HAL
open science

Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK packages

Jean-Guillaume Dumas, Pascal Giorgi, Clément Pernet

To cite this version:

Jean-Guillaume Dumas, Pascal Giorgi, Clément Pernet. Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK packages. ACM Transactions on Mathematical Software, 2008, 35 (3), pp.19:1-42. 10.1145/1391989.1391992 . hal-00018223v4

HAL Id: hal-00018223

https://hal.science/hal-00018223v4

Submitted on 14 Jan 2009

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

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

Author manuscript, published in "ACM Transactions on Mathematical Software 35, 3 (2009) article 19"

hal-00018223, version 4 - 14 Jan 2009

Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK packages*

Jean-Guillaume Dumas†
Université de Grenoble

Pascal Giorgi‡
Université de Montpellier

Clément Pernet§
Université de Grenoble

January 14, 2009

Abstract

In the past two decades, some major efforts have been made to reduce exact (e.g. integer, rational, polynomial) 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 known that modular techniques such as the Chinese remainder algorithm or the p-adic lifting allow very good practical performance, especially when word size arithmetic are used. Therefore, finite field arithmetic becomes an important core for efficient exact linear algebra libraries. In this paper, we study high performance implementations of basic linear algebra routines over word size prime fields: specially the matrix multiplication; our goal being to provide an exact alternate to the numerical BLAS library. We show that this is made possible by a careful combination of numerical computations and asymptotically faster algorithms. Our kernel has several symbolic linear algebra applications enabled by diverse matrix multiplication reductions: symbolic triangularization, system solving, determinant and matrix inverse implementations are thus studied.

Keywords: Word size prime fields; BLAS level 1-2-3; Linear Algebra Package; Winograd's symbolic Matrix Multiplication; Matrix Factorization; Exact Determinant; Exact Inverse.

*This material is based on work supported in part by the Institut de Mathématiques Appliquées de Grenoble, project IMAG-AHA. This work was mostly done while the second author was a postdoctoral fellow of the Symbolic Computation Group, D.R. Cheriton School of Computer Science, University of Waterloo, Canada.

†Laboratoire Jean Kuntzmann, umr CNRS 5224, 51, rue des Mathématiques BP 53 IMAG-LMC, F38041 Grenoble, France; `Jean-Guillaume.Dumas@imag.fr`

‡Laboratoire d'Informatique de Robotique et de Microlectronique de Montpellier, umr CNRS 5506; `Pascal.Giorgi@lirmm.fr`

§MOAIS (INRIA Rhône-Alpes / CNRS LIG Laboratoire d'Informatique de Grenoble); `Clement.Pernet@imag.fr`

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Finite field arithmetic 5
2.1.1 Implementations 5
2.1.2 Ring homomoprphism and delayed reduction 6
2.2 Recursion materials for arithmetical complexity 7

3 Matrix multiplication **9**
3.1 Cache tuning using BLAS 9
3.2 Winograd fast algorithm 10
3.2.1 A Cascade structure 11
3.2.2 Schedule of the algorithm 12
3.2.3 Control of the overflow 12
3.3 Timings and comparison with numerical routines 13

4 Triangular system solving with matrix right/left hand side **14**
4.1 The block recursive algorithm 15
4.2 Delaying reductions globally 16
4.3 Delaying reductions in the update phase only 19
4.4 Experiments 21

5 Finite Field Matrix Factorizations **23**
5.1 Triangularizations 24
5.2 Performance and comparison with numerical routines 25
5.3 Comparison with the multiplication 27

6 Applications **27**
6.1 Nullspace basis 28
6.2 Triangular multiplications 28
6.2.1 Triangular matrix multiplication 28
6.2.2 Upper-lower Triangular matrix multiplication 28
6.2.3 Upper-Upper Triangular matrix multiplication 29
6.3 Squaring 29
6.3.1 $A \times A^T$ 29
6.3.2 Symmetric case 30
6.3.3 Triangular case 30
6.4 Symmetric factorization 30
6.5 Matrix inverse 31
6.5.1 Triangular matrix inverse 31
6.5.2 Matrix inverse 31
6.5.3 Symmetric inverse 31
6.5.4 Full-rank Moore-Penrose pseudo-inverse 31
6.5.5 Rank deficient Moore-Penrose pseudo-inverse 32
6.5.6 Performances and comparisons with numerical routines 32

