Dense Linear Algebra over Finite Fields: the FFLAS and FFPACK packages

Jean-Guillaume Dumas, Thierry Gautier, Pascal Giorgi, Clément Pernet

Dense Linear Algebra over Finite Fields: the FFLAS and FFPACK packages

Jean-Guillaume Dumas
Université de Grenoble

Thierry Gautier
INRIA Rhône-ALpes

Pascal Giorgi
University of Waterloo / Université de Perpignan

Clément Pernet
Université de Grenoble

January 31, 2006

Abstract

In the last past two decades, several efforts have been made to reduce exact linear algebra problems to matrix multiplication in order to provide algorithms with optimal asymptotic complexity. To provide efficient implementations of such algorithms one need to be careful with the underlying arithmetic. It is well know that modular technique such as Chinese remainder algorithm or p-adic lifting allow in practice to achieve better performances especially when word size arithmetic are used. Therefore, finite field arithmetics becomes an important core for efficient exact linear algebra libraries.

In this paper we study different implementations of finite field in order to achieve efficiency for basic linear algebra routines such as dot product or matrix multiplication; our goal being to provide an exact alternate to numerical BLAS library. Following matrix multiplication reductions, our kernel has many symbolic linear algebra applications: symbolic triangularization, system solving, exact determinant computation and matrix inversion are then studied and we demonstrate the efficiency of these reductions in practice.

1 Introduction

Finite fields play a crucial role in computational algebra. Indeed, finite fields are the basic representation used to solve many integer problems. The whole solutions are then gathered via the Chinese remainders or lifted p-adically. Among those problems are integer polynomial factorization [58], integer system solving [12, 55], integer matrix normal forms [25] or integer determinant [41]. Finite

fields are also of intrinsic use in polynomial linear algebra [28] but also in cryptology (e.g. large integer factorization [47], discrete logarithm computations [49]) or for error correcting codes. Moreover, nearly all of these problems involve linear algebra resolutions. Therefore, a fundamental issue is to implement efficient elementary arithmetic operations and very fast linear algebra routines over finite fields.

We propose a way to implement the equivalent of the basic BLAS level 1, 2, and 3 numerical routines (respectively dot product, matrix-vector product and matrix-matrix product), but over finite fields. We will focus on implementations over fields with small cardinality, namely not exceeding machine word size, but with any characteristic (consequently, we do not deal with optimizations for powers of 2 cardinalities). For instance, we show that **symbolic matrix multiplication can be as fast as numerical matrix multiplication** (see section 3) when using word size finite fields. Our aim is *not* to rebuild some specialized routines for each field instance. Instead, the main idea is to use a very efficient and automatically tuned numerical library as a kernel (namely ATLAS [57]) and to make some conversions in order to perform an *exact* matrix multiplication (i.e. *without any loss of precision*). The performances will be reached by performing as few conversions as possible. Moreover, when dealing with symbolic computations, fast matrix multiplication algorithms, such as Strassen's or Winograd's variant [26], do not suffer from instability problems. Therefore their implementation can only focus on high efficiency.

Many algorithms have been designed to use matrix multiplication in order to be able to prove an optimal theoretical complexity. In practice those algorithms were only seldom used. This is the case e.g. in many linear algebra problems such as determinant, rank, inverse, system solution or minimal and characteristic polynomial. We believe that with our kernel, each one of those optimal complexity algorithms can also be the most efficient. One goal of this paper is then to show the actual effectiveness of this belief. In particular we focus on matrix factorization of any shape and any rank matrices. The application of such factorization to determinant, rank, and inverse is presented as well.

Some of the ideas from preliminary versions of this paper [21], in particular the BLAS-based matrix multiplication for small prime fields, are now incorporated into the Maple computer algebra system since its version 8 and also into the 2005 version of the computer algebra system Magma. Therefore an effort towards effective reduction has been made [22] in C++ and within Maple by A. Storjohann[7]. Effective reduction for minimal and characteristic polynomial were proposed in [23] and A. Steel has reported on similar efforts within his implementation of some Magma routines.

The matrix factorization, namely the exact equivalent of the LU factorization is thus extensively studied. Indeed, unlike numerical matrices, exact matrices are very often singular, even more so if the matrix is not square ! Consequently, Ibarra, Moran and Hui have developed generalizations of the LU factorization, namely the LSP and LQUP factorizations [39]. Then we adapt the scheme for rank, determinant, inverse (classical or Moore-Penrose), nullspace computa-

tions, etc.

There, we will give not only the asymptotic complexity measures but the constant factor of the dominant term. Most of these terms will give some constant factor to the multiplication time and we will compare those theoretical ratios to the practical performances we achieve. This will enable us to give a measure of the effectiveness of our reductions (see especially section 4.3). The following two lemmas will be useful there, the first one giving the order of magnitude when both matrix dimensions are equal:

Lemma 1.1. *Let m be a positive integer and suppose that*

1. *$T(m) = CT(\frac{m}{2}) + am^\omega + \epsilon(m)$, with $\epsilon(m) \leq gm^2$ for some constant g.*
2. *$T(1) = e$ for some constant e.*
3. *$\log_2(C) \leq \omega$.*

Then $T(n) = \mathcal{O}(m^\omega)$.

Proof. Let $t = \log_2(m)$. The recursion gives,

$$T(m) = C^t T(1) + am^\omega \frac{1 - \left(\frac{C}{2^\omega}\right)^t}{1 - \frac{C}{2^\omega}} + \sum_{i=0}^{t-1} C^i \epsilon(\frac{m}{2^i}).$$

Then, on the one hand, if $C \neq 4$ this yields $T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + kC^t + g'm^2$, where $g' < \frac{4g}{4-C}$ and $k < T(1) - \frac{a2^\omega}{2^\omega - C} - g'$. On the other hand, when $C = 4$, we have $T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + k'C^t + gm^2 \log_2(m)$, where $k' < T(1) - \frac{a2^\omega}{2^\omega - C}$. In both case, with $C^t = m^{\log_2(C)}$, this gives $T(m) = \frac{a2^\omega}{2^\omega - C} m^\omega + o(m^\omega)$. □

Now we give the order of magnitude when the matrix dimensions differ:

Lemma 1.2. *Let m and n be two positive integers and suppose that*

1. *$T(m;n) = \sum_{i=1}^k c_i T(\frac{m}{2}; n - d_i\frac{m}{2}) + am^\omega + bm^{\omega-1}n + \epsilon(m;n)$, with $C = \sum_{i=1}^k c_i$, $D = \sum_{i=1}^k c_i d_i$ and $\epsilon(m;n) \leq gm^2 + hmn$.*
2. *$T(1,F) \leq eF$ for a constant e.*
3. *$\log_2(C) \leq \omega - 1$*

Then $T(m;n) = \mathcal{O}(m^\omega + m^{\omega-1}n)$.

Proof. As in the preceding lemma, we use the recursion and geometric sums to

get

$$T(m;n) = \sum_{i_1=1}^{k} c_{i_1} \dots \sum_{i_t=1}^{k} c_{i_t} T(1; n - f(d_1, \dots, d_t, m)) + \\ m^\omega \left(a \frac{1 - \left(\frac{C}{2^\omega}\right)^t}{1 - \frac{C}{2^\omega}} - bD \frac{1 - \left(\frac{C}{2^{\omega-1}}\right)^t}{1 - \frac{C}{2^{\omega-1}}} \right) + bm^{\omega-1} n \frac{1 - \left(\frac{C}{2^{\omega-1}}\right)^t}{1 - \frac{C}{2^{\omega-1}}} \\ + \sum_{i_1=1}^{k} c_{i_1} H(m/2, n - d_i m/2) \dots + \sum_{i_1=1}^{k} c_{i_1} \dots \sum_{i_t=1}^{k} c_{i_t} H(1; n - f(d_1, \dots, d_t, m)) \tag{1}$$

Thus, we get $\alpha m^\omega + \beta m^{\omega-1} n \leq T(m;n) \leq \alpha m^\omega + \beta m^{\omega-1} n + C^t T(1;n) + \sum_{i=1}^{t} C^i H(\frac{m}{2^i}; n)$. The last term is bounded by $gm^2 \frac{1-\left(\frac{C}{4}\right)^t}{1-\frac{C}{4}} + fmn \frac{1-\left(\frac{C}{2}\right)^t}{1-\frac{C}{2}}$ when $C \neq 4$ and $C \neq 2$. In this case $C^t T(1;n) + \sum_{i=1}^{t} C^i H(\frac{m}{2^i}; n) \leq m^{\log_2(C)} \left((e + \frac{2g}{C-2})n + \frac{4g}{C-4} \right) = \mathcal{O}(m^\omega + m^{\omega-1} n)$. When $C = 2$, a supplementary $\log_2(m)$ factor arises in the small factors, but the order of magnitude is preserved since $\log_2(C) + 1 = 2 < \omega$. □

These two lemmas are useful in the following sections where will apply the lemmas and solve (e.g. $T(n) = \alpha m^\omega$ solved via the recurring relation for α) to get the actual constant of the dominant term. Thus, when we give an equality on complexities, this equality means that the dominant terms of both complexities are equal. In particular, some lower order terms may differ.

Now, we provide a full C++ package available directly[1] or through the exact linear algebra library LinBox[2] [20]. Extending the work undertaken by the authors et al.[50, 21, 5, 27, 19, 22, 23], this paper focuses on finite field arithmetic with more implementations ; on vector dot products with more experiments and algorithms, in particular a fast centered representation ; on matrix multiplication with an extended Winograd variant optimizing memory allocation ; on simultaneous triangular system solving; on matrix factorization and improved constant factors of complexity for many linear algebra equivalent routines (inverse, squaring, upper-lower or upper-upper triangular multiplication, etc.).

The paper is organized as follows. Section 2 deals with the choice of data structures to represent elements of a finite field ; with different ways to implement the basic arithmetic operations and the dot product. Then section 3 presents efficient ways to *generically* implement matrix multiplication over prime fields (finite fields with prime cardinality), including a study of fast matrix multiplication, and a matrix multiplication based simultaneous resolution of n triangular systems. Therefore, section 4 presents the matrix factorizations and their applications with a study of complexity and practical performances.

[1] `www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/FFLAS`
[2] `www.linalg.org`

2 Finite field arithmetic

The first task, to implement linear algebra routines is to develop the underlying arithmetic implementation. As this is crucial for high performances, we present in this section several variant of implementations and analyze their behavior on nowadays hardware architectures. These implementations present distinct constraints on the representable fields. We thus provide their respective range of use for 32 and 64 bits machines.

2.1 Prime field representations

We present here various methods implementing seven of the basic arithmetic operations:

- addition, subtraction, negation, multiplication, division,
- a multiplication followed by an addition ($r \leftarrow a * x + y$) or $AXPY$ (also called "fused-mac" within hardware).
- a multiplication followed by an in-place addition ($r \leftarrow a * x + r$) or $AXPYIN$.

Within linear algebra in general (e.g. Gaussian elimination, or matrix-vector iterations) and for dot product in particular, these last two operations are the most widely used. We now present different ways to implement these operations.

2.1.1 Classical representation with integer division

The classical representation, with positive values between 0 and $p-1$, for p the prime, will be denoted by "*Zpz*".

- Addition is a signed addition followed by a test. An additional subtraction of the modulus is made when necessary. Subtraction is similar.
- Multiplication is machine multiplication and machine remainder.
- Division is performed via the extended gcd algorithm.
- $AXPY$ is a machine multiplication and a machine addition followed by only one machine remainder.

For the results to be correct, the intermediate AXPY value must not overflow. For a $m-$bit machine integer, the prime must therefore be below $2^{\frac{m-1}{2}} - 1$ if signed values are used. For 32 and 64 bits this gives primes below 46337 and below 3037000493.

Note that with some care those operations can be extended to work with unsigned machine integers (e.g. an additional test is required for the subtraction). The possible primes then being below 65521 and 4294967291.

2.1.2 Montgomery representation

To avoid the costly machine remainders, Montgomery designed another reduction [46]:

> Given an integer B such that $gcd(p, B) = 1$. Let $n_{im} \equiv -p^{-1} \mod B$ and T such that $0 \leq T \leq pB$; if $U \equiv Tn_{im} \mod B$ then $(T + Up)/B$ is an integer and $(T + Up)/B \equiv TB^{-1} \mod p$.
>
> Moreover, $(T + Up)/B$ is an integer between 0 and $2p$ and, thus, the mod p reduction has almost been performed.

The idea of Montgomery is to set B to half the word size. Thus, multiplication and divisions by B will just be shifts and remaindering by B which is just the application of a bit-mask. Then, one can use the reduction to perform the remainderings by p. Indeed the example implementation of this reduction shown below with one shift, two bit-masks and two machine multiplications is often much less expensive than a machine remaindering:

Montgomery reduction

```
#define MASK 65535UL
#define B 65536UL
#define HALF_BITS 16
/* nim is precomputed to -1/p mod B with the extended gcd */
    ...
    unsigned long c0 (c & MASK);        /* c     mod B          */
    c0 = (c0 * nim) & MASK;             /* -c/p  mod B          */
    c += c0 * p;                        /* c = 0 mod B          */
    c >>= HALF_BITS;                    /* high bits of c       */
    return (c>p?c-p:c);                 /* c is between 0 and 2p */
```

The idea is then to change the representation of the elements: every element a is stored as $aB \mod p$. Then additions, subtractions are unchanged and the prime field multiplication is now a machine multiplication, followed by only one Montgomery reduction. Nevertheless, one has to be careful when implementing the AXPY operator since axB^2 cannot be added to yB directly. We will see section 2.4.3 that this is actually not a problem for the dot product. Finally, the primes must verify $(p-1)^2 + p*(B-1) < B^2$, which gives $p \leq 40499$ (resp. $p \leq 2654435761$) for $B = 2^{16}$ (resp. $B = 2^{32}$).

2.1.3 Floating point representation

Yet another way to perform the reduction is to use the floating point routines. According to [11, Table 3.1], for most of the architectures (alpha, AMD, Pentium IV, Itanium, Sun Solaris, etc.) those routines are faster than the integer ones (except for the Pentium III). The idea is then to compute $T \mod p$ by way of

a precomputation of a high precision numerical inverse of p:

$$T \mod p = T - \lfloor T * \frac{1}{p} \rfloor * p.$$

The idea here is that floating point division and truncation are quite fast when compared to machine remaindering. Now on floating point architectures the round-off can induce a ± 1 error when the flooring is computed. This requires then an adjustment as implemented e.g. in Shoup's NTL [54] :

NTL's floating point reduction

```
double P, invP, T;
...
T  -= floor(T*invP)*P;
if (T >= P)     T -= P;
else if (T < 0) T += P;\\
```

A variant of this scheme is to use a floating point computation of the remainder via `fmod` directly instead of flooring. There also round-off errors have to be taken into account.

2.1.4 Discrete logarithms

This representation is also known as Zech logarithms, see e.g. [16] and references therein. The idea is to use a generator of the multiplicative group, namely a primitive element [17]. Then, every non zero element is a power of this primitive element and this exponent can be used as an internal representation:

$$\begin{cases} 0 & \text{if } x = 0 \\ q-1 & \text{if } x = 1 \\ i & \text{if } x = g^i \text{and } 1 \leq i < q-1 \end{cases}$$

Then many tricks can be used to perform the operations that require some extra tables see e.g. [36, 3]. This representation can be used for prime fields as well as for their extensions, we will therefore use the notation GFq. The operations are then:

- Multiplication and division of invertible elements are just an index addition and subtraction modulo $\overline{q} = q - 1$.
- Negation is identity in characteristic 2 and addition of $i_{-1} = \frac{q-1}{2}$ modulo $\overline{q}$ in odd characteristic.
- Addition is now quite complex. If g^i and g^j are invertibles to be added then their sum is $g^i + g^j = g^i(1 + g^{j-i})$. The latter can be implemented using index addition and subtraction and access to a "plus one" table ($t\_plus1[]$) of size q. This table gives the exponent h of any number of the form $1 + g^k$, so that $g^h = 1 + g^k$.

Operation	Elements	Indices	Cost		
			+/-	Tests	Accesses
Multiplication	$g^i * g^j$	$i + j\ (-\overline{q})$	1.5	1	0
Division	g^i / g^j	$i - j\ (+\overline{q})$	1.5	1	0
Negation	$-g^i$	$i - i_{-1}\ (+\overline{q})$	1.5	1	0
Addition	$g^i + g^j$	$k = j - i\ (+\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3	2	1
Subtraction	$g^i - g^j$	$k = j - i + i_{-1}\ (\pm\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3.75	2.875	1

Table 1: Number of elementary operations to implement Zech logarithms for an odd characteristic

Table 1 shows the number of elementary operations to implement Zech logarithms for an odd characteristic finite field. Only one table of size q is considered. Those operations are of three types: mean number of exponent additions and subtractions (+/-), number of tests and number of table accesses.

We have counted 1.5 index operations when a correction by $\overline{q}$ actually arises only for half the possible values. The fact that the mean number of index operations is 3.75 for the subtraction is easily proved in view of the possible values taken by $j - i + \frac{q-1}{2}$ for varying i and j. In this case, $j - i + i_{-1}$ is between $-\frac{\overline{q}}{2}$ and $\frac{3\overline{q}}{2}$ and requires a correction by $\overline{q}$ only two eighth of the time.

The total number of additions or subtractions is then $2+0.25+1+0.5 = 3.75$ and the number of tests $1 + 0.875 + 1 = 2.875$ follows (one test towards zero, one only in case of a positive value i.e. seven eighth of the time, and a last one after the table lookup). It is possible to actually reduce the number of index exponents, (for instance replacing $i + x$ by $i + x - \frac{q-1}{2}$) but to the price of an extra test. In general, such a test ($a > q$?) is as costly as the $a - q$ operation. We therefore propose an implementation minimizing the total cost with a single table.

These operations are valid as long as the index operations do not overflow, since those are just signed additions or subtractions. This gives maximal prime value of e.g. 1073741789 for 32 bits integer. However, the table size is now a limiting factor: indeed with a field of size 2^{26} the table is already 256 Mb. Table reduction is then mandatory. We will not deal with such optimizations in this paper, see e.g. [37, 16] for more details.

2.1.5 Fully tabulated multiplication

One can also further tabulate the Zech logarithm representation. The idea is to code 0 by $2\overline{q}$ instead of 0. Then a table can be made for the multiplication:

- $t\_mul[k] = k$ for $0 \leq k < \overline{q}$.
- $t\_mul[k] = k - \overline{q}$ for $q - 1 \leq k < 2\overline{q}$.
- $t\_mul[k] = 2\overline{q}$ for $2\overline{q} \leq k \leq 4\overline{q}$.

The same can be done for the division via a shift of the table and creation of the negative values, thus giving a table of size $5q$. For the addition, the t_plus1 has also to be extended to other values, to a size of $4q$. For subtraction, an extra table of size $4q$ has also to be created. When adding the back and forth conversion tables, this gives a total of $15q$. Even with some table reduction techniques, this becomes quite huge and quite useless nowadays when memory accesses are a lot more expensive than arithmetic operations [18].

2.1.6 Quadratic tabulation

One can even further tabulate, by using tables of size p^2 so that every arithmetic operation is precomputed and tabulated[43]. For extremely small primes one can have certain speed-ups for single operations. This happens only on slow machines where memory speed was comparable to the one of arithmetic operations. For bigger primes, one need to have smaller tables. The idea is to have tables of size $\left(\frac{p}{k}\right)^2$. Then one computes the k parts of the operand separately and adds the k results together. This is only useful when $2k - 1$ modular additions and k memory accesses are faster than 1 multiplication. In practice Kawame and Murao [43] shows that this is nowadays seldom the case even for $k = 1$.

2.2 Field extensions

Some of the presented implementations for prime fields can be used for other fields or rings. The classical representation can e.g. be used with non prime modulus. Extension fields, or Galois fields of size p^d can also be implemented the following way:

1. **Integer division.** Extension fields would be implemented with polynomial arithmetic.
2. **Montgomery Reduction.** Direct use of Montgomery reduction is not possible, but there exists some efficient polynomial reductions. See e.g. [3] and references therein.
3. **Zech logarithms.** A very interesting property is that whenever this implementation is not at all valid for non prime modulus, it remains identical for field extensions. Indeed one can also find generators modulo an irreducible polynomial or even build extensions with primitive polynomials (X is thus a generator) [33, 18]. In this case the classical representation would introduce polynomial arithmetic. This discrete logarithm representation, on the contrary, would remain atomic, thus inducing a speed-up factor of $O(d^2)$, for d the extension degree. See e.g. [21, §4] for more details.

4. **A $q-$adic representation.** Another idea, by B. D. Saunders [53], is to go back to the polynomial arithmetic but in a $q-$adic way, with q a sufficiently big prime or power of a single prime.

