Q-adic Transform revisited

Jean-Guillaume Dumas

Q-adic Transform revisited

Jean-Guillaume Dumas *

October 26, 2007

Abstract

We present an algorithm to perform fast modular polynomial multiplication. The idea is to convert the X-adic representation of modular polynomials, with X an indeterminate, to a q-adic representation where q is a prime power larger than the field characteristic. With some control on the different involved sizes it is then possible to perform some of the q-adic arithmetic directly with machine integers or floating points. Depending also on the number of performed numerical operations one can then convert back to the q-adic or X-adic representation and eventually mod out high residues. In this note we present a new version of both conversions: more tabulations and a way to reduce the number of divisions involved in the process are presented. The polynomial multiplication is then applied to arithmetic in small finnite field extensions.

1 Introduction

The FFLAS/FFPACK project has demonstrated the need of a wrapping of cache-aware routines for efficient small finite field linear algebra [2, 3].

A conversion between a modular representation of prime fields and e.g. floating points used exactly is natural. It uses the homomorphism to the integers. Now for extension fields (isomorphic to polynomials over a prime field) such a conversion is not direct. In [2] we propose to transform the polynomials into a q-adic representation where q is a prime power larger than the field characteristic. We call this transformation DQT for Discrete Q-adic Transform. With some care, in particular on the size of q, it is possible to map the operations in the extension field into the floating point arithmetic realization of this q-adic representation and convert back using an inverse DQT.

In this note we propose some implantation improvements: we propose to use a tabulated Zech logarithm for the DQT and give a trick to reduce the number of machine divisions involved in the inverse. This thus gives rise to an improved DQT which we thus call FQT (Fast Q-adic Transform). Therefore we recall in section 2 the previous conversion algorithm. We then present our new

*Laboratoire J. Kuntzmann, 51, rue des Mathématiques. Université de Grenoble. UMR CNRS 5224, BP 53X, F38041 Grenoble, France, Jean-Guillaume.Dumas@imag.fr

reduction in section 3. We then show in section 4 how a time-memory trade-off can make this reduction very fast. This can then be applied on modular polynomial multiplication with small prime fields in section 5 as well as for small extension field arithmetic and fast matrix multiplication in section 6.

2 Q-adic representation of polynomials

We follow here the presentation of [2] of the idea of [5]: polynomial arithmetic is performed a $q-$adic way, with q a sufficiently big prime or power of a single prime.

Suppose that $a = \sum_{i=0}^{k-1} \alpha_i X^i$ and $b = \sum_{i=0}^{k-1} \beta_i X^i$ are two polynomials in $\mathbb{Z}/\mathrm{p}\mathbb{Z}[X]$. One can perform the polynomial multiplication ab via $q-$adic numbers. Indeed, by setting $\tilde{a} = \sum_{i=0}^{k-1} \alpha_i q^i$ and $\tilde{b} = \sum_{i=0}^{k-1} \beta_i q^i$, the product is computed in the following manner (we suppose that $\alpha_i = \beta_i = 0$ for $i > k-1$):

$$\widetilde{ab} = \sum_{j=0}^{2k-2} \left(\sum_{i=0}^{j} \alpha_i \beta_{j-i} \right) q^j \tag{1}$$

Now if q is large enough, the coefficient of q^i will not exceed q. In this case, it is possible to evaluate a and b as machine numbers (e.g. floating point or machine integers), compute the product of these evaluations, and convert back to polynomials by radix computations (see e.g. [4, Algorithm 9.14]). There just remains then to perform modulo p reductions on every coefficient. We call DQT the evaluation of polynomials modulo p at q and DQT inverse the radix conversion of a q-adic development followed by a modular reduction. Depending on the size of q, the results can still remain exact:

Algorithm 1 Polynomial multiplication by DQT

Require: Two polynomials v_1 and v_2 in $\mathbb{Z}/\mathrm{p}\mathbb{Z}[X]$ of degree less than k.
Require: a prime power q.
Ensure: $R \in \mathbb{Z}/\mathrm{p}\mathbb{Z}[X]$, with $R = v_1.v_2$.

