Fast Polynomial Multiplication over $\mathbb{F}_{2^{60}}$

David Harvey

School of Mathematics and Statistics
University of New South Wales
Sydney NSW 2052
Australia
d.harvey@unsw.edu.au

Joris van der Hoeven, Grégoire Lecerf
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
{vdhoeven,lecerf}@lix.polytechnique.fr

ABSTRACT

Can post-Schönhage–Strassen multiplication algorithms be competitive in practice for large input sizes? So far, the GMP library still outperforms all implementations of the recent, asymptotically more efficient algorithms for integer multiplication by Fürer, De–Kurur–Saha–Saptharishi, and ourselves. In this paper, we show how central ideas of our recent asymptotically fast algorithms turn out to be of practical interest for multiplication of polynomials over finite fields of characteristic two. Our Mathemagix implementation is based on the automatic generation of assembly codelets. It outperforms existing implementations in large degree, especially for polynomial matrix multiplication over finite fields.

Categories and Subject Descriptors

F.2.1 [**Analysis of algorithms and problem complexity**]: Numerical Algorithms and Problems—*Computations in finite fields*; G.4 [**Mathematical software**]: Algorithm design and analysis

General Terms

Algorithms, Theory

Keywords

Finite fields; Polynomial multiplication; Mathemagix

1. INTRODUCTION

Let $\mathbb{F}_{2^k}$ denote the finite field with 2^k elements. In this article we are interested in multiplying polynomials in $\mathbb{F}_{2^k}[x]$. This is a very basic and classical problem in complexity theory, with major applications to number theory, cryptography and error correcting codes.

Version submitted to ISSAC'16
January 30, 2016

1.1 Motivation

Let $\mathsf{M}_{2^k}(n)$ denote the bit complexity of multiplying two polynomials of degree $<n$ in $\mathbb{F}_{2^k}[x]$. Until recently, the best asymptotic bound for this complexity was $\mathsf{M}_{2^k}(n) = O(k\ n\ \log(k\ n)\ \log\log(k\ n))$, using a triadic version [32] of the classical Schönhage–Strassen algorithm [33].

In [21], we improved this bound to $\mathsf{M}_{2^k}(n) = O(k\ n\ \log(k\ n)\ 8^{\log^*(kn)})$, where $\log^* x = \min\{i \in \mathbb{N} |\ \log \circ \overset{i\times}{\cdots} \circ \log\ x \leqslant 1\}$. The factor $8^{\log^*(kn)}$ increases so slowly that it is impossible to observe the asymptotic behavior of our algorithm in practice. Despite this, the present paper demonstrates that some of the new techniques introduced in [20, 21] can indeed lead to more efficient implementations.

One of the main reasons behind the observed acceleration is that [21] contains a natural analogue for the *three primes FFT* approach to multiplying integers [30]. For a single multiplication, this kind of algorithm is more or less as efficient as the Schönhage–Strassen algorithm: the FFTs involve more expensive arithmetic, but the inner products are faster. On recent architectures, the three primes approach for integer multiplication generally has performance superior to that of Schönhage–Strassen due to its cache efficiency [25].

The compactness of the transformed representations also makes the three primes approach very useful for linear algebra. Accordingly, the implementation described in this paper is very efficient for matrix products over $\mathbb{F}_{2^k}[x]$. This is beneficial for many applications such as half gcds, polynomial factorization, geometric error correcting codes, polynomial system solving, etc.

1.2 Related work and our contributions

Nowadays the Schönhage–Strassen algorithm is widely used in practice for multiplying large integers [18] and polynomials [4, 19]. For integers, it was the asymptotically fastest known until Fürer's algorithm [11, 12] with cost $n \log n\ K^{\log^* n}$, for input bit sizes n, where $K > 1$ is some constant (an optimized version [20] yields the explicit value $K = 16$). However, no-one has yet demonstrated a practical implementation for sizes supported by current technology. The implementation of the modular variant proposed in [7] has even been discussed in detail in [28]: their conclusion is that the break-even point seems to be beyond astronomical sizes.

In [20, 21] we developed a unified alternative strategy for both integers and polynomials. Roughly speaking, products are performed *via discrete Fourier transforms* (DFTs) that are split into smaller ones. Small transforms then reduce to smaller products. When these smaller products are still large enough, the algorithm is used recursively. Since the input size decreases logarithmically between recursive calls, there is of course just one such recursive call in practice. Our implementation was guided by these ideas, but, in the end, only a few ingredients were retained. In fact, we do not recurse at all; we handle the smaller subproducts directly over $\mathbb{F}_{2^{60}}$ with the Karatsuba algorithm.

For polynomials over finite fields, one key ingredient of [21] is the construction of suitable finite fields: we need the cardinality of their multiplicative groups to have sufficiently many small prime factors. A remarkable example is $\mathbb{F}_{2^{60}}$, outlined at the end of [21], for which we have:

$$2^{60}-1=3^2\cdot 5^2\cdot 7\cdot 11\cdot 13\cdot 31\cdot 41\cdot 61\cdot 151\cdot 331\cdot 1321.$$

Section 3 contains our first contribution: a detailed optimized implementation of DFTs over $\mathbb{F}_{2^{60}}$. We propose several original tricks to make it highly competitive on processors featuring *carry-less products*. For DFTs of small prime sizes we appeal to seminal techniques due to Rader [31] and Winograd [36]. Then the *prime-factor algorithm*, also called Good–Thomas [17, 35], is used in medium sizes, while the Cooley–Tukey algorithm [5] serves for large sizes.

In Section 4, we focus on other finite fields $\mathbb{F}_{2^k}$. When k is a divisor of 60, we design efficient embeddings into $\mathbb{F}_{2^{60}}$, and compare to the use of Kronecker segmentation and padding. For values of k up to 59, we propose a new algorithm to reduce computations to $\mathbb{F}_{2^{60}}$, in the vein of the three prime FFT technique. This constitutes our second contribution.

The practical performance of our implementation is reported in Section 5. We compare to the optimized reference library GF2X, developed by Brent, Gaudry, Thomé and Zimmermann [4], which assembles Karatsuba, Toom–Cook and the triadic Schönhage–Strassen algorithms with automatically tuned thresholds. The high performance of our implementation is of course a major outcome of this paper.

For detailed references on asymptotically fast algorithms to univariate polynomials over $\mathbb{F}_{2^k}$ and some applications, we refer the reader to [4, 20, 21]. The present article does not formally rely on [21]. In the next section we recall and slightly adapt classical results, and present our implementation framework. Our use of *codelets* for small and moderate sizes of DFT is customary in other high performance software, such as FFTW3 and SPIRAL [10, 29].

2. PRELIMINARIES

This section gathers classical building blocks used by our DFT algorithm. We also describe our implementation framework. If a and b are elements of a Euclidean domain, then a quo b and a rem b respectively represent the related quotient and remainder in the division of a by b, so that $a=(a\,\mathrm{quo}\,b)\,b+(a\,\mathrm{rem}\,b)$ holds.