   Suppose that $a = \sum_{i=0}^{k-1} \alpha_i X^i$ and $b = \sum_{i=0}^{k-1} \beta_i X^i$ are two elements of $\mathtt{GF}(p^k)$ represented by $\mathbb{Z}/\mathrm{p}\mathbb{Z}[X]/Q$. One can perform the polynomial multiplication ab via $q-$adic numbers. Indeed, by setting $\tilde{a} = \sum_{i=0}^{k-1} \alpha_i q^i$ and $\tilde{b} = \sum_{i=0}^{k-1} \beta_i q^i$, the product is computed in the following manner (we suppose that $\alpha_i = \beta_i = 0$ for $i > k-1$):

$$\widetilde{ab} = \sum_{j=0}^{2k-2} \left(\sum_{i=0}^{j} \alpha_i \beta_{j-i} \right) q^j \tag{2}$$

   Now if q is big enough, the coefficient of q^i will not exceed q. In this case, it is possible to evaluate a and b as floating point numbers, compute the product of these evaluations, and convert back to finite field element, via a $q-$adic reconstruction, a division by p and a division by Q. This e.g. enables the use of floating point numerical routines for finite field extensions as we will see in section 2.4.7.

   Remark 2.1. *For the performances, a first naïve implementation would only give limited speed-up as the conversion cost is then very expensive. However, the prime power q can be chosen to be a power of 2. Then the Horner like evaluation of the polynomials at q is just a left shift. One can then compute this shift with exponent manipulations in floating point arithmetic and use then native C++ << operator as soon as values are within the 32 bits range, or use the native C++ << on 64 bits when available. This choice also speeds up the radix inverse reconversion.*

2.3 Atomic comparisons

We now present a comparison between the preceding implementation possibilities. The idea is to compare just the atomic operations. "%" denotes an implementation using machine remaindering (machine division) for every operation. This is just to give a comparing scale. "NTL" denotes NTL's floating point flooring for multiplication ; "Z/pZ" denotes our implementation of the classical representation when tests ensure that machine remaindering is used only when really needed. Last "GFq" denotes the discrete logarithm implementation of section 2.1.4. In order to be able to compare those single operations, the experiment is an application of the arithmetic operator on vectors of a given size (e.g. 256 for figures 1, 2 and 3).

We compare the number of millions of field arithmetic operations per second, *Mop/s*.

Figure 1 shows the results on a UltraSparc II 250 Mhz, with compiler "gcc version solaris2.9/3.3.2". First one can see that the need of Euclid's algorithm

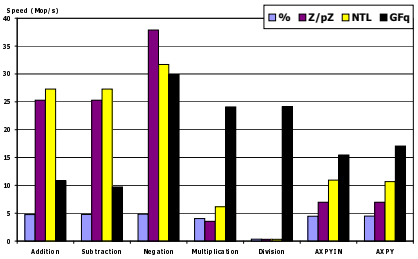

Figure 1: Single arithmetic operation modulo 32749 on a sparc ultra II, 250 MHz

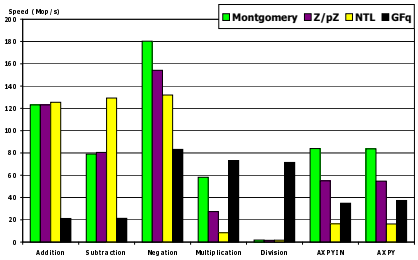

Figure 2: Single arithmetic operation modulo 32749 on a Pentium III, 1 GHz, cache 256 Kb

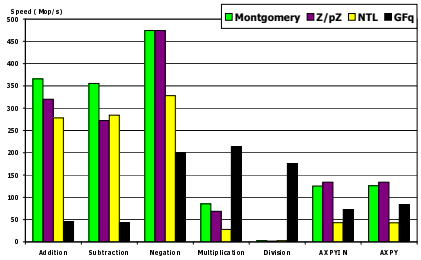

Figure 3: Single arithmetic operation modulo 32749 on a Pentium IV, 2.4 GHz, cache 512 Kb

for the field division is a huge drawback of all the implementations save one. Indeed division over "GFq" is just an index subtraction. Next, we see that floating point operations are quite faster than integer operations: NTL's multiplication is better than the integer one. Now, on this machine, memory accesses are not yet much slower than arithmetic operations. This is the reason why discrete logarithm addition is only 2 to 3 times slower than one arithmetic call. This, enables the "GFq" AXPY (base operation of most of the linear algebra operators) to be the fastest.

On newer PC, namely Intel Pentium III and IV of figures 2 and 3, the compiler used for the C/C++ programs was "gcc version 3.2.3 20030309 (Debian prerelease)". One can see that now memory accesses have become much too slow. Then any tabulated implementation is penalized, except for extremely small modulus. NTL's implementation is also penalized, both because of better integer operations and because of a pretty bad flooring (casting to integer) of the floating point representation. Now, for Montgomery reduction, this trick is very efficient for the multiplication. However; it could seems that it becomes less useful as the machine division improves as shown by the AXPY results of figure 3. Actually, we looking closely to figures 2 and 3 one sees that the "Zpz" AXPY on the Pentium III can reach 5.5% of the peak performances, and this is the same (5.4%) for the Pentium IV. Thus division has not improved, it is Montgomery's representation which is slowed down ! As shown section 2.1.2 the Montgomery AXPY is less impressive because one has to compute first the multiplication, one reduction and then only the addition and tests. This is due to our choice of representation aB. "Zpz", not suffering from this distinction between multiplied and added values can perform the multiplication and addition before the reduction so that one test and sometimes a correction by p are saved. Nevertheless, we will see in next section that our choice of representation is not anymore a disadvantage for the dot product.

2.4 Dot products

In this section, we extend the results of [19, §3]. Two main techniques are used: regrouping operations before performing the remaindering, and performing this remaindering only on demand. Several new variants of the representations of section 2.1 are tested and compared. For "GFq" and "Montgomery" representations the dot products are of course performed with their representations. In particular the timings presented do not include conversions. The argument is that the dot product is computed to be used within another higher level computation. In this paradigm, conversions will only be useful for reading the values in the beginning and for writing the results at the end of the whole program.

2.4.1 53 and 64 bits

The first idea is to use a representation where no overflow can occur during the course of the dot product. The division is then delayed to the end of the product:

if the available mantissa is of m bits and the modulo is p, the division happens at worst every λ multiplications where λ verifies the following condition:

$$\lambda(p-1)^2 < 2^m \leq (\lambda+1)(p-1)^2 \tag{3}$$

For instance when using 64 bits integers with numbers modulo 40009, a dot product of vectors of size lower than $1.15\ 10^{10}$ will never overflow. Hence one has just to perform the AXPY without any division. A single machine remaindering is needed at the end for the whole computation. This produces very good speed ups for 53 (double representation) and 64 bits storage as shown on curves (5) and (6) in figure 4.

There, the floating point representation performs the division "à la NTL" using a floating point precomputation of the inverse and is slightly better than the 64−bit integer representation. Note also the very good behavior of an implementation of P. Zimmermann [59] of Montgomery reduction over 32−bit integers.

2.4.2 AXPY blocks

The extension of this idea is to specialize dot product in order to make several multiplications and additions before performing the division (which is then delayed), even with a small storage. Indeed, one needs to perform a division only when the intermediate result is able to overflow.

Blocked dot product

```
res = 0;
unsigned long i=0; if (K<DIM) while ( i < (DIM/K)*K ) {
    for(unsigned long j = 0; j < K; ++j, ++i) res += a[i]*b[i];
    res %= P;
}
for(; i< DIM; ++i) res += a[i]*b[i];
res %= P;
```

This method will be referred as "block-XXX". Figure 5 shows that it is optimal for small primes since it performs nearly one arithmetic operation per processor cycle. Then, the step shape of the curve reflects the thresholds when an additional division is made.

2.4.3 Choice of Montgomery representation

We see here, that our choice of representation (aB) for Montgomery is interesting. Indeed, the basic dot product operation is a cumulative AXPY. A classical AXPY would then be $axB^2 + yB$, henceforth needing an additional reduction before the addition of yB. Now, within a dot product, each one of the added values is in fact the result of a multiplication. Therefore, the additions are between elements of the form $sB^2 += x_i y_i B^2$. This proves that the reduction can indeed be delayed.

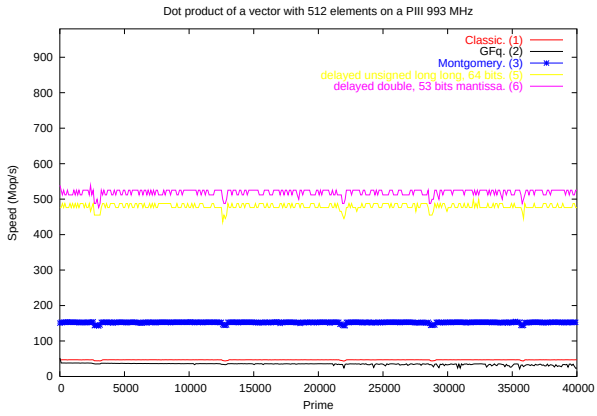

Figure 4: Dot product by delayed division, on a PIII

2.4.4 Centered representation

Another idea is to use a centered representation for "Z/pZ": indeed if elements are stored between $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, one can double the sizes of the blocks; equation 3 now becomes

$$\lambda_{centered}\left(\frac{p-1}{2}\right)^2 < 2^{m-1} \leq (1+\lambda_{centered})\left(\frac{p-1}{2}\right)^2 \tag{4}$$

Still and all, for small primes, any one of the blocked representation is better than a floating point representation. The slight differences between the three being the different thresholds for a single additional reduction.

2.4.5 Division on demand

The second idea is to let the overflow occur ! Then one should detect this overflow and correct the result if needed. Indeed, suppose that we have added a product ab to the accumulated result t and that an overflow has occurred. The variable t now contains actually $t - 2^m$.

Well, the idea is just to precompute a correction $CORR = 2^m \bmod p$ and add this correction whenever an overflow has occurred.

Now for the overflow detection, we use the following trick: since $0 < ab < 2^m$, an overflow has occurred if and only if $t + ab < t$. The "Z/pZ" code now should look like the following:

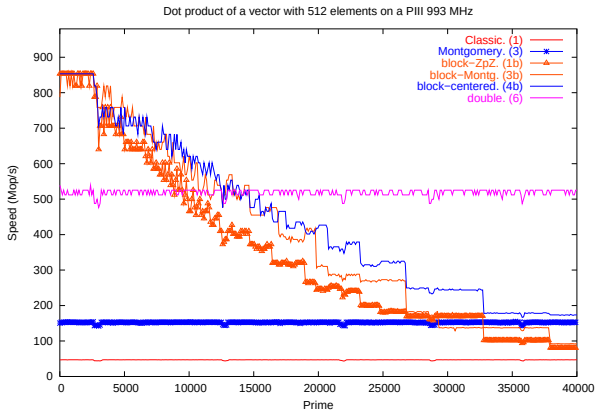

Figure 5: Dot product by blocked and delayed division, on a PIII

Unsigned Overflow detection trick

```
sum = 0;
for(unsigned long i = 0; i<DIM; ++i) {
      product = a[i]*b[i];
      sum += product; if (sum < product) sum += CORR;
}
```

Of course one can also apply this trick to Montgomery reduction. Indeed, as shown on curve (3c) in figure 6, the trick of using a representation storing $aB \mod p$ for any element a enables to perform only one reduction at the end of each block. We see also, that as soon as one reduction is needed, the drop of performances is tremendous. The pipeline is completely broken and "overflow-ZpZ" as well as "overflow-Montgomery" are rapidly outperformed by the floating point representation.

The centered representation can be used also in this case. This gives the better performances for small primes and enables a slower drop of performances. This is due to its higher threshold. However, for this representation, the unsigned trick does not apply directly anymore. The overflow and underflow need to be detected each by two tests:

Signed overflow detection trick

```
      if      ((sum < oldsum) && (oldsum - sum < 0)) sum += CORR;
      else if ((oldsum < sum) && (sum - oldsum < 0)) sum -= CORR;
```

Thus, the total of four tests is costlier and for bigger primes this overhead is too expensive as shown on curve (4c) of figure 6.

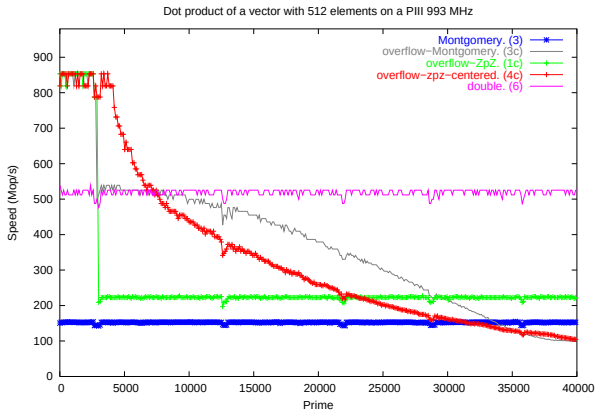

Figure 6: Overflow detection, on a PIII

2.4.6 Hybrid

Of course, one can mix both 2.4.2 and 2.4.5 approaches and delay even the overflow test when p is small. One has just to slightly change the bound on λ so that, when adding the correction, no new overflow occur:

$$\lambda(p-1)^2 + (p-1) = \lambda p(p-1) < 2^m. \tag{5}$$

This method will be referred as "block-overflow-XXX".

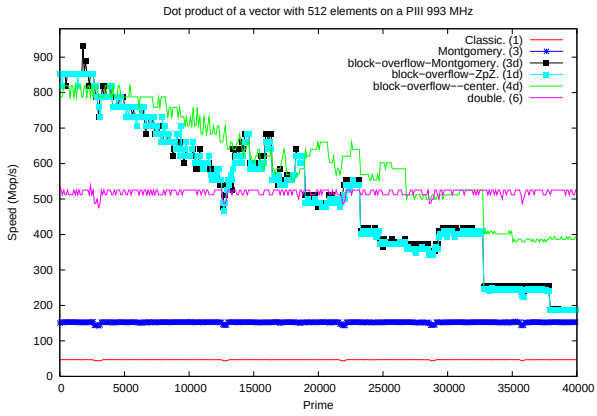

Figure 7: Hybrid (block AND overflow detection), on a PIII

We compared those ideas on a vector of size 512 with 32 bits (`unsigned long` on PIII). First, we see that as long as $512p(p-1) < 2^m$, we obtain *quasi optimal performances*, since only one division is performed at the end. Then,

when the prime exceeds the bound (i.e. for $p(p-1) > 2^{32-9}$, which is $p > 2897$) an extra division has to be made. On the one hand, this is dramatically shown by the drops of figure 6.

On the other hand, however, those drops are levelled by our hybrid approach as shown in figure 7. There the step form of the curve shows that every time a supplementary division is added, performances drop accordingly. Now, as the prime size augments, the block management overhead becomes too important.

One can remark than no significant difference exist between the performances of "block-overflow-Zpz" and "block-overflow-Montgomery". Indeed, the code is now exactly the same except for a single reduction for the whole dot product. This makes the Montgomery version better but only extremely slightly.

Lastly, here also, an hybrid *centered* version is useful. When compared to Montgomery or Zpz, one can remark that the signed overflow detection is two times as costly as the unsigned overflow detection. However, the block size is twice as big (one looses a bit for the sign but gains 2 bits since the multiplied numbers are then both of absolute value less than $\frac{p-1}{2}$).

2.4.7 Extension fields dotproduct

In an extension of degree k, one needs to compute the sum of products of polynomials of degree $k-1$. The first idea is to use the delayed paradigm:

- Delay the modular reduction of every coefficient of the resulting polynomial, using the previous sections.
- Delay the polynomial reduction. There the degrees remains $2k-1$ all along the dotproduct and no overflow as to be feared.

That way only one polynomial reduction is made thus reducing the cost of the dotproduct by a factor k.

One can also use the q-adic representation of section 4 to perform efficiently the dotproduct over an extension field.

Theorem 2.2. *Let m be the number of available mantissa bits within the machine floating point numbers. If*

$$q > nk(p-1)^2 \text{ and } (2k-1)\log[2](q) < m,$$

then Algorithm 1 is correct.

Proof. In equation 2 we can see that any coefficient of q^l in the product $\widetilde{ab}$ is a sum of at most k elements over $\mathbb{Z}/\mathrm{p}\mathbb{Z}$, therefore its value is at most $k(p-1)^2$. First, we can state in the same manner that any coefficient $\widetilde{\gamma_l}$ is at most $nk(p-1)^2$ as it is a sum of n products. Therefore if q is bigger than $nk(p-1)^2$, there is no carry in the floating point dot product and the $q-$adic to polynomial conversion is correct. Then, as the products double the $q-$adic degree, $\tilde{r}$ is strictly lower than q^{2k-1}. The result follows from the fact that an overflow occurs only if $\tilde{r}$ is bigger than 2^m. □

Algorithm 1 Dot product over Galois fields via $q-$adic conversions to floating point numbers

Require: a field $\mathtt{GF}(p^k)$ represented as polynomials mod p and mod Q, for Q a degree k irreducible polynomial over $\mathbb{Z}/\mathrm{p}\mathbb{Z}$.
Require: Two vectors v_1 and v_2 of n elements of $\mathtt{GF}(p^k)$ each, as polynomials.
Require: a prime power q.
Ensure: $R \in \mathtt{GF}(p^k)$, with $R = v_1^T.v_2$. {Polynomial to $q-$adic conversion}
1: Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point vectors of the evaluations at q of the elements of v_1 and v_2. {Using Horner's formula, for instance} {One computation}
2: Compute $\tilde{r} = \widetilde{v_1}^t \widetilde{v_2}$ {Building the solution}
3: $\tilde{r} = \sum_{l=0}^{2k-2} \widetilde{\gamma_l} q^l$. {Using radix conversion, see [26, Algorithm 9.14] for instance}
4: For each l, set $\gamma_l = \widetilde{\gamma_l} \mod p$
5: set $R = \sum_{l=0}^{2k-2} \gamma_l X^l \mod Q$

Typically, $m = 53$ for 64-bits double precision; in that case, the biggest implementable fields are $\mathtt{GF}(2^8)$, $\mathtt{GF}(3^6)$, $\mathtt{GF}(7^4)$, $\mathtt{GF}(23^3)$ and $\mathtt{GF}(317^2)$, with $n = 1$. Table 19 in appendix B gives the biggest possible block size for different fields, and the associated prime for the $q-$adic decomposition. Of course a highest block order of 1 is of absolutely no interest in practice. However, on the Pentium, for instance, as soon as the size approaches 200, the conversion cost will be compensated by the cache effects. We can see this for curve (4) on figure 10.

This method is already interesting for quite a few cases. Nonetheless, on a 64-bits architecture (DEC alpha for instance) where machine integers have 8 bytes, this can be applied to even more cases. Table 20, in appendix B, shows that about a factor of 10 can be gained for the maximum matrix size, when compared to 53-bits mantissas. Also the biggest implementable fields are now $\mathtt{GF}(2^9)$, $\mathtt{GF}(3^7)$, $\mathtt{GF}(5^5)$, $\mathtt{GF}(11^4)$, $\mathtt{GF}(47^3)$ and $\mathtt{GF}(1129^2)$. As shown in [21, figure 6 and 9], on theses machines, as a good use of the cache is essential, our wrapping is unavoidable to obtain good performances.

Remark 2.3. *Some* sparse *primitive polynomials modulo p can be chosen to build $\mathtt{GF}(p^k)$. Then the division to get the remainders can be simplified. The idea is to consider primitive trinomials, or when generating is not the bottleneck, irreducible binomials. Indeed at least one of those exists for nearly every prime field [3, Theorem 1]. In this case, we suppose that $Q = X^k + \beta X^t + \alpha$ is primitive over $\mathbb{Z}/p\mathbb{Z}$ with $t < \frac{k}{2}$ (this is not a restriction since whenever Q is primitive, its reciprocal is primitive also and no primitive polynomial is self reciprocal [45, Theorem 3.13]). Now, as we are working in $\mathbb{Z}/p\mathbb{Z}[X]/Q$, we want to compute the division of $W = \sum_{l=0}^{2k-2} \gamma_l X^l$ by Q. Therefore, we first split W into its higher and lower parts: $W = HX^k + L = H(-\beta X^t - \alpha) + L \mod Q$. We then pursue by splitting H also into its higher and lower parts: $H = H_h X^{k-t} + H_l$, leading to $W = -\beta H_h X^k - \beta H_l X^t - \alpha H + L$ which is also*

$W = \beta^2 H_h X^t + \alpha\beta H_h - \beta H_l X^t - \alpha H + L \mod Q$. *We conclude by using the fact that* W *is of degree at most* $2k-2$*, so that* H_h *is of degree less than* $t-1$*. This proves that the expression above is of degree strictly less than* k *and that it is exactly the remainder of the division of* W *by* Q*. A careful counting of the operations, taking advantage of the respective degrees of the terms, would also show that this computation requires less than* $5k$ *field operations. That way, the division can be replaced by the equivalent of two and a half simple polynomial subtractions ! This speed up is even better when* β *is zero, because then the division is only one subtraction:* $-\alpha H + L$*.*

As shown on figure 10 for the matrix multiplication, with those optimization we can reach very high peak performances, quite close to those obtained with prime fields, namely 420 Mop/s on the PIII, 735 MHz.

