Efficient Multiplication of Dense Matrices over GF(2)

MARTIN ALBRECHT
Information Security Group, Royal Holloway, University of London
GREGORY BARD
Department of Mathematics, Fordham University
and
WILLIAM HART
Department of Mathematics, University of Warwick

We describe an efficient implementation of a hierarchy of algorithms for multiplication of dense matrices over the field with two elements ($\mathbb{F}_2$). In particular we present our implementation – in the M4RI library – of Strassen-Winograd matrix multiplication and the "Method of the Four Russians" multiplication (M4RM) and compare it against other available implementations. Superior performance is demonstrated on Intel's Core 2 Duo and good performance on AMD's Opteron. The open-source M4RI library is available stand-alone and as part of the Sage mathematics software.

Categories and Subject Descriptors: G.4 [**MATHEMATICAL SOFTWARE**]:

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: GF(2), matrix, linear algebra, multiplication, Strassen, greasing

1. INTRODUCTION

We describe an efficient implementation of a hierarchy of algorithms for multiplication of dense matrices over the field with two elements ($\mathbb{F}_2$). Matrix/matrix multiplication is an important primitive in computational linear algebra and as such the fundamental algorithms we implement are well-known. Therefore this paper focuses on the numerous techniques employed for the special case of $\mathbb{F}_2$ in the M4RI library (`http://m4ri.sagemath.org`).

We note that even for problems that do not reduce to matrix/matrix multiplication many of the techniques presented in this paper are still applicable. For instance, Gaussian elimination can be achieved via the "Method of the Four Russians" Inversion (M4RI)(cf. [Bard 2007, Ch. 5] and [Bard 2008]) and borrows ideas from the "Method of the Four Russians" Multiplication (M4RM) [Arlazarov et al. 1970] which we present here.

The first author is supported by the Royal Holloway Valerie Myerscough Scholarship. The final author was supported by EPSRC grant EP/D079543/1.

The M4RI library implements dense linear algebra over $\mathbb{F}_2$ and is used by Sage [The SAGE Group 2008] and PolyBoRi [Brickenstein and Dreyer 2007].

Our optimization efforts focus on 64 bit x86 architectures (x86_64), specifically the Intel Core 2 Duo and the AMD Opteron. Thus, we assume in this paper that each native CPU word has 64-bits: $w_s = 64$. However it should be noted that our code also runs on 32-bit CPUs and on non-x86 CPUs such as the PowerPC.

In machine terms, addition in $\mathbb{F}_2$ is logical-XOR, and multiplication is logical-AND, thus a machine word of 64-bits allows one to operate on 64 elements of $\mathbb{F}_2$ in parallel: at most one CPU cycle for 64 parallel additions or multiplications. As such, element-wise operations over $\mathbb{F}_2$ are relatively cheap. In fact, in this paper, we conclude that the actual bottlenecks are memory reads and writes and issues of data locality. We present our empirical findings in relation to minimizing these and give an analysis thereof.

The second author proposed, in [Bard 2006] and [Bard 2007, Ch. 5], to count memory accesses rather than arithmetic operations to estimate the complexity of such algorithms and the empirical results of this paper lend further support to this model. However, this model is a simplification as memory access is not uniform, i.e. an algorithm which randomly accesses memory will perform much worse than an algorithm with better spatial and temporal locality. While these differences only affect the constant of a complexity estimation, in practice they make a very significant difference, as our results will demonstrate.

The paper is structured as follows. We proceed from basic arithmetic (Section 2) via the classical cubic multiplication algorithm (Section 2.3), through a detailed discussion of the "Method of the Four Russians" (Section 3) to the Strassen-Winograd algorithm (Section 4).

We start by introducing our basic data structures and conclude by presenting timing experiments to show the validity of our approach (Section 6).

Note, that all timings in this paper time Strassen-Winograd multiplication (cf. Section 4) but with different base cases.

