The M4RIE library for dense linear algebra over small fields with even characteristic

Martin R. Albrecht

The M4RIE library for dense linear algebra over small fields with even characteristic

Martin R. Albrecht
INRIA, Paris-Rocquencourt Center, POLSYS Project
UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
malb@lip6.fr

February 5, 2015

Abstract

We describe algorithms and implementations for linear algebra with dense matrices over $\mathbb{F}_{2^e}$ for $2 \leq e \leq 10$. Our main contributions are: (1) a specialisation of precomputation tables to $\mathbb{F}_{2^e}$, called Newton-John tables in this work, to avoid scalar multiplications in Gaussian elimination and matrix multiplication, (2) an efficient implementation of Karatsuba-style multiplication for matrices over extension fields of $\mathbb{F}_2$ and (3) a description of an open-source library – called M4RIE – providing the fastest known implementation of dense linear algebra over $\mathbb{F}_{2^e}$ with $2 \leq e \leq 10$.

1 Introduction

Linear algebra over small finite fields has many direct applications, such as cryptography and coding theory. Other applications include efficient linear algebra over the rationals, e.g., by reducing such computations to a series of computations modulo small primes, and solving non-linear systems of equations using Gröbner bases [19]. For the latter recent work has emphasised that the right choice of linear algebra algorithms and implementations can make a significant impact on the performance of Gröbner basis algorithms [13]. Furthermore, dense linear algebra over finite extension fields $\mathbb{F}_{p^k}$ can be used to achieve asymptotically fast matrix-matrix multiplication of dense matrices over $\mathbb{F}_p[x]$ which in turn has applications such as the Block Wiedemann algorithm for sparse matrices [15].

Compared to other small finite fields, those with even characteristic have some special properties which make them a prominent choice for designing cryptographic and coding systems – cf., the AES [10] as a prime example. For instance, addition is simply XOR and is hence natively available on modern CPUs, the same cannot be said for other small finite fields. Yet, this family of finite fields has not received much attention in the literature on linear algebra. That the current state of the art in the literature leaves something to be desired can be observed from the following simple benchmark: multiplying two random $1,000 \times 1,000$ matrices over $\mathbb{F}_4$ on a 2.66 Ghz Intel i7 CPU takes 210ms using GAP 4.4.12 [14], 460ms using LinBox/ FFLAS-FFPACK [12], or even 85s and 97s using NTL 5.4.2 [21] and Sage [23] respectively. For comparison, the closed source system Magma [8] can multiply two dense $1,000 \times 1,000$ matrices over $\mathbb{F}_4$ in 13ms and the same operation over $\mathbb{F}_2$ takes 1.7ms using the M4RI library [3].

In this work, we present the M4RIE library which implements efficient algorithms for linear algebra with dense matrices over $\mathbb{F}_{2^e}$ for $2 \leq e \leq 10$.[1] As the name of the library indicates, it makes heavy use of the M4RI library [2] both directly (i.e., by calling it) and indirectly (i.e., by using its concepts). The contributions of this work are as follows. We provide an open-source GPLv2+ C library for efficient linear algebra over $\mathbb{F}_{2^e}$ with $2 \leq e \leq 10$. In this library we implemented an idea due to Bradshaw and Boothby [7] which reduces matrix multiplication over $\mathbb{F}_{p^n}$ to a series of matrix multiplications over $\mathbb{F}_p$. Furthermore, we propose a caching technique – *Newton-John* tables – to avoid finite field multiplications which is inspired by Kronrod's method ("M4RM") [5, 1] for matrix multiplication

[1] Future versions will support $e \leq 16$.

over $\mathbb{F}_2$. Using these two techniques we provide asymptotically fast triangular solving with matrices (TRSM) and PLE-based [17] Gaussian elimination. As a result, we are able to significantly improve upon the state of the art in dense linear algebra over $\mathbb{F}_{2^e}$ with $2 \leq e \leq 10$. The example mentioned above is completed in 5.5ms by our library.

2 Notation

We represent elements in $\mathbb{F}_{2^e} \cong \mathbb{F}_2[x]/\langle f \rangle$, with $f \in \mathbb{F}_2[x]$, $\deg(f) = e$ and f irreducible, as polynomials $\sum_{i=0}^{e-1} a_i x^i$ or as coefficient vectors $(a_{e-1}, \ldots, a_0)$ where $a_i \in \mathbb{F}_2$. We sometimes identify the coefficient vector $(a_{e-1}, \ldots, a_0)$ with the integer $\sum_{i=0}^{e-1} a_i 2^i$, e.g., when indexing tables by finite field elements. By α we denote some root of the primitive polynomial f of $\mathbb{F}_{2^e}$. By A_i we denote the i-th row of the matrix A and by $A_{i,j}$ the entry in row i and column j of A. We start counting at zero. We represent permutation matrices as LAPACK-style permutation vectors. That is to say that for example the permutation matrix

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

is stored as $P = [0, 2, 2]$, where for each index i the entry P_i encodes which (row or column) swap is to be performed on the input matrix. In the given example, we would swap row/column 0 with row/column 0 (identity operation), row/column 1 with row/column 2, and finally row/column 2 with row/column 2 (identity operation). This allows to apply permutations in-place.