2.1 Discrete Fourier transforms

Let $n_1, \ldots, n_d$ be positive integers, let $v=(n_1, \ldots, n_d)$, $n=n_1\cdots n_d$, and $i\in\{0,\ldots,n-1\}$. The *v-expansion* of i is the sequence of integers $i_0,\ldots,i_{d-1}$ such that $0\leqslant i_j<n_{j+1}$ and

$$i=i_{d-1}+i_{d-2}\,n_d+i_{d-3}\,n_{d-1}\,n_d+\cdots+i_0\,n_2\cdots n_d.$$

The *v-mirror*, written $[i]_v$, of i relative to v, is the integer defined by the following reverse expansion:

$$[i]_v=i_0+i_1\,n_1+\cdots+i_{d-2}\,n_1\cdots n_{d-2}+i_{d-1}\,n_1\cdots n_{d-1}.$$

Let $\mathbb{K}$ be a commutative field, and let $\omega\in\mathbb{K}$ be a primitive n-th root of unity, which means that $\omega^n=1$, and $\omega^j\neq 1$ for all $j\in\{1,\ldots,n-1\}$. The *discrete Fourier transform* (with respect to ω) of an n-tuple $(a_0,\ldots,a_{n-1})\in\mathbb{K}^n$ is the n-tuple $(\hat{a}_0,\ldots,\hat{a}_{n-1})\in\mathbb{K}^n$ given by $\hat{a}_i=\sum_{j=0}^{n-1}\omega^{ij}\,a_j$. In other words we have $\hat{a}_i=A(\omega^i)$, where $A(x)=\sum_{j=0}^{n-1}a_j\,x^j$. For efficiency reasons, in this article, we compute the $\hat{a}_i$ in the v-mirror order, where v corresponds to a certain ordering of the prime factors of n, counted with multiplicities, that will be optimized later. We shall use the following notation:

$$\mathrm{DFT}_{\omega,v}(a)=(\hat{a}_{[0]_v},\ldots,\hat{a}_{[n-1]_v}).$$

Let $1\leqslant h<d$, and let us split $v=(n_1,\ldots,n_d)$ into $v_1=(n_1,\ldots,n_h)$ and $v_2=(n_{h+1},\ldots,n_d)$. We recall the *Cooley–Tukey algorithm* [5] in the present setting: it decomposes $\mathrm{DFT}_{\omega,v}$ into $\mathrm{DFT}_{\omega^{m_2},v_1}$ and $\mathrm{DFT}_{\omega^{m_1},v_2}$, where $m_1=n_1\cdots n_h$ and $m_2=n_{h+1}\cdots n_d$.

Algorithm 1 (In-place Cooley–Tukey algorithm)

Input. $v=(n_1,\ldots,n_d)$, $h\in\{1,\ldots,d-1\}$, an n-th primitive root of unity ω, and $a=(a_0,\ldots,a_{n-1})\in\mathbb{K}^n$.

Output. $\mathrm{DFT}_{\omega,v}(a)$, stored in a.

1. For $j\in\{0,\ldots,m_2-1\}$ do
   Let $b=(a_j,a_{j+m_2},\ldots,a_{j+(m_1-1)m_2})$
   Compute $c=\mathrm{DFT}_{\omega^{m_2},v_1}(b)$
   Replace a_{j+im_2} by c_i for all $i\in\{0,\ldots,m_1-1\}$
2. For $i\in\{0,\ldots,m_1-1\}$ and $j\in\{0,\ldots,m_2-1\}$ do
   Replace a_{j+im_2} by $a_{j+im_2}\omega^{[i]_{v_1}j}$
3. For $i\in\{0,\ldots,m_1-1\}$ do
   Let $b=(a_{im_2},a_{im_2+1},\ldots,a_{im_2+m_2-1})$
   Compute $c=\mathrm{DFT}_{\omega^{m_1},v_2}(b)$
   Replace a_{j+im_2} by c_j for all $j\in\{0,\ldots,m_2-1\}$

PROPOSITION 1. *Algorithm 1 is correct.*

PROOF. Let α, β, γ be the successive values of the vector a at the end of steps 1, 2, and 3. We have $\alpha_{j+im_2}=\sum_{k=0}^{m_1-1}a_{j+km_2}\,\omega^{km_2[i]_{v_1}}$, $\beta_{j+im_2}=\alpha_{j+im_2}\,\omega^{[i]_{v_1}j}$, and $\gamma_{j+im_2}=\sum_{l=0}^{m_2-1}\beta_{l+im_2}\;\omega^{lm_1[j]_{v_2}}$. It follows that $\gamma_{j+im_2}=\sum_{l=0}^{m_2-1}\sum_{k=0}^{m_1-1}a_{l+km_2}\,\omega^{km_2[i]_{v_1}}\,\omega^{[i]_{v_1}l}\,\omega^{lm_1[j]_{v_2}}=\sum_{l=0}^{m_2-1}\sum_{k=0}^{m_1-1}a_{l+km_2}\,\omega^{(km_2+l)[i]_{v_1}+lm_1[j]_{v_2}}$. The conclusion follows from $[j+i\,m_2]_v=[i]_{v_1}+m_1\,[j]_{v_2}$, which implies $(l+k\,m_2)\,[i]_{v_1}+l\,m_1\,[j]_{v_2}\equiv[j+i\,m_2]_v\,(l+k\,m_2)\bmod n$. □

Notice that the order of the output depends on v, but not on h. If the input vector is stored in a contiguous segment of memory, then the first step of Algorithm 1 may be seen as an $m_1\times m_2$ matrix transposition, followed by m_2 in-place DFTs of size m_1 on contiguous data. Then the transposition is inverted. The constants $\omega^{[i]_{v_1}j}$ in step 2 are often called *twiddle factors*. Transpositions affect the performance of the strided DFTs when input sizes do not fit in the last level cache memory of the processor.

For better locality, the multiplications by twiddle factors in step 2 are actually merged into step 3, and all the necessary primitive roots and fixed multiplicands (including the twiddle factors) are pre-computed once, and cached in memory. We finally notice that the inverse DFT can be computed straightforwardly by inverting Algorithm 1.

In addition to the Cooley–Tukey algorithm we shall also use the method attributed to *Good* and *Thomas* [17, 35], and that saves all the multiplications by the twiddle factors. First it requires the n_i to be pairwise coprime, and

second, input and output data must be re-indexed. The rest is identical to Algorithm 1—see [8], for instance, for complete formulas. With $\mathbb{F}_{2^{60}}$, the condition on the n_i fails only when two n_i are 3 or 5. In such a case, it is sufficient to replace v by a new vector where the two occurrences of 3 (resp. of 5) are replaced by 9 (resp. by 25), and to modify the re-indexing accordingly. In sizes 9 and 25, we build codelets upon the Cooley–Tukey formulas. A specific codelet for $n = (3\ 5)^2$ might further improve performance, but we did not implement this yet. We hardcoded the re-indexing into codelets, and we restrict to using the Good–Thomas algorithm up to sizes that fit into the cache memory. We shall discuss later the relative performance between these two DFT algoritms.

2.2 Rader reduction

When the recurrence of the Cooley–Tukey algorithm ends with a DFT of prime size n, then we may use Rader's algorithm [31] to reduce such a DFT to a cyclic polynomial product by a fixed multiplicand.