2. BASIC ARITHMETIC

2.1 Our Matrix Data Structure

We use a "flat row-major representation" for our matrices. Thus 64 consecutive entries in one row are packed into one machine word. Consequently, bulk operations on whole rows are considerably cheaper than on whole columns and addressing a single column is more expensive than addressing a single row. Additionally, we maintain an array – called `rowswap` – containing the address in memory of the first word for each row in the matrix. To represent in-place submatrices (i.e. without copying out the data) we also use this `rowswap` array. We call these in-place submatrices "matrix windows" and they consist of addresses of the first word of each row and the number of columns each row contains. This approach is limited to "matrix windows" which start and end at full word borders but this is sufficient for our application. The advantages and disadvantages of the "flat row-major" data structure are, for instance, analyzed in [Pernet 2001].

2.2 Row Additions

Since this basic operation – addition of two rows – is at the heart of every algorithm in this paper we should briefly mention the SSE2 instruction set [Fog 2008] which is available on modern x86_64 architectures. This instruction set offers an XOR operation for 128-bit wide registers, allowing one to handle two 64-bit machine words in one instruction. The use of these instructions does provide a considerable speed improvement on Intel CPUs. Table I shows that up to a 25% improvement is possible when enabling SSE2 instructions. However, in our experiments performance declined on Opteron CPUs when using SSE2 instructions. The authors were unable to identify a cause of this phenomenon.

Matrix Dimensions	Using 64-bit	Using 128-bit (SSE2)
$10,000 \times 10,000$	1.981	1.504
$16,384 \times 16,384$	7,906	6.074
$20,000 \times 20,000$	14.076	10.721
$32,000 \times 32,000$	56.931	43.197

Table I. Strassen-Winograd multiplication on 64-bit Linux, 2.33Ghz Core 2 Duo

2.3 Cubic Multiplication

The simplest multiplication operation involving matrices is a matrix/vector product which can easily be extended to classical cubic matrix/matrix multiplication. To compute the matrix/vector product Ab we have to compute the dot product of each row i of A and the vector b. If the vector b is stored as a row rather than a column, this calculation becomes equivalent to word-wise logical-AND and accumulation of the result in a word p via logical-XOR. Finally, the parity of p needs to be computed. However, as there is no native parity instruction in the x86_64 instruction set this last step is quite expensive compared to the rest of the routine. To account for this, 64 parity bits can be computed in parallel [Warren 2002, Ch. 5]. To extend this matrix/vector multiplication to matrix/matrix multiplication B must be stored transposed.

3. THE METHOD OF THE FOUR RUSSIANS

The "Method of the Four Russians" matrix multiplication algorithm can be derived from the original algorithm published by Arlazarov, Dinic, Kronrod, and Faradzev [Arlazarov et al. 1970], but does not directly appear there. It has appeared in books including [Aho et al. 1974, Ch. 6].

Consider a product of two matrices $C = AB$ where A is an $m \times l$ matrix and B is an $l \times n$ matrix, yielding an $m \times n$ for C. A can be divided into l/k vertical "stripes" $A_0 \dots A_{(l-1)/k}$ of k columns each, and B into l/k horizontal stripes $B_0 \dots B_{(l-1)/k}$ of k rows each. (For simplicity assume k divides l). The product of two stripes, $A_i B_i$ requires an $m \times l/k$ by $l/k \times n$ matrix multiplication, and yields an $m \times n$ matrix C_i. The sum of all k of these C_i equals C.

$$C = AB = \sum_{0}^{(l-1)/k} A_i B_i.$$

Example: Consider $k = 1$ and

$$A = \begin{pmatrix} a_0 & a_1 \\ a_2 & a_3 \end{pmatrix}, B = \begin{pmatrix} b_0 & b_1 \\ b_2 & b_3 \end{pmatrix}.$$