7 Conclusions **35**

hal-00018223, version 4 - 14 Jan 2009

A Proof of theorem 3.1 **36**
A.1 Some properties on the series of the type $2u - v$ 36
A.2 Notations 37
A.3 Some invariants 38
A.4 Induction for $K = 2^l q$ 39
A.5 Case of an arbitrary k 42
A.6 Optimality of the bound 43

References **44**

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 [47], integer system solving [9, 44], integer matrix normal forms [23] or integer determinant [34]. Finite fields are of intrinsic use in polynomial linear algebra [26] but also in cryptology (e.g. large integer factorization [38], discrete logarithm computations [40]) 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 prime 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 (e.g. ATLAS [46]) and to make some conversions in order to perform an *exact* matrix multiplication (i.e. *without any loss of precision*). The efficiency will be reached by performing as few conversions as possible. Several alternatives to this approach exist: one would be to implement a core linear algebra with integer arithmetic. Unfortunately, new architectures focus on numerical arithmetic and therefore by using integer arithmetic we would lose a factor of 2 or 4 due to the SIMD (single instruction, multiple data) SSE speed-up of the numerical routines. Note that SSE4 with some integer support is announced for 2008 and might then change some of this point of view. Anyway, another feature of our approach is to rely on a large community of effort for the numerical handling of linear algebra routines. We want to show in this paper that no real gain could be obtained by trying to mimic their effort over just using it.

hal-00018223, version 4 - 14 Jan 2009

hal-00018223, version 4 - 14 Jan 2009

Then, building on this fast numerical blocks, we can use fast matrix multiplication algorithms, such as Strassen's or Winograd's variant [24, §12]. There, we use exact computation on a higher level and therefore do not suffer from instability problems [30].

Many algorithms have been designed to use matrix multiplication in order to be able to prove an optimal theoretical complexity. In practice those exact algorithms are only seldom used. This is the case, for example, 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 factorization of matrices of any shape and any rank.

Some of the ideas from preliminary versions of this paper [17], 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 [18] in C++ and within Maple by A. Storjohann[6]. Effective reduction for minimal and characteristic polynomial were proposed in [20] and A. Steel has reported on similar efforts within his implementation of some Magma routines.

In this paper, 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 [33]. Then we adapt this scheme to rank, determinant, inverse (classical or Moore-Penrose), nullspace computations, 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 efficiency that we achieve in practice. This will enable us to give a measure of the effectiveness of our reductions (see especially section 6).

Now, we provide a full C++ package available directly [13] or through the exact linear algebra library LinBox[1] [16]. Extending the work undertaken by the authors et al.[41, 17, 4, 25, 14, 18, 20], this paper focuses 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 introduces some material for the evaluation of arithmetical costs of recursive algorithms; we also motivate our choice to represent elements of a finite field; Then section 3 presents efficient ways to implement matrix multiplication over *generic* prime fields, including a study of fast matrix multiplication. Section 4 deals with the matrix multipli-

[1] `www.linalg.org`

cation based simultaneous resolution of n triangular systems. Laslty, section 5 presents implementations of several matrix factorizations and their applications with a study of complexity and of efficiency in practice.

2 Preliminaries

2.1 Finite field arithmetic

The first task, to implement exact linear algebra routines, is to develop the underlying arithmetic. Indeed, any finite field, except $GF(2)$, do not map directly to the arithmetical units of nowadays processors and a software emulation is therefore mandatory. This has been well studied in literature, and we refer to [14] and references therein for a survey on this topic. Here, we recall the different ways of implementing such arithmetic and we will motivate our choice of a particular one for efficient linear algebra routines.

2.1.1 Implementations

Representation of finite fields elements plays a crucial role in the efficiency of arithmetic operations. From now on, we will count arithmetic operations in terms of field operations, that is we will count addition, subtraction, multiplication and division in the arithmetic complexity results.

A usual way to implement prime fields arithmetic is to map the elements of the field to integers modulo a prime number, defined by its characteristic. From now on, we will focus on prime fields with characteristic no greater than a word size (e.g. 32 bits). In this basic case, various representations and arithmetics can be used:

- **Classical representation with integer divisions**.
  Integers between 0 and $p-1$ or between $(1-p)/2$ and $(p-1)/2$ are used; additive group operations are done with machine integers operations followed by a test and a correction; multiplication is followed by machine remaindering while division is performed via the extended gcd algorithm.

- **Montgomery representation**.
  This representation, proposed in [37], allows to avoid costly machine remaindering within the multiplication. A shifted representation is used and remaindering is replaced by multiplications. Note that others operations, except the division, stay identical.