3 Previous implementations

To demonstrate the viability of our approach we compare our implementation with previous implementations. In particular, we compare with GAP, LinBox/FFLAS-FFPACK and Magma, as these are the fastest known previous works. Below, we give a brief overview of algorithmic and implementation choices in these libraries.

GAP [14] packs finite field elements of size $2 < s \leq 2^8$ into words using 8 bits per entry. Arithmetic is implemented using table look ups. Multiplication is performed using cubic matrix multiplication. Row echelon forms are computed using cubic Gaussian elimination.

LinBox/FFLAS-FFPACK [12] uses floating point numbers to represent finite field elements. For extension fields, elements are represented as "sparse" integers, such that there are sufficient zeroes between two coefficients to avoid the carry travelling too far [11]. This feature is experimental and requires some patching [4]. FFLAS-FFPACK implements Strassen-Winograd multiplication as well as asymptotically fast LQUP decomposition for Gaussian elimination. However, these features do not currently work with characteristic two [4].

Magma [8] implements asymptotically fast matrix multiplication and reduces Gaussian elimination to LQUP decomposition. For $\mathbb{F}_{2^k}$ with $2 \leq k \leq 4$ a bit-sliced representation similar to our `mzd_slice_t` is used in combination with Karatsuba-like formulas for polynomial multiplication. For $5 \leq k \leq 20$ elements in $\mathbb{F}_{2^k}$ are represented using Zech logarithms. For larger k a packed polynomial representation is used similar to our `mzed_t` [22].

In summary, only Magma offers a dedicated implementation for $\mathbb{F}_{2^e}$, but only for $e \leq 4$.

4 Matrix representation

The M4RIE library features two matrix types, each of which is optimised for certain operations. Both representations use one or more M4RI matrices as data storage and hence re-use M4RI's matrix window concept [1], allocation routines and data structures.

4.1 packed: `mzed_t`

Considering the polynomial representation of elements in $\mathbb{F}_{2^e}$, we may bit-pack several such elements in one machine word. Since we are using the M4RI library as actual data storage, this means that words hold 64 bits [1]. Hence,

the `mzed_t` data type packs elements of $\mathbb{F}_{2^e}$ in 64-bit words. However, instead of packing as many elements as possible into one word, every element is padded to the next length dividing 64. Thus, for example, elements in $\mathbb{F}_{32}$ are represented as polynomials of degree 8 where the top three coefficients are always zero. While this wastes some storage space and CPU time, it allows for more compact code by reducing what cases have to be considered. The second row of Figure 1 gives an example.

In this representation additions are very cheap since we can disregard any element boundaries and simply call M4RI's addition routines. Scalar multiplication, on the other hand, is much more expensive. Either we perform a table look-up for each *element* or we perform bit operations on words which perform multiplication and modular reduction in parallel on all elements of a word. In either case, multiplication is considerably more expensive than addition.

4.2 sliced: `mzd_slice_t`

Instead of representing matrices over $\mathbb{F}_{2^e}$ as matrices over polynomials we may represent them as polynomials with matrix coefficients. That is, for each degree we store matrices over $\mathbb{F}_2$ which hold the coefficients for this degree. Hence, the data type `mzd_slice_t` for matrices over $\mathbb{F}_{2^e}$ internally stores e-tuples of matrices over $\mathbb{F}_2$ as implemented by the M4RI library. We call each M4RI matrix for some degree i a *slice* and refer to the operation converting from `mzed_t` to `mzd_slice_t` as *slicing*. The inverse operation is called *clinging*. The third row of Figure 1 gives an example of the `mzd_slice_t` representation.

Addition is performed by adding each slice independently and is thus quite efficient. Scalar multiplication, on the other hand, has to rely on similar techniques as in `mzed_t`. Thus, here too, scalar multiplication is more expensive than addition.