Then

$$A_0 = \begin{pmatrix} a_0 \\ a_2 \end{pmatrix}, A_1 = \begin{pmatrix} a_1 \\ a_3 \end{pmatrix}, B_0 = \begin{pmatrix} b_0 & b_1 \end{pmatrix}, \text{ and } B_1 = \begin{pmatrix} b_2 & b_3 \end{pmatrix}$$

and consequently

$$A_0B_0 = \begin{pmatrix} a_0b_0 & a_0b_1 \\ a_2b_0 & a_2b_1 \end{pmatrix} \text{ and } A_1B_1 = \begin{pmatrix} a_1b_2 & a_1b_3 \\ a_3b_2 & a_3b_3 \end{pmatrix}.$$

Finally, we have

$$C = AB = A_0B_0 + A_1B_1 = \begin{pmatrix} a_0b_0 + a_1b_2 & a_0b_1 + a_1b_3 \\ a_2b_0 + a_3b_2 & a_2b_1 + a_3b_3 \end{pmatrix}.$$

The principal benefit of multiplying in narrow stripes is that the bits across each row of a stripe of A determine which linear combination of rows of B will contribute to the product, e.g. in the above example $a_0, \ldots, a_3$ dictate which linear combination of b_0, b_2 and b_1, b_3 must be written to the rows of C. However, if the stripe is relatively narrow as in this example, there is only a small finite number of binary values each row of the stripe can take, and thus only a small number of possible linear combinations of the rows of B that will be "selected". If we precompute all possible linear combinations of rows of B that could be selected we can create a lookup table into which the rows of the stripes of A can index.

Returning to our example, if $a_0 = a_2$ and $a_1 = a_3$ then the same linear combination would be written to the first and the second row of C. Precomputation of all $2^4 - 1$ non-zero linear combinations, $(1 \cdot b_0 + 0 \cdot b_1, 0 \cdot b_0 + 1 \cdot b_1, 1 \cdot b_0 + 1 \cdot b_1)$, ensures that the repeated linear combination has only been computed once. In our trivial example this is not a saving, but for much larger matrices reuse of the precomputed combinations gives a saving. Precomputing a table in this fashion is also called "greasing".

The technique just described gives rise to Algorithm 1. In Algorithm 1 the subroutine `read_bits(A, r, sc, k)` reads k bits from row r starting at column sc and returns the bitstring interpreted as an integer and `add_row_from_table(C, r, T, x)` adds the row x from T to the row j of C. The subroutine `make_table(B, r, c, k)` in Algorithm 1 constructs a table T of all $2^k - 1$ non-zero linear combinations of the rows of B starting in row r and column c. For this calculation Gray codes are used.

3.1 Gray Codes

The Gray code [Gray 1953], named after Frank Gray and also known as reflected binary code, is a binary numeral system where two consecutive values differ in only one digit. Examples of Gray codes for two, three and four bits are given in Figure 3.1.

Gray code tables for n-bits can be computed efficiently from $n-1$-bit Gray code tables by prepending each entry of the $n-1$-bit Gray code table with 0. Then the

Algorithm 1 M4RM

```
def add_row_from_table(C, r, T, x):
  for 0 <= i < C.ncols():
    C[r,i] += T[x,i]

def read_bits(A, r, sc, k):
  return sum(int(A[r,i])^(k-i-1) for sc <= i < sc+k)

def m4rm(A, B, k):
  m = A.nrows(); l = A.ncols(); n = B.ncols()
  C = Matrix(GF(2), m, n) # zero matrix

  for 0 <= i < l/k:
    # create table of 2^k-1 linear combinations
    T = make_table(B, i*k, 0, k)
    for 0 <= j < m:
      # read index for table T
      x = read_bits(A, j, k*i, k)
      # add appropriate row from table T
      add_row_from_table(C, j, T, x)
  return C
```

