FLINT and Fast Polynomial Arithmetic

David Harvey

June 13, 2007

What is FLINT?

- FLINT = "**F**ast **Li**brary for **N**umber **T**heory"
- www.flintlib.org
- A new C library written by William Hart (Warwick) and myself, probably more contributors soon
- GPL
- Version 1.0 not yet released

Components of FLINT

- Bill's quadratic sieve (integer factorisation)
- Dense polynomial arithmetic in $\mathbf{Z}[x]$

I'm only discussing the polynomial arithmetic today.

Relationship to SAGE

- The quadratic sieve is already in SAGE.
- I hope that the polynomial arithmetic will eventually be used in SAGE as the native $\mathbf{Z}[x]$ code, when it's mature enough.

Why a new library?

What's wrong with $\mathbf{Z}[x]$ arithmetic in NTL? PARI? LiDIA?

- PARI: no asymptotically fast polynomial arithmetic. Best algorithm for multiplication is Karatsuba.
- LiDIA: multiplication in $\mathbf{Z}[x]$ is the *naive* algorithm only. Arithmetic in $\mathbf{F}_q[x]$ is essentially a port of NTL.
- NTL: good asymptotic performance (mostly). Extremely non-threadsafe (conscious design decision). Does not fully utilise 64-bit processors.
- Speed..... more on that later :-)

Case study: the q-expansion of Δ

> *"One of my favorite NTL-based computations is computing the q-expansion of the delta function."*

— William Stein (sage-devel, 13th May 2007)

What is Δ?

Δ is a modular form of weight 12, essentially the discriminant of an elliptic curve. Has a q-series expansion:

$$\Delta(q) = q - 24q^2 + 252q^3 - 1472q^4 + \cdots$$
$$+ 262191418612588689102548992000000q^{1000000} + \cdots$$

Our goal is to compute the coefficients of $\Delta(q)$.

Computing $\Delta(q)$ in SAGE

SAGE already knows how to compute this (code by William Stein):

```
sage: delta_qexp(10)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5
- 6048*q^6 - 16744*q^7 + 84480*q^8 - 113643*q^9
+ O(q^10)
```

What algorithm does SAGE use?

Let

$$F(q) = \sum_{n=0}^{\infty} (-1)^n (2n+1) q^{n(n+1)/2}$$
$$= 1 - 3q + 5q^3 - 7q^6 + 9q^{10} - 11q^{15} + \cdots .$$

Then

$$\Delta(q) = q\, F(q)^8.$$

What algorithm does SAGE use?

So to compute N terms of $\Delta(q)$, the algorithm is:

- Write down $F(q) = 1 - 3q + 5q^3 - 7q^6 + \cdots$ up to $O(q^N)$.
- Square it.
- Truncate it to $O(q^N)$.
- Square it.
- Truncate it to $O(q^N)$.
- Square it.
- Truncate it to $O(q^N)$.
- Multiply by q.
- Result is $\Delta(q) = q\,F(q)^8$ up to $O(q^{N+1})$.

(The first squaring step is best done 'naively' since $F(q)$ has only $O(\sqrt{N})$ nonzero terms up to $O(q^N)$.)

Implementation in SAGE

Here is the (simplified) code for delta_qexp():

```
def delta_qexp(N=10):
  # make list of index/value pairs for F(q)
  stop = int((-1+math.sqrt(1+8*N))/2.0)
  values = [(n*(n+1)//2, (-1)**n * (2*n+1)) for n in range(stop+1)]

  # compute F(q)^2
  v = [0] * N
  for (i1, v1) in values:
    for (i2, v2) in values:
      try:
        v[i1 + i2] += v1 * v2
      except IndexError:
        break

  # use NTL to compute the remaining fourth power
  f = ntl.ZZX(v)
  f = f * f
  f = f.truncate(N)
  f = f * f
  f = f.truncate(N)
  f = ntl.ZZX([0,1]) * f      # multiply by q

  # convert to power series ring
  R = ZZ[['q']]
  f = R(f.truncate(N), N, check=False)
  return f
```