Hence, in this work, we present algorithms for matrix multiplication and elimination where we avoid many scalar multiplications.

$$\begin{aligned} A = & \begin{pmatrix} \alpha^2+1 & \alpha \\ \alpha+1 & 1 \end{pmatrix} \\ = & \begin{bmatrix} \square 101 & \square 010 \\ \square 011 & \square 001 \end{bmatrix} \\ = & \left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \right) \end{aligned}$$

Figure 1: a 2×2 matrix over $\mathbb{F}_8$

4.3 Conversion

Since we convert between the two representations during the course of our algorithms, these conversion must be sufficiently efficient to not become a bottleneck. In our implementation we use a series of bitshifts and bitmasks to convert between our representations. As an example, consider the following 8-bit word which holds elements $a_{i,1}x + a_{i,0}$ of $\mathbb{F}_4$ in bit-packed representation:

$$a = [a_{3,1}, a_{3,0}, a_{2,1}, a_{2,0}, a_{1,1}, a_{1,0}, a_{0,1}, a_{0,0}].$$

We compute:

```
a0 = (a & 01010101)<<1;
a1 = (a0 & 11001100) | (a0 & 00110011)<<1;
a2 = (a1 & 11110000) | (a1 & 00001111)<<2;
```

which produces

$$\begin{aligned} a_0 &= [a_{3,0}, 0, a_{2,0}, 0, a_{1,0}, 0, a_{0,0}, 0], \\ a_1 &= [a_{3,0}, a_{2,0}, 0, 0, a_{1,0}, a_{0,0}, 0, 0], \text{ and} \\ a_2 &= [a_{3,0}, a_{2,0}, a_{1,0}, a_{0,0}, 0, 0, 0, 0]. \end{aligned}$$

and hence the leftmost bits of the bit-sliced representation of a. We produce the remaining bits and words in the same fashion.

5 Newton-John tables and applications

First, we consider the bit-packed representation `mzed_t` and introduce the notion and key idea of Newton-John tables, which are essentially precomputation tables specialised for $\mathbb{F}_{2^e}$. To explain the main idea consider matrix multiplication as given in Algorithm 1.

Algorithm 1: Cubic matrix multiplication

Input: A – $m \times \ell$ matrix
Input: B – $\ell \times n$ matrix
Output: $C = A \cdot B$

```
1 begin
2     for 0 ≤ i < m do
3         for 0 ≤ j < ℓ do
4             C_j ← C_j + A_{j,i} × B_i;
5     return C;
```

This algorithm uses $m \cdot \ell \cdot n$ finite field multiplications and the same number of additions. That is, in line 4 the row B_i is scaled by $A_{j,i}$ and then added to the row C_j. Observe that B_i is rescaled ℓ-times, while there are 2^e different values for $A_{j,i}$ and hence multiples of B_i. Indeed, if $2^e < \ell$ it is advantageous to precompute all possible 2^e multiples of B_i and to store these multiples in a table indexed by finite field elements. These precomputation tables are quite similar to Kronrod's method for matrix multiplication, also sometimes referred to as "greasing". Hence, we call these tables *Newton-John tables* to honour Olivia Newton-John's work in [18].

We also note that we create these tables in less than 2^e multiplications. That is, we first compute $\alpha^i \cdot B_i$ for all $0 \leq i < e$. Then, we compute each multiple of B_i as a linear combination of $(\alpha^0 \cdot B_i, \ldots \alpha^{e-1} \cdot B_i)$ which we just computed. Using Gray codes for the addition step we can thus construct all 2^e multiples using 2^e additions [16] (this can be improved to $2^e - e$ easily). The subroutine creating these tables is given in Algorithm 2.

The complete algorithm is given in Algorithm 3 which costs $m \cdot (2^e + \ell) \cdot n$ additions and $m \cdot e \cdot n$ multiplications. Note that e is a constant here and asymptotically we thus achieve $\mathcal{O}(n^3)$ additions and $\mathcal{O}(n^2)$ multiplications. Since additions are much cheaper than multiplications, this leads to considerable performance gains.

Many variants of this basic algorithm are possible. For instance, we may use more than one Newton-John table or process the data in blocks for better cache friendliness (cf., [1] for both techniques). Furthermore, if 2^e is too big to precompute T we may precompute only M (cf., Algorithm 2) and perform e additions in line 6. Conversely, if 2^e is very small, we may combine Newton-John tables with Kronrod's method to reduce the number of additions to $\mathcal{O}(n^3/\log n)$. Our library uses eight Newton-John tables and processes matrices in blocks that fit into L2 cache. Since e is small we always compute the full table T. However, we did not implement Kronrod's method yet.

Table 1 lists CPU times multiplying two $1,000 \times 1,000$ matrices in our implementation of Newton-John multiplication, in Magma, GAP and LinBox (cf., Section 3 for a brief discussion of Magma's, GAP's and LinBox's implementations). Note that the hex string in the header of the last column indicates which revision of the public source code repository[2] was used to produce these times.

[2] cf., `https://bitbucket.org/m4rie`.