2.4.8 Vector size influence

In this last section, we discuss the vector size influence. Until now we have used vectors of size 512. We see on figure 8 that the best vector size is indeed around

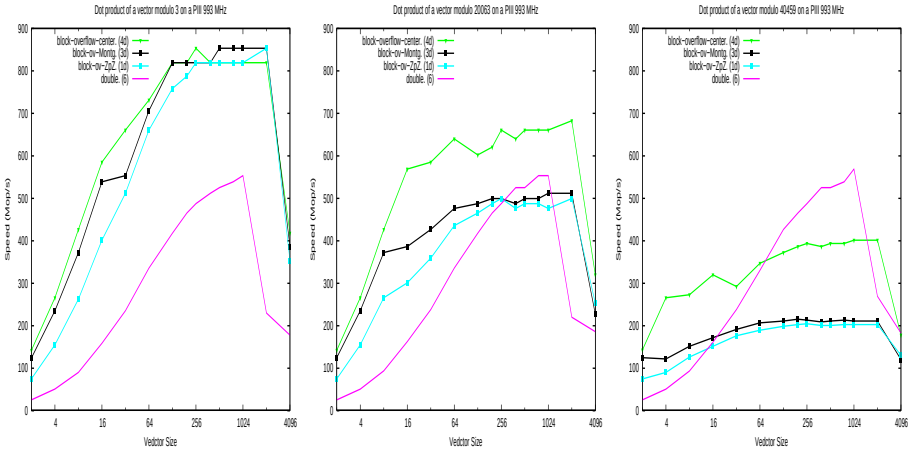

Figure 8: Hybrid (function of the vector size) modulo 3, 20063 and 40459 on a PIII 993 MHz

this size. Those three figures shows at least two things:

- On the one hand, the floating point representation is the most stable one for half word size primes. Then, vector size do not really infer with its performances. On the other hand, those performances are interesting only for big vector sizes.
- Except when the prime is too big the hybrid centered representation is nearly always the best one.

We have seen different ways to implement a dot product over word size finite fields. The conclusion is that most of the times, a floating point representation is the best implementation.

However, with some care, it is possible to improve this speed for small primes by a hybrid method using overflow detection and delayed division. Now, the floating point representation approximately doubles its performances on the P4 2.4 GHz, when compared to the 1 GHz Pentium III. But surprisingly, the hybrid versions only slightly improve. Still and all, an optimal version should switch from block methods to a floating point representation according to the vector and prime size and to the architecture.

Nevertheless, bases for the construction of an optimal dot product over word size finite fields have been presented: the idea is to use an Automated Empirical Optimization of Software [57] in order to produce a library which would determine and choose the best switching thresholds at install time.

3 Finite Field Linear Algebra Subroutines

A naïve approach to matrix-vector and matrix-matrix products would be to consider them as n or n^2 dotproducts. Thus the performances would just be the performances of previous section. However, enormous effort has been made to improve on these linear operations in order to take benefit of the memory hierarchy of nowadays machines [57].

In most of the modern computer architecture, a memory access to the R.A.M. is more than one hundred times slower than an arithmetic operation. To circumvent this slow down, the memory is structured into two or three levels of cache acting as buffers to reduce the number of accesses to the R.A.M. and reuse as much as possible the buffered data. This technique is only possible if the algorithm enables to reuse some data. This is the main reason why matrix product is the most suited operation for this kind of optimization: it is the first basic operation in linear algebra, where the time complexity ($\mathcal{O}(n^3)$) is an order of magnitude higher than the space complexity ($\mathcal{O}(n^2)$).

These considerations have driven the numericians to develop the basic linear algebra subroutines (BLAS) [14], mainly based on a tuned kernel for matrix multiplication. One of its ground idea (among many others) is to perform the matrix multiplication by blocks of small dimension, fitting into the L2 cache, and limit therefore the overhead due to memory accesses.

We show in this section how we use this effort together with delayed modulus for exact computations so as to enable higher efficiency.

3.1 Matrix multiplication

The salient features of our approach to exact matrix multiplication over a finite field are

1. The possibility to convert a finite field matrix into a matrix with floating point elements, applying the numerical routines, and converting back to the finite field. This is because nowadays processors focus on floating point arithmetic.

2. Choosing block sizes for cache optimization: this can be done over the finite fields or be left to the numerical BLAS.

3. The use of the Winograd improved variant of Strassen fast multiplication does not suffer from bad stability in the exact case.

Each one of those ideas induces some choices for block cutting of the matrices:

- First, the wrapping of the BLAS corresponds to a computation over $\mathbb{Z}$. The result is then reduced with a modulo at the conversion back to the finite field. This is similar to the delayed modulus technique, developed in section 2.4.2 for the dotproduct. Therefore, the same constraint arises for the correctness of the result: the computation over $\mathbb{Z}$ must not exceed the capacity of the floating point representation: 53 bits of mantissa. This condition is $k \max\{a_{i,j}, b_{i,j}\}^2 < 2^{53}$, where k is the common dimension between A and B. If it is not satisfied, the input matrices must be split into blocks of size $m \times k_{\max}$ for A and $k_{\max} \times n$ for B, where $k_{\max} = \frac{2^{53}}{\max\{a_{i,j}, b_{i,j}\}^2}$. The wrapping of the BLAS can then be applied to multiply these blocks, and the final result will be recovered by some additions over the finite field. This implies a limitation on both p and k. Here two different situations appear:

  1. p is given. Then an upper bound $k_{\max}$ on k can be derived from (3) or (4). The algorithm splits the operands into blocks of size $m \times k_{\max}$ and $k_{\max} \times n$, each of then are multiplied using the wrapping of BLAS. The result is recovered by adding each of these subresults.
  2. In a homomorphic computation of an integer result, several computation modulo different primes are performed. In this case, these primes can be chosen randomly and as large as possible, but still respecting the relation for k.

- To optimize the usage of the cache, one can consider to split the input matrices into small blocks so that a product of these blocks would fit into the cache. This idea is developed in [31]. Instead of performing this splitting by hand, we can let it to the BLAS. Indeed automatically tuned BLAS, such as ATLAS offer a smart decision process to determine the optimal splitting.

Hence, the wrapping of the BLAS makes it possible to benefit from the two following advantages: a delayed modulus using efficient machine floating point machine arithmetic and an optimized cache blocking.

This is emphasized on figure 9, where the naïve approach is extremely slow. On this machine the optimal block size is 33. The curve *block-33* shows the improvement that one can get doing blocking by hand. Now the numerical BLAS also takes benefit of the Pentium 4 fused-mac instruction and is able to compute more than one arithmetic operation per cycle. One can also see that the overhead we have over prime fields is negligible. For extension fields, this overhead is more important but still 4 times faster than the naïve approach as shown by curve (4) on figure 10.

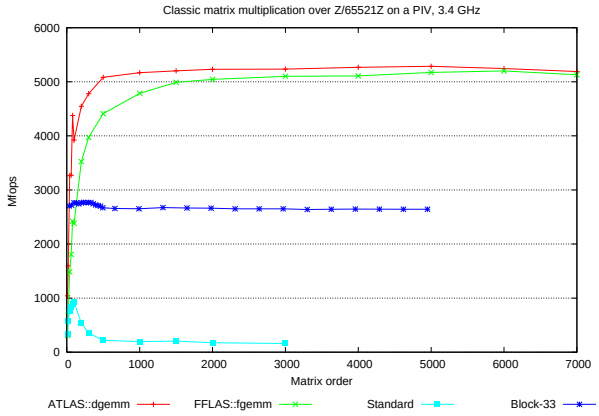

Figure 9: Blocking classical matrix multiplication, on a Pentium 4, 3.4 GHz.

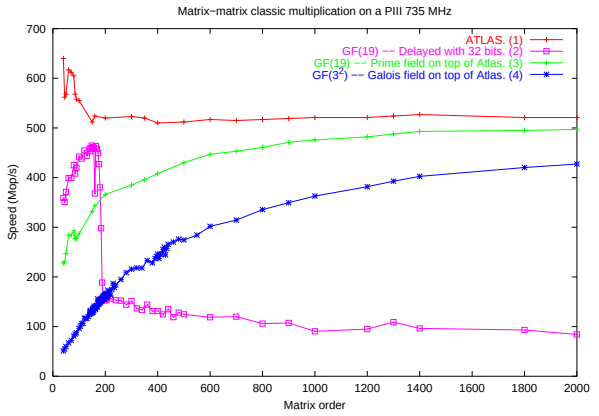

Figure 10: Speed comparison between matrix-matrix multiplications on a Pentium III 735 MHz.

3.2 Winograd fast algorithm

To improve further the efficiency of this routine, one can use a fast matrix multiplication algorithm. We will focus on Winograd's variant [26, algorithm 12.1] of Strassen's algorithm [56]. We denote by $MM(n)$ the dominant term of the arithmetic complexity of performing the matrix multiplication. The value of $MM(n)$ thus reflects the choice of algorithm, e.g. $MM(n) = 2n^3$ for the classical algorithm, and mean that the actual complexity of the classical algorithm is $2n^3 + \mathcal{O}(n^2)$. We also denote by ω the asymptotic exponent of $MM(n)$, it is thus 3 for the classical algorithm, $\log_2(7) \approx 2.807354922$ for the Strassen-Winograd variant, and the actual best known exponent is 2.375477 by [8].

In [34] Winograd's variant is discarded for numerical computations because of its bad stability and despite its better running time. In [42] aggregation-cancellation techniques of [44] are also compared. They also give better stability than the Winograd variant but worse running time. In the exact case stability has no meaning and Winograd's faster variant is thus preferred.

3.2.1 A Cascade structure

Asymptotically, this algorithm improves the number of arithmetic operations required for matrix multiplication from $MM(n) = 2n^3$ to $MM(n) = 6n^{2.8074}$. But in practice, one can improve the total number of arithmetic operations by switching after a few recursive levels of Winograd's algorithm to the classic algorithm. Table 2 compares the number of arithmetic operations depending on the matrix order and the number of recursive levels:

		Recursive levels of Winograd's algorithm					
n	Classic	1	2	3	4	5	6
4	**112**	144	214				
8	**960**	1024	1248	1738			
16	7936	**7680**	8128	9696	13126		
32	64512	59392	**57600**	60736	71712	95722	
64	520192	466944	431104	**418560**	440512	517344	685414

Table 2: Number of arithmetic operations in the multiplication of two $n \times n$ matrices

This phenomenon is amplified by the fact that additions in classic matrix multiplication are cheaper than the ones in Winograd algorithm since they are incorporated in the BLAS routine. As a consequence, the optimal number of recursive levels depends on the architecture and must be determined experimentally. It can be described by a simple parameter: the matrix order w for witch one recursive level is as fast the classic algorithm. Then the number of levels l is given by the formula

$$l = \lfloor \log_2 \frac{n}{w} \rfloor + 1.$$

We will now focus on the memory complexity.

3.2.2 Memory allocations

Consider the computation of the product $C \leftarrow A \times B$. One recursive level of Winograd's algorithm is composed by the following 22 operations:

Considering $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ and $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$.

- 8 additions:

$$\begin{array}{ll} S_1 \leftarrow A_{21} + A_{22} & T_1 \leftarrow B_{12} - B_{11} \\ S_2 \leftarrow S_1 - A_{11} & T_2 \leftarrow B_{22} - T_1 \\ S_3 \leftarrow A_{11} - A_{21} & T_3 \leftarrow B_{22} - B_{12} \\ S_4 \leftarrow A_{12} - S_2 & T_4 \leftarrow T_2 - B_{21} \end{array}$$

- 7 multiplications, by recursive calls:

$$\begin{array}{ll} P_1 \leftarrow A_{11} \times B_{11} & P_5 \leftarrow S_1 \times T_1 \\ P_2 \leftarrow A_{12} \times B_{21} & P_6 \leftarrow S_2 \times T_2 \\ P_3 \leftarrow S_4 \times B_{22} & P_7 \leftarrow S_3 \times T_3 \\ P_4 \leftarrow A_{22} \times T_4 & \end{array}$$

- 7 final additions:

$$\begin{array}{ll} U_1 \leftarrow P_1 + P_2 & U_5 \leftarrow U_4 + P_3 \\ U_2 \leftarrow P_1 + P_6 & U_6 \leftarrow U_3 - P_4 \\ U_3 \leftarrow U_2 + P_7 & U_7 \leftarrow U_3 + P_5 \\ U_4 \leftarrow U_2 + P_5 & \end{array}$$

- The result matrix is: $C = \begin{bmatrix} U1 & U5 \\ U6 & U7 \end{bmatrix}$

Using the dependencies between the tasks shown in figure 11, one can build the schedule given in table 3. It only requires the allocation of two temporary matrices X_1 and X_2 of dimensions $m/2 \times \max(n,k)/2$ and $k/2 \times n/2$.

#	operation	loc.	#	operation	loc.	#	operation	loc.
1	$S_1 = A_{21} + A_{22}$	X_1	9	$P_7 = S_3T_3$	C_{21}	17	$U_5 = U_4 + P_3$	C_{12}
2	$T_1 = B_{12} - B_{11}$	X_2	10	$S_4 = A_{12} - S_2$	X_1	18	$T_4 = T_2 - B_{21}$	X_2
3	$P_5 = S_1T_1$	C_{22}	11	$P_3 = S_4B_{22}$	C_{11}	19	$P_4 = A_{12}T_4$	C_{11}
4	$S_2 = S_1 - A_{11}$	X_1	12	$P_1 = A_{11}B_{11}$	X_1	20	$U_6 = U_3 - P_4$	C_{21}
5	$T_2 = B_{22} - T_1$	X_2	13	$U_2 = P_1 + P_6$	C_{12}	21	$P_2 = A_{12}B_{21}$	C_{11}
6	$P_6 = S_2T_2$	C_{12}	14	$U_3 = U_2 + U_7$	C_{21}	22	$U_1 = P_1 + P_2$	C_{11}
7	$S_3 = A_{11} - A_{21}$	X_1	15	$U_7 = U_3 + P_5$	C_{22}			
8	$T_3 = B_{22} - B_{12}$	X_2	16	$U_4 = U_2 + P_5$	C_{12}			

Table 3: Schedule of Winograd's algorithm for operation $C \leftarrow AB$

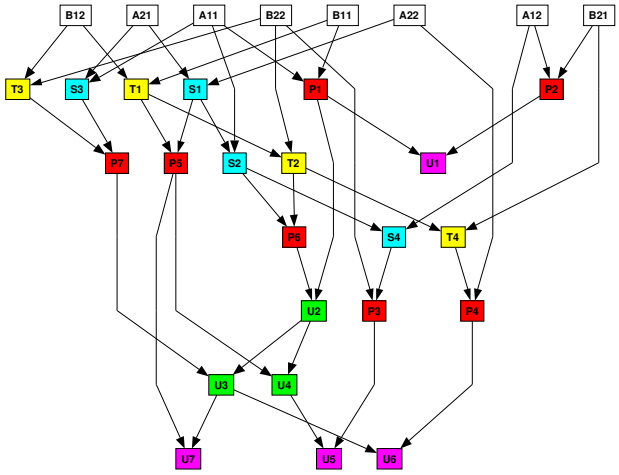

Figure 11: Tasks dependencies in one recursive level of Winograd's algorithm

Summing the temporary allocations of every recursive level, one can bound the extra memory requirements of this implementation by

$$2\sum_{i=1}^{\log n}\left(\frac{n}{2^i}\right)^2 < \frac{2}{3}n^2.$$

Note that this schedule was already presented in [15]. We recall it here before presenting the schedule for the more general operation $C \leftarrow A \times B + \beta C$:

A first method would be to store the matrix C into a temporary matrix and apply the previous schedule. This would involve a total amount of memory allocation of $(2 + 2/3)n^2$.

Alternatively, we propose the schedule of table 4, that requires the allocation of three temporary matrices X_1, X_2, X_3, of dimension $m/2 \times n/2$, $m/2 \times k/2$ and $k/2 \times n/2$.

This corresponds to a total memory allocation bounded by

$$3\sum_{i=1}^{\log n}\left(\frac{n}{2^i}\right)^2 < n^2$$

Moreover, this schedule incorporates as much as possible the extra additions into the recursive calls and at last into the BLAS call where the operations are cheaper. Only two additional additions are required, instead of four in the first method.

#	operation	loc.	#	operation	loc.	#	operation	loc.
1	$P_2 = \alpha A_{12}B_{21} + \beta C_{11}$	C_{11}	9	$P_5 = \alpha S_1 T_1 + \beta C_{12}$	C_{12}	17	$U_1 = P_1 + P_2$	C_{11}
2	$T_3 = B_{22} - B_{12}$	X_3	10	$T_2 = B_{22} - T_1$	X_3	18	$U_2 = P1 + P6$	tmpU_2
3	$S_3 = A_{11} - A_{21}$	X_2	11	$S_2 = S_1 - A_{11}$	X_2	19	$U_3 = U_2 + P_7$	tmpU_3
4	$C_{22} = C_{22} - C_{12}$	C_{22}	12	$P_6 = \alpha S_2 T_2$	X_1	20	$U_7 = U_3 + P_5$	C_{22}
5	$C_{21} = C_{21} - C_{22}$	C_{21}	13	$S_4 = A_{12} - S_2$	X_2	21	$U_4 = U_2 + P_5$	C_{12}
6	$P_7 = \alpha S_3 T_3 + \beta C_{22}$	C_{22}	14	$T_4 = T_2 - B_{21}$	X_3	22	$U_6 = U_3 - P_4$	C_{21}
7	$T_1 = B_{12} - B_{11}$	X_3	15	$P_4 = \alpha A_{12} T_4 - \beta C_{21}$	C_{21}	23	$P_3 = \alpha S_4 B_{22}$	X_1
8	$S_1 = A_{21} + A_{22}$	X_2	16	$P_1 = \alpha A_{11} B_{11}$	X_3	24	$U_5 = U_4 + P_3$	C_{12}

Table 4: Schedule of Winograd's algorithm for operation $C \leftarrow AB + \beta C$

3.2.3 Control of the overflow

As we saw at the beginning of section 3.1, one has to control the growth of the integral values stored in the floating point representation. Our main result here is to show that, in the worst case, the largest intermediate computation occurs during the recursive computation of P_6. This result generalizes [21, theorem 3.1] for the computation of $AB + \beta C$.

Theorem 3.1. *Let $A \in \mathbb{Z}^{M\times K}$, $B \in \mathbb{Z}^{K\times N}$ $C \in \mathbb{Z}^{M\times N}$ be three matrices and $\beta \in \mathbb{Z}$ with $m_A \leq a_{i,j} < M_A$, $m_B \leq b_{i,j} < M_B$ and $m_C \leq c_{i,j} < M_C$. Moreover, suppose that $0 \leq -m_A \leq M_A$, $0 \leq -m_B \leq M_B$, $0 \leq -m_C \leq M_C$, $M_C \leq M_B$ and $|\beta| \leq M_A, M_B$ Then every intermediate value z involved in the computation of $A \times B + \beta C$ with l ($l \geq 1$) recursive levels of Winograd algorithm satisfy:*

$$|z| \leq \left(\frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A\right)\left(\frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B\right)\left\lfloor\frac{K}{2^l}\right\rfloor$$

Moreover, this bound is optimal.

The proof is given in appendix A.

If the prime field elements are converted into integers between 0 and $p-1$, then the following corollary holds:

Corollary 3.2 (Positive modular representation). *Let $A \in \mathbb{Z}^{M\times K}$, $B \in \mathbb{Z}^{K\times N}$ and $C \in \mathbb{Z}^{M\times N}$ be three matrices and $\beta \in \mathbb{Z}$ with $0 \leq a_{i,j} < p$, $0 \leq b_{i,j} < p$, $0 \leq c_{i,j} < p$ and $0 \leq \beta < p$. Then every intermediate value z involved in the computation of $A \times B + \beta C$ with l ($l \geq 1$) recursive levels of Winograd algorithm satisfy:*

$$|z| \leq \left(\frac{1+3^l}{2}\right)^2 \left\lfloor\frac{K}{2^l}\right\rfloor (p-1)^2$$

Moreover this bound is optimal.

