NTL vs FLINT

Victor Shoup
`shoup@cs.nyu.edu`

February 8, 2021

0 Introduction

We have compiled some benchmarks that compare the relative performance of NTL (`http://shoup.net/ntl/`) and FLINT (`http://www.flintlib.org/`) on some fundamental benchmarks.

0.1 Methodology

All tests were carried out on a very lightly loaded machine with a 64-bit Intel "Haswell" CPU (Intel Xeon CPU E5-2698 v3 at 2.30GHz) with plenty of memory (over 250GB). The operating system was Cent OS. The compiler was gcc v7.3.1.

We compared NTL v11.4.3 with FLINT v2.7.1. These were both built using GMP v6.2.0. FLINT was built using OpenBlas v0.3.7. Both GMP and OpenBlas were configured to be optimized the Haswell architecture.

For each basic operation, the test program generated random inputs of a given size using NTL's pseudo-random generator, and then converted these NTL objects to corresponding FLINT objects. So in all cases, both libraries were working on identical objects. Also, the test program iterated the basic operation sufficiently many time to ensure that at least 3 seconds passed (for the NTL execution), to ensure fairly accurate timing. Time itself was measured using `getrusage` (system plus user time). All tests were run on a single thread — while both NTL and FLINT can exploit multiple threads, this behavior was not tested.

Test programs may be downloaded here: `http://shoup.net/ntl/benchtools.tar`.

1 Multiplication in $\mathbb{Z}_p[X]$

Fig. 1 compares the relative speed of NTL's `ZZ_pX mul` routine with FLINT's `fmpz_mod_poly_mul` routine. 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.[1] 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:

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

[1] 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.

So ratios greater than 1 mean NTL is faster, and ratios less than 1 mean FLINT is faster. The ratios are also color coded. Ratios between 1/1.2 and 1.2 are gray (essentially a tie), while ratios greater than 1.2 are green (NTL clearly wins) and those less than 1/1.2 are red (FLINT clearly wins). Emphasis is added to ratios that are greater than 2 (and 4), or less than 1/2 (and 1/4).

The ratios in the upper right-hand corner of the table essentially compare NTL's multi-modular FFT algorithm with FLINT's Kronecker-substitution algorithm. The ratios in the lower left-hand corner of the table essentially compare NTL's Schönhage-Strassen algorithm with FLINT's Schönhage-Strassen algorithm.

$k/1024$	$n/1024$												
	1/4		1/2		1		2		4		8		16
1/4	**2.76**	**2.48**	**2.84**	**2.57**	**2.54**	**2.46**	**2.61**	**2.48**	**2.59**	**2.53**	**2.55**	**2.30**	**2.52**
1/2	1.45	1.57	1.56	1.79	1.74	**2.07**	**2.08**	**2.33**	**2.44**	**2.21**	**2.54**	**2.32**	**3.26**
1	1.07	1.12	1.11	1.24	1.22	1.42	1.40	1.86	1.85	1.99	**2.94**	**2.26**	**2.83**
2	0.83	0.85	0.84	0.90	0.88	0.98	0.97	1.20	1.17	1.63	1.60	1.75	**2.17**
4	0.98	1.00	0.96	1.00	1.00	1.00	0.99	1.07	1.06	1.23	1.14	1.43	1.41
8	1.05	1.04	1.03	1.02	1.00	0.98	0.97	0.95	0.94	1.02	0.98	0.95	0.94
16	0.96	0.97	0.97	0.97	0.96	0.96	0.94	0.93	0.91	0.87	0.85	0.91	0.89

Figure 1: Multiplication in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

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

Fig. 2 compares the relative performance of NTL's `ZZ_pX MulMod` routine with FLINT's corresponding routine. The NTL routine takes as input precomputations based on f, specifically, a `ZZ_pXModulus` object. The corresponding FLINT routine is `fmpz_mod_poly_mulmod_preinv`. 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 two multiplicands were random polynomials of degree less than n.

NTL is using a multi-modular FFT strategy throughout, while FLINT is using Kronecker-substitution in the upper right region and Schönhage-Strassen in the lower left region.

The numbers in this table — and all the other tables in this report — have precisely the same meaning as in the table in Fig. 1.