0	0
0	1
1	1
1	0

2-bit Gray Code

0	0	0
0	0	1
0	1	1
0	1	0
1	1	0
1	1	1
1	0	1
1	0	0

3-bit Gray Code

0	0	0	0
0	0	0	1
0	0	1	1
0	0	1	0
0	1	1	0
0	1	1	1
0	1	0	1
0	1	0	0
1	1	0	0
1	1	0	1
1	1	1	1
1	1	1	0
1	0	1	0
1	0	1	1
1	0	0	1
1	0	0	0

4-bit Gray Code

Fig. 1. Gray Codes

order of the entries is reversed and a 1 is prepended to each entry. These two half-tables are then concatenated. These tables can then be used to construct all $2^k - 1$ non-zero linear combinations of k rows where each new entry in the table costs one row addition as its index differs in exactly one bit from that of the preceding row. Thus computing all $2^k - 1$ non-zero linear combinations of k rows can be done in $2^k - 1$ row additions, rather than $(k/2-1)2^k - 1$ as would be expected if each vector were to be tabulated separately.

From the complexity analysis in [Bard 2006] it seems one should always choose the parameter $k = \lfloor \log_2 n \rceil$ for an $n \times n$ matrix. However, in practice this is not the case. First, experimental evidence indicates [Bard 2007] that $0.75 \times \log_2 n$ seems to be a better choice. Also, for cache efficiency it makes sense to split the input matrices into blocks such that these blocks fit into L2 cache (see below). If that technique is employed then the block sizes dictate k and not the total dimensions of the input matrices. Thus, in practice, a much smaller k than $\log_2 n$ is found to be optimal (see below). However, in practice, restraining k in this way actually improves performance.

We pre-compute the Gray Code tables up to size 16. For matrices of dimension > 20 million rows and columns, this is not enough. But, such a dense matrix would have nearly half a quadrillion entries, and this is currently beyond the capabilities of existing computational hardware. Also, for these dimensions the Strassen-Winograd algorithm should be used.

3.2 A Cache Friendly Version

Note that the M4RM algorithm creates a table for each stripe of B and then iterates over all rows of C and A in the inner loop. If the matrices C and A are bigger than L2 cache then this means that for each single row addition a new row needs to be loaded from RAM. This row will evict an older row from L2. However, as this row is used only once per iteration of all rows of A and C we cannot take advantage of the fact that it is now in L2 cache. Thus if the matrices A and C do not fit into L2 cache the algorithm does not utilize this faster memory.

Thus, it is advantageous to re-arrange the algorithm in such a way that it iterates over the upper part of A completely with all tables for B before going on to the next part. This gives rise to Algorithm 2, a cache friendly version of the M4RM algorithm. For simplicity we assume that m, l, n are all multiples of some fixed block size in the presentation of Algorithm 2. This cache-friendly rearrangement is paid for by the repeated regeneration of the table T. However, compared to the inner loop, this is a cheap operation and thus is outweighed by the better data locality. Table II shows that this strategy provides considerable performance improvements.

3.3 Increasing the Number of Gray Code Tables

Recall that the actual arithmetic is quite cheap compared to memory reads and writes and that the cost of memory accesses greatly depends on where in memory data is located: the L1 cache is approximately 50 times faster than main memory. It is thus advantageous to try to fill all of L1 with Gray code tables. For example consider $n = 10000$, $k = 10$ and one Gray code table. In this situation we work on 10 bits at a time. If we use $k = 9$ and two Gray code tables, we still use the same memory for the tables but can deal with 18 bits at once. The price we pay

Algorithm 2 Cache Friendly M4RM

```
def m4rm_cf(A, B, k):
  m = A.nrows(); l = A.ncols(); n = B.ncols()
  C = Matrix(GF(2), m, n) # zero matrix

  for 0 <= start < m/block_size:
    for 0 <= i < l/k:
      T = make_table(B, i*k, 0, k)
      for 0 <= s < block_size:
        j = start*block_size + s;
        x = read_bits(A, j, k*i, k)
        add_row_from_table(C, j, T, x)
  return C
```

is one additional row addition, which is cheap if the operands are all in cache. To implement this enhancement the algorithm remains almost unchanged, except that t tables are generated for t consecutive rows of B, t values x are read for consecutive entries in A and t rows from t different tables are added to the target row of C. This gives rise to Algorithm 3 where we assume that tk divides l and fix $t = 2$.