- **Floating point inverse**.
  Another idea to reduce remaindering cost in multiplication is to precompute the inverse of the characteristic p within a floating point number. Therefore, only two floating point multiplications and some rounding are necessary. However, floating point rounding may induce a ± 1 error and then an adjustment is required, as implemented in Shoup's NTL library [43].

hal-00018223, version 4 - 14 Jan 2009

- **Discrete logarithm (also called Zech logarithm)**.
  Here, elements are seen as a power of a generator of the multiplicative group, namely a primitive element. As a consequence, multiplicative group operations can be performed only by addition or subtraction modulo $p-1$. Nevertheless, this representation makes the addition/subtraction more complicated in the field. In particular, these operations need some table lookup; see [14, §2.4].

Extension fields, denoted $GF(p^k)$, are usually implemented via polynomials over the prime field $\mathbb{Z}/\mathrm{p}\mathbb{Z}$ modulo an irreducible polynomial of degree k. Thus, operations in the extension reduce to polynomial arithmetic. An alternative is to tabulate entries and use the Zech logarithm representation also. As for prime fields, some representations can be used to avoid the costly remaindering phase within the multiplication. We will not discuss any implementations over extension field in this paper. We let the reader refer to [15] for details on data structures, arithmetic and matrix multiplication over small extension fields. From now on, when we will refer to finite fields this will mean word-size prime fields and the extensions for which the trick of [17, §4] is usable.

2.1.2 Ring homomoprphism and delayed reduction

As a primitive tool for implementing linear algebra routines, the efficiency of the finite field representation needs to be well studied. In [14] the author analyzes the efficiency of finite field arithmetic according to a chosen representation. It has been shown that atomic operations (e.g. addition, multiplication) can be performed more efficiently than with the classic method depending on the architecture. In particular, it appears that memory access based implementations (i.e. discrete logarithm) and floating point based implementations (i.e. floating point inverse) are more efficient on older architecture such as Ultra Sparc. Nevertheless, with newer architecture such as Pentium III and Pentium 4, integer machine operations become more efficient and outperform other implementations, except discrete logarithm for multiplicative group operations.

However, for linear algebra, the primary operation is the succession of two operations: a multiplication followed by an addition; this operation is commonly called $AXPY$ (also "fused-mac" or FMA within hardware). This operation clearly influences the efficiency of vectors dot product which is one of the main operations of classic linear algebra. However, optimized $AXPY$ atomic operation is deprecated since one would rather use delayed divisions. This technique consists in successive multiplications and accumulations without any division. Divisions intervene either just before an overflow occurs within the hardware data, or only after a fixed numbers of accumulations.

Indeed, any prime field $\mathbb{Z}_p$ can be naturally embedded into $\mathbb{Z}$ by representing its elements with an integer of an interval $[m, M]$, such that $M - m = p - 1$. The reverse conversion consists in applying a reduction modulo p to the integer value.

The ring structure being preserved by these homomorphisms, any ring algorithm over $\mathbb{Z}_p$ can be transposed into a ring algorithm over $\mathbb{Z}$.

hal-00018223, version 4 - 14 Jan 2009

Now the machine integer arithmetic uses a fixed number of bits γ for the integer representation: $\gamma = 32$ for `int`, $\gamma = 24$ (resp. $\gamma = 53$) for single (resp. double) precision floating point values, etc.

Using this approximate integer arithmetic, one has therefore to ensure that the computation of the integer algorithm will not overflow the representation. Hence for each integer algorithm, a bound on the maximal computed value has to be given, depending on m and M.

For example, if the representation is interval is $[0, p-1]$, one can perform λ accumulations without any divisions if

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

Note that if signed words are available, a centered representation can be used (i.e. $-\frac{p-1}{2} \leq x \leq \frac{p-1}{2}$ for the storage of an element x of the odd prime field) and the equation 1 becomes

$$\lambda\left(\frac{p-1}{2}\right)^2 < 2^{\gamma-1} \leq (1+\lambda)\left(\frac{p-1}{2}\right)^2 \tag{2}$$

which improves λ by a factor of 2.

Hence, the bottleneck of divisions can be amortized since only $\lceil \frac{n}{\lambda} \rceil$ divisions will occur in a n-dimensional vector dotproduct.

Contrary to atomic operations, floating point based implementations for dotproduct tend to be the most efficient on average. In particular, timings are constant and achieve almost half of the peak of arithmetical unit while the timings of others implementations drop as soon as the size of the finite field increases. However, when small primes are used, one can improve these timings to almost the peak of the machine by using others implementations [14, §3.4].