$k/1024$	$n/1024$												
	1/4		1/2		1		2		4		8		16
1/4	**4.31**	**3.73**	**4.25**	**3.74**	**4.00**	**3.57**	**3.94**	**3.61**	**4.04**	**3.66**	**3.88**	**3.36**	**3.88**
1/2	**2.22**	**2.27**	**2.36**	**2.61**	**2.68**	**3.12**	**3.21**	**3.49**	**3.84**	**3.27**	**3.94**	**3.44**	**5.09**
1	1.62	1.64	1.68	1.81	1.85	**2.13**	**2.15**	**2.79**	**2.83**	**3.00**	**4.58**	**3.20**	**4.31**
2	1.28	1.24	1.27	1.34	1.34	1.49	1.51	1.82	1.83	**2.39**	**2.54**	**2.48**	**3.22**
4	1.50	1.47	1.49	1.47	1.50	1.47	1.50	1.60	1.63	1.77	1.84	**2.08**	**2.07**
8	0.76	0.78	0.78	0.79	0.79	0.76	0.78	0.81	0.84	0.87	0.88	1.01	0.99
16	0.58	0.57	0.58	0.58	0.58	0.58	0.58	0.60	0.59	0.61	0.59	0.66	0.65

Figure 2: Multiplication in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

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

Fig. 3 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 corresponding FLINT routine is `fmpz_mod_poly_mulmod_preinv`. This routine internally checks if the multiplicands point to the same object, and optimizes accordingly. 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 multi-modular FFT strategy throughout, while FLINT is using Kronecker-substitution in the upper right region and Schönhage-Strassen in the lower left region.

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 $\mathbb{Z}_p[X]/(f)$.

$k/1024$	$n/1024$												
	1/4		1/2		1		2		4		8		16
1/4	**4.14**	**3.66**	**4.31**	**3.72**	**3.90**	**3.62**	**4.01**	**3.68**	**4.11**	**3.81**	**4.12**	**3.66**	**4.18**
1/2	**2.38**	**2.43**	**2.51**	**2.77**	**2.85**	**3.36**	**3.47**	**3.71**	**4.05**	**3.58**	**4.20**	**3.90**	**5.27**
1	1.78	1.82	1.87	**2.03**	**2.05**	**2.33**	**2.42**	**3.11**	**3.15**	**3.43**	**4.93**	**3.58**	**4.79**
2	1.43	1.43	1.51	1.51	1.55	1.67	1.70	**2.05**	**2.06**	**2.82**	**2.92**	**2.85**	**3.45**
4	1.62	1.65	1.65	1.61	1.67	1.61	1.66	1.77	1.80	1.89	**2.01**	**2.33**	**2.31**
8	0.85	0.89	0.86	0.89	0.86	0.90	0.87	0.89	0.94	0.97	0.94	1.08	1.09
16	0.63	0.64	0.64	0.64	0.65	0.64	0.63	0.66	0.66	0.67	0.66	0.73	0.72

Figure 3: Squaring in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

4 Pre-conditioned multiplication in $\mathbb{Z}_p[X]/(f)$

In some situations, one needs to compute $ab \bmod f$, where not only is f fixed for many operations, but so is b. This arises, for example, in a repeated squaring exponentiation over $\mathbb{Z}_p[X]/(f)$. As a second example, this arises in computing successive powers of a polynomial mod f, which happens in building the matrix used in Berlekamp's polynomial factoring algorithm, or in Brent and Kung's modular composition algorithm. A third example would be scalar/vector products over $\mathbb{Z}_p[X]/(f)$.

Fig. 4 compares the relative performance of NTL's `ZZ_pX` pre-conditioned `MulMod` routine with FLINT's corresponding routine. The NTL routine takes as input precomputations based on f and b, specifically, a `ZZ_pXModulus` object and a `ZZ_pXMultiplier` object. There is no directly comparable FLINT routine — the best choice is the same routine we used above: `fmpz_mod_poly_mulmod_preinv`. 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 two multiplicands were random polynomials of degree less than n.

NTL is using a multi-modular FFT strategy throughout, while FLINT is using Kronecker-substitution in the upper right region and Schönhage-Strassen in the lower left region.

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

Fig. 5 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 k. Both libraries use a fast "Half GCD" algorithm.

$k/1024$	$n/1024$												
	1/4		1/2		1		2		4		8		16
1/4	**7.79**	**6.38**	**7.84**	**6.58**	**6.97**	**6.36**	**7.22**	**6.38**	**7.22**	**6.54**	**7.24**	**5.91**	**7.12**
1/2	**3.93**	**4.01**	**4.21**	**4.56**	**4.88**	**5.43**	**5.76**	**6.14**	**6.86**	**5.82**	**7.10**	**6.15**	**9.30**
1	**2.86**	**2.86**	**2.98**	**3.23**	**3.33**	**3.76**	**3.87**	**5.00**	**5.04**	**5.20**	**8.38**	**5.80**	**7.95**
2	**2.32**	**2.19**	**2.26**	**2.37**	**2.33**	**2.64**	**2.72**	**3.22**	**3.32**	**3.87**	**4.60**	**4.58**	**5.94**
4	**2.61**	**2.61**	**2.70**	**2.50**	**2.72**	**2.66**	**2.71**	**2.77**	**2.94**	**3.14**	**3.33**	**3.64**	**3.72**
8	1.36	1.39	1.37	1.39	1.39	1.39	1.41	1.41	1.42	1.56	1.54	1.79	1.75
16	1.00	1.01	1.02	1.02	1.03	1.01	1.02	1.06	1.04	1.07	1.05	1.17	1.15