Algorithm 3 M4RM with Two Gray Code Tables

```
def add_2rows_from_table(C, r, T0, x0, T1, x1):
  for 0 <= i < C.ncols():
    C[r,i] += T0[x0,i] + T1[x1,i]

def m4rm_2t(A, B, k):
  m = A.nrows(); l = A.ncols(); n = B.ncols()
  C = Matrix(GF(2), m, n) # zero matrix

  for 0 <= i < l/(2*k):
    T0 = make_table(B, 2*i*k, 0, k)
    T1 = make_table(B, 2*i*k + k, 0, k)
    for 0 <= j < m:
      x0 = read_bits(A, j, 2*k*i, k)
      x1 = read_bits(A, j, 2*k*i+k, k)
      add_2rows_from_table(C, j, T0, x0, T1, x1)
  return C
```

Table II shows that increasing the number of tables is advantageous. Our implementation uses eight Gray code tables, which appears to be a good default value according to our experiments.

	"base cases" (cf. Section 5)			
Matrix Dimensions	Algorithm 1	Algorithm 2	Algorithm 3, $t = 2$	Algorithm 3, $t = 8$
$10,000 \times 10,000$	4.141	2.866	1.982	1.599
$16,384 \times 16,384$	16.434	12.214	7.258	6.034
$20,000 \times 20,000$	29.520	20.497	14.655	11.655
$32,000 \times 32,000$	86.153	82.446	49.768	44.999

Table II. Strassen-Winograd with different base cases on 64-bit Linux, 2.33Ghz Core 2 Duo

4. STRASSEN-WINOGRAD MULTIPLICATION

In 1969 Volker Strassen [Strassen 1969] published an algorithm which multiplies two block matrices

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

with only seven submatrix multiplications and 18 submatrix additions rather than eight multiplications and eight additions. Since matrix multiplication ($\mathcal{O}(n^\omega)$, $\omega \geq 2$) is considered more expensive than matrix addition ($\mathcal{O}(n^2)$) this is an improvement. Later the algorithm was improved by Winograd to use 15 submatrix additions only, the result is commonly referred to as Strassen-Winograd multiplication. While both algorithms are to a degree less numerically stable than classical cubic multiplication over floating point numbers [Higham 2002, Ch. 26.3.2] this problem does not affect matrices over finite fields and thus the improved complexity of $\mathcal{O}(n^{\log_2 7})$ [Strassen 1969; Bard 2007] is applicable here.

Let m, l and n be powers of two. Let A and B be two matrices of dimension $m \times l$ and $l \times n$ and let $C = A \times B$. Consider the block decomposition

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

where A_{00} and B_{00} have dimensions $m/2 \times l/2$ and $l/2 \times n/2$ respectively. The Strassen-Winograd algorithm, which computes the $m \times n$ matrix $C = A \times B$, is given in Algorithm 4.

The subroutine `augment` in Algorithm 4 takes two $m \times l$ and $m \times n$ matrices A and B and returns the $m \times (n + l)$ matrix $C = (A\ B)$ and the subroutine `stack` takes two $m \times n$ and $l \times n$ matrices A and B and returns the $(m + l) \times n$ matrix

$$C = \begin{pmatrix} A \\ B \end{pmatrix}.$$

At each recursion step the matrix dimensions must be divisible by two which explains the requirement of them being powers of two. However, in practice the recursion stops at a given *cutoff* dimension (c_o) and switches over to another multiplication algorithm. In our case, this is the M4RM algorithm. Thus the requirement can be relaxed to the requirement that for each recursion step the matrix dimensions must be divisible by two.