Polynomial to $q-$adic conversion

1: Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point vectors of the evaluations at q of the elements of v_1 and v_2. {Using e.g. Horner's formula}

One computation

2: Compute $\tilde{r} = \widetilde{v_1}\widetilde{v_2}$

Building the solution

3: $\tilde{r} = \sum_{i=0}^{2k-2} \widetilde{\mu_i} q^i$. {Using radix conversion, see e.g. [4, Algorithm 9.14]}
4: For each i, set $\mu_i = \widetilde{\mu_i} \mod p$
5: set $R = \sum_{i=0}^{2k-2} \mu_i X^i$

Theorem 1. *[2] Let m be the number of available mantissa bits within the machine numbers and n be the number of machine products $v_1.v_2$ accumulated, e.g. in a dot product, before the re-conversion. If*

$$q > n_q k(p-1)^2 \text{ and } (2k-1)\log[2](q) < m, \tag{2}$$

then Algorithm 1 is correct.

Note that the prime power q can be chosen to be a power of 2. Then the Horner like evaluation of the polynomials at q (line 1 of algorithm 1) is just a left shift. One can then compute this shift with exponent manipulations in floating point arithmetic and use then e.g. native C++ << operator as soon as values are within the 32 bits range, or use the native C++ << on 64 bits when available.

In the following we will thus always consider that q is a power of two.

It is shown on [2, Figures 5 & 6] that this wrapping is already a pretty good way to obtain high speed linear algebra over some small extension fields. Indeed we were able to reach high peak performance, quite close to those obtained with prime fields, namely 420 Mop/s on a PIII, 735 MHz, and more than 500 Mop/s on a 64-bit DEC alpha 500 MHz. This is roughly 20 percent below the pure floating point performance and 15 percent below the prime field implementation.

3 REDQ: modular reduction in the DQT domain

The first improvement we propose to the DQT is to replace the costly modular reduction of the polynomial coefficients by a single division by p (or, better, by a multiplication by its inverse) followed by several shifts. The idea is as follows (note that when q is a power of 2 division by q^i and flooring is just a right shift):

Algorithm 2 REDQ

Require: a prime p and a prime power q satisfying the conditions (2).
Require: $\tilde{r} = \sum_{i=0}^{d} \widetilde{\mu_i} q^i \in \mathbb{Z}$.
Ensure: $\rho \in \mathbb{Z}$, with $\rho = \sum_{i=0}^{d} \mu_i q^i$ where $\mu_i = \widetilde{\mu_i} \mod p$.

1: $rop = \left\lfloor \frac{\tilde{r}}{p} \right\rfloor$;
2: **for** $i = 0$ to d **do**
3: $\quad u_i = \left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor - p \left\lfloor \frac{rop}{q^i} \right\rfloor$;
4: **end for**
5: $\mu_d = u_d$
6: **for** $i = 0$ to $d-1$ **do**
7: $\quad \mu_i = u_i - q u_{i+1} \mod p$;
8: **end for**
9: Return $\rho = \sum_{i=0}^{d} \mu_i q^i$;

Theorem 2. *Algorithm REDQ is correct.*

We first need the following lemma:

Lemma 1. *For $r \in \mathbb{N}$ and $a,\ b \in \mathbb{N}^*$,*

$$\left\lfloor \frac{\lfloor \frac{r}{b} \rfloor}{a} \right\rfloor = \left\lfloor \frac{\lfloor \frac{r}{a} \rfloor}{b} \right\rfloor$$

Proof. We proceed by splitting the possible values of r into intervals $kab \leq r < (k+1)ab$. Then $kb \leq \frac{r}{a} < (k+1)b$ and since kb is an integer we also have that $kb \leq \lfloor \frac{r}{a} \rfloor < (k+1)b$. Thus $k \leq \frac{\lfloor \frac{r}{a} \rfloor}{b} < k+1$ and $\left\lfloor \frac{\lfloor \frac{r}{a} \rfloor}{b} \right\rfloor = k$. Obviously the same is true for the left hand side which proves the lemma. □

