Implementing fast carryless multiplication

JORIS VAN DER HOEVEN[a], ROBIN LARRIEU[b], GRÉGOIRE LECERF[c]

Laboratoire d'informatique de l'École polytechnique
LIX, UMR 7161 CNRS
Campus de l'École polytechnique
1, rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing, CS35003
91120 Palaiseau, France

a. Email: `vdhoeven@lix.polytechnique.fr`
b. Email: `larrieu@lix.polytechnique.fr`
c. Email: `lecerf@lix.polytechnique.fr`

Preliminary version of October 24, 2017

The efficient multiplication of polynomials over the finite field $\mathbb{F}_2$ is a fundamental problem in computer science with several applications to geometric error correcting codes and algebraic crypto-systems. In this paper we report on a new algorithm that leads to a practical speed-up of about two over previously available implementations. Our current implementation assumes a modern AVX2 and CLMUL enabled processor.

1. INTRODUCTION

Modern algorithms for fast polynomial multiplication are generally based on *evaluation-interpolation* strategies and more particularly on the *discrete Fourier transform* (DFT). Taking coefficients in the finite field $\mathbb{F}_2$ with two elements, the problem of multiplying in $\mathbb{F}_2[x]$ is also known as *carryless integer multiplication* (assuming binary notation). The aim of this paper is to present a practically efficient solution for large degrees.

One major obstruction to evaluation-interpolation strategies over small finite fields is the potential lack of evaluation points. The customary remedy is to work in suitable extension fields. Remains the question of how to reduce the incurred overhead as much as possible.

More specifically, it was shown in [7] that multiplication in $\mathbb{F}_2[x]$ can be done efficiently by reducing it to polynomial multiplication over the *Babylonian field* $\mathbb{F}_{2^{60}}$. Part of this reduction relied on Kronecker segmentation, which involves an overhead of a factor two. In this paper, we present a variant of a new algorithm from [11] that removes this overhead almost entirely. We also report on our MATHEMAGIX implementation that is roughly twice as efficient as before.

1.1. Related work

For a long time, the best known algorithm for carryless integer multiplication was Schönhage's triadic variant [16] of Schönhage–Strassen's algorithm [17] for integer multiplication: it achieves a complexity $O(n \log n \log \log n)$ for the multiplication of two polynomials of degree n. Recently [8], Harvey, van der Hoeven and Lecerf proved the sharper bound $O(n \log n \, 8^{\log^* n})$, but also showed that several of the new ideas could be used for faster practical implementations [7].

More specifically, they showed how to reduce multiplication in $\mathbb{F}_2[x]$ to DFTs over $\mathbb{F}_{2^{60}}$, which can be computed efficiently due to the existence of many small prime divisors of $2^{60}-1$. Their reduction relies on *Kronecker segmentation*: given two input polynomials $A(x) = \sum_{0 \leqslant i < n} a_i x^i$ and $B(x) = \sum_{0 \leqslant i < n} a_i x^i$ in $\mathbb{F}_2[x]$, one cuts them into chunks of 30 bits and forms $\tilde{A}(y,z) = \sum_{i=0}^{m-1} \sum_{j=0}^{29} a_{30i+j} z^j y^i$ and $\tilde{B}(y,z) = \sum_{i=0}^{m-1} \sum_{j=0}^{29} b_{30i+j} z^j y^i$, where $m = \lceil n/30 \rceil$ (the least integer $\geqslant n/30$). Hence $A(x) = \tilde{A}(x^{30}, x)$, $B(x) = \tilde{B}(x^{30}, x)$, and the product $C = A B$ satisfies $C(x) = \tilde{C}(x^{30}, x)$, where $\tilde{C} = \tilde{A}\tilde{B}$. Now $\tilde{A}$ and $\tilde{B}$ are multiplied in $\mathbb{F}_{2^{60}}[x]$ by reinterpreting z as the generator of $\mathbb{F}_{2^{60}}$. The recovery of $\tilde{C}$ is possible since its degree in z is bounded by $2 \cdot 29 = 58 < 60$. However, in terms of input size, half of 60 coefficients of $\tilde{A}(y, z)$ and $\tilde{B}(y, z)$ in z are "left blank", when reinterpreted inside $\mathbb{F}_{2^{60}}$. Consequently, this reduction method based on Kronecker segmentation involves a constant overhead of roughly 2. In fact, when considering algorithms with asymptotically softly linear costs, comparing relative input sizes gives a rough approximation of the relative costs.

Recently van der Hoeven and Larrieu [11] have proposed a new way to reduce multiplication of polynomials in $\mathbb{F}_q[x]$ to the computation of DFTs over an extension $\mathbb{F}_{q^\ell}$. Roughly speaking, they have shown that the DFT of a polynomial in $\mathbb{F}_{q^\ell}[x]$ could be computed almost ℓ times faster if its coefficients happen to lie in the subfield $\mathbb{F}_q$. Using their algorithm, called the *Frobenius FFT*, it is theoretically possible to avoid the overhead of Kronecker segmentation, and thereby to gain a factor of two with respect to [7]. However, application of the Frobenius FFT as described in [11] involves computations in all intermediate fields $\mathbb{F}_{q^e}$ between $\mathbb{F}_q$ and $\mathbb{F}_{q^\ell}$. This makes the theoretical speed-up of two harder to achieve and practical implementations more cumbersome.

Besides Schönhage–Strassen type algorithms, let us mention that other strategies such as the *additive Fourier transform* have been developed for $\mathbb{F}_{2^k}[x]$ [4, 15]. A competitive implementation based on the latter transform has been achieved very recently by Chen et al. [2]—notice that their preprint [2] does not take into account our new implementation. For more historical details on the complexity of polynomial multiplication we refer the reader to the introductions of [7, 8] and to the book by von zur Gathen and Gerhard [5].