Proof. Apply theorem 3.1 with $m_A = m_B = m_C = 0$ and $M_A = M_B = M_C = p-1$. □

Instead, if the prime fields elements are converted into integers between $-\frac{p-1}{2}$ and $\frac{p-1}{2}$, this bound can be improved:

Corollary 3.3 (Centered modular representation). *Let $A \in \mathbb{Z}^{M\times K}$, $B \in \mathbb{Z}^{K\times N}$ and $C \in \mathbb{Z}^{M\times N}$ be three matrices and $\beta \in \mathbb{Z}$ with $-\frac{p-1}{2} \leq a_{i,j} \leq \frac{p-1}{2}$, $-\frac{p-1}{2} \leq b_{i,j} \leq \frac{p-1}{2}$, $-\frac{p-1}{2} \leq c_{i,j} \leq \frac{p-1}{2}$ and $-\frac{p-1}{2} \leq \beta \leq \frac{p-1}{2}$. Then every intermediate value z involved in the computation of $A \times B + \beta C$ with l ($l \geq 1$) recursive levels of Winograd algorithm satisfy:*

$$|z| \leq \left(\frac{3^l}{2}\right)^2 \left\lfloor \frac{K}{2^l} \right\rfloor (p-1)^2$$

Moreover, this bound is optimal.

Proof. Apply theorem 3.1 with $m_A = m_B = m_C = -\frac{p-1}{2}$ and $M_A = M_B = M_C = \frac{p-1}{2}$. □

Corollary 3.4. *One can compute l recursive levels of Winograd algorithm without modular reduction over integers of γ bits as soon as $K < K_{max}$ where*

$$K_{max} = \left(\frac{2^{\gamma+2}}{\left((1+3^l)(p-1)\right)^2} + 1 \right) 2^l$$

for a positive modular representation and

$$K_{max} = \left(\frac{2^{\gamma+2}}{\left(3^l(p-1)\right)^2} + 1 \right) 2^l$$

for a centered modular representation.

Proof. The bounds of corollaries 3.2 and 3.3 must be below 2^γ. We set $d = \lfloor \frac{K}{2^l} \rfloor$, and solve for d. Now, $2^l d \leq K < 2^l(d+1)$ yields the results. □

3.2.4 Performances and comparison with numerical routines

In this section we discuss the practical benefit of fast matrix multiplication for exact computation. As shown in the section 3.2.1 the Winograd's algorithm allow to decrease the number of arithmetic operations as soon as a good threshold is used to switch to classical matrix multiplication. In the following we show that the use of such hybrid implementation leads in practice to outperform the performances of numerical BLAS matrix multiplication. In particular this is achieved over the prime field of integer modulo 65521.

We use two different BLAS library in our experimentation: one is tuned using ATLAS software [57] and we refer to it with the name "ATLAS"; the other comes from optimized BLAS by Kazushige Goto [30] and we refer to it with the name "GOTO".

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	0.41s	2.86s	8.81s	36.82s	101.27s	144.66s	213.69s	MT
	dgemm	0.39s	3.06s	10.28s	47.19s	129.20s	192.61s	276.43s	379.05s
	$\frac{fgemm}{dgemm}$	**1.05**	**0.93**	**0.85**	**0.78**	**0.78**	**0.75**	**0.77**	**-**
GOTO	fgemm	0.37s	2.60s	8.32s	34.80s	90.54s	128.18s	181.98s	MT
	dgemm	0.36s	2.79s	9.35s	43.36s	118.07s	178.23s	251.11s	344.73s
	$\frac{fgemm}{dgemm}$	**1.02**	**0.93**	**0.88**	**0.80**	**0.76**	**0.71**	**0.72**	**-**

MT: *Memory Trashing*

Table 5: Performance improvement using fast matrix multiplication on a P4, 3.4GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	0.50s	3.66s	11.87s	51.21s	131.78s	188.14s	273.39s	363.41s
	dgemm	0.46s	3.47s	11.74s	53.90s	147.06s	217.20s	311.57s	422.85s
	$\frac{fgemm}{dgemm}$	**1.08**	**1.05**	**1.01**	**0.95**	**0.89**	**0.86**	**0.87**	**0.85**
GOTO	fgemm	0.44s	3.35s	10.68s	46.08s	119.33s	173.24s	245.50s	328.22s
	dgemm	0.40s	3.17s	10.61s	48.90s	133.90s	200.26s	284.47s	391.01s
	$\frac{fgemm}{dgemm}$	**1.1**	**1.05**	**1.00**	**0.94**	**0.89**	**0.86**	**0.86**	**0.83**

Table 6: Performance improvement using fast matrix multiplication on Itanium2, 1.3GHz

The tables 5 and 6 report timings obtained for both exact and numeric matrix multiplication. They explicitly state the improvement brought by using fast hybrid matrix multiplication for exact computation. We refer to "fgemm" for hybrid matrix multiplication over finite field and to "dgemm" for numerical matrix multipliciation over double precision floating point numbers. From these tables one can see that fast hybrid matrix multiplication becomes more powerful as soon as matrices are getting larger and indeed more Winograd levels are used. In particular, we can notice that two levels of Winograd's algorithms are already enough to outperform BLAS matrix multiplication: in our tests the switching thresholds are around 1000 with our P4 architecture and around 2500 with our Itanium2 architecture. It seems that one level of Winograd' algorithm is not yet enough to completely amortize the cost of modular reduction involved in computation over finite field. Nevertheless, with larger matrices the cost of modular reduction is no more a problem and few Winograd's steps lead to non negligible improvements (e.g. 13% to 25% for matrix dimension 9000).

One can also notice that since our hybrid implementation is still based on top of BLAS matrix multiplication routine we directly benefit without effort of improvements made in BLAS implementation. In our examples one can see that GOTO BLAS are faster than ATLAS BLAS on our targeted architecture. This improvement is directly reflected in the performances of our hybrid implementation and the use of Winograd algorithm even amplifies this phenomenon. For example the improvement of GOTO BLAS matrix multiplication regarding ATLAS BLAS is about 10% for matrix dimension 9000 whereas for our implementation the improvement becomes roughly 15%.

3.3 Triangular system solving with matrix hand side

We now discuss the implementation of solvers for triangular systems with matrix right hand side (or equivalently left hand side). This is also the simultaneous resolution of n triangular systems. The resolution of such systems is a classical problem of linear algebra. It is e.g. one of the main operation in block Gaussian elimination as we will see in section 4.1. For solving triangular systems over finite fields, the block algorithm reduces to matrix multiplication and achieves the best known arithmetic complexity. Let us denote by $R(m,k,n)$ the arithmetical cost of a $m \times k$ by $k \times n$ rectangular matrix multiplication. Now let us suppose that $k \leq m \leq n$, then $R(k,m,n)$, $R(m,k,n)$ and $R(m,n,k)$ are all bounded by $\lceil \frac{mn}{k^2} \rceil MM(k)$ (see e.g. [35, (2.5)] for more details). In the following subsections, we present a block recursive algorithm and two optimized implementation variants of triangular system solving for which we study their behaviors and their performances.

From now on, we will denote by $\mathbb{Z}_p$the field of integer modulo the prime p.

3.3.1 Scheme of the block recursive algorithm

The classical idea is to use the divide and conquer approach. Here, we consider the upper left triangular case without loss of generality, since the any combination of upper/lower and left/right triangular cases are similar: if U is upper triangular, L is lower triangular and B is rectangular, we call `ULeft-Trsm` the resolution of $UX = B$, `LLeft-Trsm` that of $LX = B$, `URight-Trsm` that of $XU = B$ and `LRight-Trsm` that of $XL = B$.

Algorithm 2 `ULeft-Trsm`(A, B)

Require: $A \in {\mathbb{Z}_p}^{m \times m}$, $B \in {\mathbb{Z}_p}^{m \times n}$.
Ensure: $X \in {\mathbb{Z}_p}^{m \times n}$ such that $AX = B$.
1: **if** m=1 **then**
2: $X := A_{1,1}^{-1} \times B$.
3: **else**
4: *(splitting matrices into $\lfloor \frac{m}{2} \rfloor$ and $\lceil \frac{m}{2} \rceil$ blocks)*

$$\overbrace{\begin{bmatrix} A_1 & A_2 \\ & A_3 \end{bmatrix}}^{A} \overbrace{\begin{bmatrix} X_1 \\ X_2 \end{bmatrix}}^{X} = \overbrace{\begin{bmatrix} B_1 \\ B_2 \end{bmatrix}}^{B}$$

5: $X_2 :=$`ULeft-Trsm`(A_3, B_2).
6: $B_1 := B_1 - A_2 X_2$.
7: $X_1 :=$`ULeft-Trsm`(A_1, B_1).
8: **end if**
9: **return** X.

Lemma 3.5. *Algorithm* `ULeft-Trsm` *is correct and the dominant term of its arithmetic complexity over $\mathbb{Z}_p$ is*

$$TRSM(m; n) = \begin{cases} \frac{1}{2^{\omega-1}-2} \left\lceil \frac{n}{m} \right\rceil MM(m) & \text{if } m \leq n \\ \frac{1}{2^{\omega-1}-2} \left\lceil \frac{m}{n} \right\rceil^2 MM(n) & \text{if } m \geq n \end{cases}$$

The latter is $min\{mn^2, nm^2\}$ with classical multiplication.

Proof. The correctness of algorithm `ULeft-Trsm` can be proven by induction on the row dimension of the system. For this, one only has to note that

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \text{ is solution} \iff \begin{cases} A_3 X_2 = B_2 \\ A_1 X_1 + A_2 X_2 = B_1 \end{cases}$$

Let $TRSM(m, n)$ be the cost of algorithm `ULeft-Trsm` where m is the dimension of A and n the column dimension of B. It follows from the algorithm that $TRSM(m, n) = 2TRSM(\frac{m}{2}, n) + R(\frac{m}{2}, \frac{m}{2}, n)$. We note $t = \log_2(m)$, then if $m \leq n$, we have thus $TRSM(m, n) = 2TRSM(\frac{m}{2}, n) + \frac{1}{2^{\omega-1}} \left\lceil \frac{n}{m} \right\rceil MM(m) =$

$2^t TRSM(1,n) + \frac{1}{2^{\omega-1}} \left\lceil \frac{n}{m} \right\rceil MM(m) \frac{1-\left(\frac{2}{2^{\omega-1}}\right)^t}{1-\frac{2}{2^{\omega-1}}}$. As $TRSM(1,n) = 2n$ and $\left(2^{\omega-1}\right)^t = m^{\omega-1}$, we get $TRSM(m,n) = \frac{1}{2^{\omega-1}-2} \left\lceil \frac{n}{m} \right\rceil MM(m) + \mathcal{O}(m^2 + mn)$. When $m \geq n$, the trick is to consider two TRSM with the same triangular matrix, but of right hand side of size $n/2$. $R(\frac{m}{2}, \frac{m}{2}, n) = 2R(\frac{m}{2}, \frac{m}{2}, \frac{n}{2})$. Therefore the inequality $m \geq n$ is preserved all along the algorithm and the cost is thus $TRSM(m,n) = 4TRSM(\frac{m}{2}, \frac{n}{2}) + 2R(\frac{m}{2}, \frac{m}{2}, \frac{n}{2}) = 4TRSM(\frac{m}{2}, \frac{n}{2}) + \frac{1}{2^{\omega-1}} \left\lceil \frac{n}{m} \right\rceil^2 MM(n)$. Thus $TRSM(m,n) = 4^t T(1;1) + \frac{1}{2^{\omega-1}} \left\lceil \frac{m}{n} \right\rceil^2 MM(m) \frac{1-\left(\frac{4}{2^\omega}\right)^t}{1-\frac{4}{2^\omega}}$. This yields the dominant term $\frac{2}{2^\omega-4} \left\lceil \frac{m}{n} \right\rceil^2 MM(m)$.

□

3.3.2 Implementation using the BLAS "dtrsm"

Matrix multiplication speed over finite fields was improved in [21, 50] by the use of the numerical BLAS[3] library: matrices were converted to floating point representations (where the linear algebra routines are fast) and converted back to a finite field representation afterwards. The computations remained exact as long as no overflow occurred. An implementation of `ULeft-Trsm` can use the same techniques. Indeed, as soon as no overflow occurs one can replace the recursive call to `ULeft-Trsm` by the numerical BLAS *dtrsm* routine. But one can remark that approximate divisions can occur. So we need to ensure both that only exact divisions are performed and that no overflow appears. Not only one has to be careful for the result to remain within acceptable bounds, but, unlike matrix multiplication where data grows linearly, data involved in linear system grows exponentially as shown in the following.
The next two subsections first show how to deal with divisions, and then give an optimal theoretical bound on the coefficient growth and therefore an optimal threshold for the switch to the numerical call.

3.3.3 Dealing with divisions

In algorithms like `ULeft-Trsm`, divisions appear only within the last recursion's level. In the general case it cannot be predicted whether these divisions will be exact or not. However when the system is unitary (only 1's on the main diagonal) the division are of course exact and will even never be performed. Our idea is then to transform the initial system so that all the recursive calls to `ULeft-Trsm` are unitary. For a triangular system $AX = B$, it suffices to factor first the matrix A into $A = UD$, where U, D are respectively an upper unit triangular matrix and a diagonal matrix. Next the unitary system $UY = B$ is solved by any `ULeft-Trsm` (even a numerical one), without any division. The initial solution is then recovered over the finite field via $X = D^{-1}Y$. This normalization leads to an additional cost of:

- m inversions over $\mathbb{Z}_p$ for the computation of D^{-1}.

[3] `www.netlib.org/blas`

- $(m-1)\frac{m}{2} + mn$ multiplications over $\mathbb{Z}_p$ for the normalizations of U and X.

Nonetheless, in our case, we need to avoid divisions only during the numerical phase. Therefore, the normalization can take place only just before the numerical routine calls. Let β be the size of the system when we switch to a numerical computation. To compute the cost, we assume that $m = 2^i\beta$, where i is the number of recursive level of the algorithm `ULeft-Trsm`. The implementation can however handle any matrix size. Now, there are 2^i normalizations with systems of size β. This leads to an additional cost of:

- m inversions over $\mathbb{Z}_p$.
- $(\beta-1)\frac{m}{2} + mn$ multiplications over $\mathbb{Z}_p$.

This allows us to save $\left(\frac{1}{2} - \frac{1}{2^{i+1}}\right) m^2$ multiplications over $\mathbb{Z}_p$ from a whole normalization of the initial system. One iteration suffices to save $\frac{1}{4}m^2$ multiplications and we can save up to $\frac{1}{2}(m^2 - m)$ multiplications with $\log m$ iterations.

3.3.4 A theoretical threshold

We want to use the BLAS trsm routine to solve triangular systems over the integers (stored as `double` for `dtrsm` or `float` for `strsm`). The restriction is then the coefficient growth in the solution. Indeed, the k^{th} value in the solution vector is a linear combination of the $(n-k)$ already computed next values. This implies a linear growth in the coefficient size of the solution, with respect to the system dimension. Now this resolution can only be performed if every element of the solution can be stored in the mantissa of the floating point representation (e.g. 53 bits for `double`). Therefore overflow control consists in finding the largest block dimension β, such that the result of the call to BLAS trsm routine will remain exact.

We now propose a bound for the values of the solutions of such a system; this bound is optimal (in the sense that there exists a worst case matching the bound when $n = 2^i\beta$). This enables the implementation of a cascading algorithm, starting recursively and taking advantage of the BLAS performances as soon as possible.

Theorem 3.6. *Let $T \in \mathbb{Z}^{n\times n}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^n$, with $0 \le T \le p-1$ and $0 \le b \le p-1$. Let $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ be the solution of $T.X = b$ over the integers. Then, $\forall\ k \in [0..n-1]$:*

$$\begin{cases} (p-2)^k - p^k \le 2\frac{x_{n-k}}{p-1} \le p^k + (p-2)^k & \text{if } k \text{ is even} \\ -p^k - (p-2)^k \le 2\frac{x_{n-k}}{p-1} \le p^k - (p-2)^k & \text{if } k \text{ is odd} \end{cases}$$

Proof. The idea is to use an induction on k with the relation $x_k = b_k - \sum_{i=k+1}^{n} T_{k,i}x_i$. A lower and an upper bound for x_{n-k} are computed, depending whether k is even or odd: Let us define the following induction hypothesis IH_l:

$$\forall\ k \in [0..l-1] \begin{cases} -u_k \le x_{n-k} \le v_k & \text{if } k \text{ is even} \\ -v_k \le x_{n-k} \le u_k & \text{if } k \text{ is odd} \end{cases}$$

When $l = 0$, $x_n = b_n$ which implies that $-u_0 = 0 \leq x_n \leq p-1 = v_0$. Thus IH_0 is proven. Let us suppose that $\forall j \in [0..l]$ IH_j is true, and prove IH_{l+1}. There are two cases: either l is odd or not ! If l is odd, $l+1$ is even. Now, by induction,

$$\begin{aligned}
x_{n-l-1} &\leq (p-1)\left(1 + \sum_{i=0}^{\frac{l-1}{2}} u_{2i} + v_{2i+1}\right) \\
&\leq p-1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{(p-1)^2}{2}\left[p^{2i} - (p-2)^{2i} + p^{2i+1} + (p-2)^{2i+1}\right] \\
&\leq p-1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{(p-1)^2}{2}\left[p^{2i}(p+1) + (p-2)^{2i}(p-3)\right] \\
&\leq p-1 + \frac{(p-1)^2}{2}\left[(p+1)\frac{p^{l+1}-1}{p^2-1} + (p-3)\frac{(p-2)^{l+1}-1}{(p-2)^2-1}\right] \\
&\leq \frac{p-1}{2}\left[p^{l+1} + (p-2)^{l+1}\right] = v_{l+1}
\end{aligned}$$

Similarly,

$$\begin{aligned}
x_{n-l-1} &\geq -(p-1)\sum_{i=0}^{\frac{l-1}{2}} v_{2i} + u_{2i+1} \\
&\geq -\frac{(p-1)^2}{2}\sum_{i=0}^{\frac{l-1}{2}}\left[p^{2i} + (p-2)^{2i} + p^{2i+1} - (p-2)^{2i+1}\right] \\
&\geq -\frac{(p-1)^2}{2}\sum_{i=0}^{\frac{l-1}{2}}\left[p^{2i}(p+1) - (p-2)^{2i}(p-3)\right] \\
&\geq -\frac{p-1}{2}\left[p^{l+1} - (p-2)^{l+1}\right] = u_{l+1}
\end{aligned}$$

Finally, If l is even, a similar proof leads to $-v_{l+1} \leq x_{n-l+1} \leq u_{l+1}$. □

Corollary 3.7. $|X| \leq \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right]$.
Moreover, this bound is optimal.

Proof. We denote by $u_n = \frac{p-1}{2}\left[p^n - (p-2)^n\right]$ and $v_n = \frac{p-1}{2}\left[p^n + (p-2)^n\right]$ the bounds of the theorem 3.6. Now $\forall\, k \in [0..n-1]$ $u_k \leq v_k \leq v_{n-1}$. Therefore the theorem 3.6 gives $\forall\, k \in [1..n]$ $x_k \leq v_{n-1} \leq \frac{p-1}{2}\left[p^{n-1} + (p-2)^{n-1}\right]$

$$\text{Let } T = \begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ & 1 & p-1 & 0 & p-1 \\ & & 1 & p-1 & 0 \\ & & & 1 & p-1 \\ & & & & 1 \end{bmatrix}, b = \begin{bmatrix} \vdots \\ 0 \\ p-1 \\ 0 \\ p-1 \end{bmatrix}$$

Then the solution $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ of the system $T.X = b$ satisfies $\forall\ k \in [0..n-1]\ |x_{n-k}| = v_k$ □

One can derive the same kind of bound for the centered representation, but with an 2^n gain.

Theorem 3.8. *Let $T \in \mathbb{Z}^{n\times n}$ be a unit diagonal upper triangular matrix, and $b \in \mathbb{Z}^n$, with $|T| \leq \frac{p-1}{2}$ and $|b| \leq \frac{p-1}{2}$. Let $X = (x_i)_{i\in[1..n]} \in \mathbb{Z}^n$ be the solution of $T.X = b$ over the integers. Then $|X| \leq \frac{p-1}{2}\left(\frac{p+1}{2}\right)^n$.*
Moreover, this bound is optimal.

Proof. The proof is simpler than that of theorem 3.6, since the inequations are symmetric. Therefore, $u_n = v_n$ and the induction yields $u_n = \frac{p-1}{2}\left(1 + \sum_{i=0}^{n-1} u_i\right) = \frac{p-1}{2}\left(1 + \frac{p-1}{2}\frac{\left(\frac{p+1}{2}\right)^n - 1}{\frac{p+1}{2}-1}\right) = \frac{p-1}{2}\left(\frac{p+1}{2}\right)^n$. □

Thus, for a given p, the dimension n of the system must satisfy

$$\frac{p-1}{2}\left(\frac{p+1}{2}\right)^n < 2^m \tag{6}$$

where m is the size of the mantissa so that the resolution over the integers using the BLAS trsm routine is exact. For instance, with a 53 bits mantissa, this gives quite small matrices, namely at most 92×92 for $p = 2$, at most 4×4 for $p \leq 3089$, and at most $p = 416107$ for 2×2 matrices. Nevertheless, this technique is speed-worthy in most cases as shown in section 3.3.6.