Implementation in SAGE

Typical running time on sage.math:

```
sage: time f = delta_qexp(10^6)
 CPU times: user 21.60 s, sys: 0.97 s, total: 22.57 s
 Wall time: 22.57
```

But there's python overhead, conversion overhead, truncations involve unnecessary copies, etc. If we measure *just the NTL polynomial multiplication time*, we get:

- At best 5.6s for the second multiplication.
- At best 9.1s for the third multiplication.
- Therefore at least 14.7s spent inside NTL.

Implementation in Magma

Here is a Magma implementation (not as colourful unfortunately):

```
N := 200000;
ZZ := Integers();
R<x> := PolynomialRing(ZZ);

// Compute coefficients of F(q)
stop := Ceiling((-1 + Sqrt(1+8*N))/2);
values := [ZZ!0 : i in [0 .. N-1]];
for i := 0 to stop do
    r := ZZ!(i*(i+1)/2);  v := (-1)^i * (2*i+1);
    for j := 0 to stop do
        s := ZZ!(j*(j+1)/2);  w := (-1)^j * (2*j+1);
        if r + s lt N then
            values[r+s+1] := values[r+s+1] + v*w;
        end if;
    end for;
end for;

F := R!values;

// Now compute the truncated fourth power
time F := F * F;
F := R![Coefficient(F, i) : i in [0..N-1]];
time F := F * F;
F := R![Coefficient(F, i) : i in [0..N-1]];

print Coefficient(F, N-1);
quit;
```

Implementation in Magma

Running time for $N = 200000$ (Magma V2.13-5):

- At best 5.6s for the second multiplication.
- At best 9.5s for the third multiplication.

So *NTL is already winning* even though we're only computing one fifth of the terms.

Fascinating facts about Magma's polynomial arithmetic

Huh? What's going on?

I thought Magma was supposed to be wicked fast!

Surely Magma is not being trounced by a piece of *free software*??

Fascinating facts about Magma's polynomial arithmetic

Actually, Magma *can* multiply polynomials in $\mathbf{Z}[x]$ much more quickly, as long as you're not trying to *square* the polynomial:

```
> R<x> := PolynomialRing(Integers());
> f := R![Random(2^49) : i in [0..200000]];
> g := R![Random(2^49) : i in [0..200000]];
> time h := f * g;
Time: 0.970
> time h := f * f;
Time: 9.000
```

(The 0.97s is about twice as fast as NTL.)

So perhaps we can trick Magma into squaring our polynomials more quickly, by *pretending* that they are distinct.

Fascinating facts about Magma's polynomial arithmetic

No such luck. Magma has great difficulty multiplying polynomials with *signed* coefficients:

```
> R<x> := PolynomialRing(Integers());
> f := R![Random(2^49) - 2^48 : i in [0..200000]];
> g := R![Random(2^49) - 2^48 : i in [0..200000]];
> h := f * g;

[Interrupt twice in half a second; exiting]

Total time: 42.270 seconds,
Total memory usage: 49192.68 MB
```

Fascinating facts about Magma's polynomial arithmetic

But Magma is quite capable of *squaring* a polynomial with signed coefficients:

```
> R<x> := PolynomialRing(Integers());
> f := R![Random(2^49) - 2^48 : i in [0..200000]];
> time h := f * f;
Time: 9.240
```

It's just a whole lot slower than multiplying *distinct* polynomials with *non-negative* coefficients.

Fascinating facts about Magma's polynomial arithmetic

William Hart reported these weirdnesses to the Magma folks, and Allan Steel replied that the bugs are now fixed.

While we wait for the patch to come through ("`magma -upgrade`" appears to be broken)....

Fascinating facts about Magma's polynomial arithmetic

To make a reasonably fair comparison, instead of trying to compute $\Delta(q)$, let's use Magma to compute the non-modular-form

$$\nabla(q) = q(1 + 3q + 5q^3 + 7q^6 + 9q^{10} + 11q^{15} + \cdots)^8.$$

This is the same formula, without the minus signs.

