HAL
archives-ouvertes.fr

Faster Multiplication in GF(2)[x]

Richard Brent, Pierrick Gaudry, Emmanuel Thomé, Paul Zimmermann

To cite this version:

Richard Brent, Pierrick Gaudry, Emmanuel Thomé, Paul Zimmermann. Faster Multiplication in GF(2)[x]. van der Poorten, Alfred and Stein, Andreas. ANTS-VIII, May 2008, Banff, Canada. Springer-Verlag, 5011, pp.153-166, 2008, Lecture notes in computer science. <10.1007/978-3-540-79456-1>. <inria-00188261v4>

HAL Id: inria-00188261

https://hal.inria.fr/inria-00188261v4

Submitted on 7 Nov 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Faster Multiplication in GF(2)[x]

Richard P. Brent[1], Pierrick Gaudry[2],
Emmanuel Thomé[3], and Paul Zimmermann[3]

[1] Australian National University, Canberra, Australia
[2] LORIA/CNRS, Vandœuvre-lès-Nancy, France
[3] INRIA Nancy - Grand Est, Villers-lès-Nancy, France

Abstract. In this paper, we discuss an implementation of various algorithms for multiplying polynomials in GF(2)[x]: variants of the window methods, Karatsuba's, Toom-Cook's, Schönhage's and Cantor's algorithms. For most of them, we propose improvements that lead to practical speedups.

Introduction

The arithmetic of polynomials over a finite field plays a central role in algorithmic number theory. In particular, the multiplication of polynomials over GF(2) has received much attention in the literature, both in hardware and software. It is indeed a key operation for cryptographic applications [22], for polynomial factorisation or irreducibility tests [8, 3]. Some applications are less known, for example in integer factorisation, where multiplication in GF(2)[x] can speed up Berlekamp-Massey's algorithm inside the (block) Wiedemann algorithm [20, 1].

We focus here on the classical dense representation — called "binary polynomial" — where a polynomial of degree $n-1$ is represented by the bit-sequence of its n coefficients. We also focus on software implementations, using classical instructions provided by modern processors, for example in the C language.

Several authors already made significant contributions to this subject. Apart from the classical $O(n^2)$ algorithm, and Karatsuba's algorithm which readily extends to GF(2)[x], Schönhage in 1977 and Cantor in 1989 proposed algorithms of complexity $O(n \log n \log\log n)$ and $O(n(\log n)^{1.5849\ldots})$ respectively [18, 4]. In [16], Montgomery invented Karatsuba-like formulæ splitting the inputs into more than two parts; the key feature of those formulæ is that they involve no division, thus work over any field. More recently, Bodrato [2] proposed good schemes for Toom-Cook 3, 4, and 5, which are useful cases of the Toom-Cook class of algorithms [7, 21]. A detailed bibliography on multiplication and factorisation in GF(2)[x] can be found in [9].

Discussions on implementation issues are found in some textbooks such as [6, 12]. On the software side, von zur Gathen and Gerhard [9] designed a software tool called BiPolAr, and managed to factor polynomials of degree up to 1 000 000, but BiPolAr no longer seems to exist. The reference implementation for the last decade is the NTL library designed by Victor Shoup [19].

A.J. van der Poorten and A. Stein (Eds.): ANTS-VIII 2008, LNCS 5011, pp. 153–166, 2008.
© Springer-Verlag Berlin Heidelberg 2008

The contributions of this paper are the following: (a) the "double-table" algorithm for the word-by-word multiplication and its extension to two words using the SSE-2 instruction set (§1); (b) the "word-aligned" variants of the Toom-Cook algorithm (§2); (c) a new view of Cantor's algorithm, showing in particular that a larger base field can be used, together with a truncated variant avoiding the "staircase effect" (§3.1); (d) a variant of Schönhage's algorithm (§3.2) and a splitting technique to improve it (§3.3); (e) finally a detailed comparison of our implementation with previous literature and current software (§4).

Notation: w denotes the machine word size (usually $w = 32$ or 64), and we consider polynomials in $\mathrm{GF}(2)[x]$. A polynomial of degree less than d is represented by a sequence of d bits, which are stored in $\lceil d/w \rceil$ consecutive words.

The code that we developed for this paper, and for the paper [3], is contained in the GF2X package, available under the GNU General Public License from `http://wwwmaths.anu.edu.au/~brent/gf2x.html`.

1 The Base Case (Small Degree)

We first focus on the "base case", that is, routines that multiply full words (32, 64 or 128 bits). Such routines eventually act as building blocks for algorithms dealing with larger degrees. Since modern processors do not provide suitable hardware primitives, one has to implement them in software.

Note that the treatment of "small degree" in general has also to deal with sizes which are not multiples of the machine word size: what is the best strategy to multiply, e.g., 140-bit polynomials? This case is not handled here.

1.1 Word by Word Multiplication (`mul1`)

Multiplication of two polynomials $a(x)$ and $b(x)$ of degree at most $w-1$ can be performed efficiently with a "window" method, similar to base-2^s exponentiation, where the constant s denotes the window size. This algorithm was found in version 5.1a of NTL, which used $s = 2$, and is here generalized to any value of s:

1. Store in a table the multiples of b by all 2^s polynomials of degree $< s$.
2. Scan bits of a, s at a time. The corresponding table data are shifted and accumulated in the result.

Note that Step 1 discards the high coefficients of $b(x)$, which is of course undesired[1], if $b(x)$ has degree $w-1$. The computation must eventually be "repaired" with additional steps which are performed at the end.