1.2. Results and outline of the paper

This paper contains two main results. In section 3, we describe a variant of the Frobenius DFT for the special extension of $\mathbb{F}_{2^{60}}$ over $\mathbb{F}_2$. Using a single rewriting step, this new algorithm reduces the computation of a Frobenius DFT to the computation of an ordinary DFT over $\mathbb{F}_{2^{60}}$, thereby avoiding computations in any intermediate fields $\mathbb{F}_{2^e}$ with $1 < e < 60$ and $e \mid 60$.

Our second main result is a practical implementation of the new algorithm and our ability to indeed gain a factor that approaches two with respect to our previous work. We underline that in both cases, DFTs over $\mathbb{F}_{2^{60}}$ represent the bulk of the computation, but the lengths of the DFTs are halved for the new algorithm. In particular, the observed acceleration is due to our new algorithm and not the result of *ad hoc* code tuning or hardware specific optimizations.

In section 4, we present some of the low level implementation details concerning the new rewriting step. Our timings are presented in section 5. Our implementation outperforms the reference library GF2X version 1.2 developed by Brent, Gaudry, Thomé and Zimmermann [1] for multiplying polynomials in $\mathbb{F}_2[x]$. We also outperform the recent implementation by Chen et al. [2]. Finally, the evaluation-interpolation strategy used by

our algorithm is particularly well suited for multiplying matrices of polynomials over $\mathbb{F}_2$, as reported in section 5.

2. PREREQUISITES

Discrete Fourier transforms

Let ω be a primitive root of unity of order n in $\mathbb{F}_q$. The *discrete Fourier transform* (DFT) of an n-tuple $a = (a_0, ..., a_{n-1}) \in \mathbb{F}_q^n$ with respect to ω is $\mathrm{DFT}_\omega(a) := (\hat{a}_0, ..., \hat{a}_{n-1}) \in \mathbb{F}_q^n$, where

$$\hat{a}_i := a_0 + a_1 \omega^i + \cdots + a_{n-1} \omega^{(n-1)i}.$$

Hence $\hat{a}_i$ is the evaluation of the polynomial $A(x) = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ at ω^i. For simplicity we often identify A with a and we simply write $\mathrm{DFT}_\omega(A)$. The inverse transform is related to the direct transform via $\mathrm{DFT}_\omega^{-1} = n^{-1} \mathrm{DFT}_{\omega^{-1}}$, which follows from the well known formula

$$\mathrm{DFT}_{\omega^{-1}}(\mathrm{DFT}_\omega(a)) = n a.$$

If n properly factors as $n = n_1 n_2$, then ω^{n_1} is an n_2-th primitive root of unity and ω^{n_2} is an n_1-th primitive root of unity. Moreover, for any $i_1 \in \{0, ..., n_1 - 1\}$ and $i_2 \in \{0, ..., n_2 - 1\}$, we have

$$\begin{aligned}
\hat{a}_{i_1 n_2 + i_2} &= \sum_{0 \leqslant k_1 < n_1} \sum_{0 \leqslant k_2 < n_2} a_{k_2 n_1 + k_1} \omega^{(k_2 n_1 + k_1)(i_1 n_2 + i_2)} \\
&= \sum_{0 \leqslant k_1 < n_1} \omega^{k_1 i_2} \left(\sum_{0 \leqslant k_2 < n_2} a_{k_2 n_1 + k_1} (\omega^{n_1})^{k_2 i_2} \right) (\omega^{n_2})^{k_1 i_1}.
\end{aligned} \tag{1}$$

If $\mathcal{A}_1$ and $\mathcal{A}_2$ are algorithms for computing DFTs of length n_1 and n_2, we may use (1) to construct an algorithm for computing DFTs of length n as follows. For each $k_1 \in \{0, ..., n_1 - 1\}$, the sum inside the brackets corresponds to the i_2-th coefficient of a DFT of the n_2-tuple $(a_{0 n_1 + k_1}, ..., a_{(n_2 - 1) n_1 + k_1}) \in \mathbb{F}_q^{n_2}$ with respect to ω^{n_1}. Evaluating these *inner DFTs* requires n_1 calls to $\mathcal{A}_2$. Next, we multiply by the *twiddle factors* $\omega^{k_1 i_2}$, at a cost of n operations in $\mathbb{F}_q$. Finally, for each $i_2 \in \{0, ..., n_2 - 1\}$, the outer sum corresponds to the i_1-th coefficient of a DFT of an n_1-tuple in $\mathbb{F}_q^{n_1}$ with respect to ω^{n_2}. These *outer DFTs* require n_2 calls to $\mathcal{A}_1$. Iterating this decomposition for further factorizations of n_1 and n_2 yields the seminal Cooley–Tukey algorithm [3].

Frobenius Fourier transforms

Let A be a polynomial in $\mathbb{F}_q[x]$ and let ω be a primitive root of unity in some extension $\mathbb{F}_{q^\ell}$ of $\mathbb{F}_q$. We write ϕ_q for the Frobenius map $a \mapsto a^q$ in $\mathbb{F}_{q^\ell}$ and notice that

$$A(\phi_q(a)) = \phi_q(A(a)), \tag{2}$$

for any $a \in \mathbb{F}_{q^\ell}$. This formula implies many nontrivial relations for the DFT of A: if $\omega^i = \phi_q^{\circ k}(\omega^j)$, then we have $A(\omega^i) = \phi_q^{\circ k}(A(\omega^j))$. In other words, some values of the DFT of A can be deduced from others, and the advantage of the Frobenius transform introduced in [11] is to restrict the bulk of the evaluations to a minimum number of points.

Let n denote the order of the root ω, and consider the set $\Omega = \{1, \omega, \omega^2, ..., \omega^{n-1}\}$. This set is clearly globally stable under ϕ_q, so the group $\langle \phi_q \rangle$ generated by ϕ_q acts naturally on it. This action partitions Ω into disjoint orbits. Assume that we have a section Σ of Ω that contains exactly one element in each orbit. Then formula (2) allows us to recover $\mathrm{DFT}_\omega(A)$ from the evaluations of A at each of the points in Σ. The vector $(A(\sigma))_{\sigma \in \Sigma}$ is called the *Frobenius DFT* of A.