(And remember, we have to trick Magma into multiplying, not squaring!)

Fascinating facts about Magma's polynomial arithmetic

Here are the results for $N = 10^6$:

- At best 3.8s for the second multiplication.
- At best 7.6s for the third multiplication.
- Total is at least 11.4s.

Recall that NTL took at least 14.7s. (And NTL was doing slightly less work, since the coefficients were slightly smaller, and it could take advantage of the squaring.)

For problems of this type, squaring is asymptotically at best 1/3 faster than multiplying. So when Magma is repaired, we might optimistically project a running time of 7s.

Implementation in FLINT

WARNING: the FLINT interface suggested by the following code is

- preliminary;
- not yet fully implemented;
- not yet adequately documented;
- not yet thoroughly tested; and
- subject to change.

Implementation in FLINT (page 1/2)

```c
#include <stdio.h>
#include <gmp.h>
#include <math.h>
#include "flint.h"
#include "fmpz_poly.h"

int main(int argc, char* argv[])
{
    // number of terms to compute
    long N = atoi(argv[1]);

    // compute coefficients of F(q)^2
    long* values = malloc(sizeof(long) * N);
    for (long i = 0; i < N; i++)
        values[i] = 0;

    long stop = (long) ceil((-1.0 + sqrt(1.0 + 8.0*N)) / 2.0);
    for (long i = 0; i <= stop; i++)
    {
        long index1 = i*(i+1)/2;
        long value1 = (i & 1) ? (-2*i-1) : (2*i+1);
        for (long j = 0; j <= stop; j++)
        {
            long index2 = j*(j+1)/2;
            if (index1 + index2 >= N)
                break;
            long value2 = (j & 1) ? (-2*j-1) : (2*j+1);
            values[index1 + index2] += value1 * value2;
        }
    }
```

Implementation in FLINT (page 2/2)

```
    // Create some polynomial objects
    fmpz_poly_t F2, F4, F8;
    fmpz_poly_init(F2);
    fmpz_poly_init(F4);
    fmpz_poly_init(F8);

    // Initialise F2 with coefficients of F(q)^2
    fmpz_poly_ensure_space(F2, N);
    for (long i = 0; i < N; i++)
        fmpz_poly_set_coeff_si(F2, i, values[i]);
    free(values);

    // compute F(q)^8, truncated to length N
    fmpz_poly_mul(F4, F2, F2);
    fmpz_poly_truncate(F4, N);
    fmpz_poly_mul(F8, F4, F4);
    fmpz_poly_truncate(F8, N);

    // print out last coefficient
    mpz_t x;
    mpz_init(x);
    fmpz_poly_get_coeff_mpz(x, F8, N-1);
    gmp_printf("coefficient of q^%d is %Zd\n", N, x);

    // clean up
    mpz_clear(x);
    fmpz_poly_clear(F8);
    fmpz_poly_clear(F4);
    fmpz_poly_clear(F2);
    return 0;
}
```

Live demo

>> insert live demo here <<

(download, build, and run)

Summary of running times

- NTL: 14.7s
- Magma: 7s (optimistic post-bug-removal estimate)
- FLINT: 5.2s

I also tried $N = 5 \cdot 10^7$ in SAGE/NTL. After three minutes, it reports

```
Polynomial too big for FFT
```

(This could be fixed, but NTL would need to be recompiled.)

FLINT does this N in about 7 minutes and 13GB RAM. Coefficient of q^N is $-204669665570074073125865416474214400000000$.

Code walkthrough

FLINT is organised as a collection of modules. One needs to import the main flint.h header, as well as any required modules. Our example used only the fmpz_poly module.

```
#include "flint.h"
#include "fmpz_poly.h"
```

Code walkthrough

FLINT has two completely separate data types for representing polynomials in $\mathbf{Z}[x]$.