The "repair step" (Step 3) exploits the following observation. Whenever bit $w-j$ of b is set (where $0 < j < s$), then bits at position j' of a, where $j' \bmod s \geq j$, contribute to a missing bit at position $w + j' - j$ in the product. Therefore only the high result word has to be fixed. Moreover, for each j, $0 < j < s$,

[1] The multiples of b are stored in one word, i.e., modulo 2^w; alternatively, one could store them in two words, but that would be much slower.

the fixing can be performed by an exclusive-or involving values easily derived from a: selecting bits at indices j' with $j' \bmod s \geq j$ can be done inductively by successive shifts of a, masked with an appropriate value.

```
mul1(ulong a, ulong b)
multiplies polynomials a and b. The result goes in l (low part) and h (high part).
    ulong u[2^s] = { 0, b, 0, ... };                    /* Step 1 (tabulate) */
    for(int i = 2 ; i < 2^s ; i += 2)
        u[i] = u[i >> 1] << 1; u[i + 1] = u[i] ^ b;
    ulong g = u[a & (2^s − 1)], l = g, h = 0;           /* Step 2 (multiply) */
    for(int i = s ; i < w ; i += s)
        g = u[a >> i & (2^s − 1)]; l ^= g << i; h ^= g >> (w - i);
    ulong m = (2^s − 2) × (1 + 2^s + 2^2s + 2^3s + ···) mod 2^w; /* Step 3 (repair) */
    for(int j = 1 ; j < s ; j++)
        a = (a << 1) & m;
        if (bit w − j of b is set) h ^= a;
    return l, h;
```

Fig. 1. Word-by-word multiplication with repair steps

The pseudo-code in Fig. 1 illustrates the word-by-word multiplication algorithm (in practice s and w will be fixed for a given processor, thus the for-loops will be replaced by sequences of instructions). There are many alternatives for organizing the operations. For example, Step 1 can also be performed with a Gray code walk. In Step 2, the bits of a may be scanned either from right to left, or in reverse order. For an efficient implementation, the `if` statement within Step 3 should be replaced by a masking operation to avoid branching[2]:

```
h ^= a & -(((long) (b << (j-1))) < 0);
```

A non trivial improvement of the repair steps comes from the observation that Steps 2 and 3 of Fig. 1 operate associatively on the result registers `l` and `h`. The two steps can therefore be swapped. Going further, Step 1 and the repair steps are independent. Interleaving of the code lines is therefore possible and has actually been found to yield a small speed improvement. The GF2X package includes an example of such an interleaved code.

The double-table algorithm. In the `mul1` algorithm above, the choice of the window size s is subject to some trade-off. Step 1 should not be expanded unreasonably, since it costs 2^s, both in code size and memory footprint. It is possible, without modifying Step 1, to operate as if the window size were $2s$ instead of s. Within Step 2, replace the computation of the temporary variable `g` by:

[2] In the C language, the expression $(x < 0)$ is translated into the `setb` x86 assembly instruction, or some similar instruction on other architectures, which does not perform any branching.

$$\texttt{g = u[a >> i \& } (2^s - 1)\texttt{] ^ u[a >> (i+s) \& } (2^s - 1)\texttt{] << s}$$

so that the table is used twice to extract $2s$ bits (the index i thus increases by $2s$ at each loop). Step 1 is faster, but Step 2 is noticeably more expensive than if a window size of $2s$ were effectively used.

A more meaningful comparison can be made with window size s: there is no difference in Step 1. A detailed operation count for Step 2, counting loads as well as bitwise operations `&`, `^`, `<<`, and `>>` yields 7 operations for every s bits of inputs for the code of Fig. 1, compared to 12 operations for every $2s$ bits of input for the "double-table" variant. A tiny improvement of 2 operations for every $2s$ bits of input is thus obtained. On the other hand, the "double-table" variant has more expensive repair steps. It is therefore reasonable to expect that this variant is worthwhile only when s is small, which is what has been observed experimentally (an example cut-off value being $s = 4$).

1.2 Extending to a `mul2` Algorithm

Modern processors can operate on wider types, for instance 128-bit registers are accessible with the SSE-2 instruction set on the Pentium 4 and Athlon 64 CPUs. However, not all operations are possible on these wide types. In particular, arithmetic shifts by arbitrary values are not supported on the full 128-bit registers with SSE-2. This precludes a direct adaptation of our `mul1` routine to a `mul2` routine (at least with the SSE-2 instruction set). We discuss here how to work around this difficulty in order to provide an efficient `mul2` routine.

To start with, the algorithm above can be extended in a straightforward way so as to perform a $k \times 1$ multiplication (k words by one word). Step 1 is unaffected by this change, since it depends only on the second operand. In particular, a 2×1 multiplication can be obtained in this manner.

Following this, a 2×2 `mul2` multiplication is no more than two 2×1 multiplications, where only the second operand changes. In other words, those two multiplications can be performed in a "single-instruction, multiple-data" (SIMD) manner, which corresponds well to the spirit of the instruction set extensions introducing wider types. In practice, a 128-bit wide data type is regarded as a vector containing two 64-bit machine words. Two 2×1 multiplications are performed in parallel using an exact translation of the code in Fig. 1. The choice of splitting the wide register into two parts is fortunate in that all the required instructions are supported by the SSE-2 instruction set.