3.3.5 Recursive with delayed modulus

In the previous section we noticed that BLAS routines within `Trsm` are used only for small systems. An alternative is to change the cascade: instead of calling the BLAS, one could switch to the classical iterative algorithm: Let $A \in {\mathbb{Z}_p}^{m\times m}$ and $B, X \in {\mathbb{Z}_p}^{m\times n}$ such that $AX = B$, then

$$\forall i, X_{i,*} = \frac{1}{A_{i,i}}(B_{i,*} - A_{i,[i+1..m]}X_{[i+1..m],*}) \tag{7}$$

The idea is that the iterative algorithm computes only one row of the whole solution at a time. Therefore its threshold t is greater than the one of the BLAS routine, namely it requires only

$$\begin{array}{ll} t(p-1)^2 < 2^m & \text{with an unsigned representation (i.e. } [0..p-1]\text{), or} \\ t(p-1)^2 < 2^{m+1} & \text{with a signed one (i.e. } [\frac{1-p}{2}..\frac{p-1}{2}]\text{).} \end{array} \tag{8}$$

Resultantly, an implementation of this iterative algorithm depends mainly on the matrix-vector product. The arithmetical cost of such an algorithm is now cubic in the size of the system, where blocking improved the theoretical complexity.But in practice fast matrix multiplication algorithms are not better than the classical one for such small matrices as shown in [21, §3.3.2].

3.3.6 "Trsm" implementations behavior

As shown in section 3.3.1 the block recursive algorithm `Trsm` is based on matrix multiplications and allows us to reuse our efficient matrix multiplication routines presented in section 3.1 and 3.2. In the following we compare three implementation variants of `Trsm` based on the classic matrix multiplication of section 3.1. The pure recursive version of section 3.3.1 is labeled "pure rec" while the optimized variant of section 3.3.2 with optimal threshold is designed by "blas". The label "delayed$_t$" denotes the variant of section 3.3.5 where t satisfies equation 8 with unsigned representation and corresponds to the switching threshold.

Our comparisons use classical prime field arithmetic of section 2.1.1 with word size prime number and two datatype representations (i.e. 32-bits integer and 64-bits floating point number). Remember that the latter only uses the 53-bits of mantissa in order to guarantee an exact arithmetic. The two implementations are denoted respectively: `ZpZ-int` and `ZpZ-double`.

	n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
$\mathbb{Z}/5\mathbb{Z}$	pure rec.	853	1216	1470	1891	2059	2184
	blas	**1306**	**1715**	**1851**	**2312**	**2549**	**2660**
	delayed$_{100}$	1163	1417	1538	1869	2042	2137
	delayed$_{50}$	1163	1491	1639	1955	2067	2171
$\mathbb{Z}/32749\mathbb{Z}$	pure rec.	810	1225	1449	1886	2037	2184
	blas	1066	1504	**1639**	**2099**	**2321**	**2378**
	delayed$_{100}$	1142	1383	1538	1860	2019	2143
	delayed$_{50}$	**1163**	**1517**	**1639**	1955	2080	2172
	delayed$_3$	914	1279	1449	1941	2139	2159

Table 7: Comparing speed (Mfops) of `Trsm` using `Zpz-double`, on a P4, 2.4GHz

	n	*400*	*700*	*1000*	*2000*	*3000*	*5000*
$\mathbb{Z}/5\mathbb{Z}$	pure rec.	571	853	999	1500	1708	1960
	blas	688	1039	1190	**1684**	**1956**	**2245**
	delayed$_{150}$	799	**1113**	909	1253	1658	2052
	delayed$_{100}$	**831**	1092	**1265**	1571	1669	2046
	delayed$_{23}$	646	991	1162	1584	1796	2086
$\mathbb{Z}/32749\mathbb{Z}$	pure rec.	551	786	1010	1454	1694	1929
	blas	547	828	990	1449	1731	1984
	delayed$_{100}$	703	958	1162	1506	1570	1978
	delayed$_{50}$	**842**	**1113**	**1282**	**1731**	**1890**	**2174**
	delayed$_3$	528	769	900	1367	1664	1911

Table 8: Comparing speed (Mfops) of `Trsm` using `ZpZ-int`, on a P4, 2.4GHz

One can see from table 7 that "blas" `Trsm` implementation with a `ZpZ-double`

representation is the most efficient choice for small primes (here switching to BLAS happens for $n = 23$ when $p = 5$). Now for larger primes, despite a very small granularity (e.g switching to BLAS happens only for $n = 3$ when $p = 32749$), this choice remains the best as soon as systems are large (i.e. $n > 1000$). This is because grouping operations into blocks speeds up the computation. Now in case of smaller systems, the "delayed$_t$" variant becomes more efficient, due to the good behavior of dot product. However the threshold t has to be chosen carefully as shown in table 8. As a comparison, we provide performances for several thresholds, in particular the same as within "blas" variant (i.e. 3 and 23). Indeed using a threshold of 50 enables better performances than "blas" variant. This is because conversions from 32-bits integers to floating points numbers becomes too big a price to pay. However, for larger matrices, conversions ($O(n^2)$) are dominated by computations ($O(n^\omega)$), and then "blas" variant becomes again the fastest one, provided that the field is small enough.

To summarize, one would rather use `ZpZ-double` representation and "blas" `Trsm` variant in most cases. However, when the base field is already specified "delayed$_t$" could provide slightly better performances. This requires a search for optimal thresholds which again could be done through an Automated Empirical Optimizations of Software [57].

3.3.7 Performances and comparison with numerical routines

In the previous section we showed that `Trsm` optimized variant based on numerical solving allows us to achieve the best performances. In this section we compare these performances with pure numerical solving and with matrix multiplication. In order to achieve the best performances we use as much as possible fast matrix multiplication of section 3.2. For this purpose we use an experimental switching threshold to classic multiplication since table 2 reflects only theoretical behavior. As for matrix multiplication in section 3.2.4, we compare our routines according to two different BLAS optimizations (i.e. ATLAS and GOTO) and two different architectures. Nevertheless, we do not present the results with ATLAS on P4 architecture due to really poor performances of ATLAS "dtrsm" routine during our tests. We use a `ZpZ-double` representation with a 16-bits prime (i.e. 65521) for exact computation labeled "ftrsm" in the following.

Tables 9 and 10 show that our implementation of exact `Trsm` solving is not far from numerical performances. In particular, "ftrsm" performances tend to catch up with BLAS ones as soon as the dimensions of matrices increase. Moreover, with our P4 architecture and GOTO BLAS, we are able to achieve even better performances than numerical solving for matrices of dimension 10 000.

The good performances of our implementation is mostly achieved with the reduction to matrix multiplication. The figure 12 shows the performances ratio of our `Trsm` implementation with our matrix multiplication routine. One can see from this figure that our experimental ratio converges to the theoretical one. In particular, the theoretical ratio is slightly more than $\frac{1}{2}$ since fast matrix multiplication algorithm is used. According to lemma 3.5 and complexity exponent

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
GOTO	ftrsm	0.35s	2.18s	6.38s	25.66s	64.38s	91.19s	127.99s	170.44s
	dtrsm	0.19s	1.50s	4.92s	22.94s	59.97s	90.08s	127.05s	173.67s
	$\frac{ftrsm}{dtrsm}$	**1.84**	**1.45**	**1.29**	**1.11**	**1.07**	**1.01**	**1.00**	**0.98**

Table 9: Timings of triangular solver with matrix hand side on a P4, 3.4GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	ftrsm	0.52s	3.17s	9.33s	38.48s	96.83s	139.56s	192.34s	259.80s
	dtrsm	0.27s	2.04s	6.71s	29.62s	79.44s	117.00s	166.18s	224.15s
	$\frac{ftrsm}{dtrsm}$	**1.92**	**1.55**	**1.39**	**1.29**	**1.21**	**1.19**	**1.15**	**1.15**
GOTO	ftrsm	0.48s	2.87s	8.60s	35.85s	87.29s	123.91s	172.49s	240.53s
	dtrsm	0.30s	2.01s	6.37s	27.14s	72.21s	106.99s	150.86s	223.75s
	$\frac{ftrsm}{dtrsm}$	**1.6**	**1.42**	**1.35**	**1.32**	**1.20**	**1.15**	**1.14**	**1.07**

Table 10: Timings of triangular solver with matrix hand side on Itanium2, 1.3GHz

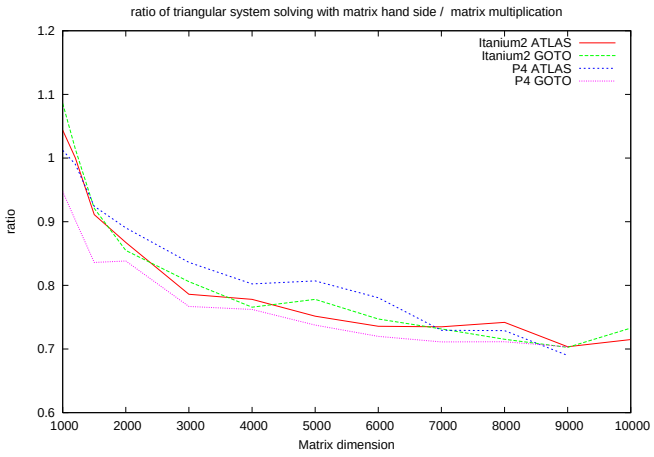

Figure 12: Comparing triangular matrix system solving with matrix multiplication

of Winograd fast matrix multiplication [56] the theoretical ratio tends to be around $\frac{2}{3}$. For matrix dimension 10 000 our experimental ratio is around 0.7 which is very close from the theoretical one (i.e. 0.6666).

4 Finite Field Matrix Factorizations and Applications

We now come to one of the major interest of linear algebra over finite field: matrix multiplication based algorithms. The classical block Gaussian elimination is one of the most common algorithm to achieve such a reduction [56]. Nevertheless, our main concern here is the singularity of the matrices since we want to derive efficient algorithms for most problems (e.g. rank or nullspace). One approach to solve these reductions is then to simplify the problem using a triangular form of the input matrix. Hence, matrix triangularization algorithm plays a central role for this approach. In this section we focus on practical implementations of triangularization in order to efficiently deal with rank profile, unbalanced dimensions, memory management, recursive thresholds, etc. In particular we demonstrate the efficiency of matrix multiplication reduction in practice for many linear algebra problems.

4.1 Triangularizations

Indeed, the classical LDU or LUP factorizations (see [1]) can not be used due to their restriction to non-singular case. Therefore, in this section we present and study three variants of recursive exact triangularization allowing singularity. First the classical `LSP` [39] is sketched. In order to reduce its memory requirements, a first version, `LUdivine`, stores L in-place, but temporarily uses some extra memory. Our last implementation is fully in-place without any extra memory requirements and corresponds to Ibarra's `LQUP`. Note that one can easily recover one to another triangularization by simply using extractions and permutations.

4.1.1 LSP Factorization

The `LSP` factorization is a generalization of the well known block `LUP` factorization for the singular case [6]. Let A be a $m \times n$ matrix, we want to compute the triple $< L, S, P >$ such that $A = LSP$. The matrices `L` and `P` are as in `LUP` factorization and `S` reduces to a non-singular upper triangular matrix when zero rows are deleted. The algorithm with best known complexity computing this factorization uses a divide and conquer approach and reduces to matrix multiplication [39]. Let us describe briefly the behavior of this algorithm. The algorithm is recursive: first, it splits A in halves and performs a recursive call on the top block. It thus gives the T, Y and L_1 blocks of figure 13. Then, after some permutations ($[XZ] = [A_{21}A_{22}]P$), it computes G such that $GT = X$ via `Trsm`, replaces X by zeroes and eventually updates $Z = Z - GY$. The third step

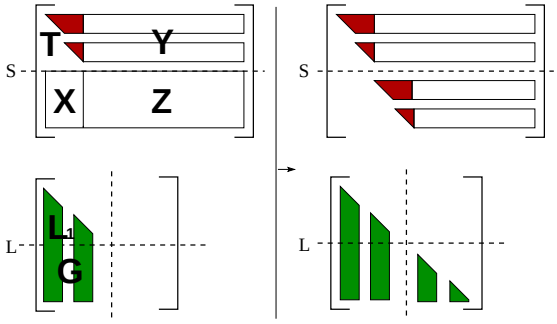

Figure 13: Principle of the LSP factorization

is a recursive call on Z. We let the readers refer e.g. to [4, (2.7c)] for further details.

Lemma 4.1. *Algorithm* `LSP` *is correct. The dominant term of its complexity when* $m \leq n$ *is*

$$LSP(m;n) = \left(\left\lceil \frac{n}{m} \right\rceil \frac{1}{2^{\omega-1}-2} - \frac{1}{2^{\omega}-2} \right) MM(m).$$

The latter is $nm^2 - \frac{1}{3}m^3$ *with classical multiplication.*

Proof. Lemma 1.2 ensures that the cost is $\mathcal{O}(m^\omega + nm^{\omega-1})$. We thus just have to look for the constant factors. Then we write $LSP(m;n) = \alpha m^\omega + \beta nm^{\omega-1} = LSP(m/2;n) + TRSM(m/2;r) + R(m/2;r;n-r) + LSP(m/2;n-r)$, where r is the rank of the first $m/2$ rows. This gives $\alpha m^\omega + \beta nm^{\omega-1} = \alpha(m/2)^\omega + \beta n(m/2)^{\omega-1} + \frac{1}{2^{\omega-1}-2}\left\lceil \frac{m}{2r} \right\rceil MM(r) + \left\lceil \frac{m(n-r)}{2r^2} \right\rceil MM(r) + \alpha(m/2)^\omega + \beta(n-r)(m/2)^{\omega-1}$. With $m \leq n$, the latter is maximal for $r = m/2$, and then, writing $MM(x) = C_\omega x^\omega$, we identify the coefficient on both sides: $\beta = \frac{\beta}{2^{\omega-1}} + \frac{C_\omega}{2^{\omega-1}} + \frac{\beta}{2^{\omega-1}}$, and $\alpha = 2\frac{\alpha}{2^\omega} - \frac{\beta}{2^\omega}$. Solving for α and β gives the announced terms. □

The point here is that, L being square $m \times m$ does not fit in place under S. Therefore a first implementation produces an extra triangular matrix. The following subsections address this memory issue.

4.1.2 LUdivine

The main concern with the direct implementation of the `LSP` algorithm, is the storage of the matrix L: it can not be stored directly with its zero columns under S (as shown in figure 13). Actually, there is enough room under S to store all the non zero entries of L, as shown in figure 14. Storing only the non zero columns of L is the goal of the `LUdivine` variant. One can notice that this operation corresponds to the storage of $\tilde{L} = LQ$ instead of L, where

Q is a permutation matrix such that $Q^T S$ is upper triangular. Consequently, the recovery of L from the computed $\tilde{L}$ is straightforward. Note that this $\tilde{L}$ corresponds to the echelon form of [40, §2] up to some transpositions.

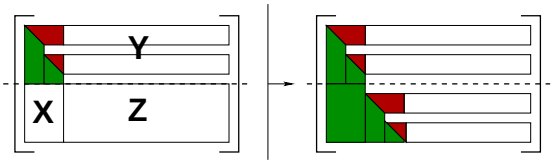

Figure 14: Principle of the LUdivine factorization

Further developments on this implementation are presented in [5, 51]. However, this implementation is still not fully in place. Indeed, to solve the triangular system $G = X.T^{-1}$, one has then to convert T to an upper triangular matrix stored in a temporary memory space. In the same way, the matrix product $Z = Z - GY$ also requires a temporary memory allocation, since rows of Y have to be shifted. This motivates the introduction of the `LQUP` decomposition.

4.1.3 LQUP

To solve the data locality issues, due to zero rows inside S, one can prefer to compute the `LQUP` factorization, also introduced in [39]. It consists in a slight modification of the `LSP` factorization: S is replaced by U, the corresponding upper triangular matrix, after the permutation of the zero rows. The transpose of this row permutation is stored in Q.

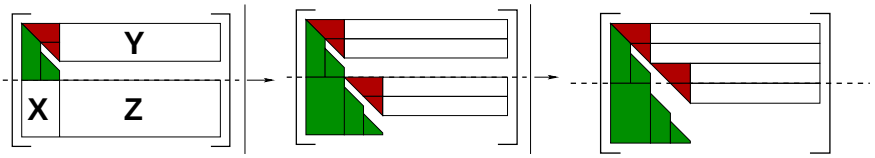

Figure 15: Principle of the LQUP factorization

This prevents the use of temporaries for Y and T, since the triangles in U are now contiguous. Moreover, the number of instructions to perform the row permutations is lower than the number of instructions to perform the block copies of `LUdivine` or `LSP`. Furthermore, our implementation of `LQUP` also uses the trick of `LUdivine`, namely storing L in its compressed form $\tilde{L}$. Thanks to all these improvements, this triangulation appears to be fully in place. As will be shown in section 4.1.4, it is also more efficient. Here again, the `LSP` and `LQUP` factorizations are simply connected via $S = QU$. So the recovery of the `LSP` is still straightforward.

4.1.4 triangularization implementations behavior

As shown in previous sections the three variants of triangularization mainly differ by their memory management. Indeed, the main operations remain matrix multiplication and triangular system solving. Therefore, the implementation of all these variants are based on our matrix multiplication routines of section 3.1 and triangular system solver of section 3.3.2. The results are impressive: for example, table 11 shows that it is possible to triangularize a 5000×5000 matrix over a finite field in 29.9 seconds. We now compare the three routine speed and memory usage with the same kernels: a `ZpZ-double` representation (so that no conversion overhead occur) and classic matrix multiplication routine.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	0.05	0.48	8.01	32.54	404.8	1804
`LUdivine`	0.05	0.47	7.79	30.27	403.9	1691
`LQUP`	0.05	0.45	7.59	29.90	201.7	1090

Table 11: Comparing real time (seconds) of `LSP`, `LUdivine`, `LQUP` over $\mathbb{Z}_{101}$, on a P4, 2.4GHz

For table 11, we used random dense square matrices (but with $3n$ non-zero entries) so as to have rank deficient matrices. The timings given in table 11 are close since the dominating operations of the three routines are similar. `LSP` is slower, since it performs some useless zero matrix multiplications when computing $Z = Z - GY$ (section 4.1.2). `LQUP` is slightly faster than `LUdivine` since row permutations involve less operations than the whole block copy of `LUdivine` (section 4.1.3). However these operations do not dominate the cost of the factorization, and they are therefore of little influence on the total timings. This is true until the matrix size induces some disk swapping, around matrix dimension 8000.

n	*400*	*1000*	*3000*	*5000*	*8000*	*10000*
`LSP`	2.83	17.85	160.4	444.2	1136	1779
`LUdivine`	1.60	10.00	89.98	249.9	639.6	999.5
`LQUP`	1.28	8.01	72.02	200.0	512.1	800.0

Table 12: Comparing memory usage (Mega bytes) of `LSP`, `LUdivine`, `LQUP` over $\mathbb{Z}_{101}$, on a P4, 2.4GHz with 512 Mb RAM

Now for the memory usage, the fully in-place implementation of `LQUP` saves 20% of memory (table 12) when compared to `LUdivine` and 55% when compared to `LSP`. Actually, the memory usage of the original `LSP` is approximately that of `LUdivine` augmented by the extra matrix storage (which corresponds exactly to that of `LQUP`: e.g. $5000 * 5000 * 8bytes = 200Mb$). This memory reduction is of high interest when dealing with large matrices and so with disk swapping (further improvements on the memory management are presented section 4.2).

4.1.5 Performances and comparison with numerical routines

Fast matrix multiplication routine of section 3.2 allowed us to speed up matrix multiplication as well as triangular system solving. These improvements are of great interest since they directly improve performances of triangularization. We now compare our exact triangularization over finite field with numerical triangularization provided within LAPACK library [2]. In particular, we use an optimized version of this library provided by ATLAS software in which we use two different BLAS kernel: ATLAS and GOTO.

Tables 13 and 14 show performances obtained with our exact triangularization based on fast matrix multiplication and the one obtained with numerical computation. Exact computation is done in the prime field of integers modulo 65521. The performances of exact computation are really stupefying since we are mostly able to obtain the same performances as numerical computation. More precisely, we are able to compute the triangularization of a $10\,000 \times 10\,000$ matrix over a finite field in about 2 minutes on a P4 3.4GHz architecture. This is only 10% slower than numerical computation.

We could have expected that our performances would have been even better than numerical approach since we take advantage of fast matrix multiplication while numerical computation not. However, in practice we do not fully benefit from fast matrix multiplication since we work at most with matrices of half dimension of the input matrix. Then the number of Winograd calls is at least one less then within matrix multiplication routines. In our tests, it appears that we only use 3 calls on our P4 architecture and 1 call on the Itanium2 architecture according to matrix multiplication threshold. This explains the better performances on the P4 compare to numerical routines than the Itanium2 architecture.