However, this still is not general enough. Additionally, in case of $\mathbb{F}_2$ the optimal case is when m, n, l are 64 times powers of 2 to avoid cutting within words. To deal with odd-dimensional matrices two strategies are known in the literature [Huss-Lederman et al. 1996]: One can either increase the matrix dimensions – this is

Algorithm 4 Strassen-Winograd

```
def strassen_winograd(A,B):
  m = A.nrows(); l = A.ncols(); n = B.ncols()
  A00 = A.submatrix(0, 0, m/2, l/2)
  A01 = A.submatrix(0, l/2, m/2, l)
  A10 = A.submatrix(m/2, 0, m, l/2)
  A11 = A.submatrix(m/2, l/2, m, l)

  B00 = B.submatrix(0, 0, l/2, n/2)
  B01 = B.submatrix(0, n/2, l/2, n)
  B10 = B.submatrix(l/2, 0, l, n/2)
  B11 = B.submatrix(l/2, n/2, l, n)

  # 8 additions
  S0 = A10 + A11; S1 = S0 - A00
  S2 = A00 - A10; S3 = A01 - S1
  T0 = B01 - B00; T1 = B11 - T0
  T2 = B11 - B01; T3 = T1 - B10

  # 7 recursive multiplications
  P0 = A00 * B00; P1 = A01 * B10
  P2 = S3 * B11 ; P3 = A11 * T3
  P4 = S0 * T0; P5 = S1 * T1
  P6 = S2 * T2

  # 7 final additions
  U0 = P0 + P1; U1 = P0 + P5
  U2 = U1 + P6; U3 = U1 + P4
  U4 = U3 + P2; U5 = U2 - P3
  U6 = U2 + P4

  C0 = augment(U0, U4)
  C1 = augment(U5, U6)
  C = stack(C0,C1)
  return C
```

called “padding” – to the next “good” value and fill the additional entries with zeros, yielding A^+ and B^+. Then one can compute $C^+ = A^+B^+$ and finally cut out the actual product matrix C from the bigger matrix C^+. A variant of this approach is to only virtually append rows and columns, i.e. we pretend they are present. Another approach is to consider the largest submatrices A^- and B^- of A and B so that the dimensions of A^- and B^- match our requirements – this is called “peeling”. Then once the product $C^- = A^-B^-$ is computed, one fixes up the remaining rows and columns of C from the remaining rows and columns of A and B that are not in A^- and B^- (cf. [Huss-Lederman et al. 1996]). For those remaining pieces Strassen-Winograd is not used but an implementation which does not cut the matrices into submatrices. We use the "peeling" strategy in our implementation, but note that it is easy to construct a case where our strategy is clearly not optimal, Table III gives an example where "padding" would only add one row and one column, while "peeling" has to remove many rows and columns. This is an area for future improvements.

Matrix Dimensions	Time in s
$2^{14} - 1 \times 2^{14} - 1$	7.86
$2^{14} \times 2^{14}$	6.09
$2^{14} + 1 \times 2^{14} + 1$	6.11

Table III. "Peeling" strategy on 64-bit Linux, 2.33Ghz, Core 2 Duo

To represent the submatrices in Algorithm 4 we use "matrix windows" as described earlier. While this has the benefit of negligible required additional storage compared to out-of-place submatrices, this affects data locality negatively. To remedy this, we copy out the target matrix C when switching from Strassen-Winograd to M4RM. On the other hand, copying out A and B at this crossover point does not improve performance. Data locality for B is achieved through the Gray code tables and it appears that the read of x from A (cf. Algorithm 1) does not significantly contribute to the runtime.

However, even with "matrix windows" Strassen-Winograd requires more memory than classical cubic multiplication. Additional storage is required to store intermediate results. The most memory-efficient scheduler (cf. [Dumas and Pernet 2007]) uses two additional temporary submatrices and is utilized in our implementation. We also tried the "proximity schedule" used in FFLAS [Pernet 2001] but did not see any improved performance.