- mpz_poly_t is essentially an array of mpz_t's. (This is similar to NTL's ZZX polynomial type, but NTL uses ZZ instead of mpz_t.)
- fmpz_poly_t stores the coefficients as raw data in a single contiguous block. Each coefficient takes up the same amount of space.

As far as possible, the interfaces for the two formats are consistent, and conversions between them are provided.

Code walkthrough

fmpz_poly_t is *much* more efficient, if you are working with dense polynomials whose coefficients are all about the same size. Especially when the coefficients are relatively small, it avoids a tremendous amount of memory management overhead.

mpz_poly_t is better suited if the coefficients have very different sizes. Also, GMP's mpz-layer functions can be used directly on the coefficients, which might be convenient in some contexts.

Code walkthrough

Each coefficient in a fmpz_poly_t is stored in a FLINT multiple-precision integer format called fmpz_t.

An fmpz_t is not a struct; it's just a typedef for a pointer to memory. Basically it's an "mpz_t without memory management":

- The first limb (word) is a control limb. The integer is positive/negative/zero according to whether the control limb is positive/negative/zero. The absolute value of the control limb is the number of limbs used to store the value.
- The absolute value of the integer itself follows directly in memory, in GMP's mpn format.

Code walkthrough

The fmpz module will provide arithmetic on this format. The user is of course responsible for memory management.

Obviously it's more difficult for a user to work with this format than with mpz_t.

But it's much more efficient than mpz_t for building other data structures (such as polynomials), if you want more control over memory management.

Code walkthrough

The fmpz_poly_init() function must be called prior to using a polynomial object:

```
fmpz_poly_t F2, F4, F8;
fmpz_poly_init(F2);
fmpz_poly_init(F4);
fmpz_poly_init(F8);
```

The fmpz_poly_clear() function must be called when the polynomial is no longer needed:

```
fmpz_poly_clear(F8);
fmpz_poly_clear(F4);
fmpz_poly_clear(F2);
```

We use some tricks we learned from GMP to enable pass-by-reference semantics in C.

Code walkthrough

In this block we copy the coefficients of $F(q)^2$ into F2:

```
fmpz_poly_ensure_space(F2, N);
for (long i = 0; i < N; i++)
    fmpz_poly_set_coeff_si(F2, i, values[i]);
```

The call to fmpz_poly_ensure_space() ensures that enough space is allocated in advance. If it was left out, the code would still work, but the loop would occasionally need to reallocate.

The function fmpz_poly_set_coeff_si () sets a given coefficient to the given signed integer. Generally we try to use function naming conventions that would be familiar to a GMP programmer (such as the 'si' suffix).

Code walkthrough

Note that fmpz_poly_set_coeff_si () performs bounds checking and automatic reallocation. If this is unacceptable overhead, one can use instead _fmpz_poly_set_coeff_si (), which performs no bounds checking.

Many FLINT functions have a 'managed' and an 'unmanaged' version like this. The 'managed' ones take care of everything (in particular memory management) so that the user doesn't need to think.

The 'unmanaged' ones require more vigilance by the programmer, but will be more efficient. They are intended to be fully documented and user-accessible. They are generally not intended to be private.

Code walkthrough

The following code multiplies F2 by itself, puts the result in F4, and then truncates F4 to length N:

```
fmpz_poly_mul(F4, F2, F2);
fmpz_poly_truncate(F4, N);
```

FLINT automatically selects an appropriate multiplication algorithm based on the degree and coefficient sizes of the input polynomials, and takes advantage of the two input polynomials being identical.

The polynomial multiplication code is currently about 5000 lines, and climbing.

Code walkthrough

Finally, the following block extracts the last coefficient of F8 as an mpz_t object, and uses GMP to print it out:

```
mpz_t x;
mpz_init(x);
fmpz_poly_get_coeff_mpz(x, F8, N-1);
gmp_printf("coefficient of q^%d is %Zd\n", N, x);
mpz_clear(x);
```