Algorithm 2: MAKETABLE

Input: B – an $1 \times n$ matrix
Output: T – a $2^e \times n$ matrix with each row a multiple of B

```
begin
    M ← e × n matrix;
    T ← 2^e × n matrix;
    for 0 ≤ k < e do
        M_k ← α^k · B;
    T ← all linear combinations of rows of M;
return T;
```

Algorithm 3: Newton-John multiplication

Input: A – $m \times \ell$ matrix
Input: B – $\ell \times n$ matrix
Output: $C = A \cdot B$

```
begin
    for 0 ≤ i < m do
        T ← MAKETABLE(B_i);
        for 0 ≤ j < ℓ do
            x ← A_{j,i} as an integer;
            C_j ← C_j + T_x;
    return C;
```

Of course, this algorithm is not asymptotically fast. Hence, we only use it as a base case for the Strassen-Winograd algorithm [24] for matrix multiplication which has complexity $\mathcal{O}\left(n^{\log_2 7}\right)$. In our implementation we cross over to the base case roughly when the submatrices fit into L2 cache; however, the exact value depends on the size of the finite field. For the machine used to produce all timing results in this paper, the crossover dimensions are 2608 for $e = 2$, 1773 for $e \leq 8$, and 1254 for $e \leq 16$. Table 3 lists timings for Strassen-Winograd multiplication on top of Newton-John tables (abbreviated as "S-W/N-J") in comparison with other implementations. However, in our implementation, as Karatsuba-based matrix multiplication is always faster, we only use Strassen-Winograd in combination with Newton-John tables by default where Karatsuba is not implemented (currently, $e > 8$).

Instead, the main application of Newton-John tables in our implementation is Gaussian elimination and PLE decomposition.

5.1 Gaussian elimination

Newton-John tables can also be used in Gaussian elimination, as shown in Algorithm 4. This algorithm uses $r \cdot (n + 2^e) \cdot n$ additions and $r \cdot (e + 1) \cdot n$ multiplications, which gives an asymptotic complexity of $\mathcal{O}\left(n^3\right)$ additions and $\mathcal{O}\left(n^2\right)$ multiplications. Again, a variety of variants are possible such as multiple Newton-John tables (similar to [1]). Our implementation uses six Newton-John tables.

In Table 2 we give CPU times for computing the reduced row echelon form of random $1,000 \times 1,000$ matrices over $\mathbb{F}_{2^e}$ in Magma, GAP and our implementation. Note that GAP's `SemiEchelonMat` command does not compute the *reduced* row echelon form. Hence, to normalise the data we multiplied all GAP timings in Table 2 by two.

We note that there is a considerable jump between $e = 8$ and $e = 9$ due to our bit-packed matrix representation: Elements of $\mathbb{F}_{2^9}$ take up 16 bits, while elements of $\mathbb{F}_{2^8}$ only take up 8 bits (cf. Section 4.1). It is future work to reduce this jump to a factor of 2 from the current factor 5.

e	Magma 2.15-10	GAP 4.4.12	LinBox svn 20111216	Newton-John 6b24b839a46f
2	0.013s	0.216s	0.468s	0.012s
3	0.036s	0.592s	0.480s	0.020s
4	0.074s	0.588s	0.760s	0.022s
5	1.276s	1.568s	1.932s	0.048s
6	1.286s	1.356s	3.532s	0.059s
7	1.316s	1.276s	3.884s	0.082s
8	1.842s	1.328s	–	0.160s
9	3.985s	64.700s	–	0.626s
10	4.160s	59.131s	–	1.080s

Table 1: Multiplication of $1,000 \times 1,000$ matrices on 2.66 Ghz Intel i7

5.2 PLE decomposition

Algorithm 4 can be modified to compute the PLE decomposition instead of the row echelon form. Since this definition is lesser well-known we reproduce it below. For a more detailed treatment of PLE decomposition see [17].

Definition 1 (PLE) *Let A be a $m \times n$ matrix over a field K. A PLE decomposition of A is a triple of matrices P, L and E such that P is a $m \times m$ permutation matrix, L is a unit lower triangular matrix, and E is a $m \times n$ matrix in row-echelon form, and $A = PLE$.*

Lemma 1 ([17]) *For a PLE decomposition of any $m \times n$ matrix A, the factors L and E can be stored in-place in A.*

For the sake of simplicity, we compute a minor variant of PLE in our library. That is, L is not necessarily unit lower triangular, i.e., we do rescale the pivot row to get leading entry 1. Then, the changes necessary for Algorithm 4 to compute this variant of PLE decomposition are:

- Store i and j in two vectors P and Q in line 6;
- only start addition in column $j + 1$ in line 13 in order to preserve L below the main diagonal;
- only eliminate below the pivot ($k > r$);
- perform column swaps below and on the main diagonal right before line 16 to compress L.