Figure 4: Pre-conditioned multiplication 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	1.51	1.60	1.84	1.86	**2.18**	**2.10**	**2.52**	**2.31**	**2.75**	**2.46**	**2.96**	**2.59**	**3.11**
1/2	1.40	1.47	1.58	1.64	1.75	1.77	1.88	1.89	**2.06**	**2.07**	**2.28**	**2.25**	**2.45**
1	1.22	1.30	1.31	1.38	1.41	1.45	1.49	1.53	1.60	1.63	1.69	1.73	1.83
2	1.13	1.24	1.17	1.28	1.22	1.29	1.25	1.33	1.30	1.37	1.36	1.42	1.42
4	1.16	1.30	1.27	1.37	1.33	1.40	1.37	1.45	1.42	1.45	1.47	1.48	1.52
8	1.00	1.14	1.01	1.13	1.02	1.10	1.02	1.08	1.00	1.06	1.00	1.03	0.98
16	0.95	1.08	0.92	1.04	0.89	0.99	0.87	0.97	0.86	0.94	0.85	0.91	0.84

Figure 5: Computing GCDs in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

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

Fig. 6 compares the relative performance of NTL's `ZZ_pX CompMod` routine and the corresponding FLINT routine `fmpz_mod_poly_compose_mod`. These routines compute $g(h) \bmod f$ for polynomials $f, g, h \in \mathbb{Z}_p[X]$ using Brent and Kung's modular composition algorithm.

The modulus p was chosen to be a random k-bit prime. The polynomial f was chosen to be a random monic polynomial of degree n, and the polynomials g and g were chosen to be random polynomials of degree less than n.

$k/1024$	$n/1024$								
	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**

Figure 6: Composition modulo a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

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

Fig. 7 compares the relative performance of NTL's `ZZ_pX CanZass` factoring routine and the corresponding FLINT routine `fmpz_mod_poly_factor_kaltofen_shoup`. Both routines implement the same algorithm, and for the range of parameters that were benchmarked, both routines are the best 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.

$k/1024$	$n/1024$								
	1/4		1/2		1		2		4
1/4	**4.35**	**3.96**	**4.76**	**4.54**	**5.38**	**4.62**	**5.55**	**4.51**	**5.68**
1/2	**2.62**	**2.64**	**2.60**	**3.48**	**3.77**	**4.50**	**4.46**	**4.59**	**5.69**
1	1.96	1.99	**2.18**	**2.47**	**2.96**	**3.11**	**2.98**	**3.78**	**4.35**
2	1.47	1.58	1.66	1.75	**2.02**	**2.09**	**2.13**	**2.63**	**3.10**
4	1.65	1.65	1.71	1.81	1.99	1.86	**2.10**	**2.29**	**2.09**

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

8 Matrix multiplication over $\mathbb{Z}_p$

In 2016, NTL had its matrix multiplication over $\mathbb{Z}_p$ upgraded, so as to use a multi-modular approach that exploits the recent upgrade of its routines for matrix multiplication modulo single-precision numbers (see §16). Note that NTL's modular composition routines also use these faster matrix multiplication routines, which in turn speeds up NTL's polynomial factoring routine significantly.

Fig. 8 compares the relative performance of NTL's `mat_ZZ_p mul` to the corresponding FLINT routine `fmpz_mod_mat_mul`.

The modulus p was chosen to be a random k-bit, odd number, and the matrices multiplied were two random $n \times n$ matrices of $\mathbb{Z}_p$.

FLINT is also using a multi-modular algorithm. NTL is exploiting AVX instructions for the small prime matrix multiplications (see §16). Also, NTL's CRT computation exploits the fact that the results are ultimately computed modulo p, which speeds up the CRT computations a bit.

$k/1024$	$n/1024$						
	1/4		1/2		1		2
1/4	1.98	**2.27**	**2.17**	**2.13**	**2.18**	**2.12**	**2.16**
1/2	1.85	**2.10**	**2.05**	1.98	**2.00**	1.95	**2.03**
1	1.99	**2.24**	**2.22**	**2.12**	**2.13**	**2.03**	**2.06**
2	1.84	**2.17**	**2.11**	**2.06**	**2.08**	1.97	**2.03**

Figure 8: Multiplication of $n \times n$ matrices over $\mathbb{Z}_p$, $k = \#$bits in p

9 Single precision: Multiplication in $\mathbb{Z}_p[X]$

Fig. 9 compares the relative speed of NTL's `zz_pX mul` routine with FLINT's `nmod_poly_mul` routine. 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.

