. Table 12 compares the relative performance of NTL's `zz_pX CompMod` routine for modular composition and the corresponding FLINT routine. Table 13 compares the relative performance of NTL's `zz_pX CanZass` factoring routine and the corresponding FLINT routine.

5.2 Special case where $p = 2$

An important special case for arithmetic in $\mathbb{Z}_p[X]$ is when $p = 2$. For this, NTL implements a special class `GF2X` that represents the ring of polynomials $\mathbb{Z}_2[X]$.

In this implementation, each coefficient is a single bit, and these bits are packed into machine words (actually, an `unsigned long` type).

k	$n/1024$												
	1		2		4		8		16		32		64
5	0.89	***0.82***	0.84	***0.78***	***0.81***	***0.76***	0.84	***0.75***	0.86	***0.77***	0.90	***0.80***	0.96
10	0.93	0.89	0.91	0.86	0.90	0.86	0.97	0.91	1.03	0.92	1.08	1.00	1.19
15	0.99	0.97	1.00	0.95	1.04	0.98	1.10	1.03	1.19	1.08	**1.30**	1.18	**1.44**
20	***0.78***	***0.74***	***0.74***	***0.68***	***0.72***	***0.67***	***0.75***	***0.70***	***0.80***	***0.72***	0.86	***0.79***	0.98
25	0.88	***0.80***	***0.82***	***0.76***	***0.83***	***0.76***	0.86	***0.78***	0.95	0.85	1.02	0.92	1.12
30	1.03	0.95	1.00	0.90	1.01	0.92	1.09	0.97	1.17	1.03	**1.29**	1.10	**1.41**
35	**1.28**	**1.20**	1.19	1.14	**1.20**	1.13	**1.28**	1.17	**1.38**	**1.27**	**1.47**	**1.35**	**1.67**
40	**1.32**	**1.26**	**1.30**	**1.20**	**1.32**	1.17	**1.42**	**1.27**	**1.53**	**1.39**	**1.64**	**1.50**	**1.86**
45	**1.42**	**1.30**	**1.38**	**1.27**	**1.42**	**1.29**	**1.50**	**1.37**	**1.67**	**1.50**	**1.86**	**1.61**	**2.03**
50	**1.38**	1.16	1.18	1.07	1.18	1.07	**1.21**	1.13	**1.32**	**1.21**	**1.43**	**1.36**	**1.57**
55	**1.43**	**1.22**	**1.24**	1.14	**1.22**	1.15	**1.31**	**1.22**	**1.40**	**1.32**	**1.58**	**1.43**	**1.72**
60	**1.47**	**1.27**	**1.32**	**1.21**	**1.32**	**1.26**	**1.43**	**1.34**	**1.57**	**1.46**	**1.73**	**1.58**	**1.97**

Table 11: Single precision: Computing GCDs in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

k	$n/1024$								
	1		2		4		8		16
5	**3.84**	**3.19**	**4.44**	**3.65**	**4.88**	**3.23**	**3.88**	**3.86**	**4.56**
10	**4.30**	**3.75**	**5.07**	**4.46**	**5.86**	**4.93**	**6.08**	**5.74**	**6.81**
15	**4.99**	**4.53**	**6.02**	**5.53**	**7.22**	**6.73**	**8.20**	**7.66**	**8.92**
30	**4.52**	**4.08**	**5.34**	**4.89**	**6.24**	**5.85**	**6.70**	**6.21**	**7.28**
60	**5.21**	**4.68**	**6.67**	**5.93**	**8.13**	**7.75**	**9.23**	**8.22**	**9.94**

Table 12: Single precision: Composition modulo a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

k	$n/1024$								
	1		2		4		8		16
5	**1.22**	1.11	**1.39**	**1.46**	**1.77**	**1.98**	**2.03**	**2.23**	**2.40**
10	**1.49**	**1.82**	**1.70**	**1.99**	**2.71**	**2.41**	**2.94**	**3.11**	**3.63**
15	**2.15**	**2.32**	**2.45**	**2.76**	**3.22**	**3.70**	**3.55**	**3.89**	**4.31**
30	**2.49**	**2.76**	**2.97**	**2.70**	**3.01**	**3.16**	**3.53**	**3.36**	**3.73**
60	**3.61**	**3.51**	**4.01**	**3.96**	**4.50**	**4.21**	**5.97**	**4.66**	**4.59**