1.3 Larger Base Case

To multiply two binary polynomials of n words for small n, it makes sense to write some special code for each value of n, as in the NTL library, which contains hard-coded Karatsuba routines for $2 \leq n \leq 8$ [19,22]. We wrote such hard-coded routines for $3 \leq n \leq 9$, based on the above `mul1` and `mul2` routines.

2 Medium Degree

For medium degrees, a generic implementation of Karatsuba's or Toom-Cook's algorithm has to be used. By "generic" we mean that the number n of words of the input polynomials is an argument of the corresponding routine. This section shows how to use Toom-Cook without any extension field, then discusses the word-aligned variant, and concludes with the unbalanced variant.

2.1 Toom-Cook without Extension Field

A common misbelief is that Toom-Cook's algorithm cannot be used to multiply binary polynomials, because Toom-Cook 3 (TC3) requires 5 evaluation points, and we have only 3, with both elements of GF(2) and ∞. In fact, any power of the transcendental variable x can be used as evaluation point. For example TC3 can use $0, 1, \infty, x, x^{-1}$. This was discovered by Michel Quercia and the last author a few years ago, and implemented in the `irred-ntl` patch for NTL [23]. This idea was then generalized by Bodrato [2] to any polynomial in x; in particular Bodrato shows it is preferable to choose $0, 1, \infty, x, 1 + x$ for TC3.

A small drawback of using polynomials in x as evaluation points is that the degrees of the recursive calls increase slightly. For example, with points $0, 1, \infty, x, x^{-1}$ to multiply two polynomials of degree less than $3n$ by TC3, the evaluations at x and x^{-1} might have up to $n + 2$ non-zero coefficients. In any case, this will increase the size of the recursive calls by at most one word.

For Toom-Cook 3-way, we use Bodrato's code; and for Toom-Cook 4-way, we use a code originally written by Marco Bodrato, which we helped to debug[3].

2.2 Word-Aligned Variants

In the classical Toom-Cook setting over the integers, one usually chooses 0, 1, 2, 1/2, ∞ for TC3. The word-aligned variant uses 0, 1, 2^w, 2^{-w}, ∞, where w is the word-size in bits. This idea was used by Michel Quercia in his `numerix` library[4], and was independently rediscovered by David Harvey [13]. The advantage is that no shifts have to be performed in the evaluation and interpolation phases, at the expense of a few extra words in the recursive calls.

The same idea can be used for binary polynomials, simply replacing 2 by x. Our implementation **TC3W** uses 0, 1, x^w, x^{-w}, ∞ as evaluation points (Fig. 2). Here again, there is a slight increase in size compared to using x and x^{-1}: polynomials of $3n$ words will yield two recursive calls of $n+2$ words for x^w and x^{-w}, instead of $n+1$ words for x and x^{-1}. The interpolation phase requires two exact divisions by $x^w + 1$, which can be performed very efficiently.

2.3 Unbalanced Variants

When using the Toom-Cook idea to multiply polynomials $a(x)$ and $b(x)$, it is not necessary to assume that $\deg a = \deg b$. We only need to evaluate $a(x)$ and

[3] `http://bodrato.it/toom-cook/binary/`

[4] `http://pauillac.inria.fr/~quercia/cdrom/bibs/`, version 0.21a, March 2005.

TC3W`(a, b)`
Multiplies polynomials $A = a_2X^2 + a_1X + a_0$ *and* $B = b_2X^2 + b_1X + b_0$ *in* $\mathrm{GF}(2)[x]$
`Let` $W = x^w$ `(assume` X `is a power of` W `for efficiency).`
$c_0 \leftarrow a_1W + a_2W^2$, $c_4 \leftarrow b_1W + b_2W^2$, $c_5 \leftarrow a_0 + a_1 + a_2$, $c_2 \leftarrow b_0 + b_1 + b_2$
$c_1 \leftarrow c_2 \times c_5$, $c_5 \leftarrow c_5 + c_0$, $c_2 \leftarrow c_2 + c_4$, $c_0 \leftarrow c_0 + a_0$
$c_4 \leftarrow c_4 + b_0$, $c_3 \leftarrow c_2 \times c_5$, $c_2 \leftarrow c_0 \times c_4$, $c_0 \leftarrow a_0 \times b_0$
$c_4 \leftarrow a_2 \times b_2$, $c_3 \leftarrow c_3 + c_2$, $c_2 \leftarrow c_2 + c_0$, $c_2 \leftarrow c_2/W + c_3$
$c_2 \leftarrow (c_2 + (1 + W^3)c_4)/(1 + W)$, $c_1 \leftarrow c_1 + c_0$, $c_3 \leftarrow c_3 + c_1$
$c_3 \leftarrow c_3/(W^2 + W)$, $c_1 \leftarrow c_1 + c_2 + c_4$, $c_2 \leftarrow c_2 + c_3$
`Return` $c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0$.

Fig. 2. Word-aligned Toom-Cook 3-way variant (all divisions are exact)

$b(x)$ at $\deg a + \deg b + 1$ points in order to be able to reconstruct the product $a(x)b(x)$. This is pointed out by Bodrato [2], who gives (amongst others) the case $\deg a = 3$, $\deg b = 1$. This case is of particular interest because in sub-quadratic polynomial GCD algorithms, of interest for fast polynomial factorisation [3, 17], it often happens that we need to multiply polynomials a and b where the size of a is about twice the size of b.