Note that in this in the following eight sections, we are working with "single precision" moduli p, i.e., moduli that fit into a single machine word. On 64-bit machines, NTL limits such moduli to 60 bits, while FLINT supports moduli up to the full 64 bits.

NTL is using a multi-modular FFT throughout, while FLINT is using Kronecker substitution throughout.

k	$n/1024$												
	1		2		4		8		16		32		64
5	0.54	0.62	0.66	0.68	0.73	0.75	0.78	0.73	0.90	0.97	1.19	1.07	1.26
10	0.74	0.77	0.85	0.87	0.92	1.08	1.20	1.24	1.39	1.42	1.58	1.59	1.61
15	0.92	1.05	1.16	1.18	1.28	1.35	1.49	1.57	1.68	**2.00**	**2.21**	**2.04**	**2.45**
20	0.54	0.57	0.62	0.64	0.68	0.79	0.87	0.94	1.00	1.13	1.11	1.13	1.11
25	0.63	0.72	0.77	0.82	0.90	0.96	1.05	1.10	1.25	1.40	1.47	1.38	1.51
30	0.87	0.88	0.96	0.97	1.09	1.22	1.39	1.40	1.66	1.51	1.68	1.53	1.68
35	0.96	1.03	1.13	1.19	1.28	1.43	1.49	1.58	1.77	1.84	1.80	1.92	1.84
40	1.10	1.15	1.28	1.29	1.44	1.62	1.78	1.94	**2.04**	1.98	**2.15**	**2.16**	**2.31**
45	1.22	1.31	1.46	1.49	1.72	1.86	1.98	**2.12**	**2.17**	**2.36**	**2.30**	**2.30**	**2.35**
50	0.93	1.00	1.07	1.12	1.24	1.38	1.51	1.52	1.54	1.55	1.53	1.56	1.59
55	1.00	1.08	1.22	1.30	1.40	1.48	1.60	1.77	1.82	1.95	1.95	**2.00**	**2.18**
60	1.20	1.26	1.39	1.49	1.69	1.78	1.89	1.94	**2.10**	1.97	**2.06**	**2.03**	**2.40**

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

10 Single precision: Multiplication in $\mathbb{Z}_p[X]/(f)$

Fig. 10 compares the relative performance of NTL's `zz_pX MulMod` routine with FLINT's corresponding routine. The NTL routine takes as input precomputations based on f, specifically, a `zz_pXModulus` object. The corresponding FLINT routine is `nmod_poly_mulmod_preinv`. 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 two multiplicands are random polynomial of degree less than n.

NTL is using a multi-modular FFT throughout, while FLINT is using Kronecker substitution throughout.

k	$n/1024$												
	1		2		4		8		16		32		64
5	0.77	0.80	0.93	1.00	1.16	1.14	1.36	1.38	1.60	1.58	1.79	1.76	1.94
10	1.13	1.18	1.38	1.41	1.63	1.74	1.98	1.99	**2.21**	**2.09**	**2.34**	**2.39**	**2.69**
15	1.50	1.63	1.89	1.97	**2.28**	**2.31**	**2.67**	**2.54**	**2.76**	**2.84**	**2.99**	**3.21**	**3.44**
20	0.93	0.98	1.16	1.19	1.32	1.42	1.54	1.60	1.61	1.69	1.74	1.82	1.89
25	1.18	1.24	1.43	1.50	1.69	1.73	1.92	1.83	**2.09**	**2.09**	**2.37**	**2.29**	**2.53**
30	1.52	1.55	1.79	1.80	**2.13**	**2.12**	**2.34**	**2.15**	**2.55**	**2.20**	**2.50**	**2.45**	**2.84**
35	1.79	1.80	**2.12**	**2.13**	**2.44**	**2.52**	**2.64**	**2.61**	**2.75**	**2.86**	**2.93**	**3.19**	**3.17**
40	**2.07**	**2.13**	**2.46**	**2.44**	**2.70**	**2.73**	**3.12**	**3.04**	**3.22**	**3.06**	**3.83**	**3.48**	**3.80**
45	**2.37**	**2.42**	**2.81**	**2.91**	**3.24**	**3.18**	**3.37**	**3.23**	**3.47**	**3.72**	**3.68**	**3.83**	**4.00**
50	1.80	1.75	**2.06**	**2.05**	**2.31**	**2.28**	**2.46**	**2.33**	**2.47**	**2.43**	**2.53**	**2.60**	**2.68**
55	1.99	**2.03**	**2.32**	**2.31**	**2.57**	**2.59**	**2.64**	**2.74**	**2.73**	**2.77**	**3.11**	**3.20**	**3.27**
60	**2.37**	**2.30**	**2.62**	**2.58**	**2.91**	**2.81**	**3.10**	**3.04**	**3.46**	**3.08**	**3.38**	**3.26**	**3.91**