Table 13: Single precision: Factoring a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

	$n = 10^3$		$n = 10^6$	
	NTL	`gf2x`	NTL	`gf2x`
with PCLMUL	0.21μs	0.20μs	13.3ms	6.4ms
w/o PCLMUL	1.54μs	1.24μs	91.8ms	15.7ms

Table 14: Time to multiply two random polynomials of degree $< n$ over $\mathbb{Z}_2[X]$

To multiply two polynomials over $\mathbb{Z}_2[X]$, NTL recursively performs Karatsuba down to the "base case", which is the multiplication of two "word-sized" polynomials. Note that even before reaching this base case, the recursion switches over at some point to hand-coded, branch-free versions of Karatsuba, with an optimization from Weimerskirch, Stebila, and Shantz [34] for multiplying two 3-word polynomials.

To multiply two word-sized polynomials, one of two strategies is used, depending on the available hardware. Modern x86 machines have a built-in instruction called PCLMUL for precisely this task, and NTL will use this instruction (by way of compiler "intrinsics") if this is possible.

Failing this, NTL will use its default strategy, which is the following "window" method. To multiply two word-sized polynomials a and b, first compute the polynomial $g \cdot b \bmod X^w$ for all polynomials $g \in \mathbb{Z}_2[X]$ of degree less than a small parameter s. Here, w is the number of bits in a word. Typically, $s = 4$ when $w = 64$, and $s = 3$ when $w = 32$. The values $g \cdot b \bmod X^w$ are computed by a series of shifts and XORs (each entry in the table is computed as either the XOR of two previously computed table entries, or a shift of one previously computed table entry). After this, the bits of a are processed in $\lceil w/s \rceil$ blocks of s bits, where for each block, one table entry is fetched, and two shifts and two XORs are performed. Finally, $s - 1$ simple correction steps are executed, to compensate for the fact the values in the table were only computed mod X^w. Each correction step consists of a handful of shifts, ANDs, and XORs. All of the above is carried out using branch-free code that is generated when NTL is configured.

The above strategy was implemented in v2.0 of NTL in 1998 for a window size of $s = 2$. The strategy was later generalized in [4] to arbitrary window size, and incorporated into v5.4 of NTL in 2005.

In fact, the paper [4] mentioned above reports on an implementation of a highly-optimized `C` library called `gf2x` for multiplying polynomials over $\mathbb{Z}_2[X]$. Moreover, NTL can be configured to call the multiplication routine in the `gf2x` library instead of its own version. We compared the performance of the current version of NTL to the current version of `gf2x` (version 1.3, released December 2019). Table 14 shows the time to multiply two random polynomials of degree less than n over $\mathbb{Z}_2[X]$, where $n = 10^3$ and $n = 10^6$, for both NTL and `gf2x`, and both with and without usage of the PCLMUL instruction. As one can see, NTL is fairly competitive with `gf2x` in these ranges, except for the case where $n = 10^6$ and no PCLMUL instruction is used (`gf2x` is almost 6 times faster here).

6 Matrix arithmetic over $\mathbb{Z}_p$

NTL provides support for matrix arithmetic over $\mathbb{Z}_p$, including basic operations such as addition and multiplication, as well as operations such as inversion and solving matrix-vector equations.

Different strategies are used for single-precision and multi-precision modulus p. NTL also implements specialized strategies for $p = 2$, but we do not discuss these here.

6.1 Matrix arithmetic modulo single-precision p

Here are the strategies that NTL uses for computing the product $C = AB$, where A and B are large matrices over $\mathbb{Z}_p$, and where p is a single-precision modulus.

Strassen's divide-and-conquer algorithm. For very large matrices, NTL runs a few levels of Strassen's divide-and-conquer matrix multiplication algorithm [29].

Cache friendly memory access. While Strassen's divide-and-conquer algorithm already yields somewhat cache-friendly code, to obtain even more cache friendly code, all the matrices are organized into *panels*, which are matrices with many rows but only 32 columns. We compute the ith panel of C by computing AB_i, where B_i is ith panel of B. If multiple cores are available, we use them to parallelize the computation, as the panels of C can be computed independently.