Note that we can also recover this algorithm by considering block iterative PLE decomposition [3] with multiplication updates to the right hand side based on Newton-John tables. We remark that currently our implementation of Newton-John-based PLE decomposition only uses one Newton-John table. Increasing this number should improve performance and is future work.

Also, this algorithm is neither asymptotically fast nor in-place since it has cubic complexity and requires $\mathcal{O}(n)$ storage to hold the table T. Hence its main application is as a base case for asymptotically fast PLE decomposition [17] which reduces PLE decomposition to matrix multiplication. If the crossover dimension – when the asymptotically fast algorithm switches over to PLE decomposition based on Algorithm 4 – does not depend on n like in our implementation, then the overall algorithm is both asymptotically fast and in-place. However, one last building block is needed to implement asymptotically fast PLE decomposition: triangular system solving.

5.3 TRiangular Solving with Matrices

Triangular system solving with matrices can also be achieved using Newton-John tables. As an example, we given an algorithm for solving $X = U^{-1} \cdot B$ with U upper triangular in Algorithm 5. We note that Algorithm 5 is essentially block iterative TRSM with Newton-John table based multiplication. Yet, we present it here for completeness.

Algorithm 4: Newton-John Gauss elimination

Input: A – $m \times n$ matrix
Output: r – the rank of A
Result: A is in reduced row echelon form

```
1  begin
2      r ← 0;
3      for 0 ≤ j < n do
4          for r ≤ i < m do
5              if A_{i,j} ≠ 0 then
6                  A_i ← A_{i,j}^{-1} · A_i;
7                  swap the rows i and r in A;
8                  T ← MAKETABLE(A_r);
9                  for 0 ≤ k < m do
10                     if k = r then continue;
11                     ;
12                     x ← A_{k,j} as an integer;
13                     A_k ← A_k + T_x;
14                 r ← r + 1;
15                 break;
16     return r;
```

6 Karatsuba multiplication

Recall that `mzd_slice_t` represents matrices over $\mathbb{F}_{2^e}$ as polynomials with matrices over $\mathbb{F}_2$ as coefficients. Using this representation, matrix multiplication then can be accomplished by performing polynomial multiplication and subsequent modular reduction. For example, assume we want to compute $C = A \cdot B$ where A and B are over $\mathbb{F}_4$. We rewrite A as $A_1x + A_0$ and B as $B_1x + B_0$, the product is then $\tilde{C} = A_1B_1x^2 + (A_1B_0 + A_0B_1)x + A_0B_0$ which reduces to $C = (A_1B_1 + A_1B_0 + A_0B_1)x + A_0B_0 + A_1B_1$ modulo the primitive polynomial $f = x^2 + x + 1$ of $\mathbb{F}_4$. Hence, matrix multiplication over $\mathbb{F}_{2^e}$ can be reduced to matrix multiplication and addition over $\mathbb{F}_2$. Using naive polynomial arithmetic we get that matrix multiplication over $\mathbb{F}_{2^e}$ costs e^2 matrix multiplications over $\mathbb{F}_2$. However, using Karatsuba polynomial multiplication we can reduce this to $e^{\log_2 3} \approx e^{1.584}$. To get back to the above example, we can rewrite it as $C = ((A_1 + A_0)(B_1 + B_0) + A_0B_0)x + A_0B_0 + A_1B_1$ and hence multiplication costs 3 instead of 4 multiplications over $\mathbb{F}_2$. This was first explicitly proposed for matrices over $\mathbb{F}_{p^n}$ by Bradshaw and Boothby in [7]. However, this technique has been used for linear algebra over $\mathbb{F}_{2^k}$ with $2 \leq k \leq 4$ in Magma for some time [22] and seems to be folklore among some researchers.

In our implementation we use [20] for Karatsuba-like formulas up to degree $e = 8$. Concrete costs are given in Table 3 where the first column lists the CPU time for multiplying two $4,000 \times 4,000$ matrices using Strassen-Winograd on top of Newton-John multiplication. The column "(S-W/N-J)/M4RI" indicates how many $4,000 \times 4,000$ matrix multiplications over $\mathbb{F}_2$ can be achieved in the same time using the M4RI library (this time is given in row $e = 1$). The column "naive" lists how many multiplications would be needed by naive polynomial multiplication. The column "[20]" lists the best known complexity for Karatsuba-like formulas. The absolute time of our Karatsuba-like implementation is given in the column "Bit-slice". The last column shows the number of multiplications which our Karatsuba-like implementation actually achieves. That is, it divides the column "Bit-slice" by the column "Bit-slice" for $e = 1$ which simply holds the M4RI time. Finally, Table 3 also compares our implementation with the previous two best implementations GAP and Magma.