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

11 Single precision: Squaring in $\mathbb{Z}_p[X]/(f)$

Fig. 11 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 corresponding FLINT routine is `nmod_poly_mulmod_preinv`. This routine internally checks if the multiplicands point to the same object, and optimizes accordingly. 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 multi-modular FFT throughout, while FLINT is using Kronecker substitution throughout.

k	$n/1024$												
	1		2		4		8		16		32		64
5	0.84	0.85	1.00	1.07	1.23	1.23	1.44	1.53	1.64	1.59	1.95	1.76	**2.06**
10	1.22	1.25	1.50	1.52	1.78	1.87	**2.15**	**2.13**	**2.36**	**2.26**	**2.49**	**2.52**	**2.91**
15	1.66	1.77	**2.06**	**2.12**	**2.48**	**2.50**	**2.97**	**2.76**	**2.99**	**2.98**	**3.14**	**3.50**	**3.54**
20	1.00	1.03	1.25	1.26	1.44	1.50	1.66	1.69	1.73	1.75	1.93	1.97	1.98
25	1.25	1.31	1.54	1.59	1.80	1.85	**2.06**	1.96	**2.24**	**2.23**	**2.56**	**2.46**	**2.71**
30	1.62	1.66	1.92	1.91	**2.29**	**2.24**	**2.52**	**2.30**	**2.70**	**2.36**	**2.67**	**2.60**	**2.99**
35	1.92	1.93	**2.28**	**2.26**	**2.62**	**2.67**	**2.83**	**2.81**	**2.88**	**3.00**	**3.10**	**3.38**	**3.38**
40	**2.21**	**2.27**	**2.65**	**2.58**	**2.88**	**2.88**	**3.34**	**3.22**	**3.37**	**3.19**	**3.88**	**3.64**	**4.16**
45	**2.56**	**2.57**	**2.96**	**3.09**	**3.47**	**3.39**	**3.59**	**3.35**	**3.41**	**3.88**	**3.87**	**4.01**	**4.24**
50	1.94	1.94	**2.21**	**2.19**	**2.48**	**2.33**	**2.63**	**2.41**	**2.59**	**2.54**	**2.73**	**2.73**	**2.90**
55	**2.15**	**2.17**	**2.48**	**2.46**	**2.75**	**2.75**	**2.84**	**2.83**	**2.84**	**2.82**	**3.27**	**3.39**	**3.47**
60	**2.46**	**2.47**	**2.93**	**2.75**	**3.12**	**2.96**	**3.34**	**3.17**	**3.47**	**3.21**	**3.58**	**3.43**	**4.07**

Figure 11: Single precision: Squaring in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

12 Single precision: Pre-conditioned multiplication in $\mathbb{Z}_p[X]/(f)$

As in §4, we consider the computation of $ab \bmod f$, where not only is f fixed for many operations, but so is b.

Fig. 12 compares the relative performance of NTL's `zz_pX` pre-conditioned `MulMod` routine with FLINT's corresponding routine. The NTL routine takes as input precomputations based on f and b, specifically, a `zz_pXModulus` object and a `zz_pXMultiplier` object. There is no directly comparable FLINT routine — the best choice is the same routine we used above: `nmod_poly_mulmod_preinv`. 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 two multiplicands are random polynomial of degree less than n.

NTL is using a multi-modular FFT throughout, while FLINT is using Kronecker substitution throughout.

13 Single precision: Computing GCDs in $\mathbb{Z}_p[X]$

Fig. 13 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 k. Both libraries use a fast "Half GCD" algorithm.

14 Single precision: Modular composition in $\mathbb{Z}_p[X]$

Fig. 14 compares the relative performance of NTL's `zz_pX CompMod` routine and the corresponding FLINT routine `nmod_poly_compose_mod`. These routines compute $g(h) \bmod f$ for polynomials $f, g, h \in \mathbb{Z}_p[X]$ using Brent and Kung's modular composition algorithm.