In fact, let $\mu = n-1$, let γ be a generator of the multiplicative group of $\mathbb{Z}/n\,\mathbb{Z}$, and let η be its modular inverse. We let $B(x) = \sum_{i=0}^{\mu-1} a_{\gamma^i \bmod n}\, x^i$ and $W(x) = \sum_{i=0}^{\mu-1} \omega^{\eta^i \bmod n}\, x^i$, and we compute $C(x) = B(x)\, W(x) \bmod (x^\mu - 1)$. The coefficient c_i of x^i in $C(x)$ equals $\sum_{j=1}^{n-1} a_j\, \omega^{(\eta^i j) \bmod n}$. Consequently we have $\mathrm{DFT}_{\omega,n}(a) = (\sum_{j=0}^{n-1} a_j,\ a_0 + c_{\sigma(1)-1},\ \ldots,\ a_0 + c_{\sigma(\mu)-1})$, where σ is the permutation of $\{1, \ldots, \mu\}$ defined by $\sigma(\eta^i \bmod \mu) = i$. In this way, the DFT of a reduces to one cyclic product of size μ, by the fixed multiplicand W.

Remark 2. Bluestein reduction [2] allows for the conversion of DFTs into cyclic products even when n is not prime. In [21], this is crucially used for controlling the size n of recursive DFTs. This suggests that Bluestein reduction might be useful in practice for DFTs of small composite orders, say $n \leqslant 50$. For DFTs over $\mathbb{F}_{2^{60}}$, this turns out to be wrong: so far, the strategies to be described in Section 3 are more efficient.

2.3 Implementation framework

Throughout this article, we consider a platform equipped with an Intel(R) Core(TM) i7-4770 CPU at 3.40 GHz and 8 GB of 1600 MHz DDR3 memory, which features AVX2 and CLMUL technologies (family number 6 and model number 0x3C). The platform runs the Jessie GNU Debian operating system with a 64 bit Linux kernel version 3.14. Care has been taken to avoid *CPU throttling* and *Turbo Boost* issues while measuring timings. We compiled with GCC [15] version 4.9.2.

In order to efficiently and easily benefit from AVX and CLMUL instruction sets, we decided to implement the lowest level of our DFTs directly in assembly code. In fact there is no standard way to take full advantage of these technologies within languages such as C and C++, where current SIMD types are not even *bona fide*. It is true that programming *via intrinsics* [26] is a reasonable compromise, but there remain a certain number of technical pitfalls such as memory alignment, register allocation, and instruction latency management.

For our convenience we developed *dynamic compilation* features (also known as *just in time* compilation) from scratch, dedicated to high performance computing within Mathemagix (`http://www.mathemagix.org`). It is only used to tune assembly code for DFTs of orders a few thousands. Our implementation in the Runtime library partially supports the x86, SSE, and AVX instruction sets for the amd64 *application binary interface*—missing instructions can be added easily. This of course suits most personal computers and computation clusters.

The efficiency of an SSE or AVX instruction is not easy to determine. It depends on the types of its arguments, and is usually measured in terms of latency and throughput. In ideal conditions, the *latency* is the number of CPU clock cycles needed to make the result of an instruction available to another instruction; the *reciprocal throughput*, sometimes called *issue latency*, is the (fractional) number of cycles needed to actually perform the computation—for brevity, we drop "reciprocal". For detailed definitions we refer the reader to [26], and also to [9] for useful additional comments.

In this article, we shall only use AVX-128 instructions, and 128-bit registers are denoted `xmm0`,..., `xmm15` (our code is thus compatible with the previous generation of processors). A 128-bit register may be seen as a vector of two 64-bit integers, that are said to be *packed* in it. We provide unfamiliar readers with typical examples for our aforementioned processor, with cost estimates, and using the Intel syntax, where the destination is the first argument of instructions:

- `vmovq` loads/stores 64-bits integers from/to memory. `vmovdqu` loads/stores packed 64-bits integers not necessary aligned to 256-bit boundaries. `vmovdqa` is similar to `vmovdqu` when data is aligned on a 256-bit boundary; it is also used for moving data between registers. Latencies of these instructions are between 1 and 4, and throughputs vary between 0.3 and 1.
- `vpand`, `vpor`, and `vpxor` respectively correspond to bitwise "and", "or" and "xor" operations. Latencies are 1 and throughputs are 0.33 or 0.5.
- `vpsllq` and `vpsrlq` respectively perform left and right logical shifts on 64-bit packed integers, with latency 1 or 2, and throughput 1. We shall also use `vpunpckhqdq xmm1, xmm2, xmm3/m128` to unpack and interleave in `xmm1` the 64-bit integers from the high half of `xmm2` and `xmm3/m128`, with latency and throughput 1. Here, `xmm1`, `xmm2`, and `xmm3` do not mean that the instruction only acts on these specific registers: instead, the indices 1, 2, and 3 actually refer to argument positions. In addition, the notation `xmm3/m128` means that the third argument may be either a register or an adress pointing to 128-bit data to be read from the memory.
- `vpclmulqdq xmm1, xmm2, xmm3/m128, imm8` carry-less multiplies two 64-bit integers, selected from `xmm2` and `xmm3/m128` according to the constant integer `imm8`, and stores the result into `xmm1`. The value 0 for `imm8` means that the multiplicands are the 64-bit integers from the low half of `xmm2` and `xmm3/m128`. Mathematically speaking, this corresponds to multiplying two polynomials in $\mathbb{F}_2[x]$ of degrees <64, which are packed into integers: such a polynomial is thus represented by a 64-bit integer, whose i-th bit corresponds to the coefficient of degree i. This instruction has latency 7 and throughput 2. This high latency constitutes an important issue when optimizing the assembly code. This will be discussed later.

3. DFTs OVER $\mathbb{F}_{2^{60}}$

In order to perform DFTs efficiently, we are interested in finite fields $\mathbb{F}_{2^k}$ such that $2^k - 1$ admits many small prime factors. This is typically the case [21] when k admits many small prime factors itself. Our favorite example is $k = 60$, also because 60 is only slightly smaller than the bit size 64 of registers on modern architectures.

Using the eight smallest prime divisors of $2^{60}-1$ allows us to perform DFTs up to size $3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 = 17461919475$, which is sufficiently large for most of the applications. We thus begin with building DFTs in size 3, 5, 7, 11, 13, 31, 41, 61, and then combine them using the Good–Thomas and Cooley–Tukey algorithms.

3.1 Basic arithmetic in $\mathbb{F}_{2^{60}}$

The other key advantage of $k=60$ is the following defining polynomial $P(z) = (z^{61}-1)/(z-1) \in \mathbb{F}_2[z]$. Elements of $\mathbb{F}_{2^{60}}$ will thus be seen as polynomials in $\mathbb{F}_2[z]$ modulo P.

For multiplying a and b in $\mathbb{F}_{2^{60}}$, we perform the product $\tilde{a}\,\tilde{b}$ of the preimage polynomials, so that the preimage $\tilde{c}$ of $c = a\,b$ may be obtained as follows

$$\tilde{c} = ((\tilde{a}\,\tilde{b}) \operatorname{rem} (z^{61}-1)) \operatorname{rem} P.$$

The remainder by $z^{61}-1$ may be performed efficiently by using bit shifts and bitwise xor. The final division by P corresponds to a single conditional subtraction of P.