We have implemented a word-aligned version **TC3U** of this case, using the same evaluation points 0, 1, x^w, x^{-w}, ∞ as for **TC3W**, and following the algorithm given in [2, p. 125]. If a has size $4n$ words and b has size $2n$ words, then one call to **TC3U** reduces the multiplication $a \times b$ to 5 multiplications of polynomials of size $n + O(1)$. In contrast, two applications of Karatsuba's algorithm would require 6 such multiplications, so for large n we expect a speedup of about 17% over the use of Karatsuba's algorithm.

3 Large Degrees

In this section we discuss two efficient algorithms for large degrees, due to Cantor and Schönhage [4, 18]. A third approach would be to use segmentation, also known as Kronecker-Schönhage's trick, but it is not competitive in our context.

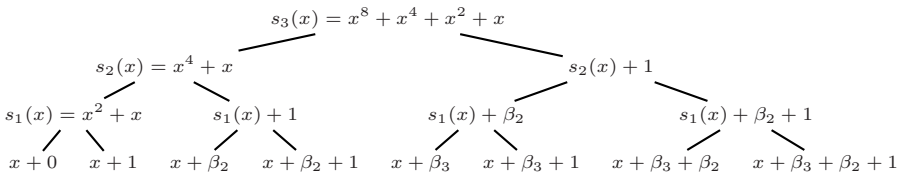
3.1 Cantor's Algorithm

Overview of the Algorithm. Cantor's algorithm provides an efficient method to compute with polynomials over finite fields of the form $F_k = \mathrm{GF}(2^{2^k})$. Cantor proposes to perform a polynomial multiplication in $F_k[x]$ using an evaluation/interpolation strategy. The set of evaluation points is carefully chosen to form an additive subgroup of F_k. The reason for the good complexity of Cantor's algorithm is that polynomials whose roots form an additive subgroup are sparse: only the monomials whose degree is a power of 2 can occur. Therefore it is possible to build a subproduct tree, where each internal node corresponds to a translate of an additive subgroup of F_k, and the cost of going up and down the tree will be almost linear due to sparsity.

We refer to [4, 8] for a detailed description of the algorithm, but we give a description of the subproduct tree, since this is useful for explaining our improvements. Let us define a sequence of polynomials $s_i(x)$ over GF(2) as follows:

$$s_0(x) = x, \quad \text{and for all } i > 0,\ s_{i+1}(x) = s_i(x)^2 + s_i(x).$$

The s_i are linearized polynomials of degree 2^i, and for all i, $s_i(x) \mid s_{i+1}(x)$. Furthermore, one can show that for all k, $s_{2^k}(x)$ is equal to $x^{2^{2^k}} + x$, whose roots are exactly the elements of F_k. Therefore, for $0 \leq i \leq 2^k$, the set of roots of s_i is a subvector-space W_i of F_k of dimension i. For multiplying two polynomials whose product has a degree less than 2^i, it is enough to evaluate/interpolate at the elements of W_i, that is to work modulo $s_i(x)$. Therefore the root node of the subproduct tree is $s_i(x)$. Its child nodes are $s_{i-1}(x)$ and $s_{i-1}(x) + 1$ whose product gives $s_i(x)$. More generally, a node $s_j(x) + \alpha$ is the product of $s_{j-1}(x) + \alpha'$ and $s_{j-1}(x) + \alpha' + 1$, where α' verifies $\alpha'^2 + \alpha' = \alpha$. For instance, the following diagram shows the subproduct tree for $s_3(x)$, where $\langle 1 = \beta_1, \beta_2, \beta_3 \rangle$ are elements of F_k that form a basis of W_3. Hence the leaves correspond exactly to the elements of W_3. In this example, we have to assume $k \geq 2$, so that $\beta_i \in F_k$.



Let c_j be the number of non-zero coefficients of $s_j(x)$. The cost of evaluating a polynomial at all the points of W_i is then $O(2^i \sum_{j=1}^{i} c_j)$ operations in F_k. The interpolation step has identical complexity. The numbers c_j are linked to the numbers of odd binomial coefficients, and one can show that $C_i = \sum_{j=1}^{i} c_j$ is $O(i^{\log_2(3)}) = O(i^{1.5849\ldots})$. Putting this together, one gets a complexity of $O(n(\log n)^{1.5849\ldots})$ operations in F_k for multiplying polynomials of degree $n < 2^{2^k}$ with coefficients in F_k.

In order to multiply arbitrary degree polynomials over GF(2), it is possible to clump the input polynomials into polynomials over an appropriate F_k, so that the previous algorithm can be applied. Let $a(x)$ and $b(x)$ be polynomials over GF(2) whose product has degree less than n. Let k be an integer such that $2^{k-1}2^{2^k} \geq n$. Then one can build a polynomial $A(x) = \sum A_i x^i$ over F_k, where A_i is obtained by taking the i-th block of 2^{k-1} coefficients in $a(x)$. Similarly, one constructs a polynomial $B(x)$ from the bits of $b(x)$. Then the product $a(x)b(x)$ in GF(2)[x] can be read from the product $A(x)B(x)$ in $F_k[x]$, since the result coefficients do not wrap around (in F_k). This strategy produces a general multiplication algorithm for polynomials in GF(2)[x] with a bit-complexity of $O(n(\log n)^{1.5849\ldots})$.

Using a Larger Base Field. When multiplying binary polynomials, a natural choice for the finite field F_k is to take k as small as possible. For instance, in [8], the cases $k = 4$ and $k = 5$ are considered. The case $k = 4$ is limited to computing

a product of 2^{19} bits, and the case $k = 5$ is limited to 2^{36} bits, that is 8 GB (not a big concern for the moment). The authors of [8] remarked experimentally that their $k = 5$ implementation was almost as fast as their $k = 4$ implementation for inputs such that both methods were available.