3. FAST REDUCTION FROM $\mathbb{F}_2[x]$ TO $\mathbb{F}_{2^{60}}[x]$

3.1. Variant of the Frobenius DFT

To efficiently reduce a multiplication in $\mathbb{F}_2[x]$ into DFTs over $\mathbb{F}_{2^{60}}$, we use an order n that divides $2^{60}-1$ and such that $n = 61\,m$ for some integer m. We perform the decomposition (1) with $n_1 = m$ and $n_2 = 61$. Let ω be a primitive n-th root of unity in $\mathbb{F}_{2^{60}}$. The discrete Fourier transform of $A \in \mathbb{F}_2[x]_{<n}$, given by $(A(1), A(\omega), A(\omega^2), ..., A(\omega^{n-1})) \in \mathbb{F}_{2^{60}}^n$, can be reorganized into 61 slices as follows

$$\mathrm{DFT}_\omega(A) = ((A(\omega^{61i}))_{0 \leqslant i < m}, (A(\omega^{61i+1}))_{0 \leqslant i < m}, ..., (A(\omega^{61i+60}))_{0 \leqslant i < m}).$$

The variant of the Frobenius DFT of A that we introduce in the present paper corresponds to computing only the second slice:

$$E_\omega\colon \mathbb{F}_2[x]_{<60m} \ \to \ \mathbb{F}_{2^{60}}^m$$
$$A \ \mapsto \ (A(\omega^{61i+1}))_{0 \leqslant i < m}.$$

Let us show that this transform is actually a bijection. The following lemma shows that the slices $(A(\omega^{61i+2}))_{0 \leqslant i < m}, ..., (A(\omega^{61i+60}))_{0 \leqslant i < m}$ can be deduced from the second slice $(A(\omega^{61i+1}))_{0 \leqslant i < m}$ using the action of the Frobenius map ϕ_2.

LEMMA 1. *Let* $\Omega_i = \{\omega^{61j+i} : 0 \leqslant j < m\}$ *for* $1 \leqslant i < 61$. *Then the action of* $\langle \phi_2 \rangle$ *is transitive on the pairwise disjoint sets* $\Omega_1, ..., \Omega_{60}$.

Proof. Let $1 \leqslant i < 61$ and $0 \leqslant j < m$, we have $\phi_2(\omega^{61j+i}) = \omega^{61j' + (2i \bmod 61)}$ for some integer $0 \leqslant j' < m$, so the action of $\langle \phi_2 \rangle$ onto $\Omega_1, ..., \Omega_{60}$ is well defined. Notice that 2 is primitive for the multiplicative group $\mathbb{F}_{61}^\times$. This implies that for any $1 \leqslant i < 61$ there exists k such that $2^k = i \bmod 61$. Consequently we have $\phi_2^{\circ k}(\omega^{61j+1}) = \omega^{61j'+i}$ for some $0 \leqslant j' < m$, whence $\phi_2^{\circ k}(\Omega_1) \subseteq \Omega_i$. Since ϕ_2 is injective the latter inclusion is an equality. $\qquad \square$

If we were needed the complete $\mathrm{DFT}_\omega(A)$, then we would still have to compute the first slice $(A(\omega^{61i}))_{0 \leqslant i < m}$. The second main new idea with respect to [11] is to discard this first slice and to restrict ourselves to input polynomials A of degrees $<60\,m$. In this way, E_ω can be inverted, as proved in the following proposition.

PROPOSITION 2. E_ω *is bijective.*

Proof. The dimensions of the source and destination spaces of E_ω over $\mathbb{F}_2$ being the same, it suffices to prove that E_ω is injective. Let $A \in \mathbb{F}_2[x]_{<60m}$ be such that $E_\omega(A) = 0$. By construction, A vanishes at m distinct values, namely ω^{61i+1} for $0 \leqslant i < m$. Under the action of $\langle \phi_2 \rangle$ it also vanishes at $60\,(m-1)$ other values by Lemma 1, whence $A = 0$. $\quad \square$

Remark 3. The transformation E_ω being bijective is due to the fact that 2 is primitive in the multiplicative group $\mathbb{F}_{61}^\times$. Among the prime divisors of $2^{60}-1$, the factors 3, 5, 11 and 13 also have this property, but taking $n_2 = 61$ allows us to divide the size of the evaluation-interpolation scheme by 60, which is optimal.

3.2. Frobenius encoding

We decompose the computation of E_ω into two routines. The first routine is written F_ω and called the *Frobenius encoding*:

$$F_\omega\colon \mathbb{F}_2[x]_{<60m} \ \to \ \mathbb{F}_{2^{60}}[x]_{<m}$$
$$A = \sum_{0 \leqslant k < 60m} a_k x^k \ \mapsto \ \sum_{0 \leqslant k < m} \omega^k \left(\sum_{0 \leqslant l < 60} a_{k+ml}\,\theta^l \right) x^k, \text{where } \theta = \omega^m. \tag{3}$$

Below, we will choose θ in such a way that F_ω is essentially a simple reorganization of the coefficients of A.

We observe that the coefficients of $F_\omega(A)$ are part of the values of the inner DFTs of A in the Cooley–Tukey formula (1), applied with $n_1 = m$ and $n_2 = 61$. The second task is the computation of the corresponding outer DFT of order m:

$$\mathrm{DFT}_{\tilde\omega}\colon \mathbb{F}_{2^{60}}[x]_{<m} \;\to\; \mathbb{F}_{2^{60}}^m$$
$$\tilde A \;\mapsto\; (\tilde A(\tilde\omega^i))_{0\leqslant i < m}, \text{ where } \tilde\omega = \omega^{61}.$$

PROPOSITION 4. $E_\omega = \mathrm{DFT}_{\tilde\omega} \circ F_\omega$.