Next consider the computation of AP, where P is a single panel. We can write $AP = \sum_j A_j P_j$, where each A_j is a panel of A and each P_j is a 32×32 square submatrix of P. We have thus reduced the problem to that of computing

$$Q \leftarrow Q + RS, \tag{10}$$

where Q and R are panels, and S is a 32×32 square matrix. The matrix S is small and fits into the first-level cache on most machines — that is why we chose a panel width of 32. While the panels Q and R typically do not fit into the first-level cache, the data in each panel is laid out in contiguous memory in row-major order. In the implementation of (10), we process the panels a few rows at a time, so the data in each panel gets processed sequentially, and we rely on hardware prefetch (which is common on modern high-performance systems) to speed up the memory access. Moreover, these fetches are paired with a CPU-intensive computation (involving S, which is in the first-level cache), so the resulting code is fairly cache friendly.

Fast modular arithmetic. The basic arithmetic operation in any matrix multiplication algorithm is the computation of the form $x \leftarrow x + yz$, where x and y are scalars. In our case, the scalars lie in $\mathbb{Z}_p$. While we could use the techniques in Section 3.1, this is not the most efficient approach.

If p is small enough, specifically, at most 23-bits in length, we can use the underlying floating-point hardware that is commonly available and typically very fast.

Indeed, if we have 23-bit numbers w and x_i and y_i, for $i = 1, \ldots, k$, then we can compute $w + \sum_i x_i y_i$ exactly in floating-point, provided k is not too big: since standard (double precision) floating-point can exactly represent 53-bit integers, we can take k up to $2^{53-23\cdot 2} = 2^7$. If k is larger than this, we can still use the fast floating-point hardware, interspersed with occasional "clean up" operations which convert the accumulated floating-point sum to an integer, reduce it mod p, and then convert it back to floating-point.

By using this floating-point implementation, we can also exploit the fact that modern x86 CPUs come equipped with very fast SIMD instructions for quickly performing several floating-point operations concurrently. Our code is geared to Intel's AVX, AVX2, and AVX512 instruction sets, which allows us to process floating-point operations 4 (or 8) at a time.

For p larger than 23 bits, the code reverts to using double-word integer arithmetic (if available) to accumulate inner products (rather than the AVX floating-point instructions), but still uses the same "cache friendly" panel/square memory organization, along with Strassen's divide-and-conquer algorithm, and utilizing multiple cores, if available.

Other matrix operations. NTL also implements other matrix operations modulo single-precision p, such as inversion and solving matrix-vector equations. The same techniques for utilizing cache-friendly code and for fast modular arithmetic are employed.

Comparison to FFLAS. One of the state-of-the-art implementations of matrix operations over finite fields is FFLAS [7]. For small, single-precision p, FFLAS also employs floating-point techniques; however, it reduces all computations to floating-point matrix operations that are then carried out using the well-know BLAS API (see `netlib.org/blas/blast-forum`). We compared the current version of FFLAS (version 2.4.3), paired with the current version of OpenBLAS (version 0.3.7, see `www.openblas.net`). For example, to multiply two 4096×4096 matrices modulo a 23-bit prime, FFLAS took 3.18s and NTL took 3.69s, so FFLAS is about 16% faster on this benchmark. As another benchmark, to invert a matrix of the same size, FFLAS took 4.15s and NTL took 4.74s, so FFLAS is about 14% faster on this benchmark.

6.2 Matrix arithmetic modulo multi-precision p

For large, multi-precision p, NTL uses a multimodular technique for matrix multiplication:

1. The coefficients of the matrix are reduced modulo several single-precision primes $p_1, \ldots, p_k$.
2. The matrix product is computed modulo each p_i, using the techniques outlined above.
3. The results are combined using Chinese remaindering.

Steps 1 and 3 are implemented using the same techniques used for the multimodular polynomial multiplication algorithm, discussed above in Section 4. Step 2 is implemented as discussed above in Section 6.1 (if AVX intructions are availble, then we use 23-bit primes). Just as for the multimodular polynomial multiplication algorithm, these steps are all trivially parallelizable, which is exploited if multiple cores are available.