k	$n/1024$												
	1		2		4		8		16		32		64
5	1.44	1.41	1.72	1.73	**2.20**	**2.05**	**2.45**	**2.54**	**2.85**	**2.87**	**3.40**	**3.21**	**3.66**
10	1.99	**2.05**	**2.55**	**2.55**	**3.07**	**3.15**	**3.69**	**3.63**	**4.18**	**3.81**	**4.48**	**4.35**	**5.11**
15	**2.81**	**2.94**	**3.49**	**3.55**	**4.30**	**4.20**	**5.07**	**4.71**	**5.27**	**5.20**	**5.73**	**6.11**	**6.43**
20	1.75	1.76	**2.16**	**2.12**	**2.44**	**2.52**	**2.83**	**2.87**	**3.04**	**3.01**	**3.38**	**3.06**	**3.56**
25	**2.19**	**2.19**	**2.67**	**2.67**	**3.23**	**3.10**	**3.57**	**3.30**	**3.97**	**3.86**	**4.54**	**3.86**	**4.80**
30	**2.82**	**2.69**	**3.35**	**3.18**	**3.96**	**3.79**	**4.18**	**3.89**	**4.85**	**4.05**	**4.86**	**4.18**	**5.32**
35	**3.28**	**3.24**	**3.98**	**3.79**	**4.54**	**4.50**	**4.93**	**4.70**	**5.23**	**5.23**	**5.60**	**5.26**	**6.06**
40	**3.81**	**3.79**	**4.59**	**4.33**	**5.02**	**4.86**	**5.88**	**5.49**	**6.09**	**5.50**	**7.04**	**6.39**	**7.49**
45	**4.36**	**4.32**	**5.23**	**5.17**	**6.05**	**5.69**	**6.18**	**5.71**	**6.60**	**6.67**	**6.94**	**6.94**	**7.60**
50	**3.33**	**3.23**	**3.86**	**3.69**	**4.33**	**4.09**	**4.64**	**4.16**	**4.67**	**4.38**	**4.86**	**4.74**	**5.20**
55	**3.51**	**3.63**	**4.36**	**4.13**	**4.81**	**4.79**	**4.97**	**4.95**	**5.19**	**5.08**	**5.98**	**5.93**	**6.33**
60	**4.25**	**4.12**	**4.94**	**4.64**	**5.47**	**5.01**	**5.81**	**5.52**	**6.58**	**5.60**	**6.42**	**6.17**	**7.54**

Figure 12: Single precision: Pre-conditioned multiplication in $\mathbb{Z}_p[X]/(f)$: n = degree bound, k = #bits in p

k	$n/1024$												
	1		2		4		8		16		32		64
5	0.95	0.89	0.91	0.85	0.89	0.84	0.92	0.83	0.94	0.85	0.98	0.87	1.04
10	0.98	0.96	0.98	0.94	1.01	0.94	1.05	0.96	1.10	1.00	1.17	1.06	1.27
15	1.05	1.05	1.08	1.04	1.12	1.06	1.19	1.11	1.29	1.16	1.39	1.26	1.52
20	0.83	0.79	0.78	0.73	0.77	0.72	0.80	0.74	0.85	0.76	0.90	0.82	1.02
25	0.90	0.86	0.83	0.81	0.87	0.81	0.90	0.83	0.98	0.89	1.05	0.96	1.15
30	1.09	1.02	1.07	0.97	1.08	0.99	1.16	1.04	1.25	1.10	1.35	1.17	1.46
35	1.27	1.19	1.19	1.14	1.24	1.16	1.33	1.22	1.43	1.29	1.53	1.37	1.71
40	1.31	1.25	1.30	1.21	1.35	1.24	1.46	1.29	1.57	1.40	1.69	1.51	1.90
45	1.39	1.29	1.36	1.27	1.44	1.30	1.54	1.38	1.71	1.51	1.87	1.62	**2.05**
50	1.40	1.16	1.18	1.08	1.17	1.09	1.23	1.15	1.34	1.22	1.44	1.32	1.58
55	1.40	1.21	1.24	1.14	1.23	1.15	1.32	1.23	1.43	1.32	1.58	1.43	1.72
60	1.44	1.26	1.31	1.21	1.34	1.27	1.45	1.35	1.60	1.48	1.75	1.59	1.93

Figure 13: Single precision: Computing GCDs in $\mathbb{Z}_p[X]$: n = degree bound, k = #bits in p

The modulus p was chosen to be a random k-bit prime. The polynomial f was chosen to be a random monic polynomial of degree n, and the polynomials g and g were chosen to be random polynomials of degree less than n.