In order to decrease the reduction cost, we allow an even more redundant representation satisfying the minimal requirement that data sizes are $\leqslant 64$ bits. If $\tilde{a}$, $\tilde{b} \in \mathbb{F}_2[z]$ have degrees <64, then $\tilde{c} = \tilde{a}\,\tilde{b}$ may be reduced in-place by $z^{64}-z^3$, using the following macro, where `xmm1` denotes any auxiliary register, and `xmm0` represents a register different from `xmm1` that contains $\tilde{c}$:

```
vpunpckhqdq xmm1, xmm0, xmm0
vpsllq xmm1, xmm1, 3
vpxor xmm0, xmm0, xmm1
```

In input `xmm0` contains the 128-bit packed polynomial, and its 64-bit reduction is stored in its low half in output. Let us mention from here that our implementation does not yet fully exploit vector instructions of the SIMD unit. In fact we only use the 64-bit low half of the 128-bit registers, except for DFTs of small orders as explained in the next paragraphs.

3.2 DFTs of small prime orders

In size $n=3$, it is classical that a DFT needs only one product and one reduction: $\mathrm{DFT}_{\omega,(3)}(a) = (a_0+a_1+a_2,\ a_0+(a_1+a_2)\,\omega+a_2,\ a_0+a_1+(a_1+a_2)\,\omega)$. This strategy generalizes to larger n as follows, *via* the Rader reduction of Section 2.2, that involves a product of the form

$$C(x) = \left(\sum_{i=0}^{\mu-1} b_i\,x^i\right)\left(\sum_{i=0}^{\mu-1} w_i\,x^i\right) \operatorname{rem} (x^\mu-1). \tag{1}$$

The coefficient c_i of degree i of C satisfies:

$$c_i = \sum_{\substack{(k+l) \operatorname{rem} \mu = i \\ k<l}} (b_k+b_l)\,(w_k+w_l) + \sum_{k=0}^{\mu-1} b_k\,w_k. \tag{2}$$

This amounts to $\mu + \mu\,(\mu-1)/2$ products, μ^2 sums (even less if the w_k+w_l are pre-computed), and μ reductions.

We handcrafted these formulas in registers for $n=3$, 5, 7. Products are computed by `vpclmulqdq`. They are not reduced immediately. Instead we perform bitwise arithmetic on 128-bit registers, so that reductions to 64-bit integers are postponed to the end. The following table counts instructions of each type. Precomputed constants are mostly read from memory and not stored in registers. The last column *cycles* contains the number of CPU clock cycles, measured with the CPU instruction `rdtsc`, when running the DFT code in-place on contiguous data already present in the level 1 cache memory.

n	clmul	shift	xor	move	cycles
3	1	2	7	6	19
5	10	8	22	10	37
7	21	12	45	14	58

For $n \geqslant 11$, these computations do not fit into the 16 registers, and using auxiliary memory introduces unwanted overhead. This is why we prefer to use the method described in the next subsection.

3.3 DFTs of larger prime orders

For larger DFTs we still use the Rader reduction (1) but it is worth using Karatsuba's method instead of formula (2). Let $\eta = \mu \operatorname{quo} 2$, and let $\delta_\mu = 1$ if μ is odd and 0 otherwise. We decompose $B(x) = B_0(x) + x^\eta\,B_1(x) + \delta_\mu\,b_{\mu-1}\,x^{\mu-1}$, and $W(x) = W_0(x) + x^\eta\,W_1(x) + \delta_\mu\,w_{\mu-1}\,x^{\mu-1}$, where B_0, B_1, W_0, W_1 have degrees $\leqslant \eta-1$. Then $B(x)\,W(x)$ may be computed as $C_0(x) + x^\eta\,C_1(x) + x^{2\eta}\,C_2(x) + \delta_\mu\,(b_{\mu-1}\,W(x) + w_{\mu-1}\,B(x))$, where $C_0(x) = B_0(x)\,W_0(x)$, $C_2(x) = B_1(x)\,W_1(x)$, and $C_1(x) = (B_0(x)+B_1(x))\,(W_0(x)+W_1(x)) - C_0(x) - C_2(x)$ are obtained by the Karatsuba algorithm.

If μ is odd, then during the recursive calls for C_0, C_1, C_2, we collect $b_i\,w_i$, for $0 \leqslant i < 2\,\eta$. Then we compute $b_{\mu-1}\,w_{\mu-1}$, so that the needed $b_{\mu-1}\,w_i + w_{\mu-1}\,b_i$ are obtained as $(b_{\mu-1}+b_i)\,(w_{\mu-1}+w_i) - b_{\mu-1}\,w_{\mu-1} - b_i\,w_i$.

During the recursive calls, reductions of products are discarded, and sums are performed over 128 bits. The total number of reductions at the end thus equals μ. We use these formulas for $n = 11, 13, 31, 41, 61$. For $n = 5, 7$ this approach leads to fewer products than with the previous method, but the number of sums and moves is higher, as reported in the following table:

n	clmul	shift	xor	move	cycles
5	9	8	34	52	63
7	18	12	76	83	83
11	42	20	184	120	220
13	54	24	244	239	450
31	270	60	1300	971	2300
41	378	80	1798	1156	3000
61	810	120	3988	2927	7300

For readability only the two most significant figures are reported in column *cycles*. The measurements typically vary by up to about 10%. It might be possible to further improve these timings by polishing register allocation, decreasing temporary memory, reducing latency effects, or even by trying other strategies [1, 3].

3.4 DFTs of composite orders

As long as the input data and the pre-computed constants fit into the level 3 cache of the processor (in our case, 8 MB), we may simply unfold Algorithm 1: we do not transpose-copy data in step 1, but rather implement DFT codelets of small prime orders, with suitable input and output strides. For instance, in size $n = 3 \cdot 5$, we perform five in-place DFTs of order 3 with stride 5, namely on (a_0, a_5, a_{10}), (a_1, a_6, a_{11}), ..., (a_4, a_9, a_{14}), then we multiply by those twiddle factors not equal to 1, and finally, we do three in-place DFTs of size 5 and stride 1. In order to minimize the effect of the latency of `vpclmulqdq`, carry-less products by twiddle factors and reductions may be organized in groups of 8, so that the result of each product is available when arriving at its corresponding reduction instructions. More precisely,

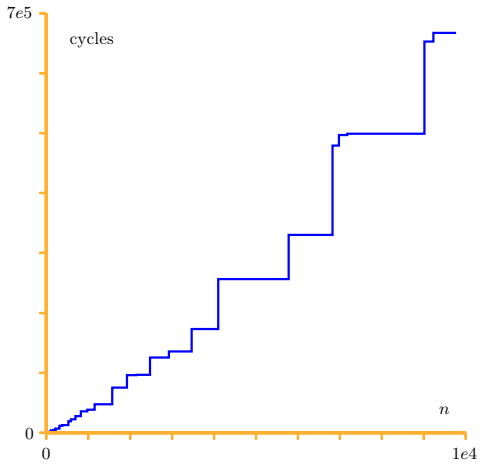

Figure 1. Timings for DFT over $\mathbb{F}_{2^{60}}$, in CPU cycles.

if `rdi` contains the current address to entries in a, and `rsi` the current address to the twiddle factors, then 8 entry-wise products are performed as follows:

```
vmovq xmm0, [rdi+8*0]
vmovq xmm1, [rdi+8*1]
  ...
vmovq xmm7, [rdi+8*7]
vpclmulqdq xmm0, xmm0, [rsi+8*0], 0
vpclmulqdq xmm1, xmm1, [rsi+8*1], 0
  ...
vpclmulqdq xmm7, xmm7, [rsi+8*7], 0
```

Then the contents of `xmm0`,...,`xmm7` are reduced in sequence modulo $z^{64} - z^3$, and finally the results are stored into memory.