At the current time, while NTL does provide implementations of other matrix operations modulo multi-precision p, such as inversion and solving matrix-vector equations, these implementations are very basic, and not as fast as they could be.

Comparison to FFLAS. We compared NTL's multimodular matrix multiplication performance to that of FFLAS (see discussion of FFLAS at the end of Section 6.1), which uses a similar multimodular approach. For example, to multiply two 1024×1024 matrices modulo a 1024-bit prime, FFLAS took 11.5s and NTL took 11.9s, essentially a tie. On the same benchmark, but using integer-only scalar arithmetic (no floating-point or AVX, using 60-bit primes), NTL's multimodular algorithm takes 22.8s, which is indicative of NTL's performance on other platforms where it would not be able to take advantage of AVX. As another comparison, on this same benchmark, NTL's "plain" matrix multiplication code (which carries out the computation naively using multi-precision integer arithmetic) takes 217s (so the multimodular algorithm is still much faster even without AVX support).

7 Polynomial and matrix arithmetic over other finite rings

NTL also provides classes for the ring of univariate polynomials $E[X]$, where E is a quotient ring of the form $\mathbb{Z}_p[T]/(m(T))$. These classes are called `ZZ_pEX`, `zz_pEX`, and `GF2EX`, respectively, depending on whether p is multi-precision, single-precision, or equal to 2, and are implemented in terms of the classes `ZZ_pX`, `zz_pX`, and `GF2X`, respectively. For low degree polynomials over E, naive, quadratic-time algorithms are used. For large degree polynomials, multiplication is implemented via Kronecker substitution [16], which reduces the problem of multiplication in $E[X]$ first to multiplication in $\mathbb{Z}_p[X,T]$ and then to multiplication in $\mathbb{Z}_p[X]$. Most other operations in $E[X]$ (e.g., division, GCD) are reduced to polynomial multiplication.

Algorithms for factoring univariate polynomials over E, computing minimal polynomials, and generating irreducible polynomials over E are also available (assuming E is a field).

NTL also provides support for basic matrix operations over E. Currently, these implementations are fairly basic, and there is plenty of room for performance improvements in this area.

8 Polynomial and matrix arithmetic over $\mathbb{Z}$

NTL provides the class `ZZX`, which represents the ring $\mathbb{Z}[X]$ of univariate polynomials over the integers. Multiplication in $\mathbb{Z}[X]$ is done using essentially the same tech-

niques as in Section 4. In particular, it implements several algorithms: plain, Karatsuba, Schönhage-Strassen, and multimodular FFT.

For a number of other operations, such as GCD, a multimodular approach is employed, reducing these operations to corresponding operations in $\mathbb{Z}_p[X]$ for several small primes p.

An algorithm for factoring univariate polynomials over $\mathbb{Z}$ is also provided. The algorithm is based on the well-known Berlekamp-Zassenhaus strategy [35], but with a number of critical improvements. The algorithm works by first ensuring that the given polynomial is squarefree (this is essentially just a GCD computation, and is usually very fast). Second, this squarefree polynomial is factored modulo several small primes, and one small prime p is selected as "best", meaning that the number r of irreducible factors mod p is minimal. Third, this factorization mod p is lifted to a factorization mod p^k for a suitably large k. This is done using an asymptotically fast Hensel lifting procedure.

The final step is to discover those subsets of the irreducible factors mod p^k that multiply out to factors of the polynomial over the integers. The naive strategy, which tries all possible subsets, takes time exponential in r. If r is not too big, NTL employs the heuristic pruning strategy introduced in [1], which speeds up this exponential-time strategy significantly. However, for larger r, NTL switches to Mark van Hoeij's algorithm [33], which reduces the problem of discovering appropriate subsets of factors to that of solving a certain type of knapsack problem. This knapsack problem is itself solved using a lattice basis reduction algorithm.