Proof. This formula follows from (1):

$$A(\omega^{61i+1}) = \sum_{0\leqslant k < m} \omega^k \left(\sum_{0\leqslant l < 61} a_{k+ml}\theta^l \right) \tilde\omega^{ki} = F_\omega(A)(\tilde\omega^i). \qquad \square$$

Summarizing, we have reduced the computation of a DFT of size $60\,n/61$ over $\mathbb{F}_2$ to a DFT of size $m = n/61$ over $\mathbb{F}_{2^{60}}$. This reduction preserves data size.

3.3. Direct transforms

The computation of F_ω involves the evaluation of m polynomials in $\mathbb{F}_2[x]_{<60}$ at $\theta = \omega^m \in \mathbb{F}_{2^{60}}$. In order to perform these evaluations fast, we fix the representation of $\mathbb{F}_{2^{60}} = \mathbb{F}_2[z]/(\mu(z))$ and the primitive root ν of unity of maximal order $2^{60}-1$ to be given by

$$\mu(z) \;=\; (z^{61}-1)/(z-1)$$
$$\nu \;=\; z^{18} + z^6 + 1 \bmod \mu(z).$$

Setting $\omega = \nu^{(2^{60}-1)/n}$ and $\theta = \nu^{(2^{60}-1)/61}$, it can be checked that $\theta = z \bmod \mu(z)$. Evaluation of a polynomial in $\mathbb{F}_2[x]_{<60}$ at θ can now be done efficiently.

Algorithm 1
Input: $A(x) = \sum_{0\leqslant i < 60m} a_i x^i$.
Output: $F_\omega(A)$.
Assumption: $n = 61\,m$ divides $2^{60}-1$.

1. For $i = 0, \ldots, m-1$, build $P_i(z) = \sum_{0\leqslant j < 60} a_{i+mj} z^j \bmod \mu(z) \in \mathbb{F}_{2^{60}}$.

2. Return $P_0 + \omega P_1 x + \omega^2 P_2 x^2 + \cdots + \omega^{m-1} P_{m-1} x^{m-1}$.

PROPOSITION 5. *Algorithm 1 is correct.*

Proof. This deduces immediately from the definition of F_ω in formula (3), using the fact that $\theta = z \bmod \mu(z)$ in our representation. $\qquad \square$

Algorithm 2
Input: $A \in \mathbb{F}_2[x]_{<60m}$.
Output: $E_\omega(A)$.
Assumption: $n = 61\,m$ divides $2^{60}-1$.

1. Compute the Frobenius encoding $\tilde A(x) \in \mathbb{F}_{2^{60}}[x]_{<m}$ of A by Algorithm 1.

2. Compute the DFT of $\tilde A$ with respect to $\tilde\omega$.

PROPOSITION 6. *Algorithm 2 is correct.*

Proof. The correctness simply follows from Propositions 4 and 5. $\qquad \square$

3.4. Inverse transforms

By combining Propositions 2 and 4, the map F_ω is invertible and its inverse may be computed by the following algorithm.

Algorithm 3
Input: $\tilde A(x) = \sum_{i\geqslant 0} \tilde a_i x^i \in \mathbb{F}_{2^{60}}[x]_{<m}$.
Output: $F_\omega^{-1}(\tilde A)$.

Assumption: $n = 61\,m$ divides $2^{60} - 1$.

1. For $i = 0, \ldots, m-1$, build the preimage $P_i(z) := \sum_{0 \leqslant j < 60} p_{i,j} z^j$ of $\omega^{-i} \tilde{a}_i$.

2. Return $\sum_{0 \leqslant i < m} \sum_{0 \leqslant j < 60} p_{i,j} x^{i+mj}$.

PROPOSITION 7. *Algorithm 3 is correct.*

Proof. This is a straightforward inversion of Algorithm 1. □

Algorithm 4
Input: $\hat{a} \in \mathbb{F}_{2^{60}}^m$.
Output: $E_\omega^{-1}(\hat{a})$.
Assumption: $n = 61\,m$ divides $2^{60} - 1$.

1. Compute the inverse DFT $\tilde{A} \in \mathbb{F}_{2^{60}}[x]_{<m}$ of $\hat{a}$ with respect to $\tilde{\omega}$.

2. Compute the Frobenius decoding A of $\tilde{A}$ by Algorithm 3 and return A.

PROPOSITION 8. *Algorithm 4 is correct.*

Proof. The correctness simply follows from Propositions 4 and 7. □

3.5. Multiplication in $\mathbb{F}_2[x]$

Using the standard technique of multiplication by evaluation-interpolation, we may now compute products in $\mathbb{F}_2[x]$ as follows:

Algorithm 5
Input: $A, B \in \mathbb{F}_2[x]_{<\ell}$.
Output: $A\,B$

1. Let $m \geqslant (2\,\ell - 1)/60$ be such that $n = 61\,m$ divides $2^{60} - 1$.

2. Let $\omega = \nu^{(2^{60}-1)/n}$ be the privileged root of unity of order n.

3. Compute $E_\omega(A)$ and $E_\omega(B)$ by Algorithm 2.

4. Compute $\hat{c}$ as the entry-wise product of $E_\omega(A)$ and $E_\omega(B)$.

5. Compute $C(x) = E_\omega^{-1}(\hat{c})$ by Algorithm 4 and return C.

PROPOSITION 9. *Algorithm 4 is correct.*

Proof. The correctness simply follows from Propositions 6 and 8 and using the fact that $E_\omega(A\,B) = E_\omega(A)\,E_\omega(B)$, since $m \geqslant (2\,\ell - 1)/60$. □

For step 1, the actual determination of m has been discussed in [7, section 3]. In fact it is often better not to pick the smallest possible value for m but a slightly larger one that is also very smooth. Since $2^{60} - 1$ admits many small prime divisors, such smooth values of m usually indeed exist.