This behaviour can be explained by analyzing the different costs involved when using F_k or F_{k+1} for doing the same operation. Let M_i (resp. A_i) denote the number of field multiplications (resp. additions) in one multipoint evaluation phase of Cantor's algorithm when 2^i points are used. Then M_i and A_i verify

$$M_i = (i-1)2^{i-1}, \quad \text{and} \quad A_i = 2^{i-1}C_{i-1}.$$

Using F_{k+1} allows chunks that are twice as large as when using F_k, so that the degrees of the polynomials considered when working with F_{k+1} are twice as small as those involved when working with F_k. Therefore one has to compare $M_i m_k$ with $M_{i-1}m_{k+1}$ and $A_i a_k$ with $A_{i-1}a_{k+1}$, where m_k (resp. a_k) is the cost of a multiplication (resp. an addition) in F_k.

Since A_i is superlinear and a_k is linear (in 2^i resp. in 2^k), if we consider only additions, there is a clear gain in using F_{k+1} instead of F_k. As for multiplications, an asymptotical analysis, based on a recursive use of Cantor's algorithm, leads to choosing the smallest possible value of k. However, as long as 2^k does not exceed the machine word size, the cost m_k should grow roughly linearly with 2^k. In practice, since we are using the 128-bit multimedia instruction sets, up to $k = 7$, the growth of m_k is more than balanced by the decay of M_i.

In the following table, we give some data for computing a product of $N = 16\,384$ bits and a product of $N = 524\,288$ bits. For each choice of k, we give the cost m_k (in Intel Core2 CPU cycles) of a multiplication in F_k, with the $\mathtt{mp}\mathbb{F}_q$ library [11]. Then we give A_i and M_i for the corresponding value of i required to perform the product.

			$N = 16\,384$			$N = 524\,288$		
k	2^k	m_k (in cycles)	i	M_i	A_i	i	M_i	A_i
4	16	32	11	10 240	26 624	16	491 520	2 129 920
5	32	40	10	4 608	11 776	15	229 376	819 200
6	64	77	9	2 048	5 120	14	106 496	352 256
7	128	157	8	896	2 432	13	49 152	147 456

The Truncated Cantor Transform. In its plain version, Cantor's algorithm has a big granularity: the curve of its running time is a staircase, with a big jump at inputs whose sizes are powers of 2. In [8], a solution is proposed (based on some unpublished work by Reischert): for each integer $\ell \geq 1$ one can get a variant of Cantor's algorithm that evaluates the inputs modulo $x^\ell - \alpha$, for all α in a set W_i. The transformations are similar to the ones in Cantor's algorithm, and the pointwise multiplications are handled with Karatsuba's algorithm. For a given ℓ, the curve of the running time is again a staircase, but the jumps are at different positions for each ℓ. Therefore, for a given size, it is better to choose an ℓ, such that we are close to (and less than) a jump.

We have designed another approach to smooth the running time curve. This is an adaptation of van der Hoeven's truncated Fourier transform [14]. Van der Hoeven describes his technique at the butterfly level. Instead, we take the general idea, and restate it using polynomial language.

Let n be the degree of the polynomial over F_k that we want to compute. Assuming n is not a power of 2, let i be such that $2^{i-1} < n < 2^i$. The idea of the truncated transform is to evaluate the two input polynomials at just the required number of points of W_i: as in [14], we choose to evaluate at the n points that correspond to the n left-most leaves in the subproduct tree. Let us consider the polynomial $P_n(x)$ of degree n whose roots are exactly those n points. Clearly $P_n(x)$ divides $s_i(x)$. Furthermore, due to the fact that we consider the left-most n leaves, $P_n(x)$ can be written as a product of at most i polynomials of the form $s_j(x) + \alpha$, following the binary expansion of the integer n: $P_n = q_{i-1}q_{i-2}\cdots q_0$, where q_j is either 1 or a polynomial $s_j(x) + \alpha$ of degree 2^j, for some α in F_k.

The multi-evaluation step is easily adapted to take advantage of the fact that only n points are wanted: when going down the tree, if the subtree under the right child of some node contains only leaves of index $\geq n$, then the computation modulo the corresponding subtree is skipped. The next step of Cantor's algorithm is the pointwise multiplication of the two input polynomials evaluated at points of W_i. Again this is trivially adapted, since we have just to restrict it to the first n points of evaluation. Then comes the interpolation step. This is the tricky part, just like the inverse truncated Fourier transform in van der Hoeven's algorithm. We do it in two steps:

1. Assuming that all the values at the $2^i - n$ ignored points are 0, do the same interpolation computation as in Cantor's algorithm. Denote the result by f.
2. Correct the resulting polynomial by reducing f modulo P_n.

In step 1, a polynomial f with 2^i coefficients is computed. By construction, this f is congruent to the polynomial we seek modulo P_n and congruent to 0 modulo s_i/P_n. Therefore, in step 2, the polynomial f of degree $2^i - 1$ (or less) is reduced modulo P_n, in order to get the output of degree $n - 1$ (or less).

Step 1 is easy: as in the multi-evaluation step, we skip the computations that involve zeros. Step 2 is more complicated: we can not really compute P_n and reduce f modulo P_n in a naive way, since P_n is (a priori) a dense polynomial over F_k. But using the decomposition of P_n as a product of the sparse polynomials q_j, we can compute the remainder within the appropriate complexity.