NTL also supports a number of operations on matrices over $\mathbb{Z}$. In addition to algorithms for basic arithmetic, NTL provides a number of algorithms for lattice basic reduction, i.e., variations on the famous LLL algorithm [17]:

- An exact integer version of LLL is implemented, which is essentially the same as one presented in [5].
- A number of faster, heuristic floating-point versions of LLL are implemented. These are variations on algorithms presented by Schnorr and Euchner [25], with significant modifications to improve performance and to deal with rounding errors more robustly.
- A number of heuristic floating-point versions of Block Korkin-Zolotarev (BKZ) reduction (also from [25]), which can produce much higher quality reduced bases than LLL, are implemented.

Much of the work on the floating-point LLL and BKZ algorithms in NTL arose out of some very pleasant interactions with Phong Nguyen, who was attempting to use NTL's initial implementations of LLL to break certain lattice-based cryptosystems [22, 20]. These attempts really "stress tested" NTL's implementation, and led to several heuristic improvements over the original floating-point algorithms proposed by Schnorr and Euchner. One such improvement was utilizing a "lazy" size-reduction condition. Indeed, the floating-point LLL algorithms presented in [25] impose the classical size-reduction condition, in which the Gramm-Schmidt coefficients of the lattice basis must be at most $1/2$ in absolute value. Unfortunately, using this classical size-reduction

was found to easily lead to infinite loops, due to the fact that the Gramm-Schmidt coefficients are only being computed approximately. NTL's implementation replaces the bound $1/2$ in the size-reduction condition by $1/2 + \epsilon$, where ϵ starts out very small, but then grows as infinite loops are (heuristically) detected. Other improvements in NTL's LLL implementation include:

- implementation of Givens rotations, in place of Gramm-Schmidt orthogonalization, which yields better numerical stability;
- implementations of LLL algorithms that use other floating-point types to yield (at the expense of performance) greater stability and/or range.

In support of these floating-point LLL implementations, as well as for other applications, NTL provides several specialized floating-point classes:

- `quad_float`, which provides (essentially) twice the precision of ordinary double-precision floating-point. This class was derived from software developed previously by Keith Briggs (see `keithbriggs.info/doubledouble.html`), which itself was derived from software developed earlier by Douglas Priest. [24]
- `xdouble`, which provides double-precision floating-point with an extended exponent range.
- `RR`, which provides arbitrary precision floating-point (with correct rounding for basic arithmetic and square root), and extended exponent range (note that `RR` provides functionality similar to the MPFR library [8], which was developed some time later than NTL's `RR`).

In recent years, better floating-point LLL algorithms have been developed [21] and implemented [30]. However, these algorithms have not yet been incorporated into NTL. For an excellent survey on more recent developments on the LLL algorithm, see [23].

9 The future of NTL

Currently, the author has no plans to add significant new functionality to NTL. Rather, the plan is to continue to improve the performance of NTL, either by implementing new algorithms or by exploiting new hardware features. In addition, the author hopes to make it easier for others to contribute to NTL by using a code-hosting facility. At that stage, other contributors may perhaps wish to add new functionality to NTL.

References

[1] Abbott, John, Shoup, Victor, and Zimmermann, Paul. 2000. Factorization in Z[x]: the searching phase. Pages 1–7 of: *Proceedings of the 2000 International Symposium on Symbolic and Algebraic Computation, ISSAC 2000, St. Andrews, United Kingdom, August 6-10, 2000*.

[2] Bluestein, L. 1970. A linear filtering approach to the computation of discrete Fourier transform. *IEEE Transactions on Audio and Electroacoustics*, **18**(4), 451–455.

[3] Brent, Richard P., and Kung, H. T. 1978. Fast Algorithms for Manipulating Formal Power Series. *J. ACM*, **25**(4), 581–595.

[4] Brent, Richard P., Gaudry, Pierrick, Thomé, Emmanuel, and Zimmermann, Paul. 2008. Faster Multiplication in GF(2)[x]. Pages 153–166 of: *Algorithmic Number Theory, 8th International Symposium, ANTS-VIII, Banff, Canada, May 17-22, 2008, Proceedings*.

[5] Cohen, Henri. 1993. *A Course in Computational Algebraic Number Theory*. Springer.

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

[7] Dumas, Jean-Guillaume, Giorgi, Pascal, and Pernet, Clément. 2008. Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages. *ACM Trans. on Mathematical Software (TOMS)*, **35**(3), 1–42.