Proof *of theorem 2.* First we need to prove that $0 \leq u_i < p$. By definition of the truncation, we have $\frac{\tilde{r}}{q^i} - 1 < \left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor \leq \frac{\tilde{r}}{q^i}$ and $\frac{\tilde{r}}{pq^i} - 1 - \frac{1}{q^i} < \left\lfloor \frac{rop}{q^i} \right\rfloor \leq \frac{\tilde{r}}{pq^i}$. Thus $-1 < u_i < p + \frac{p}{q^i}$, which is $0 \leq u_i \leq p$ since u_i is an integer. We now consider the possible case $u_i = p$ and show that it does not happen. $u_i = p$ means that $\left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor = p(1 + \left\lfloor \frac{rop}{q^i} \right\rfloor) = pg$. This means that $pgq^i \leq r < pgq^i + q^i$. So that in turns $gq^i \leq rop \leq \frac{\tilde{r}}{p} < gq^i + \frac{q^i}{p}$. Thus $g \leq \frac{rop}{q^i} < g + \frac{1}{p}$ so that $\left\lfloor \frac{rop}{q^i} \right\rfloor = g$. But then from the definition of g we have that $g = g - 1$ which is absurd. Therefore $0 \leq u_i \leq p-1$.

Second we show that $u_i = \sum_{j=i}^{d} \mu_j q^{j-i} \mod p$. Well $u_i = \left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor - p \left\lfloor \frac{\lfloor \frac{\tilde{r}}{p} \rfloor}{q^i} \right\rfloor$ and thus lemma 1 gives that $u_i = \left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor - p \left\lfloor \frac{\lfloor \frac{\tilde{r}}{q^i} \rfloor}{p} \right\rfloor$. The latter is $u_i = \left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor \mod p$. Now, since $\tilde{r} = \sum_{j=0}^{2k-2} \widetilde{\mu_j} q^j$, we have that $\left\lfloor \frac{\tilde{r}}{q^i} \right\rfloor = \sum_{j=i}^{2k-2} \widetilde{\mu_j} q^{j-i}$. Therefore, as $\mu_j = \widetilde{\mu_j} \mod p$, the equality is proven. □

In the algorithm, the computation of *rop* has to be a division in exact arithmetic. The following lemma gives bounds for which this division can be performed by a floating point multiplication with the precomputed inverse of p, as is done e.g. in NTL[1].

Lemma 2. *For a prime p and $r,\ t \in \mathbb{N}^*$ with $r < tp$, then*

$$\left\lfloor \frac{r}{p} \right\rfloor = \left\lfloor r\left(\frac{1}{p} + \epsilon\right) \right\rfloor \quad \textit{as long as} \quad 0 \leq \epsilon < \frac{1}{p^2(t + 1 - 1/p)}.$$

Proof. Consider $up \leq r < up + i$ with u, i positive integers and $u < p$. Then $\left\lfloor \frac{r}{p} \right\rfloor = u$ and $r(\frac{1}{p} + \epsilon) = u + \frac{i}{p} + (up + i)\epsilon$. Then $\frac{i}{p} + (up + i)\epsilon < 1$ if $\epsilon < \frac{p-i}{p(up+i)}$. The right hand side is decreasing in i and is therefore minimal at $i = p - 1$ for which it is $\frac{1}{p^2(u+1-1/p)}$. □

[1] `www.shoup.net/ntl`

Therefore, it is possible to perform the division by a multiplication by the precomputed inverse of the prime number. Since entries are already in floating point format this is a potential significant speed-up. Remark that for ϵ to be positive, the precomputed inverse has just to be rounded towards $+\infty$. Then since the division is correctly rounded, $\epsilon < 2^{-m}$, where m is the available floating point mantissa, and the non reduced r can be as large as

$$r < \frac{2^m}{p} + 1$$

while remaining correct.