3.2 Schönhage's Algorithm

Fig. 3 describes our implementation of Schönhage's algorithm [18] for the multiplication of binary polynomials. It slightly differs from the original algorithm, which was designed to be applied recursively; in our experiments — up to degree 30 million — we found out that TC4 was more efficient for the recursive calls. More precisely, Schönhage's original algorithm reduces a product modulo $x^{2N} + x^N + 1$ to $2K$ products modulo $x^{2L} + x^L + 1$, where K is a power of 3,

FFTMul(a, b, N, K) *Assumes $K = 3^k$, and K divides N.*
Multiplies polynomials a *and* b *modulo $x^N + 1$, with a transform of length K.*

1. Let $N = KM$, and write $a \sum_{i=0}^{K-1} a_i x^{iM}$, where $\deg a_i < M$ (idem for b)
2. Let L be the smallest multiple of $K/3$ larger or equal to M
3. Consider a_i, b_i in $R := \mathrm{GF}(2)[x]/(x^{2L} + x^L + 1)$, and let $\omega = x^{L/3^{k-1}} \in R$
4. Compute $\hat{a}_i = \sum_{j=0}^{K-1} \omega^{ij} a_j$ in R for $0 \le i < K$ (idem for b)
5. Compute $\hat{c}_i = \hat{a}_i \hat{b}_i$ in R for $0 \le i < K$
6. Compute $c_\ell = \sum_{i=0}^{K-1} \omega^{-\ell i} \hat{c}_i$ in R for $0 \le \ell < K$
7. Return $c = \sum_{\ell=0}^{K-1} c_\ell x^{\ell M}$.

Fig. 3. Our variant of Schönhage's algorithm

$L \ge N/K$, and N, L are multiples of K. If one replaces N and K respectively by $3N$ and $3K$ in Fig. 3, our variant reduces one product modulo $x^{3N} + 1$ to $3K$ products modulo $x^{2L} + x^L + 1$, with the same constraints on K and L.

A few practical remarks about this algorithm and its implementation: the forward and backward transforms (steps 4 and 6) use a fast algorithm with $O(K \log K)$ arithmetic operations in R. In the backward transform (step 6), we use the fact that $\omega^K = 1$ in R, thus $\omega^{-\ell i} = \omega^{-\ell i \bmod K}$. It is crucial to have an efficient arithmetic in R, i.e., modulo $x^{2L} + x^L + 1$. The required operations are multiplication by x^j with $0 \le j < 3L$ in steps 4 and 6, and plain multiplication in step 5. A major difference from Schönhage-Strassen's algorithm (SSA) for integer multiplication is that here K is a power of *three* instead of a power of two. In SSA, the analog of R is the ring of integers modulo 2^L+1, with L divisible by $K = 2^k$. As a consequence, in SSA one usually takes L to be a multiple of the numbers of bits per word — usually 32 or 64 —, which simplifies the arithmetic modulo 2^L+1 [10]. However assuming L is a multiple of 32 or 64 here, in addition to being a multiple of $K/3 = 3^{k-1}$, would lead to huge values of L, hence an inefficient implementation. Therefore the arithmetic modulo $x^{2L} + x^L + 1$ may not impose any constraint on L, which makes it tricky to implement.

Following [10], we can define the *efficiency* of the transform by the ratio $M/L \le 1$. The highest this ratio is, the more efficient the algorithm is. As an example, to multiply polynomials of degree less than $r = 6\,972\,593$, one can take $N = 13\,948\,686 = 2126K$ with $K = 6\,561 = 3^8$. The value of N is only 0.025% larger than the maximal product degree $2r-2$, which is close to optimal. The corresponding value of L is 2 187, which gives an efficiency M/L of about 97%. One thus has to compute $K = 6561$ products modulo $x^{4374} + x^{2187} + 1$, corresponding to polynomials of 69 words on a 64-bit processor.

3.3 The Splitting Approach to FFT Multiplication

Due to the constraints on the possible values of N in Algorithm **FFTMul**, the running time (as a function of the degree of the product ab) follows a "staircase". Thus, it is often worthwhile to split a multiplication into two smaller multiplications and then reconstruct the product.

FFTReconstruct(c′, c″, N, N′, N″)
Reconstructs the product c of length N from wrapped products c' of length N' and c'' of length N'', assuming $N' > N'' > N/2$. The result overwrites c'.

1. $\delta := N' - N''$
2. For $i := N - N' - 1$ downto 0 do
   $\{c'_{i+N'} := c'_{i+\delta} \oplus c''_{i+\delta}; \quad c'_i := c'_i \oplus c'_{i+N'}\}$
3. Return $c := c'$

Fig. 4. Reconstructing the product with the splitting approach

More precisely, choose $N' > N'' > \deg(c)/2$, where $c = ab$ is the desired product, and N', N'' are chosen as small as possible subject to the constraints of Algorithm **FFTMul**. Calling Algorithm **FFTMul** twice, with arguments $N = N'$ and $N = N''$, we obtain $c' = c \bmod (x^{N'}+1)$ and $c'' = c \bmod (x^{N''}+1)$. Now it is easy to reconstruct the desired product c from its "wrapped" versions c' and c''. Bit-serial pseudocode is given in Fig. 4.

It is possible to implement the reconstruction loop efficiently using full-word operations provided $N' - N'' \geq w$. Thus, the reconstruction cost is negligible in comparison to the cost of **FFTMul** calls.