4. IMPLEMENTATION DETAILS

We follow INTEL's terminology and use the term *quad word* to denote a unit of 64 bits of data. In the rest of the paper we use the C99 standard for presenting our source code. In particular a quad word representing an unsigned integer is considered of type `uint64_t`.

Our implementations are done for an AVX2-enabled processor and an operating system compliant to System V Application Binary Interface. The C++ library NUMERIX of MATHEMAGIX [13] (http://www.mathemagix.org) defines wrappers for AVX types. In particular, `avx_uint64_t` represents an SIMD vector of 4 elements of type `uint64_t`. Recall that the platform disposes of 16 AVX registers which must be allocated accurately in order to minimize read and write accesses to the memory.

Our new polynomial product is implemented in the JUSTINLINE library of MATH-EMAGIX. The source code is freely available from revision 10681 of our SVN server (https://gforge.inria.fr/projects/mmx/). Main sources are in justinline/src/frobenius_encode_f2_60.cpp for the Frobenius encoding and in justinline/mmx/polynomial_f2_amd64_avx2_clmul.mmx for the top level functions. Related test and bench files are also available from dedicated directories of the JUSTINLINE library. Let us further mention here that our MATHEMAGIX functions may be easily exported to C++ [12].

4.1. Packed representations

Polynomials over $\mathbb{F}_2$ are supposed to be given in *packed representation*, which means that coefficients are stored as a vector of contiguous bits in memory. For the implementation considered in this paper, a polynomial of degree $\ell - 1$ is stored into $\lceil \ell/64 \rceil$ quad words, starting with the low-degree coefficients: the constant term is the least significant bit of the first word. The last word is suitably padded with zeros.

Reading or writing one coefficient or a range of coefficients of a polynomial in packed representation must be done carefully to avoid invalid memory access. Let A be such a polynomial of type `uint64_t*`. Reading the coefficient a_i of degree i in A is obtained as `(A[i >> 6] >> (i & 63)) & 1`. However, reading or writing a single coefficient should be avoided as much as possible for efficiency, so we prefer handling ranges of 256 bits. In the sequel the function of prototype

```
void load (avx_uint64_t& d, const uint64_t* A,
   const uint64_t& ℓ, const uint64_t& i, const uint64_t& e);
```

returns the $e \leqslant 256$ bits of A starting from i into d. Bits beyond position ℓ are considered to be zero.

For arithmetic operations in $\mathbb{F}_{2^{60}}$ we refer the reader to [7, section 3.1]. In the sequel we only appeal to the function

```
uint64_t f2_60_mul (const uint64_t& a, const uint64_t& b);
```

that multiplies the two elements a and b of $\mathbb{F}_{2^{60}}$ in packed representation.

We also use a packed column-major representation for matrices over $\mathbb{F}_2$. For instance, an 8×8 bit matrix $(M_{i,j})_{0 \leqslant i < 8, \, 0 \leqslant j < 8}$ is encoded as a quad word whose $(8j+i)$-th bit is $M_{i,j}$. Similarly, a $256 \times \ell$ matrix $(M_{i,j})_{0 \leqslant i < 256, \, 0 \leqslant j < \ell}$ may be seen as a vector v of type `avx_uint64_t*`, so $M_{i,j}$ corresponds to the i-th bit of `v[j]`.

4.2. Matrix transposition

The Frobenius encoding essentially boils down to matrix transpositions. Our main building block is 256×64 bit matrix transposition. We decompose this transposition in a suitable way with regards to data locality, register allocation and vectorization.

For the computation of general transpositions, we repeatedly make use of the well-known divide and conquer strategy: to transpose an $n \times \ell$ matrix M, where n and ℓ are even, we decompose $M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, where A, B, C, D are $n/2 \times \ell/2$ matrices; we swap the anti-diagonal blocks B and C and recursively transpose each block A, B, C, D.

4.2.1. Transposing packed 8 × 8 bit matrices

The basic task we begin with is the transposition of a packed 8×8 bit matrix. The solution used here is borrowed from [18, Chapter 7, section 3].

Function 1

Input: $(M_{i,j})_{0\leqslant i<8,\,0\leqslant j<8}$ in packed representation.
Output: The transpose $(N_{i,j})_{0\leqslant i<8,\,0\leqslant j<8}$ of M in packed representation.

```
uint64_t
packed_matrix_bit_8x8_transpose (const uint64_t& M) {
 1. uint64_t N = M;
 2. static const uint64_t mask_4 = 0x00000000f0f0f0f0;
 3. static const uint64_t mask_2 = 0x0000cccc0000cccc;
 4. static const uint64_t mask_1 = 0x00aa00aa00aa00aa;
 5. uint64_t a;
 6. a = ((N >> 28) ^ N) & mask_4; N = N ^ a;
 7. a = a << 28; N = N ^ a;
 8. a = ((N >> 14) ^ N) & mask_2; N = N ^ a;
 9. a = a << 14; N = N ^ a;
10. a = ((N >> 7) ^ N) & mask_1;  N = N ^ a;
11. a = a << 7;   N = N ^ a;
12. return N; }
```

In steps 6 and 7, the anti-diagonal 4×4 blocks are swapped. In steps 8 and 9, the matrix N is seen as four 4×4 matrices whose anti-diagonal 2×2 blocks are swapped. In steps 10 and 11, the matrix N is seen as sixteen 2×2 matrices whose anti-diagonal elements are swapped. All in all, 18 instructions, 3 constants and one auxiliary variable are needed to transpose a packed 8×8 bit matrix in this way.