4.2 Data locality

To solve even bigger problems, say that the matrices do not fit in RAM, one has mainly two solutions: either perform out of core computations or parallelize the resolution. In both cases, the memory requirements of the algorithms to be used will become the main concern. This is because the memory accesses (either on hard disk or remotely via a network) dominate the computational cost. A classical solution is then to improve data locality so as to reduce the volume of these remote accesses. In such critical situations, one may have to prefer a slower algorithm having a good memory management, rather than the fastest one, but suffering from high memory requirements. We here propose to deal with this concern in the case of rank or determinant computations of large dense matrices. The generalization to the full factorization case being direct but not yet fully implemented.

To improve data locality and reduce the swapping, the idea is to use square recursive blocked data formats [31]. A variation of the `LSP` algorithm, namely the `TURBO` algorithm [24], adapts this idea to the exact case. Alike the `LQUP` algorithm which is based on a recursive splitting of the row dimension (see

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	0.34s	1.98s	5.54s	21.38s	54.67s	79.45s	110.29s	137.56s
	dgetrf	0.17s	1.22s	3.93s	17.39s	46.65s	65.09s	97.99s	133.01s
	$\frac{lqup}{dgetrf}$	**2.00**	**1.62**	**1.40**	**1.22**	**1.17**	**1.22**	**1.12**	**1.03**
GOTO	lqup	0.30s	1.78s	5.08s	19.94s	49.67s	73.22s	97.71s	129.76s
	dgetrf	0.15s	1.14s	3.54s	16.86s	41.77s	62.28s	88.00s	120.71s
	$\frac{lqup}{dgetrf}$	**2.00**	**1.56**	**1.43**	**1.18**	**1.18**	**1.17**	**1.11**	**1.07**

Table 13: Performances of matrix triangularization on P4, 3.4GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	0.45s	2.59s	7.32s	28.94s	71.19s	101.57s	141.79s	189.39s
	dgetrf	0.22s	1.53s	4.84s	21.14s	55.84s	81.81s	116.02s	156.34s
	$\frac{lqup}{dgetrf}$	**2.04**	**1.69**	**1.51**	**1.36**	**1.21**	**1.24**	**1.22**	**1.21**
GOTO	lqup	0.57s	2.48s	6.32s	22.77s	49.36s	68.90s	107.46s	184.83s
	dgetrf	0.20s	1.39s	4.33s	18.69s	49.44s	73.28s	103.07s	140.81s
	$\frac{lqup}{dgetrf}$	**2.85**	**1.78**	**1.45**	**1.21**	**0.99**	**0.94**	**1.04**	**1.31**

Table 14: Performances of matrix triangularization on Itanium2-1.3GHz

section 4.1.3), `TURBO` achieves more data locality by splitting both row *and* column dimensions. Indeed the recursive splitting with only the row dimension tend to produce "very rectangular" blocks: a large column dimension and a small row dimension. On the contrary, `TURBO` preserves the squareness of the original matrix for the first levels. More precisely each recursive level consists in a splitting of the matrix into four static blocks followed by five recursive calls to matrix triangularizations (U, V, C, D, and Z, in that order on figure 16), six `Trsm` and four matrix multiplications for the block updates. In this

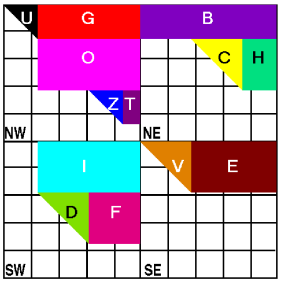

Figure 16: Principle of the TURBO decomposition

first implementation, only one recursive step of `TURBO` is used, the five recursive calls being performed by the `LQUP` algorithm. For the actual size of matrices, the quite complex implementation of more recursive levels of `TURBO` is not yet mandatory.

Now for the comparisons of figure 17, we use the full `LQUP` factorization algorithm as a reference. Factorization of matrices of size below 8000 fit in 512Mb of RAM. Then `LQUP` is slightly faster than `TURBO`, implementation of the latter producing slightly more scattered groups. Now, the first field representation chosen (curves 1 and 2) is a modular prime field representation using machine integers. As presented in [21], any matrix multiplication occurring in the decomposition over such a representation is performed by converting the three operands to three extra floating point matrices. This memory overhead is critical in our comparison. `TURBO`, having a better data locality and using square blocks whenever possible, requires smaller temporary matrices than the large and very rectangular blocks used in `LQUP`. Therefore, for matrices of order over 8000, `LQUP` has to swap a lot while `TURBO` remains more in RAM. This is strikingly true for matrices between 8000 and 8500, where `TURBO` manages to keep its top speed.

To also reduce the memory overhead due to the conversions to floating point numbers, one can use the `Zpz-double` field representation, as used in section 3.3.6. There absolutely no allocation is done beside the initial matrix storage. On the one hand, performances increase since the conversions and copy are no longer performed, as long as the computations remain in RAM (see curves 3 and 4). On the other hand, the memory complexities of both algorithms now become identical. Furthermore, this fully in-place implementation does

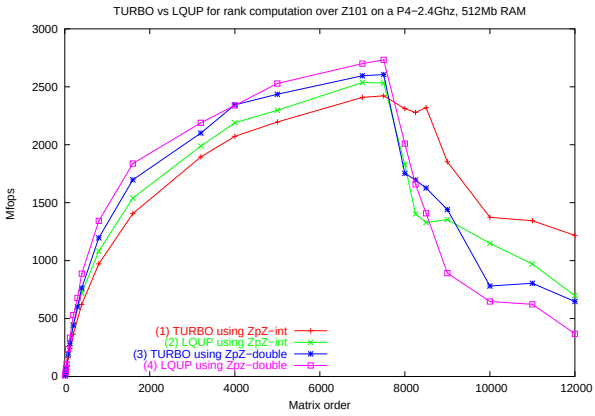

Figure 17: TURBO versus LQUP for out of core rank

not create small block copies anymore. Paradoxically, this prevents the virtual blocks from fitting in the RAM, since they are just a view of the large initial matrix. For this reason, both performance losses appear for matrices of order around 8000. However, the drop is lower for `TURBO` thanks to the recursive blocked data formats producing better data locality.

This behavior of course confirms that as soon as the RAM is full, data locality becomes more important than memory saves : `TURBO` over `ZpZ-int` is the fastest for matrices of size bigger than 8000, despite its bigger memory demand. This is advocating further uses of recursive blocked data formats and of more recursive levels of `TURBO`.

4.3 Rank, determinant

The `LQUP` factorization and the `Trsm` routines reduce to matrix multiplication as we have seen in the previous sections. Theoretically, as classic matrix multiplication requires $2n^3 - n^2$ arithmetic operations, the factorization, requiring at most $\frac{2}{3}n^3$ arithmetic operations, could be computed in about $\frac{1}{3}$ of the time. However, when Winograd fast matrix multiplication algorithm is used this ratio becomes $\frac{2}{5}$. Now, the matrix multiplication routine `Fgemm` of section 3.2.4 can compute 5000×5000 matrix multiplications in only 34.8 seconds on a 3.4GHz Pentium 4. This is achieved with 3 levels of Winograd algorithm and with very good performances of the GOTO BLAS. Well, figure 18 shows that with $n \times n$ matrices we are not very far from these quasi-optimal performances also for the factorization.

Moreover, from the two routines (i.e. `LQUP` and `Trsm`, one can also easily derive several other algorithms:

- The **rank** is the number of non-zero rows in U.

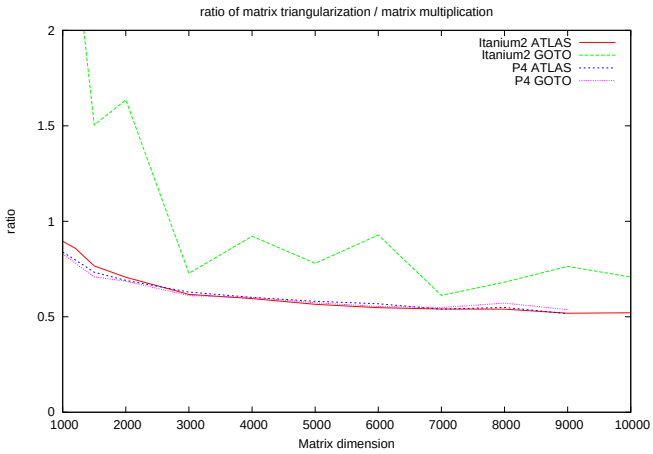

Figure 18: Comparing matrix triangularization with matrix multiplication

- The **determinant** is the product of the diagonal elements of U (stopping whenever a zero is encountered).

4.4 Nullspace basis

Computing a right nullspace basis with the LQUP factorization is immediate on a $m \times n$ full rank matrix, where $m \leq n$: if $U = [U_1 U_2]$, the matrix $U_1^{-1} U_2$ completed with identity matrix yields a basis for the nullspace of A.

This requires $NS(m;n) = LQUP(m;n) + TRSM(m;n-m)$. which gives

$$NS(m;n) = (\left\lceil \frac{n}{m} \right\rceil \frac{2}{2^{\omega-1}-2} - \frac{1}{2^{\omega}-2}) MM(m) \tag{9}$$

The latter is $(m^2 n - \frac{1}{3}m^3) + (n-m)m^2 = 2m^2 n - \frac{4}{3}m^3$ with classical multiplication. One can notice that computing a right nullspace of the transposed of the input matrix yields a left nullspace basis.

4.5 Triangular multiplications

4.5.1 Triangular matrix multiplication

To perform the multiplication of a triangular matrix by a dense matrix via a block decomposition, one requires four recursive calls and two dense matrix-matrix multiplications. The cost is thus $TRMM(n) = 4TRMM(n/2) + 2MM(n/2)$, solving for $TRMM(n) = \alpha MM(n)$ yields

$$TRMM(n) = \frac{2}{2^{\omega}-4} MM(n). \tag{10}$$

The latter is n^3 with classical multiplication.

4.5.2 Upper-lower Triangular matrix multiplication

The block multiplication of a lower triangular matrix by an upper triangular matrix is

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_3 & A_2B_4 \\ A_4B_3 & A_4B_4 \end{bmatrix}$$

The cost is thus $UTLT(n) = 2UTLT(n/2)+2TRMM(n/2)+MM(n/2)$, solving for $UTLT(n) = \alpha MM(n)$ yields

$$UTLT(n) = \frac{2^\omega}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{11}$$

The latter is $\frac{2}{3}n^3$ with classical multiplication.

4.5.3 Upper-Upper Triangular matrix multiplication

Now the block version is even simpler (of course the lower lower multiplication is similar):

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & B_2 \\ & B_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 & A_1B_2 + A_2B_4 \\ & A_4B_4 \end{bmatrix}$$

The cost is thus $UTUT(n) = 2UTUT(n/2) + 2TRMM(n/2)$, which yields

$$UTUT(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{12}$$

The latter is $\frac{1}{3}n^3$ with classical multiplication.

4.6 Squaring

4.6.1 $A \times A^T$

Suppose we want to compute A times its transpose, even with a diagonal in the middle. The block version is

$$\begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \times \begin{bmatrix} D_1 & \\ & D_4 \end{bmatrix} \times \begin{bmatrix} A_1^T & A_3^T \\ A_2^T & A_4^T \end{bmatrix} = \begin{bmatrix} A_1D_1A_1^T + A_2D_4A_2^T & A_1D_1A_3^T + A_2D_4A_4^T \\ A_3D_1A_1^T + A_4D_4A_2^T & A_3D_1A_3^T + A_4D_4A_4^T \end{bmatrix}$$

Since AA^T is symmetric, the lower left and upper right are just transpose of one another. The other corners (upper left and lower right) are computed via recursive calls. Thus the arithmetic cost of this special product is $AAT(n) = 4AAT(n/2)+2MM(n/2)+3ADD(n/2)+2(n/2)^2$ Ignoring the cost of the three additions and the diagonal multiplications, this yields

$$AAT(n) = \frac{2}{2^\omega - 4} MM(n). \tag{13}$$

The latter is n^3 with classical multiplication. One can note that when A is rectangular with $m \leq n$ the cost extends to

$$AAT(m; n) = \left\lceil \frac{n}{m} \right\rceil \frac{2}{2^\omega - 4} MM(m). \tag{14}$$

4.6.2 Symmetric case

When A is already symmetric, and if the diagonal is unitary, the constant factor improves. Indeed, in this case $A_2 = A_3^T$ and then one of the four recursive calls is saved. Also one of the remaining three recursive call is a call to a non symmetric AA^T. Therefore the cost is now: $SymAAT(n) = 2SymAAT(n/2) + AAT(n/2) + 2MM(n/2)$, once again ignoring n^2. This yields

$$SymAAT(n) = \frac{2(2^\omega - 3)}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{15}$$

The latter is $\frac{5}{6}n^3$ with classical multiplication.

4.6.3 Triangular case

We here view the explicit computation of $L^T DL$ for instance as a special case of upper-lower triangular matrix multiplication, but where both matrices are symmetric of one another. We also show that we can add an extra diagonal factor in the middle at a negligible cost. Consider then

$$\begin{bmatrix} L_1 & \\ L_3 & L_4 \end{bmatrix} \times \begin{bmatrix} D_1 & \\ & D_4 \end{bmatrix} \times \begin{bmatrix} L_1^T & L_3^T \\ & L_4^T \end{bmatrix} = \begin{bmatrix} L_1 D_1 L_1^T & L_1 D_1 L_3^T \\ L_3 D_1 L_1^T & L_2 D_1 L_2^T + L_4 D_4 L_4^T \end{bmatrix}$$

Thus it requires two recursive call, a call to AAT (with a diagonal in the middle) only one call to TRMM as both lower-left and upper-right corners are transpose of one another. This yields

$$LTL(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{16}$$

The latter is $\frac{1}{3}n^3$ with classical multiplication.

4.7 Symmetric factorization

For the sake of simplicity, we here consider the LU factorization of a generic rank profile symmetric $n \times n$ matrix A. We could describe how to perform this decomposition with the permutation and the possible rank deficiency in the blocks, but we here only analyze the cost of such a LDL^T factorization. The idea is that one can recursively decompose $A = \begin{bmatrix} A_1 & A_2 \\ A_2^T & A_4 \end{bmatrix} = \begin{bmatrix} L_1 & \\ G & L_2 \end{bmatrix} \times \begin{bmatrix} D_1 & \\ & D_2 \end{bmatrix} \times \begin{bmatrix} L_1^T & G^T \\ & L_2^T \end{bmatrix}$. Well, this requires a recursive call to compute L_1 and D_1 ; a TRSM to compute G such that $LDG^T = A_2$; an AAT to compute GDG^T and a recursive call to compute $L_2 D_2 L_2^T = A_4 - GDG^T$. The cost is thus $LDLT(n) = 2LDLT(n/2) + TRSM(n/2) + AAT(n/2)$, which yields

$$LDLT(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{17}$$

The latter is $\frac{1}{3}n^3$ with classical multiplication.

4.8 Matrix inverse

4.8.1 Triangular matrix inverse

To invert a triangular matrix via a block decomposition, one requires two recursive calls and two triangular matrix multiplications.

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix}^{-1} = \begin{bmatrix} A_1^{-1} & -A_1^{-1}A_2A_4^{-1} \\ & A_4^{-1} \end{bmatrix}$$

The cost is thus $INVT(n) = 2INVT(n/2) + 2TRMM(n/2)$ which yields

$$INVT(n) = \frac{2}{2^\omega - 2} TRMM(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{18}$$

The latter is $\frac{1}{3}n^3$ with classical multiplication.

4.8.2 Matrix inverse

To invert a dense matrix, one needs to compute an $LQUP$ decomposition, then to invert L and permute it with Q^{-1}. A TRSM is then required to solve $UX = Q^{-1}L^{-1}$. applying P^{-1} to X yields the inverse. The cost is then $INV(n) = LQUP(n) + INVT(n) + TRSM(n)$. This gives

$$INV(n) = \frac{3 \times 2^\omega}{(2^\omega - 4)(2^\omega - 2)} MM(n). \tag{19}$$

The latter is $INV(n) = 2n^3$ with classical multiplication.

4.8.3 Symmetric inverse

If A is symmetric, one can decompose it into a LDL^T factorization instead of the LU. Therefore, its inverse is then only one $INVT$ for both L^{-1} and L^{-T} followed by an LTL. The cost is then $SymINV(n) = LDLT(n) + INVT(n) + LTL(n)$ which yields

$$SymINV(n) = \frac{12}{(2^\omega - 2)(2^\omega - 4)} MM(n). \tag{20}$$

The latter is $SymINV(n) = n^3$ with classical multiplication.

4.8.4 Full-rank Moore-Penrose pseudo-inverse

A is a rectangular full rank $m \times n$ matrix. Lets suppose, without loss of genericity, that $m \leq n$. The Moore-Penrose inverse of A is thus $A^\dagger = A^T(AA^T)^{-1}$, see e.g. [52] and references therein.

Computing the Moore-Penrose inverse is then just a LDL^T decomposition of the symmetric matrix AA^T, followed by two rectangular system solvings: $MPINV(m;n) = AAT(m;n) + LDLT(m) + 2TRSM(m;n)$. The cost is then

$$MPINV(m;n) = \left(\left\lceil \frac{n}{m} \right\rceil \frac{6}{2^\omega - 4} + \frac{4}{(2^\omega - 2)(2^\omega - 4)} \right) MM(m) \tag{21}$$

The latter is $3m^2n + \frac{1}{3}m^3$ with classical multiplication. This correspond e.g. to the normal equations numerical resolution [29, algorithm 5.3.1].

4.8.5 Rank deficient Moore-Penrose pseudo-inverse

In this case, one needs to compute a full-rank decomposition of A. This is done by performing the LSP decomposition of A and if A is of rank r, selecting the first r columns of L (call them $L_r = \begin{bmatrix} L_1 \\ G \end{bmatrix}$) and the first r rows U (call them $U_r = [U_1|Y]$), forgetting the permutation P. We have $A = L_r U_r$ and we modify the formula [48, (7)] as follows:

$$A^\dagger = \begin{bmatrix} I \\ Y^T U_1^{-T} \end{bmatrix} \left((L_1 + L_1^{-T}G^TG)(U_1 + YY^TU_1^{-1})\right)^{-1} [I|L_1^{-T}G^T]. \quad (22)$$

We note $W = (L_1 + L_1^{-T}G^TG)(U_1 + YY^TU_1^{-1})$. We compute W by two squarings, two TRSM and a classical matrix multiplication. We perform a reversed LU decomposition on W to get $W = U_w L_w$. Now we compute $L_1^T U_w$ and $L_w U_1^T$ by upper-upper triangular multiplication and $H = (L_1^T U_w)^{-1} G^T$ and $Z = Y^T (L_w U_1^T)^{-1}$ by two TRSM. Now, $A^\dagger = \begin{bmatrix} W^{-1} & L_w^{-1}H \\ ZU_w^{-1} & ZH \end{bmatrix}$. W^{-1} is two triangular inverses and an upper lower product. ZH is a rectangular multiplication and the last two blocks are obtained by two triangular solvings.

$$\begin{aligned} MPINV_r(m;n) = LSP(m;n) + AAT(r;m-r) + AAT(r;n-r) + 3TRSM(r,m-r) \\ + 3TRSM(r,n-r) + MM(r) + LSP(r) + 2UTUT(r) + 2INVT(r) + UTLT(r) \\ + R(n-r;r;m-r) \end{aligned} \quad (23)$$

The latter is $2rmn + 2r^2m + 2r^2n + m^2n - \frac{1}{3}m^3 - \frac{4}{3}r^3$ with classical multiplication. To get an idea, numerical computations based on the Cholesky factorization of AA^T presented in [9] as faster than SVD or QR or iterative methods would require $3m^2n + 2r^2m + 3r^3$ flops.

4.8.6 Performances and comparisons with numerical routines

As for triangular system solving and matrix triangularization, we now compare performances of matrix inversion for triangular and dense matrices with numerical computation and with matrix multiplication. Our comparison with numerical computation is still based on LAPACK library with two different BLAS kernel (i.e. ATLAS and GOTO). We do not present the result of triangular matrix inversion over our P4 architecture according to the bad behavior of "dtrsm" function which is the main routine used by LAPACK for triangular matrix inversion. Our base field is the prime field of integers modulo 65521 using a `Zpz-double` representation and we use fast matrix multiplication of section 3.2.