Up to sizes $n < 10\,000$ made from primes $\leqslant 41$, we generated executable codes for the Cooley–Tukey algorithm, and measured timings for all the possible orderings v. This revealed that increasing orders, namely $n_1 \leqslant n_2 \leqslant \cdots \leqslant n_d$, yield the best performances. In size $n = 3 \cdot 5 \cdot 7 \cdot 11 = 1155$, one transform takes 44000 CPU cycles, among which 9000 are spent in multiplications by twiddle factors. We implemented the Good–Thomas algorithm in the same way as for Cooley–Tukey, and concluded that it is always faster when data fit into the cache memory. When $n = 1155$, this leads to only 38000 cycles for one transform, for which carryless products contribute 40%.

For each size $m < 10000$, we then deduced the smallest DFT size $n \geqslant m$, together with the best ordering, leading to the fastest calculation *via* the Good–Thomas algorithm. The graph in Figure 1 represents the number of CPU cycles in terms of m obtained in this way. Thanks to the variety of prime orders, the staircase effect is softer than with the classical FFT.

For sizes n larger than a few thousands, using internal DFTs of prime size with large strides is of course a bad idea in terms of memory management. The classical technique, going back to [16], and now known as the *4-step* or *6-step* FFT in the literature, consists in decomposing n into $m_1\, m_2$ such that m_1 and m_2 are of order of magnitude $\sqrt{n}$. In the context of Algorithm 1, with $v = (n_1, \ldots, n_d)$, and $n = n_1 \cdots n_d$, we need to determine $h \in \{1, \ldots, d\}$ such that $m_1 = n_1 \cdots n_h$ and $m_2 = n_{h+1} \cdots n_d$ are the closest to $\sqrt{n}$.

As previously mentioned, for large values of m_1, m_2, step 1 of Algorithm 1 decomposes into the transposition of a $m_1 \times m_2$ matrix (in column representation), followed by m_2 in-place DFTs of size m_1 on contiguous data. Then the transposition is performed backward. In the classical case of the FFT, m_1 and m_2 are powers of two that satisfy $m_1 = m_2$ or $m_1 = 2\, m_2$, and efficient cache-friendly and cache-oblivious solutions are known to transpose such $m_2 \times m_2$ matrices in-place with AVX2 instructions. Unfortunately, we were not able to do so in our present situation. Consequently we simply transpose rows in groups of 4 into a fixed temporary buffer of size $4\, m_2$. Then we may perform 4 DFTs of size m_2 on contiguous data, and finally transpose backward. The threshold for our 4-step DFTs has been estimated in the neighborhood of 7000.

4. OTHER GROUND FIELDS

Usually, fast polynomial multiplication over $\mathbb{F}_2$ is used as the basic building block for other fast polynomial arithmetic over finite fields in characteristic two. It is natural to ask whether we may use our fast multiplication over $\mathbb{F}_{2^{60}}$ as the basic building block. Above all, this requires an efficient way to reduce multiplications in $\mathbb{F}_{2^k}[x]$ with $k \neq 60$ to multiplications in $\mathbb{F}_{2^{60}}[x]$. The optimal algorithms vary greatly with k. In this section, we discuss various possible strategies. Timings are reported for some of them in the next section. In the following paragraphs, $a, b \in \mathbb{F}_{2^k}[x]$ represent the two polynomials of degrees $< d$ to be multiplied, and $P_k \in \mathbb{F}_2[z]$ is the defining polynomial of $\mathbb{F}_{2^k}$ over $\mathbb{F}_2$.

4.1 Lifting and Kronecker segmentation

Case when $k = 1$. In order to multiply two packed polynomials in $\mathbb{F}_2[x]$, we use standard *Kronecker segmentation* and cut the input polynomials into slices of 30 bits. More precisely, we set $m = \lceil d/30 \rceil$, $A(y, x) = \sum_{i=0}^{m-1} \sum_{j=0}^{29} a_{30i+j}\, x^j\, y^i$ and $B(y,\ x) = \sum_{i=0}^{m-1} \sum_{j=0}^{29} b_{30i+j}\ x^j\ y^i$, so that $a(x) = A(x^{30}, x)$ and $b(x) = B(x^{30}, x)$. The product $c = a\, b$ then satisfies $c(x) = C(x^{30}, x)$, where $C = A\, B$. We are thus led to multiply A by B in $\mathbb{F}_{2^{60}}[x]$ by reinterpreting y as the generator of $\mathbb{F}_{2^{60}}$. In terms of the input size, this method admits a constant overhead factor of roughly 2. In fact, when considering algorithms with asymptotically softly linear cost, comparing relative input sizes gives a relevant approximation of the relative costs.

General strategies. It is well known that one can handle any value of k by reduction to the case $k = 1$. Basically, $\mathbb{F}_{2^k}$ is seen as $\mathbb{F}_2[z] \,/\, (P_k(z))$, and a product in $\mathbb{F}_{2^k}[x]$ is lifted to one product in degree $< d$ in $\mathbb{F}_2[z][x]$, with coefficients of degrees $< k$ in z, followed by $2\, d - 1$ divisions by $P_k(z)$. Then, the *Kronecker substitution* [14, Chapter 8] reduces the computation of one such product in $\mathbb{F}_2[z][x]$, said in *bi-degree* $< (k, d)$, to one product in $\mathbb{F}_2[x]$ in degree $< (2\, k - 1)\, d$. In total, we obtain a general strategy to reduce one product in degree $< d$ in $\mathbb{F}_{2^k}$ to one product in $\mathbb{F}_{2^{60}}[x]$ in degree $\leqslant \lceil ((2\, k - 1)\, d) \,/\, 30 \rceil$, plus $O(d\, \mathsf{M}_{2^k}(1))$ bit operations. Roughly speaking, this approach increases input bit sizes by a factor at most 4 in general, but only 3 when $k = 2$.

Instead of Kronecker substitution, we may alternatively use classical *evaluation/interpolation schemes* [23, Section 2]. Informally speaking, a multiplication in $\mathbb{F}_{2^k}[x]$ in degree $< d$ is still regarded as a multiplication of $\tilde{a}$ and $\tilde{b}$ in $\mathbb{F}_2[z][x]$ of bi-degrees $< (k, d)$, but we "specialize" the variable z at sufficiently many "points" in a suitable ring $\mathbb{A}$, then perform products in $\mathbb{A}[x]$ of degrees $< d$, and finally we "interpolate" the coefficients of $\tilde{a}\, \tilde{b}$.