One advantage of the above algorithm is that it admits a straightforward AVX vectorization that we will denote by

```
avx_uint64_t
avx_packed_matrix_bit_8x8_transpose (const avx_uint64_t& M);
```

This routine transposes four 8×8 bit matrices M_0, M_1, M_2, M_3 that are packed successively into an AVX register of type `avx_uint64_t`. We emphasize that this task is *not* the same as transposing a 32×8 or 8×32 bit matrices.

Remark 10. The BMI2 technology gives another method for transposing 8×8 bit matrices:

```
uint64_t mask = 0x0101010101010101;
uint64_t N= 0;
for (unsigned i = 0; i < 8; i++)
  N |= _pext_u64 (M, mask << i) << (8 * i);
```

The loop can be unrolled while precompting the shift amounts and masks, which leads to a faster sequential implementation. Unfortunately this approach cannot be vectorized with the AVX2 technology. Other sequential solutions even exist, based on lookup tables or integer arithmetic, but their vectorization is again problematic. Practical efficiencies are reported in section 5.

4.2.2. Transposing four 8×8 byte matrices simultaneously

Our next task is to design a transposition algorithm of four packed 8×8 byte matrices simultaneously. More precisely, it performs the following operation on a packed 32×8 byte matrix:

$$\begin{pmatrix} M_0 \\ M_1 \\ M_2 \\ M_3 \end{pmatrix} \longrightarrow \begin{pmatrix} M_0^\top \\ M_1^\top \\ M_2^\top \\ M_3^\top \end{pmatrix},$$

where the M_i are 8×8 blocks. This operation has the following prototype in the sequel:

```
void avx_packed_matrix_byte_8x8_transpose
  (avx_uint64_t* dest, const avx_uint64_t* src);
```

This function works as follows. First the input `src` is loaded into eight AVX registers $r_0, ..., r_7$. Each r_i is seen as a vector of four `uint64_t`: for $j \in \{0, ..., 3\}$, $r_0[j], ..., r_7[j]$ thus represent the 8×8 byte matrix M_j. Then we transpose these four matrices simultaneously in-register by means of AVX shift and blend operations over 32, 16 and 8 bits entries in the spirit of the aforementioned divide and conquer strategy.

4.2.3. Transposing 256×64 bit matrices

Having the above subroutines at our disposal, we can now present our algorithm to transpose a packed 256×64 bit matrix. The input bit matrix of type `avx_int64_t*` is written $(M_{i,j})_{0 \leqslant i < 256, \, 0 \leqslant j < 64}$. The transposed output matrix is written $(N_{i,j})_{0 \leqslant i < 64, \, 0 \leqslant j < 256}$ and has type `uint64_t*`. We first compute the auxiliary byte matrix T as follows:

```
static avx_uint64_t T[64];
for (int i= 0; i < 8; i++) {
  avx_packed_matrix_byte_8x8_transpose (T + 8*i, M + 8*i);
  for (int k= 0; k < 8; k++)
    T[8*i+k]= avx_packed_matrix_bit_8x8_transpose(T[8*i+k]); }
```

If we write $M_{i,k:l}$ for the byte representing the packed bit vector $(M_{i,k}, ..., M_{i,l})$, then T contains the following 32×64 byte matrix:

$$\begin{pmatrix} M_{0,0:7} & \cdots & M_{56,0:7} & M_{0,8:15} & \cdots & M_{56,8:15} & \cdots & M_{0,56:63} & \cdots & M_{56,56:63} \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ M_{7,0:7} & \cdots & M_{63,0:7} & M_{7,8:15} & \cdots & M_{63,8:15} & \cdots & M_{7,56:63} & \cdots & M_{63,56:63} \\ \hline M_{64,0:7} & \cdots & M_{120,0:7} & M_{64,8:15} & \cdots & M_{120,8:15} & \cdots & M_{64,56:63} & \cdots & M_{120,56:63} \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ M_{71,0:7} & \cdots & M_{127,0:7} & M_{71,8:15} & \cdots & M_{127,8:15} & \cdots & M_{71,56:63} & \cdots & M_{127,56:63} \\ \hline M_{128,0:7} & \cdots & M_{184,0:7} & M_{128,8:15} & \cdots & M_{184,8:15} & \cdots & M_{128,56:63} & \cdots & M_{184,56:63} \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ M_{135,0:7} & \cdots & M_{191,0:7} & M_{135,8:15} & \cdots & M_{191,8:15} & \cdots & M_{135,56:63} & \cdots & M_{191,56:63} \\ \hline M_{192,0:7} & \cdots & M_{248,0:7} & M_{192,8:15} & \cdots & M_{248,8:15} & \cdots & M_{192,56:63} & \cdots & M_{248,56:63} \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ M_{199,0:7} & \cdots & M_{255,0:7} & M_{199,8:15} & \cdots & M_{255,8:15} & \cdots & M_{199,56:63} & \cdots & M_{255,56:63} \end{pmatrix}.$$

First, for all $0 \leqslant i \leqslant 7$, we load column $8i$ into the AVX register r_i. We interpret these registers as forming a 32×8 byte matrix that we transpose in-registers. This transposition is again performed in the spirit of the aforementioned divide and conquer strategy and makes use of various specific AVX2 instructions. We obtain

$$\begin{pmatrix} M_{0,0:7} & M_{1,0:7} & \cdots & M_{7,0:7} & M_{64,0:7} & M_{65,0:7} & \cdots & M_{71,0:7} & \cdots \\ M_{0,8:15} & M_{1,8:15} & \cdots & M_{7,8:15} & M_{64,8:15} & M_{65,8:15} & \cdots & M_{71,8:15} & \cdots \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \\ M_{0,56:63} & M_{1,56:63} & \cdots & M_{7,56:63} & M_{64,56:63} & M_{65,56:63} & \cdots & M_{71,56:63} & \cdots \end{pmatrix}.$$

More precisely, for $i = 0, ..., 7$, the group of four consecutive columns from $4i$ until $4i + 3$ is in the register r_i. We save the registers $r_0, ..., r_7$ at the addresses $N, N+4, N+64, N+68, N+128, N+132, N+192$ and $N+196$.