We remark that our implementation makes use of M4RI's matrix-matrix multiplication routines which implement Strassen-Winograd on top of Kronrod's method also known as the "Method of Four Russians" [1]. Hence, Karatsuba-based matrix multiplication is also asymptotically fast with respect to n.

e	Magma 2.15-10	GAP 4.4.12	Newton-John 6b24b839a46f
2	0.028s	0.184s	0.012s
3	0.045s	0.496s	0.019s
4	0.054s	0.560s	0.022s
5	0.690s	1.224s	0.042s
6	0.670s	1.168s	0.048s
7	0.700s	1.104s	0.060s
8	0.866s	1.136s	0.081s
9	1.523s	35.634s	0.427s
10	1.540s	36.154s	0.831s

Table 2: Elimination of $1,000 \times 1,000$ matrices on 2.66 Ghz Intel i7

Algorithm 5: Newton-John TRSM upper left

Input: U – $m \times m$ upper triangular matrix
Input: B – $m \times n$ matrix
Result: $X = U^{-1} \cdot B$ is stored in B

```
1 begin
2     for m > i ≥ 0 do
3         B_i ← U_{i,i}^{-1} · B_i;
4         T ← MakeTable(B_i);
5         for 0 ≤ j < i do
6             x ← U_{j,i};
7             B_j ← B_j + T_x;
```

However, we note that Karatsuba based multiplication needs more memory than Strassen on top of Newton-John multiplication. Our implementation uses three temporary matrices over $\mathbb{F}_2$. We finish this section by pointing out in principle more efficient polynomial multiplication algorithms than Karatsuba can be applied (cf., [9, 6] for a discussions dedicated to $\mathbb{F}_2[x]$ and $\mathbb{F}_{2^e}$). We leave this as an open problem for future work.

e	Magma 2.15-10	GAP 4.4.12	S-W/N-J	Bit-slice	(S-W/N-J)/ M4RI	naive	[20]	Bit-slice/ M4RI
1	0.100s	0.244s	–	0.071s	1	1	1	1.0
2	1.220s	12.501s	0.630s	0.224s	8.8	4	3	3.1
3	2.020s	35.986s	1.480s	0.448s	20.8	9	6	6.3
4	5.630s	39.330s	1.644s	0.693s	23.1	16	9	9.7
5	94.740s	86.517s	3.766s	1.005s	53.0	25	13	14.2
6	89.800s	85.525s	4.339s	1.336s	61.1	36	17	18.8
7	82.770s	83.597s	6.627s	1.639s	93.3	49	22	23.1
8	104.680s	83.802s	10.170s	2.140s	143.2	64	27	30.1

Table 3: Multiplication of $4,000 \times 4,000$ matrices over $\mathbb{F}_{2^e}$ on 2.66 Ghz Intel i7.

7 Echelon Forms

Putting the building blocks

(1) Karatsuba multiplication,

(2) Newton-John-based PLE decomposition,

(3) asymptotically-fast PLE decomposition,

(4) Newton-John based TRSM and

(5) asymptotically-fast TRSM,

together we can construct asymptotically fast Gaussian elimination, i.e.., computation of (reduced) row echelon forms (cf., [17]). Our implementation uses `mzd_slice_t` as representation for large matrices and switches over to `mzed_t` roughly when the submatrix currently considered fits into L2 cache (matrix dimension 3547 for $e = 2$, 2508 for $e < 4$, and 1773 for $e < 8$). Table 4 lists CPU times for computing the (reduced) row echelon form using Magma (reduced), GAP (not reduced) and our implementation (reduced). We also list running times for LinBox mod p (reduced) where p is the largest prime smaller than 2^e. Note that currently our implementation only implements asymptotically fast PLE decomposition up to $e = 8$, for $e \in \{9, 10\}$ Newton-John table based Gaussian elimination is used. Comparing our implementation with LinBox mod p demonstrates clearly, that in order to be competitive we need to extend asymptotically fast PLE decomposition to $e > 8$.

e	Magma 2.15-10	GAP 4.4.12	LinBox (mod p) 1.1.6	M4RIE 6b24b839a46f
2	6.04s	162.65s	49.52s	3.31s
3	14.47s	442.52s	49.92s	5.33s
4	60.37s	502.67s	50.91s	6.33s
5	659.03s	N/A	51.20s	10.51s
6	685.46s	N/A	51.61s	13.08s
7	671.88s	N/A	53.94s	17.29s
8	840.22s	N/A	64.24s	20.25s
9	1630.38s	N/A	76.18s	260.77s
10	1631.35s	N/A	76.45s	291.30s

Table 4: Elimination of $10,000 \times 10,000$ matrices on 2.66 Ghz Intel i7

Acknowledgements

We are grateful to anonymous referees for helpful comments on this work.