[8] Fousse, Laurent, Hanrot, Guillaume, Lefèvre, Vincent, Pélissier, Patrick, and Zimmermann, Paul. 2007. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, **33**(2), 13.

[9] Goldberg, David. 1991. What Every Computer Scientist Should Know About Floating Point Arithmetic. *ACM Computing Surveys*, **23**(1), 5–48.

[10] Granlund, Torbjörn, and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. `http://gmplib.org/`.

[11] Halevi, Shai, and Shoup, Victor. 2014. Algorithms in HElib. Pages 554–571 of: *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*.

[12] Harvey, David. 2014. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, **60**, 113–119.

[13] Kaltofen, Erich, and Shoup, Victor. 1998. Subquadratic-time factoring of polynomials over finite fields. *Math. Comput.*, **67**(223), 1179–1197.

[14] Karatsuba, Anatoly, and Ofman, Yuri. 1962. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, **145**, 293–294. Translation in Physics-Doklady 7, 595–596, 1963.

[15] Kedlaya, Kiran S., and Umans, Christopher. 2011. Fast Polynomial Factorization and Modular Composition. *SIAM J. Comput.*, **40**(6), 1767–1802.

[16] Kronecker, Leopold. 1882. Grundzüge einer arithmetischen theorie der algebraischen grössen. *Journal für die reine und angewandte Mathematik*, 1–122.

[17] Lenstra, A. K., Lenstra, H. W., and Lovász, L. 1982. Factoring polynomials with rational coefficients. *Mathematische Annalen*, **261**(4), 515–534.

[18] Möller, N., and Granlund, T. 2011. Improved Division by Invariant Integers. *IEEE Transactions on Computers*, **60**(2), 165–175.

[19] Montgomery, Peter L. 1985. Modular multiplication without trial division. *Mathematics of Computation*, **44**, 519–521.

[20] Nguyen, Phong Q. 1999. Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97. Pages 288–304 of: *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*.

[21] Nguyen, Phong Q., and Stehlé, Damien. 2005. Floating-Point LLL Revisited. Pages 215–233 of: *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*.

[22] Nguyen, Phong Q., and Stern, Jacques. 1998. Cryptanalysis of the Ajtai-Dwork Cryptosystem. Pages 223–242 of: *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*.

[23] Nguyen, Phong Q., and Vallée, Brigitte (eds). 2010. *The LLL Algorithm - Survey and Applications*. Information Security and Cryptography. Springer.

[24] Priest, Douglas M. 1992 (11). *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Ph.D. thesis, University of California, Berkeley. `ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[25] Schnorr, Claus-Peter, and Euchner, M. 1991. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Pages 68–85 of: *Fundamentals of Computation Theory, 8th International Symposium, FCT '91, Gosen, Germany, September 9-13, 1991, Proceedings*.

[26] Schönhage, Arnold, and Strassen, Volker. 1971. Schnelle Multiplikation großer Zahlen. *Computing*, **7**(3-4), 281–292.

[27] Shoup, Victor. 1993. Factoring polynomials over finite fields: Asymptotic complexity vs. reality. Pages 124–129 of: *Proc. IMACS Symposium*.

[28] Shoup, Victor. 1995. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation*, **20**, 363–397.

[29] Strassen, Volker. 1969. Gaussian elimination is not optimal. *Numerische Mathematik*, **13**, 354–356.

[30] The FPLLL development team. 2016. *fplll, a lattice reduction library*. Available at `https://github.com/fplll/fplll`.

[31] van der Hoeven, Joris. 2004. The Truncated Fourier Transform and Applications. Pages 290–296 of: *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC*.

[32] van der Hoeven, Joris, and Lecerf, Grégoire. 2020. Fast multivariate multi-point evaluation revisited. *J. Complexity*, **56**.

[33] van Hoeij, Mark. 2002. Factoring Polynomials and the Knapsack Problem. *Journal of Number Theory*, **95**(2), 167 – 189.

[34] Weimerskirch, André, Stebila, Douglas, and Shantz, Sheueling Chang. 2003. Generic GF(2) Arithmetic in Software and Its Application to ECC. Pages 79–92 of: *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*.

[35] Zassenhaus, Hans. 1969. On Hensel factorization, I. *Journal of Number Theory*, **1**(3), 291–311.