For instance, take the polynomial $R = 1234X^3 + 5678X^2 + 9123X + 4567$, the prime $p = 23$ and use $q = 10^6$ to see what is happening. The trick is that by dividing only once by 23 all the coefficients of R are divided at once: $rop = \lfloor 1234005678009123004567/23 \rfloor = 53652420783005348024$. Then $rop \times 23 = 1234005678009123004552$ which gives $u_0 = 15$. Then we shift to get 1234005678009123 and $53652420783005 \times 23 = 1234005678009115$ which gives $u_1 = 8$. We shift and multiply twice to get $u_2 = 18$ and $u_3 = \mu_3 = 15$. We have then to compute $\mu = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -q & 1 & 0 & 0 \\ 0 & -q & 1 & 0 \\ 0 & 0 & -q & 1 \end{bmatrix} u \mod p$ to get the final results. As is there is no benefit to use this method when compared to direct remaindering by p. The trick is that this last step from the u_i to the μ_i can be tabulated since u is of much smaller size than R.

4 Time-Memory trade-off in REDQ

Indeed, there is a bijection between the u_i and the μ_i since

$$Q_d = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ -q & \ddots & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -q & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ q & \ddots & \ddots & & \vdots \\ q^2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ q^d & \dots & q^2 & q & 1 \end{bmatrix}^{-1}$$

If this operation is fully tabulated, it requires a table of size at least p^{d+1}. But, due to the nature of Q_d, we have the following relations:

Therefore, it is very easy to tabulate with a table of size p^k only and perform $\frac{d-1}{k-1}$ table accesses. The time memory trade-off, after the computation of the u_i, which requires 1 div + $(d+1)$ mul + $2d$ shifts, is thus as shown on table 1.

In practice, indexing by a t-uple of integers mod p is made by evaluating at p, as $\sum u_i p^i$. If a few more memory space is available, one can also directly index in the binary format using $\sum u_i \left(2^{\lceil \log_2(p) \rceil}\right)^i$. On the one hand all the multiplications by p are replaced by binary shifts. On the other hand, this makes the table grow a little bit, from p^k to $2^{\lceil \log_2(p) \rceil k}$.

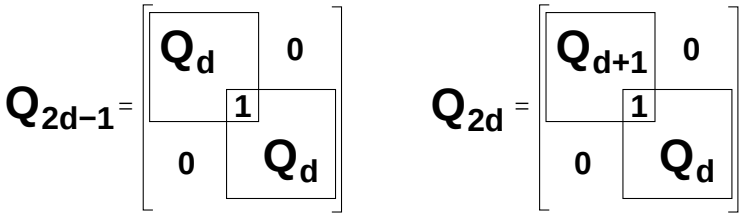

Figure 1: Recurring relations on the Q_d matrices.

Memory	0	p^2	p^k	p^{d+1}
Extra time	d (mul,add,mod)	d accesses	$\left\lceil \frac{d}{k-1} \right\rceil$ accesses	1 access

Table 1: Time-Memory trade-off in REDQ

5 Comparison with delayed reduction for polynomial multiplication

Another approach to perform modular polynomial multiplication is to use delayed reductions e.g. as in [1]: The idea is to accumulate products of the form $\sum_i a_i b_{k-i}$, without reductions, while the sum does not overflow. Thus, if we use for instance a centered representation modulo p (integers from $\frac{1-p}{2}$ to $\frac{p-1}{2}$) to accumulate n products as long as

$$n_d(p-1)^2 < 2^{m+1} \tag{3}$$

The modular reduction was made by different ways, we just call it REDC here. It is at most equivalent to 1 division.

Now the idea of the FQT (Fast Q-adic Transform) is to represent modular polynomials of the form $P = \sum_{i=0}^{N} a_i X^i$ by $P = \sum_{i=0}^{N/k} P_i X^i$ where the P_i are degree k polynomials stored in a single integer in the q-adic way. Therefore, a product PQ has the form $\sum \left(\sum P_i Q_{t-i}\right) X^t$. There, each multiplication $P_i Q_{t-i}$ is made by algorithm 1 on a single machine integer. The reduction is made by a tabulated REDQ and can also be delayed now as long as conditions (2) is garanteed.

For the complexity, table 2 gives the respective complexities of both strategies.