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**

Figure 14: Composition modulo a degree n polynomial in $\mathbb{Z}_p[X]$, k = #bits in p

15 Single precision: Factoring in $\mathbb{Z}_p[X]$

Fig. 15 compares the relative performance of NTL's `ZZ_pX CanZass` factoring routine and the corresponding FLINT routine `nmod_poly_factor_kaltofen_shoup`. Both routines implement the same algorithm, and for the range of parameters that were benchmarked, both routines are the best 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.

k	$n/1024$								
	1		2		4		8		16
5	1.27	1.15	1.44	1.50	1.81	**2.01**	**2.04**	**2.19**	**2.37**
10	1.50	1.87	1.77	**2.02**	**2.73**	**2.42**	**2.92**	**2.99**	**3.56**
15	**2.19**	**2.36**	**2.47**	**2.76**	**3.23**	**3.69**	**3.50**	**3.80**	**4.13**
30	**2.52**	**2.75**	**2.92**	**2.66**	**2.96**	**3.14**	**3.50**	**3.31**	**3.70**
60	**3.56**	**3.45**	**3.91**	**3.86**	**4.40**	**4.13**	**5.81**	**4.55**	**4.50**

Figure 15: Single precision: Factoring a degree n polynomial in $\mathbb{Z}_p[X]$, $k = \#$bits in p

16 Single precision: Matrix multiplication over $\mathbb{Z}_p$

In 2016, NTL has had its single-precision matrix arithmetic significantly upgraded to be more cache friendly and to take advantage of SIMD instructions on x86 machines. It has also been upgraded to exploit multiple threads; however, all experiments reported here are single threaded.

On modern Intel processors, NTL's implementation works (very roughly) as follows. For p up to 23-bits in length, floating point AVX instructions are used. For p up to 31-bits in length, ordinary 64-bit integer multiplication is used. For larger p, 128-bit integer multiplication is used.

Fig. 16 compares the relative performance of NTL's `mat_zz_p mul` routine and the corresponding FLINT routine `nmod_mat_mul`. Both routines use a subcubic Strassen recursion.

The modulus p was chosen to be a random k-bit odd number. The matrices were random $n \times n$ matrices.

Since v2.7.0, FLINT uses elements of the BLAS library to implement matrix arithmetic modulo small primes. Prior to this, NTL consistently performed as good as or much better than FLINT for such operations. Now FLINT has overtaken NTL's performance for some of these operations. For matrix multiplication, FLINT is as good as, and sometimes twice as good as, NTL. For other matrix operations, NTL still sometimes significantly outperforms FLINT.

17 Single precision: Matrix inversion over $\mathbb{Z}_p$

Fig. 17 compares the relative performance of NTL's `mat_zz_p inv` routine and the corresponding FLINT routine `nmod_mat_inv`. FLINT reduces to (subcubic) matrix multiplication, while NTL uses a direct (cubic) implementation.

The modulus p was chosen to be a random k-bit prime. The matrices were random $n \times n$ invertible matrices.

k	$n/1024$								
	1/2		1		2		4		8
5	0.83	0.70	0.67	0.65	0.63	0.62	0.63	0.65	0.68
10	1.14	1.05	1.01	1.04	1.05	1.04	1.19	1.10	1.27
15	1.15	1.04	1.01	1.04	1.05	1.04	1.19	1.10	1.28
20	1.15	1.06	1.01	1.06	1.04	1.03	1.18	1.10	1.28
25	1.01	0.83	0.84	0.78	0.81	0.77	0.90	0.80	0.95
30	0.72	0.59	0.80	0.72	0.76	0.71	0.85	0.75	0.91
35	0.66	0.53	0.55	**0.50**	0.53	**0.49**	0.59	0.52	0.63
40	0.65	0.53	0.57	**0.50**	0.67	0.61	0.75	0.65	0.79
45	0.87	0.69	0.72	0.66	0.64	0.62	0.75	0.65	0.80
50	0.86	0.69	0.72	0.64	0.81	0.75	0.90	0.78	0.95
55	1.01	0.83	0.83	0.76	0.81	0.74	0.90	0.77	0.95
60	0.99	0.81	0.85	0.75	0.80	0.86	1.03	0.90	1.09

Figure 16: Single precision: Multiplication of $n \times n$ matrices over $\mathbb{Z}_p$, $k = \#\text{bits in } p$

k	$n/1024$								
	1/2		1		2		4		8
5	**5.07**	**4.48**	**3.94**	**3.40**	**2.88**	**2.07**	1.86	1.51	1.28
10	**6.58**	**6.24**	**4.97**	**4.36**	**3.66**	**2.68**	**2.37**	**2.05**	1.83
15	**7.53**	**7.00**	**5.52**	**4.83**	**3.95**	**2.84**	**2.47**	**2.14**	1.90
20	**7.51**	**7.05**	**5.50**	**4.75**	**3.99**	**2.85**	**2.55**	**2.13**	1.96
25	**2.22**	**2.06**	**2.01**	1.85	1.66	1.47	1.27	1.15	1.06
30	1.84	1.71	1.73	1.58	1.33	1.26	1.07	1.00	0.91
35	1.69	1.62	1.56	1.56	1.18	1.07	0.89	0.78	0.69
40	1.68	1.65	1.49	1.45	1.17	1.06	0.87	0.87	0.74
45	1.69	1.52	1.55	1.49	1.28	1.14	1.00	0.88	0.81
50	1.75	1.59	1.50	1.48	1.26	1.14	0.98	0.91	0.82
55	1.70	1.60	1.51	1.48	1.34	1.18	1.10	1.00	0.93
60	1.67	1.58	1.49	1.46	1.33	1.21	1.07	0.98	0.91