For each $k = 1, ..., 7$, we build a similar 32×8 byte matrix from the columns $k, 8+k, ..., 56 + k$ of T, and transpose this matrix using the same algorithm. This time the result is saved at the addresses $N', N'+4, N'+64, N'+68, N'+128, N'+132, N'+192$ and $N'+196$, where $N' = N + 8k$. This yields an efficient routine for transposing M into N, whose prototype is given by

```
void packed_matrix_bit_256x64_transpose
  (uint64_t* N, (const avx_uint64_t*) M);
```

4.3. Frobenius encoding

If the input polynomial A has degree less than $\ell \leqslant 60\,m$ and is in packed representation, then it can also be seen as a $m \times 60$ matrix in packed representation (except a padding with zeros could be necessary to adjust the size).

In this setting, the polynomials P_i of Algorithm 1 are simply read as the rows of the matrix. Therefore, to compute the Frobenius encoding $F_\omega(A)$, we only need to transpose this matrix, then add 4 rows of zeros for alignment (because we store one element of $\mathbb{F}_{2^{60}}$ per quad word) and multiply by twiddle factors. This leads to the following implementation:

Function 2
Input: $A(x) = \sum_{0 \leqslant i < \ell} a_i x^i \in \mathbb{F}_2[x]$.
Output: $F_\omega(A)$ stored from pointer d to m allocated quad words.
Assumptions: $n = 61\,m$ divides $2^{60} - 1$ and $\ell \leqslant 60\,m$.

```
void encode (uint64_t* d, const uint64_t& m,
             const uint64_t* A, const uint64_t& ℓ) {
1. uint64_t c = 1, i = 0, e = 0;
2. avx_uint64_t v[64]; uint64_t w[256];
3. while (i < m) {
4.   e = min (m - i, 256);
5.   for (int j = 0; j < 64; j++)
       load (v[j], A, ℓ, i + m * j, e);
6.   packed_matrix_bit_256x64_transpose (w, v);
7.   for (int j = 0; j < e; j++) {
       d[i + j] = f2_60_mul (w[j], c);
       c = f2_60_mul (c, ω); }
8.   i += e; }
```

Remark. To optimize read accesses, it is better to run loop 5 for $j < \lceil l/m \rceil$ and to initialize the remaining $v[j]$ to zero. Indeed, for a product of degree d, we typically multiply two polynomials of degree $\simeq d/2$, which means $\ell < 30\,m$ when computing the direct transform.

The Frobenius decoding consists in inverting the encoding. The implementation issues are the same, so we refer to our source code for further details.

5. TIMINGS

The platform considered in this paper is equipped with an INTEL(R) CORE(TM) i7-6700 CPU at 3.40 GHz and 32 GB of 2133 MHz DDR4 memory. This CPU features AVX2, BMI2 and CLMUL technologies (family number 6 and model number 94). The platform runs the STRETCH GNU DEBIAN operating system with a 64 bit LINUX kernel version 4.3. We compile with GCC [6] version 5.4.

We use version 1.2 of the GF2X library (https://gforge.inria.fr/projects/gf2x/, released in July 2017)—it makes use of the CLMUL features of the platform. We tuned it to our platform during the installation process up to 32 000 000 input quad words. We also compare to the implementation of the additive Fourier transform by Chen et al. [2], using the GIT version of 2017, September, 1.
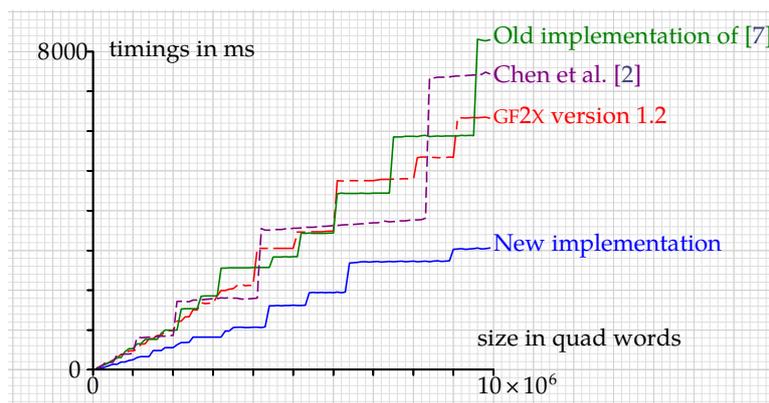
Figure 1. Products in $\mathbb{F}_2[x]_{<\ell}$, input size $\lceil \ell/64 \rceil$ quad words, timings in milliseconds.

Frobenius encoding

Concerning the cost of the Frobenius encoding and decoding, Function 1 takes about 20 CPU cycles when compiled with the sole `-O3` option. With the additional options `-mtune=native -mavx2 -mbmi2`, the BMI2 version of Remark 10 takes about 16 CPU cycles. The vectorized version of Function 1 transposes four packed 8×8 bit matrices simultaneously in about 20 cycles, which makes an average of 5 cycles per matrix.

It it interesting to examine the performance of the sole transpositions made during the Frobenius encoding and decoding (that is discarding products by twiddle factors in $\mathbb{F}_{2^{60}}$). From sizes of a few kilobytes this average cost per quad word is about 8 cycles with the AVX2 technology, and it is about 23 cycles without. Unfortunately the vectorization speed-up is not as close to 4 as we would have liked.

Since the encoding and decoding costs are linear, their relative contribution to the total computation time of polynomial products decreases for large sizes. For two input polynomials in $\mathbb{F}_2[x]$ of 2^{16} quad words, the contribution is about 15%; for 2^{22} quad words, it is about 10%.