Complexity	Multiplications	Reductions
Delayed	N^2	$\frac{N^2}{n_d}$ REDC
FQT	$\left(\frac{N}{k}\right)^2$	$\frac{1}{n_q}\left(\frac{N}{k}\right)^2$ REDQ

Table 2: Modular polynomial multiplication complexities.

For instance, with $p = 3$, $N = 100$, choosing a double floating point representation and a degree 4 DQT (i.e. $k = 4$), the DQT boils down to 993 multi-

plications and 53 divisions. For the same parameters, the classical polynomial multiplication algorithm requires 10^4 multiplications and only 1 remaindering, which is roughly 10 times more operations as shown on figure 2.

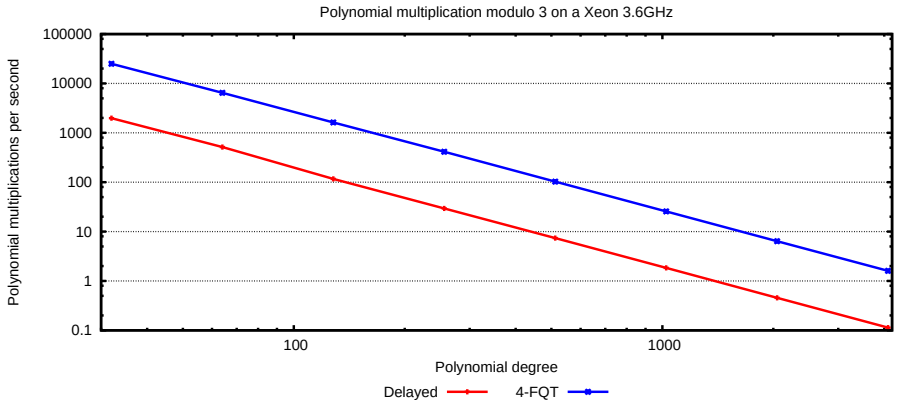

Figure 2: Polynomial multiplications modulo 3 per second on a Xeon 3.6 GHz

Even by switching to a larger mantissa, say e.g. 128 bits, so that the DQT multiplications are 4 times costlier, this can still be useful: take $p = 1009$ and choose $k = 3$, gives 1445 multiplications over 128 bits and 67 divisions. This should still be lower than the delayed.

Still this strategy is useful for small degrees and small primes. This can nonetheless improves drastically the base cases of faster recursive algorithms such as Karatsuba, Toom-Cook, etc. In particular, it improves the speed of small finite field extension's arithmetic as shown next.

6 Application to small finite field extensions

We use here the isomorphism between finite field to see any finite field extension as the set of polynomials modulo a prime p and an irreducible polynomial $\mathcal{P}$ of degree k. Clearly we can convert any finite field element to its q-adic expansion ; perform the FQT between two elements and then reduce the obtained polynomial modulo $\mathcal{P}$. In this paper we propose to improve all the conversion steps of [2, algorithm 4.1] in order to approach the performance of the prime field wrapping also for several extension fields:

1. Replace the Horner evaluation of the polynomials, to form the q-adic expansion, by a single table lookup recovering directly the floating point representation.

2. Replace the radix conversion and the costly modular reductions of each polynomial coefficient, by a single half REDQ operation.

3. Replace the polynomial division by two table lookups and a single field operation.

Indeed, suppose the internal representation of the extension field is already by Zech logarithms and uses conversion tables from polynomial to index representations. See e.g. [1] for more details. Then we choose a time-memory trade-off for the REDQ operation of the same order of magnitude, that is to say p^k. The overall memory required by these new tables only doubles and the REDQ requires only 2 accesses. Moreover, in the small extension, the polynomial multiplication must also be reduced by an irreducible polynomial, $\mathcal{P}$. We show next that this reduction can be precomputed in the REDQ table lookup and is therefore almost free.

More precisely, we propose the following algorithm 3.

Algorithm 3 Fast Dot product over Galois fields via FQT and FQT inverse

Require: a field $\mathtt{GF}(p^k)$ with elements represented as exponents of a generator of the field.
Require: Two elements v_1 and v_2 of $\mathtt{GF}(p^k)$.
Require: a prime power q.
Ensure: $R \in \mathtt{GF}(p^k)$, with $R = v_1.v_2$.