4 Experimental Results

The experiments reported here were made on a 2.66Ghz Intel Core 2 processor, using gcc 4.1.2. A first tuning program compares all Toom-Cook variants from

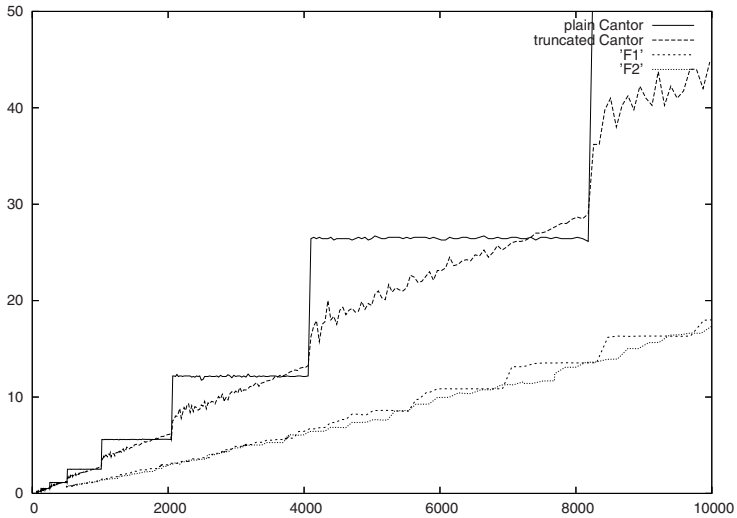


Fig. 5. Comparison of the running times of the plain Cantor algorithm, its truncated variant, our variant of Schönhage's algorithm (F1), and its splitting approach (F2). The horizontal axis represents 64-bit words, the vertical axis represents milliseconds.

Karatsuba to TC4, and determines the best one for each size. The following table gives for each algorithm the range in words where it is used, and the percentage of word sizes where it is used in this range.

Algorithm	Karatsuba	TC3	TC3W	TC4
Word range	10-65 (50%)	21-1749 (5%)	18-1760 (45%)	166-2000 (59%)

A second tuning program compared the best Karatsuba or Toom-Cook algorithm with both FFT variants (classical and splitting approach): the FFT is first used for 2461 words, and TC4 is last used for 3295 words.

In Fig. 5 the running times are given for our plain implementation of Cantor's algorithm over F_7, and its truncated variant. We see that the overhead induced by handling P_n as a product implies that the truncated version should not be used for sizes that are close to (and less than) a power of 2. We remark that

Table 1. Comparison of the multiplication routines for small degrees with existing software packages (average cycle counts on an Intel Core2 CPU)

	$N = 64$	128	192	256	320	384	448	512
NTL 5.4.1	99	368	703	1 130	1 787	2 182	3 070	3 517
LIDIA 2.2.0	117	317	787	988	1 926	2 416	2 849	3 019
ZEN 3.0	158	480	1 005	1 703	2 629	3 677	4 960	6 433
this paper	54	132	364	410	806	850	1 242	1 287

Table 2. Comparison in cycles with the literature and software packages for the multiplication of N-bit polynomials over GF(2): the timings of [16, 8, 17] were multiplied by the given clock frequency. Kn means n-term Karatsuba-like formula. In [8] we took the best timings from Table 7.1, and the degrees in [17] are slightly smaller. $F1(K)$ is the algorithm of Fig. 3 with parameter $K = 3^k$; $F2(K)$ is the splitting variant described in Section 3.3 with two calls to $F1(K)$.

reference	[16]	[8]	[17]	NTL 5.4.1	LIDIA 2.2.0	this paper
processor	Pentium 4	UltraSparc1	IBM RS6k	Core 2	Core 2	Core 2
$N = 1\,536$	1.1e5 [K3]			1.1e4	2.5e4	1.0e4 [TC3]
4 096	4.9e5 [K4]			5.3e4	9.4e4	3.9e4 [K2]
8 000			1.3e6	1.6e5	2.8e5	1.1e5 [TC3W]
10 240	2.2e6 [K5]			2.6e5	5.8e5	1.9e5 [TC3W]
16 384		5.7e6	3.4e6	4.8e5	8.6e5	3.3e5 [TC3W]
24 576	8.3e6 [K6]			9.3e5	2.1e6	5.9e5 [TC3W]
32 768		1.9e7	8.7e6	1.4e6	2.6e6	9.3e5 [TC4]
57 344	3.3e7 [K7]			3.8e6	7.3e6	2.4e6 [TC4]
65 536		4.7e7	1.7e7	4.3e6	7.8e6	2.6e6 [TC4]
131 072		1.0e8	4.1e7	1.3e7	2.3e7	7.2e6 [TC4]
262 144		2.3e8	9.0e7	4.0e7	6.9e7	1.9e7 [F2(243)]
524 288		5.2e8		1.2e8	2.1e8	3.7e7 [F1(729)]
1 048 576		1.1e9		3.8e8	6.1e8	7.4e7[F2(729)]

this overhead is more visible for small sizes than for large sizes. This figure also compares our variant of Schönhage's algorithm (Fig. 3) with the splitting approach: the latter is faster in most cases, and both are faster than Cantor's algorithm by a factor of about two. It appears from Fig. 5 that a truncated variant of Schönhage's algorithm would not save much time, if any, over the splitting approach.