Additional Timings

Tables 5 and Table 6 contain additional timings for matrix multiplication ane Gaussian elimination over $\mathbb{F}_{2^e}$ on a 2.66 Ghz Intel i7 machine. Both tables also list the CPU cycles needed on average divided by $n^{2.807}$. On the one hand, this shows that the implementations indeed achieves $\mathcal{O}\left(n^{2.807}\right)$. On the other hand, this shows that our implementation of Gaussian elimination is not as efficient as our implementation of matrix multiplication: the leading constant is 6 for multiplication and 2.8 for PLE decomposition [17]. Hence, we would expect a ratio of ≈ 2 between Table 5 and Table 6. Since the observed ratio is smaller than this, we can expect further performance gains by improving the Newton-John table-based PLE base case.

References

[1] M. Albrecht, G. V. Bard, and W. Hart. Algorithm 898: Efficient multiplication of dense matrices over GF(2). *ACM Transactions on Mathematical Software*, 37(1):14 pages, January 2010. pre-print available as `arxiv:0811.1714 [cs.MS]`.

	CPU time in seconds								
n	$e=2$	$e=3$	$e=4$	$e=5$	$e=6$	$e=7$	$e=8$	$e=9$	$e=10$
1000	0.005	0.011	0.016	0.024	0.033	0.041	0.051	0.142	0.193
2000	0.032	0.067	0.100	0.149	0.209	0.245	0.309	0.826	1.090
3000	0.102	0.206	0.312	0.453	0.651	0.753	0.951	2.575	3.262
4000	0.230	0.468	0.687	1.013	1.475	1.653	2.061	5.758	7.446
5000	0.489	0.919	1.322	2.042	2.784	3.168	3.931	11.633	14.229
6000	0.872	1.798	2.681	3.895	5.189	6.375	7.885	23.042	28.482
7000	1.311	2.682	3.962	5.758	7.549	9.450	12.153	33.950	42.866
8000	1.873	3.869	5.611	8.145	10.862	13.617	16.615	49.188	61.191
9000	2.496	5.082	7.781	10.971	15.700	20.356	22.155	63.774	81.184
10000	3.334	6.613	10.374	14.332	23.497	23.740	29.266	82.923	147.319
	CPU cycles / $n^{2.807}$								
n	$e=2$	$e=3$	$e=4$	$e=5$	$e=6$	$e=7$	$e=8$	$e=9$	$e=10$
1000	0.055	0.113	0.169	0.247	0.336	0.414	0.518	1.442	1.950
2000	0.046	0.096	0.145	0.215	0.302	0.354	0.446	1.192	1.572
3000	0.047	0.095	0.144	0.209	0.301	0.348	0.439	1.190	1.507
4000	0.047	0.096	0.141	0.208	0.304	0.340	0.424	1.186	1.534
5000	0.053	0.101	0.145	0.224	0.306	0.348	0.433	1.281	1.567
6000	0.057	0.118	0.177	0.257	0.342	0.420	0.520	1.521	1.880
7000	0.056	0.114	0.169	0.246	0.323	0.404	0.520	1.454	1.835
8000	0.055	0.113	0.165	0.239	0.319	0.400	0.489	1.448	1.801
9000	0.052	0.107	0.164	0.232	0.332	0.430	0.468	1.348	1.717
10000	0.052	0.104	0.163	0.225	0.369	0.373	0.460	1.304	2.318

Table 5: Multiplication on 2.66 Ghz Intel i7

	CPU time in seconds								
n	$e=2$	$e=3$	$e=4$	$e=5$	$e=6$	$e=7$	$e=8$	$e=9$	$e=10$
1000	0.015	0.024	0.025	0.051	0.058	0.067	0.090	0.413	0.788
2000	0.100	0.167	0.177	0.244	0.298	0.349	0.435	2.711	4.212
3000	0.310	0.338	0.380	0.647	0.730	0.896	1.089	8.171	11.735
4000	0.483	0.691	0.792	1.304	1.558	1.928	2.194	18.272	24.791
5000	0.918	1.158	1.327	1.954	2.476	3.063	3.589	33.692	43.648
6000	1.123	1.448	1.836	2.995	4.012	4.869	5.691	55.851	69.846
7000	1.854	2.279	2.777	4.471	5.738	7.085	8.648	87.633	106.048
8000	2.258	2.738	4.032	6.102	8.049	9.835	12.091	127.787	149.665
9000	2.459	3.842	4.892	9.145	11.567	13.450	16.116	169.167	201.220
10000	3.329	4.999	7.188	10.753	14.320	16.823	21.934	223.253	280.286
	CPU cycles / $n^{2.807}$								
n	$e=2$	$e=3$	$e=4$	$e=5$	$e=6$	$e=7$	$e=8$	$e=9$	$e=10$
1000	0.154	0.245	0.253	0.515	0.584	0.678	0.913	4.159	7.934
2000	0.144	0.240	0.254	0.351	0.429	0.503	0.625	3.899	6.057
3000	0.142	0.156	0.175	0.298	0.336	0.413	0.501	3.764	5.406
4000	0.098	0.142	0.162	0.267	0.320	0.396	0.450	3.753	5.092
5000	0.100	0.127	0.145	0.214	0.271	0.336	0.394	3.699	4.792
6000	0.072	0.095	0.120	0.197	0.264	0.320	0.374	3.675	4.596
7000	0.078	0.097	0.118	0.190	0.244	0.302	0.369	3.741	4.527
8000	0.066	0.080	0.118	0.179	0.236	0.288	0.354	3.750	4.392
9000	0.050	0.081	0.103	0.192	0.243	0.283	0.339	3.566	4.242
10000	0.052	0.078	0.112	0.168	0.224	0.263	0.344	3.501	4.396