Tabulated $q-$adic conversion

{Use conversion tables from exponent to floating point evaluation}

1: Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point vectors of the evaluations at q of the elements of v_1 and v_2.

The floating point computation

2: Compute $\tilde{r} = \widetilde{v_1}\widetilde{v_2}$;

Computing a radix decomposition

3: $r = \lfloor \tilde{r} \rfloor$; $\quad \{r = \tilde{r}$ but we might need a conversion to an integral type$\}$
4: $rop = \left\lfloor \frac{\tilde{r}}{p} \right\rfloor$;
5: **for** $i = 0$ to $2k-2$ **do**
6: $\quad u_i = \left\lfloor \frac{r}{q^i} \right\rfloor - p \left\lfloor \frac{rop}{q^i} \right\rfloor$;
7: **end for**

Tabulated radix conversion to exponents of the generator

$\{\mu_i$ is such that $\mu_i = \widetilde{\mu_i} \mod p$ for $\tilde{r} = \sum_{i=0}^{2k-2} \widetilde{\mu_i} q^i\}$
8: Set $L = representation(\sum_{i=0}^{k-2} \mu_i X^i)$.
9: Set $H = representation(X^{k-1} \times \sum_{i=k-1}^{2k-2} \mu_i X^{i-k+1})$.

Reduction in the field

10: Return $R = H + L \in \mathtt{GF}(p^k)$;

Theorem 2 proves that in algorithm 3, u_i satisfies $u_i = \sum_{j=i}^{2k-2} \mu_j q^{j-i} \mod p$. Therefore the representations of $\sum \mu_i X^i$ in the field can be precomputed and

stored in a table where the indexing will be made by $(u_0, \ldots, u_{k-1})$ and $(u_{k-1}, \ldots, u_{2k-2})$ and not by the μ_i's. Note also that the representation of X^{k-1} can be just $k-1$ if the irreducible polynomial used to build $\texttt{GF}(p^k)$ is primitive and X has been chosen as the generator.

Theorem 3. *Algorithm 3 is correct.*

Proof. Theorem 2 proves that $u_i = \sum_{j=i}^{2k-2} \mu_j q^{j-i} \mod p$. There remains thus to prove that it is possible to compute L and H from the u_i. From the equality above, we see that $\mu_{2k-2} = u_{2k-2}$ and $\mu_i = u_i - qu_{i+1} \mod p$, for $i = 0..(2k-3)$. Therefore a p^k elements precomputed table, indexed by $(u_0, \ldots, u_{k-1})$, can provide the representation of

$$L = \sum_{i=0}^{k-2} (u_i - qu_{i+1} \mod p) X^i.$$

Another table with p^k elements, indexed by $(u_{k-1}, \ldots, u_{2k-2})$, can provide the representation of

$$H = u_{2k-2} X^{2k-2} + \sum_{i=k-1}^{2k-3} (u_i - qu_{i+1} \mod p) X^i.$$

Finally $R = X^{k-1} \times \sum_{i=k-1}^{2k-2} \mu_i X^{i-k+1} + \sum_{i=0}^{k-2} \mu_i X^i$ needs to be reduced modulo the irreducible polynomial used to build the field. But, if we are given the representations of H and L in the field, R is then equal to their addition inside the field, directly using the internal representations. □

Table 3 recalls the respective complexities of the two presented algorithms

Conversions	Memory	Shift	Add	Axpy	Div	Table	Red
Algorithm 1	$3p^k$	$4k-2$	$4k-4$	0	$2k-1$	0	$5k$
Algorithm 3	$6p^k$	$4k-2$	0	$4k-3$	0	3	4
Algorithm 3	$4p^k + 2^{k\lceil \log_2 p \rceil + 1}$	$4k-2$	$2k-1$	$2k-1$	0	3	4

Table 3: Complexity of the back and forth conversion between extension field and floating point numbers

Figure 3 shows only the speed of the conversion after the floating point operations. The log scales prove that for q ranging from 2^1 to 2^{26} (on a 32 bit Pentium IV) our new implantation is two to three times faster than the previous one.