Tables 1 and 2 compare our timings with existing software or published material. Table 1 compares the basic multiplication routines involving a fixed small number of words. Table 2 compares the results obtained with previous ones published in the literature. Since previous authors used 32-bit computers, and we use a 64-bit computer, the cycle counts corresponding to references [16, 8, 17] should be divided by 2 to account for this difference. Nevertheless this would not affect the comparison.

5 Conclusion

This paper presents the current state-of-the-art for multiplication in GF(2)[x]. We have implemented and compared different algorithms from the literature, and invented some new variants.

The new algorithms were already used successfully to find two new primitive trinomials of record degree 24 036 583 (the previous record was 6 972 593), see [3].

Concerning the comparison between the algorithms of Schönhage and Cantor, our conclusion differs from the following excerpt from [8]: *The timings of Reischert (1995) indicate that in his implementation, it [Schönhage's algorithm] beats Cantor's method for degrees above 500,000, and for degrees around 40,000,000, Schönhage's algorithm is faster than Cantor's by a factor of* $\approx \frac{3}{2}$. Indeed, Fig. 5 shows that Schönhage's algorithm is consistently faster by a factor of about 2, already for a few thousand bits. However, a major difference is that, in Schönhage's algorithm, the pointwise products are quite expensive, whereas they are inexpensive in Cantor's algorithm. For example, still on a 2.66Ghz Intel Core 2, to multiply two polynomials with a result of 2^{20} bits, Schönhage's algorithm with $K = 729$ takes 28ms, including 18ms for the pointwise products modulo $x^{5832} + x^{2916} + 1$; Cantor's algorithm takes 57ms, including only 2.3ms for the pointwise products. In a context where a given Fourier transform is used many times, for example in the block Wiedemann algorithm used in the "linear algebra" phase of the Number Field Sieve integer factorisation algorithm, Cantor's algorithm may be competitive.

Acknowledgements

The word-aligned variant of TC3 for GF(2)[x] was discussed with Marco Bodrato. The authors thank Joachim von zur Gathen and the anonymous referees for their useful remarks. The work of the first author was supported by the Australian Research Council.

References

1. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A., Osvik, D.A.: A kilobit special number field sieve factorization. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 1–12. Springer, Heidelberg (2007)
2. Bodrato, M.: Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In: Carlet, C., Sunar, B. (eds.) WAIFI 2007. LNCS, vol. 4547, pp. 116–133. Springer, Heidelberg (2007)
3. Brent, R.P., Zimmermann, P.: A multi-level blocking distinct degree factorization algorithm. Research Report 6331, INRIA (2007)
4. Cantor, D.G.: On arithmetical algorithms over finite fields. J. Combinatorial Theory, Series A 50, 285–300 (1989)
5. Chabaud, F., Lercier, R.: ZEN, a toolbox for fast computation in finite extensions over finite rings, `http://sourceforge.net/projects/zenfact`
6. Cohen, H., Frey, G., Avanzi, R., Doche, C., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. In: Discrete Mathematics and its Applications, Chapman & Hall/CRC (2005)
7. Cook, S.A.: On the Minimum Computation Time of Functions. PhD thesis, Harvard University (1966)
8. von zur Gathen, J., Gerhard, J.: Arithmetic and factorization of polynomials over F_2. In: Proceedings of ISSAC 1996, Zürich, Switzerland, pp. 1–9 (1996)
9. von zur Gathen, J., Gerhard, J.: Polynomial factorization over F_2. Math. Comp. 71(240), 1677–1698 (2002)
10. Gaudry, P., Kruppa, A., Zimmermann, P.: A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In: Proceedings of ISSAC 2007, Waterloo, Ontario, Canada, pp. 167–174 (2007)
11. Gaudry, P., Thomé, E.: The mpFq library and implementing curve-based key exchanges. In: Proceedings of SPEED, pp. 49–64 (2007)
12. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer Professional Computing. Springer, Heidelberg (2004)
13. Harvey, D.: Avoiding expensive scalar divisions in the Toom-3 multiplication algorithm, 10 pages (Manuscript) (August 2007)
14. van der Hoeven, J.: The truncated Fourier transform and applications. In: Gutierrez, J. (ed.) Proceedings of ISSAC 2004, Santander, 2004, pp. 290–296 (2004)
15. The LiDIA Group. LiDIA, A C++ Library For Computational Number Theory, Version 2.2.0 (2006)
16. Montgomery, P.L.: Five, six, and seven-term Karatsuba-like formulae. IEEE Trans. Comput. 54(3), 362–369 (2005)
17. Roelse, P.: Factoring high-degree polynomials over F_2 with Niederreiter's algorithm on the IBM SP2. Math. Comp. 68(226), 869–880 (1999)
18. Schönhage, A.: Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2. Acta Inf. 7, 395–398 (1977)
19. Shoup, V.: NTL: A library for doing number theory, Version 5.4.1 (2007), `http://www.shoup.net/ntl/`
20. Thomé, E.: Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. J. Symb. Comp. 33, 757–775 (2002)
21. Toom, A.L.: The complexity of a scheme of functional elements realizing the multiplication of integers. Soviet Mathematics 3, 714–716 (1963)
22. Weimerskirch, A., Stebila, D., Shantz, S.C.: Generic GF(2) arithmetic in software and its application to ECC. In: Safavi-Naini, R., Seberry, J. (eds.) ACISP 2003. LNCS, vol. 2727, pp. 79–92. Springer, Heidelberg (2003)
23. Zimmermann, P.: Irred-ntl patch, `http://www.loria.fr/~zimmerma/irred/`