The classical Karatsuba transform is such a scheme. For instance, if $k = 2$ then $\alpha(z) = \alpha_0 + \alpha_1\, z$ is sent to $(\alpha_0, \alpha_0 + \alpha_1, \alpha_1)$, which corresponds to the projective evaluation at $(0, 1, \infty)$. The size overhead of this reduction is 3, but its advantage to the preceding general strategy is the splitting of the initial product into smaller independent products. The Karatsuba transform can be iterated, and even Toom–Cook transforms of [3] might be used to handle small values of k.

Another standard evaluation scheme concerns the case when k is sufficiently large. Each coefficient $\alpha(z)$ of $\tilde{a}$, $\tilde{b}$ is "evaluated" *via* one DFT of size $n \geqslant \lceil (2\ k\ -\ 1)\,/\,30 \rceil$ applied to the Kronecker segmentation of $\alpha(z)$ into slices of 30 bits seen in $\mathbb{F}_{2^{60}}[y]$. Then we perform n products in $\mathbb{F}_{2^{60}}[x]$ in degree $<d$, and "interpolate" the output coefficients by means of inverse DFTs. Asymptotically, the size overhead is again 4.

The rest of this section is devoted to other reduction strategies involving a size growth less than 4 in various particular cases.

4.2 Field embedding

Case when $k \,|\, 60$. When $k \geqslant 2$ and $k \,|\, 60$, which corresponds to $k = 2$, 3, 4, 5, 6, 10, 12, 15, 20, 30, we may embed $\mathbb{F}_{2^k}$ into $\mathbb{F}_{2^{60}}$, by regarding $\mathbb{F}_{2^{60}}$ as a field extension of $\mathbb{F}_{2^k}$. The input polynomials are now cut into slices of degrees $<m$ with $m = \lfloor (60\,/\,k + 1)\,/\,2 \rfloor$. The overhead of this method is asymptotically $60\,/\,(m\ k) \leqslant 2$. If $k = 4$, 12, 20, then $60\,/\,k$ is odd, $m = (60\,/\,k\ +\ 1)\,/\,2$, and the overhead is <2. In particular it is only $3\,/\,2$ for $k = 20$.

Let Q be an irreducible factor of P in $\mathbb{F}_{2^k}[x]$, of degree $60\,/\,k$. Elements of $\mathbb{F}_{2^{60}}$ can thus be represented by polynomials of degrees $<60\,/\,k$ in $\mathbb{F}_{2^k}[y]$, modulo Q, *via* the following isomorphism

$$\Phi\colon \mathbb{F}_2[z, y]\,/\,(P_k(z), \check{Q}(z, y)) \to \mathbb{F}_2[z]\,/\,(P(z)),$$

where $\check{Q}(z,\ y)$ represents the canonical preimage of Q in $\mathbb{F}_2[z, y]$. Let $f(z, y)$ be a polynomial of bi-degree $<(k, m)$ representing an element of $\mathbb{F}_{2^k}[y]$, obtained as a slice of a or b. The image $\Phi(f)$ can be obtained as $f(\Phi(z),\ \Phi(y))$. Consequently, if we pre-compute $\Phi(z^j\, y^i)$ for all $j < k$ and $i < m$, then deducing $\Phi(f)$ requires 59 sums in $\mathbb{F}_2[z]\,/\,(P(z))$.

The number of sums may be decreased by using larger look-up tables. Let b_0, ..., b_{59} represent the basis $z^j\, y^i$ for $j < k$ and $i < m$, whatever the order is. Then all the $\{\Phi(\sum_{j=0}^{5} \beta_{6i+j}\, b_{6i+j}) |\ (\beta_{6i},\ ...,\ \beta_{6i+5}) \in \mathbb{F}_2^6\}$ for $0 \leqslant i < 10$ can be stored in 10 look-up tables of size 64, which allows $\Phi(f)$ to be computed using 9 sums. Of course the cost for reordering and packing bits of elements of $\mathbb{F}_{2^k}[y]$ must be taken into account, and sizes of look-up tables must be carefully adjusted in terms of k. The details are omitted.

Conversely, let $g \in \mathbb{F}_2[z]\,/\,(P(z))$. We wish to compute $\Phi^{-1}(g)$. As for direct images of Φ, we may pre-compute $\Phi^{-1}(z^i)$ for all $i < 60$ and reduce the computation of $\Phi^{-1}(g)$ to 59 sums in $\mathbb{F}[z, y]\,/\,(P_k(z), \check{Q}(z, y))$. Larger look-up tables can again help to reduce the number of sums.

Let us mention that the use of look-up tables cannot benefit from SIMD technologies. On the current platform this is not yet a problem: fully vectorized arithmetic in $\mathbb{F}_{2^{60}}[z,\ y]\,/\,(P_k(z),\ \check{Q}(z,\ y))$ would also require SIMD versions of carry-less multiplication which are not available at present.

Case when $\gcd(60,\ k) \geqslant 2$. We may combine the previous strategies when k is not coprime with 60. Let $t = \gcd(60, k)$. We rewrite elements of $\mathbb{F}_{2^k}[x]$ into polynomials in $\mathbb{F}_{2^t}[y][x]$ of bi-degrees $<(k\,/\,t, d)$, and then use the Kronecker substitution to reduce to one product in $\mathbb{F}_{2^t}[x]$ in degree $(2\ k\,/\,t\ -\ 1)\ d$. For example, when $60 \mid k$, elements of $\mathbb{F}_{2^k}$ may be represented by polynomials in $\mathbb{F}_{2^{60}}[y]$. We are thus led to multiplying polynomials in $\mathbb{F}_{2^{60}}[y][x]$ in bi-degree $<(k\,/\,60, d)$, which yields an overhead of 2. Another favorable case is when $20 \,|\, k$, yielding an overhead of 3.

4.3 Double-lifting

When $k \leqslant 30$, we may of course directly lift products in $\mathbb{F}_{2^k}[x]$ to products in $\mathbb{F}_2[z][x]$ of output degree in z at most 58. The latter products may thus be computed as products in $\mathbb{F}_{2^{60}}[x]$. This strategy has an overhead of $60\,/\,k$, so it outperforms the general strategies for $k \geqslant 16$.

For $31 \leqslant k \leqslant 59$, we lift products to $\mathbb{F}_2[z][x]$ and multiply in $(\mathbb{F}_2[z]\,/\,(P(z)))[x]$ and $(\mathbb{F}_2[z]\,/\,(P(z+1)))[x]$. We then perform Chinese remaindering to deduce the product modulo $P(z)\,P(z+1)$.

Multiplying A and B in $(\mathbb{F}[z]\,/\,(P(z\ +\ 1)))[x]$ may be obtained as $(A(z+1, x)\ B(z+1, x) \operatorname{rem} P(z))(z+1)$. In this way all relies on the same DFT routines over $\mathbb{F}_{2^{60}}$.