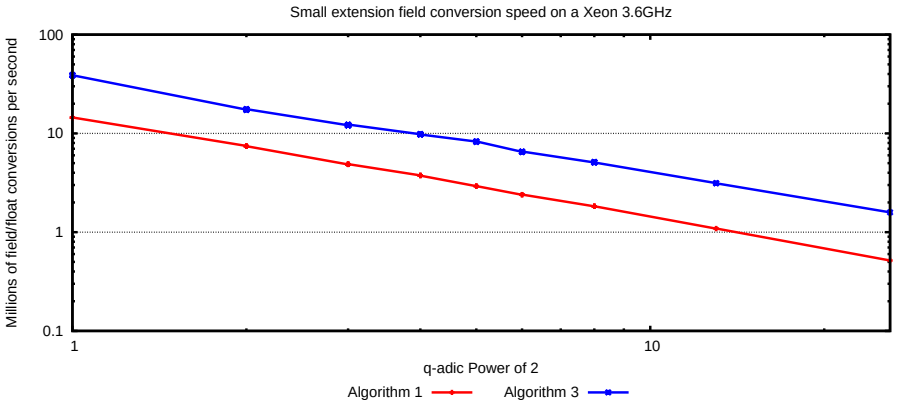

Figure 3: Small extension field conversion speed on a Xeon 3.6GHz

Furthermore, these improvements e.g. allow the extension field routines to reach the speed of 7500 millions of GF(9) operations per second (on a XEON, 3.6 GHz) as shown on figure 4.

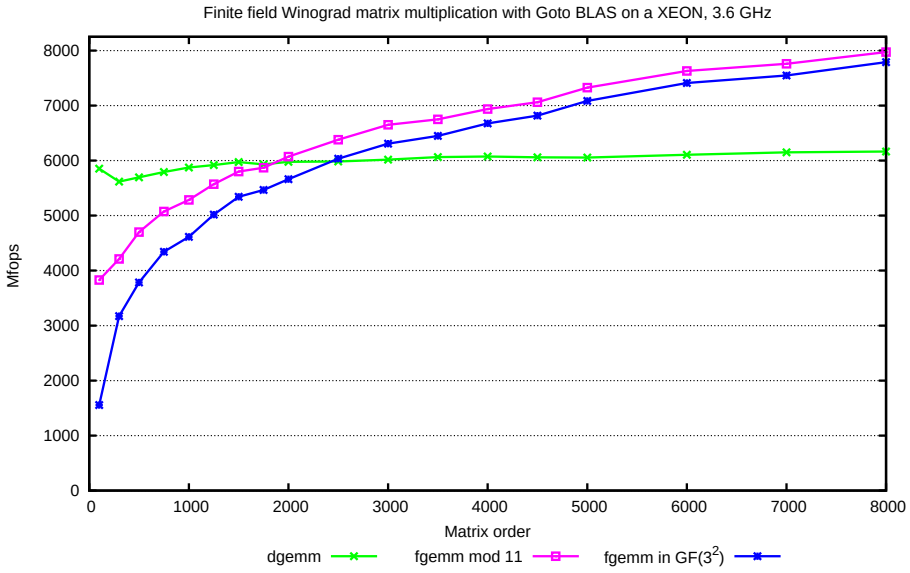

Figure 4: Finite field Winograd matrix multiplication with Goto BLAS on a XEON, 3.6 GHz

This represent a reduction from the 15 percent overhead of the previous implementation to less than 4 percent now, when compared to GF(11).

References

[1] Jean-Guillaume Dumas. Efficient dot product over finite fields. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Proceedings of the seventh International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 139–154. Technische Universität München, Germany, July 2004.

[2] Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet. Finite field linear algebra subroutines. In Teo Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, July 2002.

[3] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: Finite field linear algebra package. In Jaime Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*, pages 119–126. ACM Press, New York, July 2004.

[4] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[5] B. David Saunders. Personal communication, 2001.