Figure 17: Single precision: Inversion of $n \times n$ matrices over $\mathbb{Z}_p$, $k = \#\text{bits in } p$

18 Single precision: Nullspace computation over $\mathbb{Z}_p$

Fig. 18 compares the relative performance of NTL's `mat_zz_p kernel` routine and the corresponding FLINT routine `nmod_mat_nullspace`. FLINT reduces to matrix multiplication, while NTL uses a direct implementation.

The modulus p was chosen to be a random k-bit prime. The matrices were (roughly) random $n \times n$ matrices of rank $n/2$.

19 Multiplication in $\mathbb{Z}[X]$

Fig. 19 compares the relative speed of NTL's `ZZX mul` routine with FLINT's `fmpz_poly_mul` routine. The polynomials were generated at random to have degree less than n, and coefficients in the range $0, \ldots, 2^k - 1$.

In the upper right region, NTL is using a multi-modular FFT, while FLINT is using Kronecker substitution. In the lower left region, both are using Schönhage-Strassen.

k	$n/1024$								
	$1/2$		1		2		4		8
5	**3.90**	**3.36**	**3.24**	**2.86**	**2.39**	1.88	**2.01**	1.52	1.33
10	**4.50**	**4.26**	**3.80**	**3.45**	**2.90**	**2.39**	**2.39**	**2.00**	1.64
15	**5.44**	**5.24**	**4.20**	**3.75**	**3.18**	**2.56**	**2.52**	**2.04**	1.75
20	**5.41**	**5.18**	**4.27**	**3.77**	**3.10**	**2.55**	**2.59**	**2.02**	1.71
25	1.81	1.66	1.58	1.49	1.32	1.19	1.13	0.97	0.85
30	1.60	1.42	1.41	1.26	1.11	1.03	0.94	0.84	0.72
35	1.51	1.34	1.27	1.19	1.00	0.92	0.78	0.69	0.56
40	1.48	1.36	1.27	1.17	0.99	0.90	0.78	0.72	0.60
45	1.50	1.33	1.27	1.19	1.04	0.94	0.85	0.76	0.64
50	1.51	1.38	1.26	1.18	1.04	0.93	0.85	0.76	0.64
55	1.47	1.35	1.27	1.18	1.07	0.97	0.92	0.82	0.72
60	1.48	1.40	1.28	1.19	1.10	0.98	0.94	0.83	0.73

Figure 18: Single precision: Nullspace computation of $n \times n$ matrices over $\mathbb{Z}_p$, $k = \#$bits in p

$k/1024$	$n/1024$												
	$1/4$		$1/2$		1		2		4		8		16
$1/4$	1.63	1.50	1.69	1.57	1.55	1.56	1.64	1.59	1.67	1.67	1.67	1.53	1.68
$1/2$	0.77	0.87	0.88	1.02	1.03	1.31	1.28	1.49	1.54	1.41	1.65	1.42	**2.13**
1	0.70	0.64	0.62	0.73	0.73	0.89	0.87	1.20	1.17	1.29	1.93	1.51	1.96
2	0.82	0.79	0.77	0.74	0.72	0.77	0.78	0.73	0.72	1.05	1.02	1.18	1.49
4	1.30	1.30	1.28	1.24	1.21	1.06	1.03	1.12	1.06	1.08	1.01	0.99	0.96
8	1.08	1.06	1.03	1.02	1.00	0.97	0.94	0.95	0.90	1.01	0.96	0.93	0.90
16	0.95	0.96	0.95	0.94	0.93	0.92	0.90	0.88	0.86	0.82	0.79	0.89	0.86

Figure 19: Multiplication in $\mathbb{Z}[X]$: $n =$ degree bound, $k = \#$bits in each coefficient

20 Concluding remarks

We attempt to draw some conclusions from these benchmarks.

- NTL could perhaps be improved by improving its Schönhage-Strassen implementation, by using Kronecker substitution in place of multi-modular FFT *in some parameter ranges*, and by fine tuning some of its algorithm crossover points.
- FLINT could perhaps be improved by using a multi-modular FFT in place of Kronecker substitution *in some parameter ranges*.