If $\alpha(z) \in \mathbb{F}_2[z]$ has degree $<l$, with l a power of 2, we decompose $\alpha(z) = \alpha_0(z) + z^{l/2}\ \alpha_1(z)$ with α_0 and α_1 of degrees $<l\,/\,2$, then $\alpha(z\ +\ 1)$ is obtained efficiently by the following induction:

$$\begin{aligned} \alpha(z+1) &= \alpha_0(z+1) + (z+1)^{l/2}\,\alpha_1(z+1) \\ &= \alpha_0(z+1) + \alpha_1(z+1) + z^{l/2}\,\alpha_1(z+1). \end{aligned} \tag{3}$$

Chinese remaindering can also be done efficiently. Let $U(z) = z^2\ +\ 1$ be the inverse of $P(z\ +\ 1)$ modulo $P(z)$. Residues $\gamma_0(z) \bmod P(z)$ and $\gamma_1(z) \bmod P(z+1)$ then lift into $\gamma_1(z) + (U(z)\ (\gamma_0(z) - \gamma_1(z)) \operatorname{rem} P(z))\ P(z+1)$ modulo $P(z)\ P(z\ +\ 1)$. Since $\deg\ \gamma_i \leqslant 58$, this formula involves only one carry-less product. Asymptotically, the overhead of this method is $2 < 120\,/\,k < 4$.

On our platform, formula (3) may be implemented in degree <64 as follows. Assume that `xmm0` contains `101010...10`, `xmm1` contains `11001100...1100`, `xmm2` contains `11110000...11110000`, ..., and `xmm5` is filled with `11...1100...00`. Then, using `xmm15` for input and output, we simply do

```
vpand xmm14, xmm15, xmm0
vpsrlq xmm14, xmm14, 1
vpxor xmm15, xmm14, xmm15
vpand xmm14, xmm15, xmm1
vpsrlq xmm14, xmm14, 2
vpxor xmm15, xmm14, xmm15
  ...
vpand xmm14, xmm15, xmm5
vpsrlq xmm14, xmm14, 32
vpxor xmm15, xmm14, xmm15
```

4.4 The Crandall–Fagin reduction

For "lucky" $60 < k < 2^{30}$ such that $(x^{k+1} - 1)\,/\,(x - 1)$ is irreducible over $\mathbb{F}_2$, multiplication in $\mathbb{F}_{2^k}$ reduces to cyclic multiplication over $\mathbb{F}_2$ of length $k + 1$. Using our adaptation [21] of Crandall–Fagin's algorithm [6], multiplying two polynomials of degrees $<d$ in $\mathbb{F}_{2^k}[x]$ therefore reduces to one product in $(\mathbb{F}_{2^{60}}[y]\,/\,(y^m - 1))[x]$ in degree $<d$, where m is the first integer such that $\lceil (k+1)\,/\,m \rceil \leqslant 30$. The asymptotic overhead of this method is ≈ 2. This strategy generalizes to the case when P_k divides $x^{k+r} - 1$, with $r \ll k$.

For various $k > 60$, the polynomial $(x^{k+1} - 1)\,/\,(x - 1)$ factors into r irreducible polynomials of degree $k\,/\,r$. In that case, we may use the previous strategy to reduce r

$\log_2 d$	16	17	18	19	20	21
us	13	28	52	140	290	640
GF2X	79	160	400	800	1600	3500

Table 1. Products in degree $<d$ in $\mathbb{F}_{2^{60}}[x]$, in milliseconds.

$\log_2 d/64$	16	17	18	19	20	21
us	29	70	170	300	730	1300
GF2X	39	89	190	490	900	2100
FLINT	840	2200	4400	11000	...	

Table 2. Products in degree $<d$ in $\mathbb{F}_2[x]$, in milliseconds.

products in $\mathbb{F}_{2^{k/r}}[x]$ to multiplications in $\mathbb{F}_{2^{60}}[x]$, using r different defining polynomials of $\mathbb{F}_{2^{k/r}}$. Asymptotically, the overhead reaches again ≈ 2, although working with different defining polynomials of $\mathbb{F}_{2^{k/r}}$ probably involves another non trivial overhead in practice. Again, the strategy generalizes to the case when $(x^{k+1}-1)/(x-1)$ admits r irreducible factors of degree $k/r-\delta$ with $\delta \ll k/r$.

5. TIMINGS

Our polynomial products are implemented in the JUSTINLINE library of MATHEMAGIX. The source code is freely available from revision 10434 of our SVN server (`http://gforge.inria.fr/projects/mmx/`). Sources for DFTs over $\mathbb{F}_{2^{60}}$ are in the file `dft_f2_60_amd64_avx_clmul.mmx`. Those for our polynomial products in $\mathbb{F}_{2^k}[x]$ are in `polynomial_f2k_amd64_avx_clmul.mmx`. Let us recall here that MATHEMAGIX functions can also be easily exported to C++ [24].

We use version 1.1 of the GF2X library, tuned to our platform during installation. The top level function, named `gf2x_mul`, multiplies packed polynomials in $\mathbb{F}_2[x]$, and makes use of the carry-less product instruction. Triadic variants of the Schönhage–Strassen algorithm start to be used from degree $64 \cdot 6485$. GF2X can be used from versions $\geqslant$ 5.5 of the NTL library [34], that uses Kronecker substitution to multiply in $\mathbb{F}_{2^k}[x]$. Consequently, we do not need to compare to NTL. We also compare to FLINT 2.5.2, tuned according to §13 of the documentation.

5.1 $\mathbb{F}_2[x]$ and $\mathbb{F}_{2^{60}}[x]$

Table 1 displays timings for multiplying polynomials of degrees $<d$ in $\mathbb{F}_{2^{60}}[x]$. The row "us" concerns the natural product *via* DFTs, the other row is the running time of `gf2x_mul` used to multiply polynomials in $\mathbb{F}_2[x]$ built from Kronecker substitution.

In Table 2, we report on timings for multiplying polynomials of degrees $<d$ in $\mathbb{F}_2[x]$. The row "us" concerns our DFT based implementation *via* Kronecker segmentation, as recalled in Section 4.1. The row GF2X is the running time for the direct call to `gf2x_mul`. The row FLINT concerns the performance of the function `nmod_poly_mul`, which reduces to products in $\mathbb{Z}[x]$ *via* Kronecker substitution. Since the packed representation is not implemented, we could not obtain timings until the end. This comparison is not intended to be fair, but rather to show unfamiliar readers why dedicated algorithms for $\mathbb{F}_2[x]$ may be worth it.

In both cases, our DFT based products turn out to be more efficient than the triadic version of the Schönhage–Strassen of GF2X, for large degrees. One major advantage of the DFT based approach concerns linear algebra. Instead of multiplying $r \times r$ matrices over $\mathbb{F}_{2^{60}}[x]$ naively

r	1	2	4	8	16	32
us	29	120	500	2200	9800	48000
GF2X	39	320	2500	20000	160000	1300000

Table 3. Products of $r \times r$ matrices over $\mathbb{F}_2[x]$, in degree $<64 \cdot 2^{16}$, in milliseconds.