Tables 15 and 16 illustrate the performances of our exact triangular matrix inversion regarding performances of LAPACK routine "dtrtri". Results are a

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
GOTO	tri. inv	0.10s	0.65s	1.99s	8.38s	22.00s	32.34s	45.67s	62.01s
	dtrtri	0.19s	1.06s	2.97s	11.27s	27.89s	40.13s	55.43s	74.44s
	$\frac{tri.inv}{dtrtri}$	**0.52**	**0.61**	**0.67**	**0.74**	**0.78**	**0.80**	**0.82**	**0.83**

Table 15: Timings of triangular matrix inversion on a P4, 3.4GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	tri. inv	0.20s	1.11s	3.17s	12.67s	32.05s	46.26s	64.96s	86.27s
	dtrtri	0.10s	0.75s	2.42s	10.66s	28.26s	41.65s	58.83s	79.21s
	$\frac{tri.inv}{dtrtri}$	**2.00**	**1.48**	**1.30**	**1.18**	**1.13**	**1.11**	**1.10**	**1.08**
GOTO	tri. inv	0.14s	0.84s	2.50s	10.20s	26.37s	38.61s	54.15s	73.34s
	dtrtri	0.16s	0.94s	2.72s	10.83s	27.57s	40.14s	56.16s	75.74s
	$\frac{tri.inv}{dtrtri}$	**0.87**	**0.89**	**0.91**	**0.94**	**0.95**	**0.96**	**0.96**	**0.96**

Table 16: Timings of triangular matrix inversion on Itanium2, 1.3GHz

bit surprising since they show a different behavior according to the BLAS kernel. When ATLAS BLAS are used as kernel, we see that performances of our routines tend to catch up with the numerical ones following a decreasing gap with matrix dimension. As soon as GOTO BLAS are used as kernel numerical, the performances become more efficient only when matrices are greater than 5000. As a consequence our exact computation are alway faster than numerical one but following an increasing gap. Despite this surprising behavior, one can see that exact computation of triangular matrix inverse yields about the same performances as numerical one, sometime slower sometime faster according to the BLAS kernel.

Now, Tables 17 and 18 provide the same comparisons for dense matrix inversion. For numerical computation references we use the routine "dgetri" in combination with the factorization routine "dgetrf" to yield matrix inverse. On both architecture with ATLAS BLAS kernel, exact computations become the most efficient when matrix dimension is getting larger. Numerical computation is only better than exact on the Itanium 2 architecture with GOTO BLAS kernel. In this particular application, the benefit of fast matrix multiplication is important since it allows to outperform numerical performances.

As shown in previous section, matrix inversion algorithms reduce to matrix multiplication. Figures 19 and 20 show the correlation between matrix inversion performances and matrix multiplication performances; triangular and dense case are studied.

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	inverse	0.88s	5.22s	15.00s	59.03s	152.20s	218.15s	310.94s	394.56s
	dgetrf+dgetri	0.55s	4.06s	13.85s	64.51s	176.34s	298.89s	422.42s	604.83s
	$\frac{inverse}{dgetrf+dgetri}$	**1.6**	**1.28**	**1.08**	**0.91**	**0.86**	**0.72**	**0.73**	**0.65**
GOTO	inverse	0.78s	4.66s	13.63s	54.15s	136.55s	197.26s	271.65s	362.91s
	dgetrf+dgetri	0.48s	3.52s	11.81s	62.68s	137.21s	201.81s	287.78s	477.27s
	$\frac{inverse}{dgetrf+dgetri}$	**1.62**	**1.32**	**1.15**	**0.86**	**0.99**	**0.97**	**0.94**	**0.76**

Table 17: Timings of matrix inversion on P4, 3.4GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	inverse	1.22s	7.02s	20.13s	80.91s	201.58s	289.50s	401.41s	538.93s
	dgetrf+dgetri	0.65s	4.83s	15.88s	73.17s	206.11s	308.30s	441.29s	603.26s
	$\frac{inverse}{dgetrf+dgetri}$	**1.87**	**1.45**	**1.26**	**1.10**	**0.97**	**0.93**	**0.90**	**0.89**
GOTO	inverse	1.05s	6.14s	18.10s	73.85s	180.21s	258.17s	356.79s	496.39s
	dgetrf+dgetri	0.71s	4.56s	13.96s	59.26s	155.71s	230.29s	323.15s	440.82s
	$\frac{inverse}{dgetrf+dgetri}$	**1.47**	**1.34**	**1.29**	**1.24**	**1.15**	**1.12**	**1.10**	**1.12**

Table 18: Timings of matrix inversion on Itanium2, 1.3GHz

According to section 4.8.1, the ratio of triangular matrix inversion and matrix multiplication is $4/(2^\omega - 4)(2^\omega - 2)$; which gives a theoretical ratio of $1/6$ when classic matrix multiplication is used. However this ratio increase to ≈ 0.267 when Winograd fast matrix multiplication is used (i.e. $\omega = \log_2 7$). Since our matrix multiplication routine is using fast matrix multiplication, the asymptotic behavior of this ratio should tend to the latter. However we observe in practice that our performances are beyond this ratio. This is due to the hybrid matrix multiplication which uses both Winograd and classic algorithms. So the practical ratio obtained here is really close to the theoretical one since it should asymptotically lie between 0.2674 and 0.166.

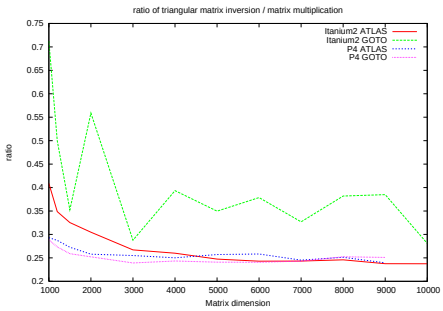

Figure 19: Comparing triangular matrix inversion with matrix multiplication

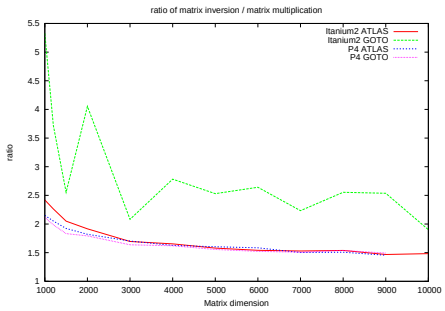

Figure 20: Comparing matrix inversion with matrix multiplication

From section 4.8.2 one can express the ratio between dense matrix inversion and matrix multiplication as respectively 1 with classic algorithm and 1.4 with Winograd algorithm. In practice we observe that dense matrix inversion ratio is just above the asymptotic behavior of Winograd based inversion. This certainly could be explained by the number of different algorithm involved in this application. In particular it involves three different reduction to matrix multiplications; which may be of a little influence on the final performances. Moreover, we do not take into account memory effect which can play a crucial role in performances as already demonstrated by ATLAS software with optimized BLAS [57]. In our test we used a naive approach which leads us to use $2n^2$ elements in memory. Decreasing this memory will certainly allow us to get better performances. In particular, it is not known yet how to perform matrix inversion in place using a reduction to matrix multiplication.

5 Conclusions

We have achieved the goal of approaching the efficiency of the numerical linear algebra library but for finite fields. We showed that exact computation can

benefit from Winograd fast matrix multiplication algorithm and then even leads to outperform the performances of the well known BLAS and LAPACK libraries.

This performances are achieved through efficient reduction to matrix multiplication where we took care of minimizing the ratio and also by reusing the numerical computation as much as possible. We also showed that from our routines one can easily implement efficient algorithms for many linear algebra problem (e.g. null-space, generalized inverse,...). Note that approximate timings for these algorithms can be derived from the timings provided with our main routines.

One can try to design block algorithms where the blocks fit in the cache of a specific machine to reach very good performances. By reusing BLAS library this has been proven to not be mandatory for matrix multiplication in [21] and we think we proved here that this is not mandatory for any dense linear algebra routine. Therefore, using recursive block algorithms, efficient numerical BLAS and fast matrix multiplication algorithms one can approach the numerical performances or even surpass them over finite fields. Moreover, long range efficiency and portability are warranted as opposed to every day tuning with at most 10% loss in specific cases (e.g. see table 8 where delayed can beat BLAS only for big primes and with a specific empirical threshold).

Besides, the exact equivalent of stability constraints for numerical computations is coefficient growth. Therefore, whenever possible, we computed and improved theoretical bounds on this growth (see bounds 3.7 and [21, Theorem3.1]). Those optimal bounds enable further uses of the BLAS routines.

Further developments include:

- A Self-adapting Software [13] would allow to provide hybrid implementations with best empirical thresholds (e.g. switch to different algorithms during the recursive course of `Trsm` and `TURBO`).
- The other case where our wrapping of BLAS is insufficient is for very small matrices where benefits of BLAS are limited and fast algorithms are not useful (see tables 7 and 8). Here also, automated tuning could produce improved versions.
- The technique of wrapping BLAS becomes useless when finite fields are larger than the corresponding bound of feasibility (e.g. $p > 2^{26}$ for matrix multiplication). At a non negligible price the Chinese remainder algorithm could be used to authorize the use of BLAS. Optimizing this scheme would be then a interesting way to provide similar results for larger finite fields.
- Finally, extending the out of core work of section 4.2 to design a parallel library is promising. Also, in the case of parallelism, our all-recursive approach enables a very efficient "sequential-first" parallelization as shown e.g. in [10] for triangular system solving.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] Daniel V. Bailey and Christof Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[4] Dario Bini and Victor Pan. *Polynomial and Matrix Computations, Volume 1: Fundamental Algorithms.* Birkhauser, Boston, 1994.

[5] Morgan Brassel, Pascal Giorgi, and Clement Pernet. LUdivine: A symbolic block LU factorisation for matrices over finite fields using blas. In *East Coast Computer Algebra Day, Clemson, South Carolina, USA*, April 2003. Poster.

[6] James R. Bunch and John E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28:231–236, 1974.

[7] Zhuliang Chen and Arne Storjohann. Effective reductions to matrix multiplication, July 2003. ACA'2003, 9th International Conference on Applications of Computer Algebra, Raleigh, North Carolina State University, USA.

[8] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[9] Pierre Courrieu. Fast computation of moore-penrose inverse matrices. *Neural Information Processing - Letters and Reviews*, 8(2):25–29, August 2005.

[10] Van-Dat Cung, Vincent Danjean, Jean-Guillaume Dumas, Thierry Gautier, Guillaume Huard, Bruno Raffin, Christophe Rapine, Jean-Louis Roch, and Denis Trystram. Adaptive and hybrid algorithms: classification and illustration on triangular system solving. In Jean-Guillaume Dumas, editor, *Proceedings of Transgressive Computing 2006, Granada, Spain*. Universidad de Granada, Spain, April 2006.

[11] David Defour. *Fonctions élémentaires : algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, September 2003.

[12] John D. Dixon. Exact solution of linear equations using p-adic expansions. *Numerische Mathematik*, 40:137–141, 1982.

[13] Jack Dongarra and Victor Eijkhout. Self-adapting numerical software and automatic tuning of heuristics. *Lecture Notes in Computer Science*, 2660:759–770, January 2003.

[14] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *Transactions on Mathematical Software*, 16(1):1–17, March 1990. `www.acm.org/pubs/toc/Abstracts/0098-3500/79170.html`.

[15] C. C. Douglas, M. Heroux, G. Slishman, and R. M. Smith. Gemmw: A portable level 3 blas winograd variant of strassen's matrix-matrix multiply algorithm. *Journal of Computational Physics*, 110:1–10, 1994.

[16] Pierre Douillet. Zech logarithms and finite fields. Technical report, Faculté des Sciences, Paris, February 2001.

[17] Jacques Dubrois and Jean-Guillauem Dumas. Efficient polynomial time algorithms computing industrial-strength primitive roots. *Information Processing letters*, 97(2):41–45, January 2006.

[18] Jean-Guillaume Dumas. *Algorithmes parallèles efficaces pour le calcul formel : algèbre linéaire creuse et extensions algébriques*. PhD thesis, Institut National Polytechnique de Grenoble, France, December 2000. `ftp://ftp.imag.fr/pub/Mediatheque.IMAG/theses/2000/-Dumas.Jean-Guillaume`.

[19] Jean-Guillaume Dumas. Efficient dot product over finite fields. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Proceedings of the seventh International Workshop on Computer Algebra in Scientific Computing, Yalta, Ukraine*, pages 139–154. Technische Universität München, Germany, July 2004.

[20] Jean-Guillaume Dumas, Thierry Gautier, Mark Giesbrecht, Pascal Giorgi, Bradford Hovinen, Erich Kaltofen, B. David Saunders, Will J. Turner, and Gilles Villard. LinBox: A generic library for exact linear algebra. In Arjeh M. Cohen, Xiao-Shan Gao, and Nobuki Takayama, editors, *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*, pages 40–50. World Scientific Pub, August 2002.

[21] Jean-Guillaume Dumas, Thierry Gautier, and Clément Pernet. Finite field linear algebra subroutines. In Teo Mora, editor, *Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, Lille, France*, pages 63–74. ACM Press, New York, July 2002.

[22] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: Finite field linear algebra package. In Gutierrez [32], pages 119–126.

[23] Jean-Guillaume Dumas, Clément Pernet, and Zhendong Wan. Efficient computation of the characteristic polynomial. In Manuel Kauers, editor,

Proceedings of the 2005 International Symposium on Symbolic and Algebraic Computation, Beijing, China, pages 140–147. ACM Press, New York, July 2005.

[24] Jean-Guillaume Dumas and Jean-Louis Roch. On parallel block algorithms for exact triangularizations. *Parallel Computing*, 28(11):1531–1548, November 2002.

[25] Jean-Guillaume Dumas, B. David Saunders, and Gilles Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, 32(1/2):71–99, July–August 2001.

[26] Joachim von zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

[27] Pascal Giorgi. From blas routines to finite field exact linear algebra solutions, July 2003. ACA'2003, 9th International Conference on Applications of Computer Algebra, Raleigh, North Carolina State University, USA.

[28] Pascal Giorgi, Claude-Pierre Jeannerod, and Gilles Villard. On the complexity of polynomial matrix computations. In Rafael Sendra, editor, *Proceedings of the 2003 International Symposium on Symbolic and Algebraic Computation, Philadelphia, Pennsylvania, USA*, pages 135–142. ACM Press, New York, August 2003.

[29] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[30] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication. Technical report, University of Texas, 2002. FLAME working note #9.

[31] F. Gustavson, A. Henriksson, I. Jonsson, and B. Kaagstroem. Recursive blocked data formats and BLAS's for dense linear algebra algorithms. *Lecture Notes in Computer Science*, 1541:195–206, 1998.

[32] Jaime Gutierrez, editor. *ISSAC'2004. Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*. ACM Press, New York, July 2004.

[33] Tom Hansen and Gary L. Mullen. Primitive polynomials over finite fields. *Mathematics of Computation*, 59(200):639–643, S47–S50, October 1992.

[34] Nicholas J. Higham. Exploiting fast matrix multiplication within the level 3 BLAS. *Trans. on Mathematical Software*, 16(4):352–368, December 1990.

[35] Xiaohan Huang and Victor Y. Pan. Fast rectangular matrix multiplications and improving parallel matrix computations. In ACM, editor, *PASCO '97. Proceedings of the second international symposium on parallel symbolic*

computation, July 20–22, 1997, Maui, HI, pages 11–23, New York, NY 10036, USA, 1997. ACM Press.

[36] Klaus Huber. Some comments on Zech's logarithm. *IEEE Transactions on Information Theory*, IT–36:946–950, July 1990.

[37] Klaus Huber. Solving equations in finite fields and some results concerning the structure of GF(p^m). *IEEE Transactions on Information Theory*, IT–38:1154–1162, May 1992.

[38] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press. `www.supercomp.org/sc96/proceedings/SC96PROC/JACOBSON/-INDEX.HTM`.

[39] Oscar H. Ibarra, Shlomo Moran, and Roger Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, March 1982.

[40] Erich Kaltofen, Mukkai S. Krishnamoorthy, and B. David Saunders. Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, 136:189–208, 1990.

[41] Erich Kaltofen and Gilles Villard. On the complexity of computing determinants. *Computational Complexity*, 13(3-4):91–130, 2005.

[42] Igor Kaporin. The aggregation and cancellation techniques as a practical tool for faster matrix multiplication. *Theoretical Computer Science*, 315(2-3):469–510, 2004.

[43] Yasuhiro Kawame and Hirokazu Murao. MBLAS: Modular basic linear algebra subprograms, design and speedup techniques. In Gutierrez [32]. Poster.

[44] Julian Laderman, Victor Pan, and Xuan-He Sha. On practical algorithms for accelerated matrix multiplication. *Linear Algebra Appl.*, 162–164:557–588, 1992.

[45] Rudolf Lidl and Harald Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, revised edition, 1994.

[46] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[47] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over $gf(2)$. In Louis C. Guillou and Jean-Jacques Quisquater, editors, *Proceedings of the 1995 International Conference on the Theory and Application of Cryptographic Techniques, Saint-Malo, France*, volume 921 of *Lecture Notes in Computer Science*, pages 106–120, May 1995.

[48] Ben Noble. A method for computing the generalized inverse of a matrix. *SIAM Journal on Numerical Analysis*, 3(4):582–584, December 1966.

[49] Andrew M. Odlyzko. Discrete logarithms: The past and the future. *Designs, Codes, and Cryptography*, 19:129–145, 2000.

[50] Clément Pernet. Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical report, Laboratoire Informatique et Distribution, July 2001. `www-id.imag.fr/Apache/RR/RR011122FFLAS.ps.gz`.

[51] Clément Pernet. Calcul du polynôme caractéristique sur des corps finis. Master's thesis, University of Delaware, June 2003.

[52] B. D. Saunders. Black box methods for least squares problems. In Bernard Mourrain, editor, *ISSAC 2001: July 22–25, 2001, University of Western Ontario, London, Ontario, Canada: proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, pages 297–302, 2001.

[53] B. David Saunders. Personal communication, 2001.

[54] Victor Shoup. NTL 5.3: A library for doing number theory, 2002. `www.shoup.net/ntl`.

[55] Arne Storjohann. The shifted number system for fast linear algebra on integer matrices. *Journal of Complexity*, 21(4):609–650, 2005.

[56] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[57] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, January 2001. `www.elsevier.nl/gej-ng/10/35/21/47/-25/23/article.pdf`.

[58] Hans Zassenhaus. A remark on the Hensel factorization method. *Mathematics of Computation*, 32(141):287–292, January 1978.

[59] Paul Zimmermann, 2002. Personal communication.

A Proof of theorem 3.1

To bound the intermediate values in the computation of l recursive levels of Winograd's algorithm, we will show that the worst case occurs in the computation of P_6. We will first consider the case $K = 2^l q$ and then generalize the result for every K. To end the proof we will provide an instance of a computation for which the bound is attained.

A.1 Some properties on the series of the type $2u - v$

Consider the series defined recursively by:

$$\begin{cases} u_{l+1} = 2u_l - v_l \\ v_{l+1} = 2v_l - u_l \\ u_0 \leq 0 \\ v_0 \geq 0 \end{cases}$$

Since

$$\begin{cases} u_{l+1} + v_{l+1} = u_l + v_l = \cdots = u_0 + v_0 \\ v_{l+1} - u_{l+1} = 3(v_l - u_l) = \cdots = 3^{l+1}(v_0 - u_0) \end{cases}$$

It comes

$$\begin{cases} u_l = u_0 \frac{(1+3^l)}{2} + v_0 \frac{(1-3^l)}{2} \\ v_l = v_0 \frac{(1+3^l)}{2} + u_0 \frac{(1-3^l)}{2} \end{cases}$$

Thus, the following properties hold:

$$u_l \leq 0 \text{ and } v_l \geq 0 \tag{24}$$

$$u_l \text{ is decreasing and } v_l \text{ is increasing} \tag{25}$$

$$v_l > -u_l \text{ if } v_0 > -u_0 \tag{26}$$

Now define v^A and v^B, two series of the type v by setting $u_0^A = m_A$, $v_0^A = M_A$, $u_0^B = m_B$ and $v_0^B = M_B$.

Let us also define $t_j = \frac{1+3^j}{2}$ and $s_j = \frac{1-3^j}{2}$. Thus $t_j + s_j = 1$ and $t_j - s_j = 3^j$.

The following property holds:

$$(2M_A - m_A)t_j + (2m_A - M_A)s_j = M_A t_{j+1} + m_A s_{j+1} = v_{j+1}^A \tag{27}$$

A.2 Notations

Let

$$b_l = \left(\frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A\right)\left(\frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B\right)\left\lfloor\frac{K}{2^l}\right\rfloor.$$

The series $(b_l)_{l>0}$ is increasing since (25).

Remark that the result of the computation is independent of the algorithm and is always bounded by $K\max(|m_A|,|M_A|)\max(|m_B|,|M_B|)+\beta\max(|m_C|,|M_C|) \leq$

$(K+1)M_AM_B$. Now this value is always smaller than b_1 for $k \geq 1$ and also smaller than b_l $\forall l \geq 1$. Therefore, the coefficients of the blocks U_1, U_5, U_6, U_7 always satisfy the bound.