5. TUNING PARAMETERS

Our final implementation calls Strassen-Winograd, which switches over to M4RM if the input matrix dimensions are less than a certain parameter c_o. If B then has fewer columns than w_s the classical cubic algorithm is called. This last case is quite common in the fix-up step of "peeling". This strategy gives three parameters for tuning. The first is c_o, the crossover point where we switch from Strassen-Winograd to M4RM. Second, b_s is the size for block decomposition inside M4RM for cache friendliness. Third, k dictates the size of the used Gray code tables. We always fix the number of Gray code tables to $t = 8$.

By default c_s is chosen such that *two* matrices fit into L2 cache, because this provides the best performance in our experiments. For the Opteron (1MB of L2 cache) this results in $c_s = 2048$ and for the Core 2 Duo (4MB of L2 cache) this results in $c_s = 4096$. We only fit two matrices, rather than all three matrices in L2 cache as b_s reduces the size of the matrices we are working with to actually fit three matrices in L2 cache. The default value is fixed at $b_s = c_s/2$. The value k is set to $\lfloor 0.75 \times \log_2 b_s \rfloor - 2$. We subtract 2 as a means to compensate for the use of 8 Gray code tables. However, if additionally reducing k by 1 would result in fitting all Gray code tables in L1 cache, we do that. Thus, k is either $\lfloor 0.75 \times \log_2 b_s \rfloor - 2$

or $\lfloor 0.75 \times \log_2 b_s \rfloor - 3$ depending on the input dimensions and the size of the L1 cache. These values have been determined empirically and seem to provide the best compromise across platforms.

On the Opteron these values — $c_s = 2048$, $b_s = 1024$, $k = 5$, $t = 8$ Gray code tables — mean that the two input matrices fit into the 1MB of L2 cache, while the 8 Gray code tables fit exactly into L1: $8 \cdot 2^5 \cdot 2048/8 = 64$Kb. The influence of the parameter b_s in the final implementation is shown in Table IV for fixed $k = 5$ and $c_s = 2048$.

On the Core 2 Duo these values are $c_s = 4096$, $b_s = 2048$, $k = 6$, $t = 8$ and ensure that all data fits into L2 cache. Since the Core 2 Duo has only 32kb of L1 cache we do not try to fit all tables into it. So far in our experiments, performance did not increase when we tried to optimize for L1 cache.

Matrix Dimensions	$b_s = 2048$	$b_s = 1024$	$b_s = 768$
$10,000 \times 10,000$	2.96	2.49	2.57
$16,384 \times 16,384$	13.23	10.49	10.37
$20,000 \times 20,000$	21.19	17.73	18.11
$32,000 \times 32,000$	67.64	67.84	69.14

Table IV. Strassen-Winograd multiplication, 64-bit Linux, 2.6Ghz Opteron

6. RESULTS

To evaluate the performance of our implementation we provide benchmark comparisons against the best known implementations we are aware of. First, Magma [Bosma et al. 1997] is widely known for its high performance implementations of many algorithms. Second, GAP [The GAP Group 2007] (or equivalently the C-MeatAxe [Ringe 2007]) is to our knowledge the best available open-source implementation of dense matrix multiplication over $\mathbb{F}_2$. Note, that the high-performance FFLAS [Pernet 2001] library does not feature a dedicated implementation for $\mathbb{F}_2$. In the Tables V and VI we give the average of ten observed runtimes and RAM usage for multiplying two random square matrices. The timings for M4RI were obtained using Sage [The SAGE Group 2008]. M4RI was compiled with GCC 4.3.1 on both machines and we used the options `-O2` on the Opteron machine and `-O2 -msse2` on the Core 2 Duo machine.