$\log_2 d$	16	17	18	19	20	21
us	28	58	110	290	600	1300
GF2X	76	160	400	790	1600	3400

Table 4. Products in degree $<d$ in $\mathbb{F}_{2^{59}}[x]$, in milliseconds.

$\log_2 d/32$	16	17	18	19	20	21
us	30	73	170	310	750	1400
GF2X	54	120	270	570	1500	2700

Table 5. Products in degree $<d$ in $\mathbb{F}_{2^2}[x]$, in milliseconds.

in time $O(r^3\,\mathsf{M}_{2^{60}}(d))$, we compute the $3\,r^2$ DFTs and $\approx 2\,d$ products of $r \times r$ matrices over $\mathbb{F}_{2^{60}}$, in time $O(r^2\,\mathsf{M}_{2^{60}}(d) + r^3\,d)$. Matrix multiplication over $\mathbb{F}_{2^k}[x]$ is reduced to matrix multiplication over $\mathbb{F}_{2^{60}}[x]$ using similar techniques as in Section 4. Timings for matrices over $\mathbb{F}_2[x]$, reported in Table 3, confirm the practical gain.

5.2 $\mathbb{F}_4[x]$ and $\mathbb{F}_{2^{59}}[x]$

Table 4 displays timings for multiplying polynomials of degrees $<d$ in $\mathbb{F}_{2^{59}}[x]$. The row "us" concerns the *double-lifting* strategy of Section 4.3. The next row is the running time of `gf2x_mul` used to multiply polynomials in $\mathbb{F}_2[x]$ built from Kronecker substitution. As expected, timings for GF2X are similar to those of Table 1, and the overhead with respect to $\mathbb{F}_{2^{60}}$ is close to 2. Overall, our implementation is about twice as fast than *via* GF2X.

Table 5 finally concerns our implementation of the strategy from Section 4.2 for products in degree $<d$ in $\mathbb{F}_4[x]$. As expected, timings are similar to those of Table 2 when input sizes are the same. We compare to the sole time needed by `gf2x_mul` used as follows: we rewrite the product in $\mathbb{F}_4[x]$ into a product in $\mathbb{F}_2[x][y]/(y^2+y+1)$, which then reduces to 3 products in $\mathbb{F}_2[x]$ in degrees $\leqslant d/2$ thanks to Karatsuba's formula.

6. CONCLUSION

We are pleased to observe that key ideas from [20, 21] turn out to be of practical interest even for polynomials in $\mathbb{F}_{2^k}[x]$ of medium sizes. Besides Schönhage–Strassen-type algorithms, other strategies such as the *additive Fourier transform* have been developed for $\mathbb{F}_{2^k}[x]$ [13, 27], and it would be worth experimenting them carefully in practice.

Let us mention a few plans for future improvements. First, vectorizing our code should lead to a significant speed-up. However, in our implementation of multiplication in $\mathbb{F}_2[x]$, we noticed that about the third of the time is spent in carry-less products. Since `vpclmulqdq` does not admit a genuine vectorized counterpart over 256-bit registers, we cannot hope for a speed-up of two by fully exploiting the AVX-256 mode. Then, the graph from Figure 1 can probably be further smoothed by adapting the truncated Fourier transform algorithm [22]. We are also investigating further accelerations of DFTs of prime orders in Section 3.3. For instance, for $n = 7$ and $\mu = 6$, we might exploit the factorization $z^6 - 1 = (z+1)^2\,(z^2+z+1)^2$ in order to compute cyclic

products using Chinese remaindering. In the longer run, we finally expect the approach in this paper to be generalizable to finite fields of higher characteristic $3, 5, 7, 11, ...$

7. REFERENCES

[1] S. Ballet and J. Pieltant. On the tensor rank of multiplication in any extension of $\mathbb{F}_2$. *J. Complexity*, 27(2):230–245, 2011.

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

[3] M. Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields*, volume 4547 of *Lect. Notes Comput. Sci.*, pages 116–133. Springer Berlin Heidelberg, 2007.

[4] 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.

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

[6] R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comp.*, 62(205):305–324, 1994.

[7] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. *SIAM J. Comput.*, 42(2):685–699, 2013.

[8] P. Duhamel and M. Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, 1990.

[9] A. Fog. *Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Number 4 in Optimization manuals. `http://www.agner.org`, Technical University of Denmark, 1996–2016.

[10] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proc. IEEE*, 93(2):216–231, 2005.

[11] M. Fürer. Faster integer multiplication. In *Proceedings of the Thirty-Ninth ACM Symposium on Theory of Computing, STOC 2007*, pages 57–66, New York, NY, USA, 2007. ACM Press.

[12] M. Fürer. Faster integer multiplication. *SIAM J. Comp.*, 39(3):979–1005, 2009.

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

[14] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, second edition, 2003.

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

[16] W. M. Gentleman and G. Sande. Fast Fourier transforms: For fun and profit. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 563–578. ACM Press, 1966.

[17] I. J. Good. The interaction algorithm and practical Fourier analysis. *J. R. Stat. Soc. Ser. B*, 20(2):361–372, 1958.

[18] T. Granlund et al. GMP, the GNU multiple precision arithmetic library. `http://gmplib.org`, from 1991.

[19] W. Hart et al. FLINT: Fast Library for Number Theory. `http://www.flintlib.org`, from 2008.

[20] D. Harvey, J. van der Hoeven, and G. Lecerf. Even faster integer multiplication. `http://arxiv.org/abs/1407.3360`, 2014.

[21] D. Harvey, J. van der Hoeven, and G. Lecerf. Faster polynomial multiplication over finite fields. `http://arxiv.org/abs/1407.3361`, 2014.

[22] 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 Press, 2004.

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

[24] 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 Press, 2013.

[25] J. van der Hoeven, G. Lecerf, and G. Quintin. Modular SIMD arithmetic in Mathemagix. `http://hal.archives-ouvertes.fr/hal-01022383`, 2014.

[26] Intel Corporation, 2200 Mission College Blvd., Santa Clara, CA 95052-8119, USA. *Intel (R) Architecture Instruction Set Extensions Programming Reference*, 2015. Ref. 319433-023, `http://software.intel.com`.

[27] 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.

[28] C. Lüders. Implementation of the DKSS algorithm for multiplication of large numbers. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '15, pages 267–274. ACM Press, 2015.

[29] L. Meng and J. Johnson. High performance implementation of the TFT. In K. Nabeshima, editor, *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 328–334. ACM, 2014.

[30] J. M. Pollard. The fast Fourier transform in a finite field. *Math. Comp.*, 25(114):365–374, 1971.

[31] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. IEEE*, 56(6):1107–1108, 1968.

[32] A. Schönhage. Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. *Acta Infor.*, 7(4):395–398, 1977.

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

[34] V. Shoup. *NTL: A Library for doing Number Theory*, 2015. Software, version 9.6.2. `http://www.shoup.net`.

[35] L. H. Thomas. Using a computer to solve problems in physics. In W. F. Freiberger and W. Prager, editors, *Applications of digital computers*, pages 42–57. Boston, Ginn, 1963.

[36] S. Winograd. On computing the discrete Fourier transform. *Math. Comp.*, 32:175–199, 1978.