Table 6: Gaussian elimination on 2.66 Ghz Intel i7

[2] M. R. Albrecht and G. V. Bard. *The M4RI Library – Version 20111004*. The M4RI Team, 2011. http://m4ri.sagemath.org.

[3] M. R. Albrecht, G. V. Bard, and C. Pernet. Efficient dense Gaussian elimination over the field with two elements. arXiv:1111.6549 [cs.MS], 2011.

[4] M. R. Albrecht and J.-G. Dumas. e-mail conversation on [linbox-use] mailing list, November 2011. https://groups.google.com/group/linbox-use/t/2f64a0ae683a2c.

[5] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk.*, 194(11), 1970. (in Russian), English Translation in Soviet Math Dokl.

[6] R. Barbulescu, J. Detrey, N. Estibals, and P. Zimmermann. Finding optimal formulae for bilinear maps. Cryptology ePrint Archive, Report 2012/110, 2012. http://eprint.iacr.org/.

[7] T. J. Boothby and R. Bradshaw. Bitslicing and the Method of Four Russians over larger finite fields. arxiv:0901.1413, 2009.

[8] W. Bosma, J. Cannon, and C. Playoust. The MAGMA Algebra System I: The User Language. In *Journal of Symbolic Computation 24*, pages 235–265. Academic Press, 1997.

[9] R. P. Brent, P. Gaudry, E. Thomé, and P. Zimmermann. Faster multiplication in GF(2)[x]. In *Proceedings of the 8th International Conference on Algorithmic Number Theory*, ANTS-VIII'08, pages 153–166, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - the Advanced Encryption Standard*. Springer Verlag, Berlin, Heidelberg, New York, 2002.

[11] J.-G. Dumas, L. Fousse, and B. Salvy. Simultaneous modular reduction and kronecker substitution for small finite fields. *J. Symb. Comput.*, 46:823–840, July 2011.

[12] J.-G. Dumas, P. Giorgi, and C. Pernet. FFPACK: finite field linear algebra package. In *ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, ISSAC '04, pages 119–126, New York, NY, USA, 2004. ACM.

[13] J.-C. Faugère and S. Lachartre. Parallel Gaussian Elimination for Gröbner bases computations in finite fields. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 89–97, 2010.

[14] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2007.

[15] P. Giorgi, C.-P. Jeannerod, and G. Villard. On the complexity of polynomial matrix computations. In *ISSAC '03: Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation*, pages 135–142. ACM Press, 2003.

[16] F. Gray. Pulse code communication, March 1953. US Patent No. 2,632,058.

[17] C.-P. Jeannerod, C. Pernet, and A. Storjohann. Rank-profile revealing Gaussian elimination and the CUP matrix decomposition. arXiv:1112.5717, page 35 pages, 2012.

[18] R. Kleiser, J. Travolta, O. Newton-John, J. Jacobs, W. Casey, B. Woodard, and A. Carr. *Grease*. Paramount Pictures, 1978.

[19] D. Lazard. Gröbner-bases, Gaussian elimination and resolution of systems of algebraic equations. In *Proceedings of the European Computer Algebra Conference on Computer Algebra*, volume 162 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, 1983. Springer Verlag.

[20] P. L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Trans. on Computers*, 53(3):362–369, 2005.

[21] V. Shoup. NTL. http://www.shoup.net/ntl/, 2009. version 5.4.2.

[22] A. Steel. Private communication. 29. November, 2011.

[23] W. Stein et al. *Sage Mathematics Software (Version 4.7.1)*. The Sage Development Team, 2011. http://www.sagemath.org.

[24] V. Strassen. Gaussian elimination is not optimal. *Nummerische Mathematik*, 13:354–256, 1969.