Now if the nine following intermediate computations are bounded by b_l, we will be done.

$$
\begin{aligned}
P_1 &= A_{11} \times B_{11} \\
P_2 &= A_{12} \times B_{21} + \beta C_{11} \\
P_3 &= (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22} \\
P_4 &= A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) + \beta(C_{22} - C_{12} - C_{21}) \\
P_5 &= (A_{21} + A_{22}) \times (B_{12} - B_{11}) + \beta C_{12} \\
P_6 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{22} - B_{12}) + \beta(C_{22} - C_{12}) \\
U_2 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} \\
U_3 &= A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} + \beta(C_{22} - C_{12}) \\
U_4 &= (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22}) + \beta C_{12}
\end{aligned}
$$

We will prove that the largest intermediate value always occurs in the computation of P_6. Consider l recursive levels indexed by j: $j = l$ is the first splitting of the matrices into four blocks and $j = 0$ corresponds to the last level where the product is done by a classic matrix multiplication algorithm. The recursive algorithm can be seen as a back and forth process: the splitting is done from $j = l$ to $j = 0$ and then the multiplications are done from $j = 0$ to $j = l$.

We also define the following notations:

- $M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$ is an upper bound on the intermediate computations of $X = A \times B + \beta C$ with j recursive levels and $m_A \leq a_{i,j} \leq M_A$, $m_B \leq b_{i,j} \leq M_B$ and $m_C \leq c_{i,j} \leq M_C$. k is the common dimension of A and B
- $M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max_X M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$.
- $M(X)\frac{k}{2^{j+1}}$ for $M^{j+1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(X)$.

The following formulas correspond to the seven recursive calls:

$$M^{j+1,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max \begin{pmatrix} M(P_1) = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,0,0} \\ M(P_2) = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,m_C,M_C} \\ M(P_3) = M^{j,\frac{k}{2}}_{2m_A-2M_A,2M_A-2m_A,m_B,M_B,0,0} \\ M(P_4) = M^{j,\frac{k}{2}}_{m_A,M_A,2m_B-2M_B,2M_B-2m_B,m_C-2M_C,M_C-2m_C} \\ M(P_5) = M^{j,\frac{k}{2}}_{2m_A,2M_A,m_B-M_B,M_B-m_B,m_C,M_C} \\ M(P_6) = M^{j,\frac{k}{2}}_{2m_A-M_A,2M_A-m_A,2m_B-M_B,2M_B-m_B,0,0} \\ M(P_7) = M^{j,\frac{k}{2}}_{m_A-M_A,M_A-m_A,m_B-M_B,M_B-m_B,m_C-M_C,M_C-m_C} \end{pmatrix} \tag{28}$$

Moreover, the classic algorithm is used for $j = 0$:

$$M^{0,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max \begin{pmatrix} M_A M_B k + \beta M_C \\ -m_A M_B k - \beta m_C \\ -M_A m_B k - \beta m_C \end{pmatrix} \tag{29}$$

A.3 Some invariants

Lemma A.1. *The following invariants hold in every recursive call:*

1. $0 \leq -m_A \leq M_A,\ 0 \leq -m_B \leq M_B,\ 0 \leq -m_C \leq M_C$
2. $m_C \geq m_B$ *and* $M_C \leq M_B$
3. $M_C - m_C \leq M_B - m_B$

Proof. From equation (28), one gets invariants (1) and (2). Then invariant (3) is a consequence of (1) and (2). □

A.4 Induction for $K = 2^l q$

Let IH_j be the following induction hypothesis:

If the invariants of section A.3 are satisfied then

$$M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = [v^A_j][v^B_j]\frac{k}{2^j}$$

Suppose that the previous invariants are satisfied and that IH_j is true. We will prove that the maximum of (28) is reached during the computation of P_6 to show that IH_{j+1} is satisfied.

The conditions on m_A, M_A, m_B are M_B satisfied for every recursive call. We can therefore apply IH_j to every product $X \in \{P_1, P_2, P_3, P_4, P_5, P_6\}$ in order to compare $M(X)$ with $M(P_6)$.

- For $P_1 = A_{11} \times B_{11}$:

$$\begin{aligned} M(P_6) - M(P_1) &= [(2M_A - m_A)t_j + (2m_A - M_A)s_j] \times \\ &\quad [(2M_B - m_B)t_j + (2m_B - M_B)s_j] - v_j^{A_{11}} v_j^{B_{11}} \\ &= v_{j+1}^A v_{j+1}^B - v_j^{A_{11}} v_j^{B_{11}} \\ &\geq v_{j+1}^A v_{j+1}^B - v_j^A v_j^B \end{aligned}$$

  And since v^A and v^B are increasing and positive, we have $M(P_6) \geq M(P_1)$.

- For $P_2 = A_{12} \times B_{21} + \beta C_{11}$: with the same argument $M(P_6) \geq M(P_2)$.

- For $P_3 = (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22}$:

$$\begin{aligned} M(P_6) - M(P_3) &= v_{j+1}^A v_{j+1}^B - v_j^{A_{11}+A_{12}-A_{21}-A_{22}} v_j^{B_{22}} \\ &= v_{j+1}^A v_{j+1}^B - [(2M_A - 2m_A)t_j + (2m_A - 2M_A)s_j] v_j^B \\ &= v_{j+1}^A v_{j+1}^B - (v_{j+1}^A - m_A t_j - M_A s_j) v_j^B \quad (27) \\ &= v_{j+1}^A [v_{j+1}^B - v_j^B] - u_j^A v_j^B \\ &\geq v_{j+1}^A [v_{j+1}^B - v_j^B] - v_{j+1}^A v_j^B \quad (26) \\ &\geq v_{j+1}^A [v_{j+1}^B - 2v_j^B] \\ &\geq v_{j+1}^A 3^j [M_B - m_B] \geq 0 \end{aligned}$$

- For $P_4 = A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) + \beta(C_{22} - C_{12} - C_{21})$: with the same argument,

$$M(P_6) - M(P_4) = v_{j+1}^A v_{j+1}^B - v_j^{A_{22}} v_j^{B_{22}+B_{11}-B_{12}-B_{21}} \geq 0$$

- For $P_5 = (A_{21} + A_{22}) \times (B_{12} - B_{11}) + \beta C_{12}$:

$$\begin{aligned} M(P_6) - M(P_5) &= v_{j+1}^A v_{j+1}^B - v_j^{A_{21}+A_{22}} v_j^{B_{12}-B_{11}} \\ &= v_{j+1}^A v_{j+1}^B - 2v_j^A \left[v_j^B - u_j^B\right] \\ &= \left[2v_j^A - u_j^A\right] v_{j+1}^B - v_j^A \left[v_{j+1}^B - u_j^B\right] \\ &= v_j^A v_{j+1}^B - u_j^A v_{j+1}^B + v_j^A u_j^B \\ &= v_j^A \left[v_{j+1}^B + u_j^B\right] - u_j^A v_{j+1}^B \\ &= v_j^A \left[2v_j^B\right] - u_j^A v_{j+1}^B \end{aligned}$$

  and since $u_j^A \leq 0 \leq v_j^A, v_j^B, v_{j+1}^B$ it comes $M(P_6) - M(P_5) \geq 0$.

- For $P_7 = (A_{11} - A_{21}) \times (B_{22} - B_{12}) + \beta(C_{22} - C_{12})$: using P_5,

$$\begin{aligned} M(P_5) - M(P_7) &= v_j^{A_{21}+A_{22}} v_j^{B_{12}-B_{11}} - v_j^{A_{11}-A_{21}} v_j^{B_{22}-B_{12}} \\ &= [2M_A t_j + 2m_A s_j - (M_A - m_A)t_j - (m_A - M_A)s_j] \times \\ &\quad [(M_B - m_B)t_j + (m_B - M_B)s_j] \\ &= [(M_A + m_A)(t_j + s_j)] \left[(M_B - m_B)(t_j - s_j)\right] \\ &\geq 0 \end{aligned}$$

The coefficients of the blocks U_1, U_5, U_6 and U_7 are bounded by $kM_AM_B + \beta M_C$ and are therefore smaller than the ones in P_6.

Lastly, we must control the size of the coefficients in $U_2 = P_1 + P_6$, $U_3 = U_2 + P_7$ and $U_4 = U_2 + P_7$.

- For $U_2 = (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11}$:

$$\forall x \in U_2, |x| \leq \max \begin{pmatrix} (2M_A - m_A)(M_B - m_B) + 2M_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2m_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2M_Am_B \end{pmatrix} k/2^j \tag{30}$$

Now $2M_A - m_A - (-2m_A + M_A) = M_A + m_A \geq 0$ and $0 \leq -m_A \leq M_A$, so the 30 simplifies into $\forall x \in U_2, |x| \leq (2M_A - m_A)(M_B - m_B) + 2M_AM_B$.

$$\begin{aligned} M(P_6) - M(U_2) &\geq (2M_A - m_A)(2M_B - m_B) - (2M_A - m_A)(M_B - m_B) \\ &\quad -2M_AM_B \\ &= (2M_A - m_A)(M_B) - 2M_AM_B \\ &= -m_AM_B \geq 0 \end{aligned}$$

- For $U_3 = A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} + \beta(C_{22} - C_{12})$: with the same argument

$$\forall x \in U_3, |x| \leq \max \begin{pmatrix} (M_A(M_B - m_B) + 2M_AM_B)k/2^j + |\beta|(M_C - m_C) \\ (M_A(M_B - m_B) - 2m_AM_B)k/2^j + |\beta|(M_C - m_C) \\ (M_A(M_B - m_B) - 2M_Am_B)k/2^j + |\beta|(M_C - m_C) \end{pmatrix} k/2^j$$

The max is always equal to its first argument, and since $k/2^j \geq 1$, $\beta \leq M_A - m_A$ and $M_C - m_C \leq M_B - m_B$, we have:

$$\begin{aligned} |x| &\leq (M_A(M_B - m_B) + 2M_AM_B)k/2^j + \beta(M_C - m_C) \\ &\leq (2M_A - m_A)(M_B - m_B) + 2M_AM_B)k/2^j \\ &\leq M(U2) \leq M(P6) \end{aligned}$$

Lastly

$$M(U_3) \leq M(P_6)$$

- For $U_4 = (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22}) + \beta C_{12}$: with the same argument as for U_3,

$$\forall x \in U_4, |x| \leq (M_A(M_B - m_B) + 2M_AM_B)k/2^j + |\beta|M_C$$

Since $M_C \leq M_B - m_B$, $-m_A \leq M_A$ and $-m_B \leq M_B$, we have

$$M(U_4) \leq M(U_3) \leq M(P_6).$$

Finally $M^{j+1,k}_{m_A,M_A,m_B,M_B} = M(P_6)\frac{k}{2^{j+1}} = v^A_{j+1}v^B_{j+1}\frac{k}{2^{j+1}}$, and IH_{j+1} is satisfied.

For the initialization of the induction ($j = 1$), the products of the blocks are done by the classical algorithm. From (28) and (29), one gets:

$$\begin{aligned}
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_1) &= M_A M_B k/2 \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_2) &= M_A M_B k/2 + |\beta| M_C \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_3) &= 2(M_A - m_A) M_B k/2 \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_4) &= 2M_A(M_B - m_B)k/2 + |\beta|(2M_C - m_C) \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_5) &= 2M_A(M_B - m_B)k/2 + |\beta| M_C \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_6) &= (2M_A - m_A)(2M_B - m_B)k/2 \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_7) &= (M_A - m_A)(M_B - m_B)k/2 + |\beta|(M_C - m_C) \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_2) &= (2M_A - m_A)(M_B - m_B)k/2 + 2M_A M_B k/2 \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_3) &= M_A(M_B - m_B)k/2 + 2M_A M_B k/2 + |\beta|(M_C - m_C) \\
M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(U_4) &= 2M_A M_B k/2 + M_A(M_B - m_B)k/2 + |\beta| M_C
\end{aligned}$$

Again, we will prove that $M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_6)$ reaches the highest value, using invariants of section A.3, and the fact that $|\beta| \leq M_A, M_B$ and $k \geq 2$.

It is straightforward for P_1 and P_2.

- For P_3:

$$\begin{aligned}
M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_3) &= ((2M_A - m_A)(2M_B - m_B) - 2(M_A - m_A)M_B)k/2 \\
&= (2M_A M_B k - (2M_A - m_A)m_B)k/2 \geq 0
\end{aligned}$$

- For P_4: Since $-|\beta|(2M_C - m_C) \geq -M_A(2M_B - m_B)$, we have

$$\begin{aligned}
M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_4) &= ((2M_A - m_A)(2M_B - m_B) - 2M_A(M_B - m_B))k/2 \\
&\quad -|\beta|(M_C - 2m_C) \\
&\geq (M_A - m_A)(2M_B - m_B) - 2M_A(M_B - m_B) \\
&= m_A(m_B - 2M_B) \geq 0
\end{aligned}$$

- For P_5: $M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_5) \leq M^{1,k}_{m_A,M_A,m_B,M_B,m_C,M_C}(P_4)$
- For P_7:

$$\begin{aligned}
M^{1,k}_{m_A,\ldots}(P_6) - M^{1,k}_{m_A,\ldots}(P_7) &= ((2M_A - m_A)(2M_B - m_B) \\
&\quad -(M_A - m_A)(M_B - m_B)k/2 - |\beta|(M_C - m_C) \\
&\geq M_A(2M_B - m_B) + (M_A - m_A)M_B - M_A(M_B - m_B) \\
&\geq (2M_A - m_A)M_B \geq 0
\end{aligned}$$

- For U_2, U_3, U_4: using the same argument as for the case of arbitrary j.

IH_1 is then satisfied.

A.5 Case of an arbitrary k

Let l be such that $2^l d \leq k < 2^l(d+1)$ ($d = \left\lfloor \frac{k}{2^l} \right\rfloor$). A *dynamic peeling* technique [38] is used to deal with odd dimensions: at each recursive level, the largest blocks with even dimensions at the top left hand corner of the input matrices are are multiplied using Winograd's algorithm. Then an optional rank 1 update is applied, with the odd dimensions.

These updates are using matrix-vector products, dot products and tensor products. Every intermediate result during these computations are therefore bounded in absolute value by $kM_AM_b + |\beta|M_C \leq (k+1)M_AM_B$

We show now that this bound is always under the one of Winograd's algorithm.

$$\forall l \geq 1 \; 2^l(d+1)M_AM_B \leq v_l^A v_l^B \left\lfloor \frac{k}{2^l} \right\rfloor$$

(since $(k+1)M_AM_B \leq 2^l(d+1)M_AM_B$).

- For $l = 1$, the inequation is satisfied: $2M_AM_B(d+1) \leq (2M_A - m_A)(2M_B - m_B)d$ (since $d \geq 1$)
- Let us suppose that it is satisfied for $l \geq 1$ and prove it for $l+1$:

$$\begin{aligned} v_{l+1}^A v_{l+1}^B \left\lfloor \frac{k}{2^{l+1}} \right\rfloor &= [(2M_A - m_A)t_l + (2m_A - M_A)s_l] \\ &\times [(2M_B - m_B)t_l + (2m_B - M_B)s_l]d \\ &\geq 2[M_At_l + m_As_l][M_Bt_l + m_Bs_l]2d \\ &\geq v_l^A v_l^B \left\lfloor \frac{k}{2^l} \right\rfloor \\ &\geq 2(2^l M_AM_B(2d+1)) \\ &\geq 2^{l+1}M_AM_B(d+1)) \end{aligned}$$

By induction, the bound of section A.4 is valid for any k.

A.6 Optimality of the bound

We simply build a sequence of square matrices A_l and B_l of order 2^l for which l recursive calls to Winograd's algorithm will involve intermediate results equals to the bound.

Let $(A_l)_{l \in \mathbb{N}^*}$ and $(B_l)_{l \in \mathbb{N}^*}$ be recursively defined as follows:

$$\begin{cases} A_1 = \begin{bmatrix} m_A & 0 \\ M_A & M_A \end{bmatrix}, \quad B_1 = \begin{bmatrix} M_B & m_B \\ 0 & M_B \end{bmatrix} \\ A_{l+1} = \begin{bmatrix} \overline{A_l} & 0 \\ A_l & A_l \end{bmatrix}, \quad B_{l+1} = \begin{bmatrix} B_l & \overline{B_l} \\ 0 & B_l \end{bmatrix} \end{cases}$$

where $\overline{A_{i,j}} = M_A + m_A - A_{i,j}$ et $\overline{B_{i,j}} = M_B + m_B - B_{i,j}$.

Since at each recursive level, the computation of $P6 = (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12})$ involves the largest possible intermediate values, let us define:

$$S(A_l) = (A_l)_{2,1} + (A_l)_{2,2} - (A_l)_{1,1} = 2A_{l-1} - \overline{A_{l-1}} = 3A_{l-1} - J_{l-1}$$

where J_k is the square matrix of order 2^k whose coefficients are all equals to $M_A + m_A$.

Moreover $S(J_k) = J_{k-1}$. Thus, applying P_6 l times recursively, since S is linear:

$$S(S(\dots(S(A_l)))) = S^l(A_l) = 3^{l-1}S(A_1) - \left(\sum_{k=0}^{l-2} 3^k\right) J_1$$

Then $S(A_1) = 2M_A - m_A$ and $J_1 = M_A + m_A$ imply:

$$S^l(A_l) = 3^{l-1}(2M_A - m_A) - \frac{3^{l-1}-1}{3-1}(M_A + m_A) = \frac{1+3^l}{2}M_A + \frac{1-3^l}{2}m_A.$$

The same holds for B_l:

$$S^l(B_l) = \frac{1+3^l}{2}M_B + \frac{1-3^l}{2}m_B$$

The order of A_l and B_l is $k = 2^l$, so $\left\lfloor \frac{k}{2^l} \right\rfloor = 1$

Therefore, the computation of $A_l \times B_l$ with l recursive levels of Winograd's algorithm involves intermediate values equals to $v_l^{A_l} v_l^{B_l} \left\lfloor \frac{k}{2^l} \right\rfloor$. This proves the optimality of the bound.

Note that this bound is unchanged for computations of the type $A \times B + \beta C$.

B Sizes of non prime Galois fields for which matrix multiplication over numerical BLAS is possible

Here are some of the galois fields implementable with our q-adic representation. In the tables, n_{max} is the biggest matrix size for which this field is usable without loss of precision and $q_b(n_{max})$ is the best prime power to use with this maximal size.

These two tables shows that on 32 bits architectures the q-adic approach is interesting mainly on quadratic fields ($\texttt{GF}(p^2)$), whereas, on 64 bits architectures our approach speeds-up cubic fields ($\texttt{GF}(p^3)$) also.

GF	n_{max}	$q_b(n_{max})$
2^2	104028	208057
2^3	516	1549
2^4	45	181
2^5	11	59
2^6	4	27
2^7	2	16
2^8	1	11
3^2	26007	208057
3^3	129	1549
3^4	11	181
3^5	2	59
3^6	1	27
5^2	6501	208057
5^3	32	1549
5^4	2	181
7^2	2889	208057
7^3	14	1549
7^4	1	181

GF	n_{max}	$q_b(n_{max})$
11^2	1040	208057
11^3	5	1549
13^2	722	208057
13^3	3	1549
17^2	406	208057
17^3	2	1549
19^2	321	208057
19^3	1	1549
23^2	214	208057
23^3	1	1549
29^2	132	208057
31^2	115	208057
47^2	49	208057
53^2	38	208057
101^2	10	208057
139^2	5	208057
227^2	2	208057
317^2	1	208057

Table 19: Highest block order for some non-prime Galois fields, with a 53 bits mantissa

GF	n_{max}	$q_b(n_{max})$	GF	n_{max}	$q_b(n_{max})$
2^2	1321119	2642239	13^2	9174	2642239
2^3	2376	7129	13^3	16	7129
2^4	140	563	17^2	5160	2642239
2^5	27	137	17^3	9	7129
2^6	8	53	19^2	4077	2642239
2^7	4	29	19^3	7	7129
2^8	2	19	23^2	2729	2642239
2^9	1	13	23^3	4	7129
3^2	330279	2642239	29^2	1685	2642239
3^3	594	7129	29^3	3	7129
3^4	35	563	31^2	1467	2642239
3^5	6	137	31^3	2	7129
3^6	2	53	37^2	1019	2642239
3^7	1	29	37^3	1	7129
5^2	82569	2642239	41^2	825	2642239
5^3	148	7129	41^3	1	7129
5^4	8	563	47^2	624	2642239
5^5	1	137	47^3	1	7129
7^2	36697	2642239	53^2	488	2642239
7^3	66	7129	101^2	132	2642239
7^4	3	563	139^2	69	2642239
11^2	13211	2642239	227^2	25	2642239
11^3	23	7129	317^2	13	2642239
11^4	1	563	1129^2	1	2642239

Table 20: Highest block order for some non-prime Galois fields, with a 64 bits mantissa