Polynomial product

In Figure 1 we report timings in milliseconds for multiplying two polynomials in $\mathbb{F}_2[x]_{<\ell}$, hence each of input size $\lceil \ell/64 \rceil$ quad words—indicated in abscissa and obtained from `justinline/bench/polynomial_f2_bench.mmx`. Notice that our implementation in [7] was faster than version 1.1 of GF2X, but is now of similar speed as version 1.2. The additive FFT strategy of [2] achieves a noticeable speed-up in favorable cases, but because of its staircase-effect its runtime is roughly similar to the one of GF2X in average. With respect to our old implementation, the new one finally achieves a speed-up that is not far from the factor 2 predicted by the asymptotic complexity analysis. Let us mention that our new implementation becomes faster than GF2X when $\lceil \ell/64 \rceil$ is larger than 2048.

Polynomial matrix product

As in [7], one major advantage of DFTs over the Babylonian field $\mathbb{F}_{2^{60}}$ is the compactness of the evaluated FFT-representation of polynomials. This makes linear algebra over $\mathbb{F}_2[x]$ particularly efficient: instead of multiplying $r \times r$ matrices over $\mathbb{F}_2[x]_{<\ell}$ naively by means of r^3 polynomial products of degree $<\ell$, we use the standard evaluation-interpolation approach. In our context, this comes down to: (a) computing the $2r^2$ Frobenius encodings, (b) the $2r^2$ direct DFTs of all entries of the two matrices to be multiplied, (c) performing the $\approx 2\ell/60$ products of $r \times r$ matrices over $\mathbb{F}_{2^{60}}$, (d) computing the r^2 inverse DFTs and Frobenius decodings of the so-computed matrix products.

r	1	2	4	8	16	32
this paper	12	51	212	896	3969	18953
GF2X	22	182	1457	11856	92858	745586

Table 1. Products of $r \times r$ matrices over $\mathbb{F}_2[x]$, for degree $64 \cdot 2^{16}$, in milliseconds.

Timings for matrices over $\mathbb{F}_2[x]$ are obatined from `justinline/bench/matrix_polynomial_f2_bench.mmx` and are reported in Table 1. The row "this paper" confirms the practical gain of this fast approach within our implementation. For comparison, the row "GF2X" shows the cost of computing the product naively, by doing r^3 polynomial multiplications using GF2X. More efficient evaluation-interpolation based approaches [10, Section 2] for matrix multiplication can in principle be combined with Schönhage's triadic polynomial multiplication [16] as implemented in GF2X. However, this would require an additional implementation effort and also lead to an extra constant overhead with respect to our approach.

6. CONCLUSION

The present paper describes a major new approach for the efficient computation of large carryless products. It confirms the excellent arithmetic properties of the Babylonian field $\mathbb{F}_{2^{60}}$ for practical purposes, when compared to the fastest previously available strategies.

Improvements are still possible for our implementation of DFTs over $\mathbb{F}_{2^{60}}$. First, taking advantage of the more recent AVX-512 technologies is an important challenge. This is difficult due to the current lack of 256 or 512 bit SIMD counterparts for the `vpclmulqdq` assembly instruction (carryless multiplication of two quad words). However, larger vector instruction would be beneficial for matrix transposition, and even more taking into account that there are twice as many 512 bit registers as 256 bit registers; so we can expect a significant speed-up for the Frobenius encoding/decoding stages. The second expected improvement concerns the use of truncated Fourier transforms [9, 14] in order to smoothen the graph from Figure 1. Finally we expect that our new ideas around the Frobenius transform might be applicable to other small finite fields.

BIBLIOGRAPHY

[1] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in GF(2)[x]. In A. van der Poorten and A. Stein, editors, *Algorithmic Number Theory*, volume 5011 of *Lect. Notes Comput. Sci.*, pages 153–166. Springer Berlin Heidelberg, 2008.

[2] Ming-Shing Chen, Chen-Mou Cheng, Po-Chun Kuo, Wen-Ding Li, and Bo-Yin Yang. Faster multiplication for long binary polynomials. https://arxiv.org/abs/1708.09746, 2017.

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

[4] S. Gao and T. Mateer. Additive fast Fourier transforms over finite fields. *IEEE Trans. Inform. Theory*, 56(12):6265–6272, 2010.

[5] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 3rd edition, 2013.

[6] GCC, the GNU Compiler Collection. Software available at http://gcc.gnu.org, from 1987.

[7] D. Harvey, J. van der Hoeven, and G. Lecerf. Fast polynomial multiplication over $F_{2^{60}}$. In M. Rosenkranz, editor, *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, pages 255–262. ACM, 2016.

[8] D. Harvey, J. van der Hoeven, and G. Lecerf. Faster polynomial multiplication over finite fields. *J. ACM*, 63(6), 2017. Article 52.

[9] J. van der Hoeven. The truncated Fourier transform and applications. In J. Schicho, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 290–296. ACM, 2004.

[10] J. van der Hoeven. Newton's method and FFT trading. *J. Symbolic Comput.*, 45(8):857–878, 2010.

[11] J. van der Hoeven and R. Larrieu. The Frobenius FFT. In M. Burr, editor, *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 437–444. ACM, 2017.

[12] J. van der Hoeven and G. Lecerf. Interfacing Mathemagix with C++. In M. Monagan, G. Cooperman, and M. Giesbrecht, editors, *Proceedings of the 2013 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '13, pages 363–370. ACM, 2013.

[13] J. van der Hoeven and G. Lecerf. Mathemagix User Guide. https://hal.archives-ouvertes.fr/hal-00785549, 2013.

[14] R. Larrieu. The truncated Fourier transform for mixed radices. In M. Burr, editor, *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '17, pages 261–268. ACM, 2017.

[15] Sian-Jheng Lin, Wei-Ho Chung, and S. Yunghsiang Han. Novel polynomial basis and its application to Reed-Solomon erasure codes. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 316–325. IEEE, 2014.

[16] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Infor.*, 7:395–398, 1977.

[17] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.

[18] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2nd edition, 2012.