	Magma 2.14-14		GAP 4.4.10		M4RI-20080821	
Matrix Dimensions	Time	RAM	Time	RAM	Time	RAM
$10,000 \times 10,000$	2.210 s	85 MB	6.691 s	60 MB	1.504 s	60 MB
$16,384 \times 16,384$	8.670 s	219 MB	26.341 s	156 MB	6.074 s	156 MB
$20,000 \times 20,000$	16.030 s	331 MB	—	—	10.721 s	232 MB
$32,000 \times 32,000$	58.730 s	850 MB	—	—	43.197 s	589 MB

Table V. 64-bit Debian/GNU Linux, 2.33Ghz Core 2 Duo

	Magma 2.14-13		GAP 4.4.10		M4RI-20080811	
Matrix Dimensions	Time	RAM	Time	RAM	Time	RAM
$10,000 \times 10,000$	2.656 s	85 MB	10.472 s	60 MB	2.490 s	60 MB
$16,384 \times 16,384$	10.260 s	219 MB	43.658 s	156 MB	10.490 s	156 MB
$20,000 \times 20,000$	18.156 s	331 MB	—	—	17.730 s	232 MB
$32,000 \times 32,000$	67.237 s	850 MB	—	—	67.840 s	589 MB

Table VI. 64-bit Debian/GNU Linux, 2.6Ghz Opteron,

REFERENCES

AHO, A., HOPCROFT, J., AND ULLMAN., J. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

ARLAZAROV, V., DINIC, E., KRONROD, M., AND FARADZEV, I. 1970. On economical construction of the transitive closure of a directed graph. *Dokl. Akad. Nauk. 194,* 11. (in Russian), English Translation in Soviet Math Dokl.

BARD, G. 2008. Matrix inversion (or LUP-factorization) via the Method of Four Russians, in $\theta(n^3/\log n)$ time. In Submission.

BARD, G. V. 2006. Accelerating Cryptanalysis with the Method of Four Russians. Cryptology ePrint Archive, Report 2006/251. Available at `http://eprint.iacr.org/2006/251.pdf`.

BARD, G. V. 2007. Algorithms for solving linear and polynomial systems of equations over finite fields with applications to cryptanalysis. Ph.D. thesis, University of Maryland. Available at `http://www.cs.umd.edu/~jkatz/THESES/bard_thesis.pdf`.

BOSMA, W., CANNON, J., AND PLAYOUST, C. 1997. The MAGMA Algebra System I: The User Language. In *Journal of Symbolic Computation 24*. Academic Press, 235–265.

BRICKENSTEIN, M. AND DREYER, A. 2007. PolyBoRi: A framework for Gröbner basis computations with Boolean polynomials. In *Electronic Proceedings of MEGA 2007.* Available at `http://www.ricam.oeaw.ac.at/mega2007/electronic/26.pdf`.

DUMAS, J.-G. AND PERNET, C. 2007. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. Available at `http://www.citebase.org/abstract?id=oai:arXiv.org:0707.2347`.

FOG, A. 2008. Optimizing software in C++. Available at `http://www.agner.org/optimize`.

GRAY, F. 1953. Pulse code communication. USA Patent 2,632,058.

HIGHAM, N. 2002. *Accuracy and Stability of Numerical Algorithms*, second ed. Society for Industrial and Applied Mathematics.

HUSS-LEDERMAN, S., JACOBSON, E. M., JOHNSON, J. R., TSAO, A., AND TURNBULL, T. 1996. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of Supercomputing '96*.

PERNET, C. 2001. Implementation of Winograd's algorithm over finite Fields using ATLAS Level3 Blas. Tech. rep., ID-Laboratory.

RINGE, M. 2007. Meataxe 2.4.8. Available at `http://www.math.rwth-aachen.de/~MTX/`.

STRASSEN, V. 1969. Gaussian elimination is not optimal. *Nummerische Mathematik 13*, 354–256.

The GAP Group 2007. *GAP – Groups, Algorithms, and Programming, Version 4.4.10.* The GAP Group.

The SAGE Group 2008. *SAGE Mathematics Software (Version 3.1.2).* The SAGE Group. Available at `http://www.sagemath.org`.

WARREN, H. S. 2002. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.