According to these results and the necessity of genericity, we provide implementations based on generic finite fields (e.g. use of *C++ template mechanism*). However, in this paper, we mainly use a floating point based implementation for our finite fields arithmetic, called `Zpz-double`. This choice is principally motivated by the use of optimized numerical basic linear algebra operations through the BLAS library. Indeed, one can easily benefit from these libraries by simply mapping linear algebra operations over finite fields to numeric computations and delayed divisions. This will be extensively explained in sections 3 and 4. Therefore, the choice of floating point based representations for finite field elements will be an asset since it will avoid any data conversion. Possibly, we may use a different finite field implementation in order to compare efficiencies. There, we will use the notation `Zpz-int`, meaning a word size integer based implementation. As we will see throughout the rest of the paper, the combination of BLAS and `Zpz-double` implementation will allow us to approach numerical efficiency for linear algebra problems over finite fields.

2.2 Recursion materials for arithmetical complexity

The following two lemmas will be useful to study the constant factor of linear algebra algorithms compared to matrix multiplication. The first one gives the

hal-00018223, version 4 - 14 Jan 2009

order of magnitude when the involved matrices will be square:

Lemma 2.1. *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 constants* C, a, ω, g.
2. $T(1) = e$ *for some constant* e.
3. $\log_2(C) < \omega$.

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

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 cases, 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 2.2. *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* $C = \sum_{i=1}^k c_i$, $D = \sum_{i=1}^k c_i d_i$, $2 < \omega$ *and* $\epsilon(m,n) \leq gm^2 + hmn$.
2. $T(1,F) \leq eF$ *for a constant* e.
3. $\log_2(C) < \omega - 1$

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

$$\begin{aligned} T(m,n) = \sum_{i_1=1}^k c_{i_1} \ldots \sum_{i_t=1}^k c_{i_t} T(1, n - f(d_1, \ldots, d_t, m)) + \\ m^\omega \left(a \frac{1 - \left(\frac{C}{2^\omega}\right)^t}{1 - \frac{C}{2^\omega}} - bD \frac{1 - \left(\frac{C}{2^{\omega-1}}\right)^t}{1 - \frac{C}{2^{\omega-1}}} \right) + bm^{\omega-1} n \frac{1 - \left(\frac{C}{2^{\omega-1}}\right)^t}{1 - \frac{C}{2^{\omega-1}}} \\ + \sum_{i_1=1}^k c_{i_1} H(m/2, n - d_i m/2) \ldots + \sum_{i_1=1}^k c_{i_1} \ldots \sum_{i_t=1}^k c_{i_t} H(1, n - f(d_1, \ldots, d_t, m)) \end{aligned} \tag{3}$$

hal-00018223, version 4 - 14 Jan 2009

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 we solve (e.g.suppose $T(m) = \alpha m^\omega$ in a 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.

3 Matrix multiplication

We propose a design for a matrix multiplication kernel routine over a word-size finite field, based on the three following features:

1. delayed modular redution, as explained section 2.1.2,
2. cache tuning and floating point arithmetic optimizations using BLAS,
3. Strassen-Winograd fast algorithm.

3.1 Cache tuning using BLAS

In most of the modern computer architecture, a memory access to the RAM is more than one hundred times slower than an arithmetic operation. To circumvent this slowdown, the memory is structured into two or three levels of cache acting as buffers to reduce the number of accesses to the RAM and reuse as much as possible the buffered data. This approach is only valid if the algorithm involves many computations with local data.

In linear algebra, matrix multiplication is the better suited operation for cache optimization: it is the first basic operation, for which the time complexity $\mathcal{O}(n^3)$ is an order of magnitude higher than the space complexity $\mathcal{O}(n^2)$. Furthermore it plays such a central role in linear algebra, that every other algorithm will take advantage of the tuning of this kernel routine.

These considerations have driven the development of basic linear algebra subroutines (BLAS) [11, 46] for numeric computations. One of its main achievement is the level 3 set of routines, based on a highly tuned matrix multiplication kernel.

hal-00018223, version 4 - 14 Jan 2009

For computations on a word-size finite field, a similar approach could be developed, e.g. following [29] for block decomposition. Instead, we propose to simply wrap these numerical routines to form the integer algorithm of the delayed modular approach of the previous section. This will enable to take benefit from both the efficiency of the floating point arithmetic and the cache tuning of the BLAS libraries. Furthermore relying on the generic BLAS interface makes it possible to benefit from the large variety of optimizations for all existing architectures and ensures a long term efficiency thanks to the much larger development effort existing for numerical computations.

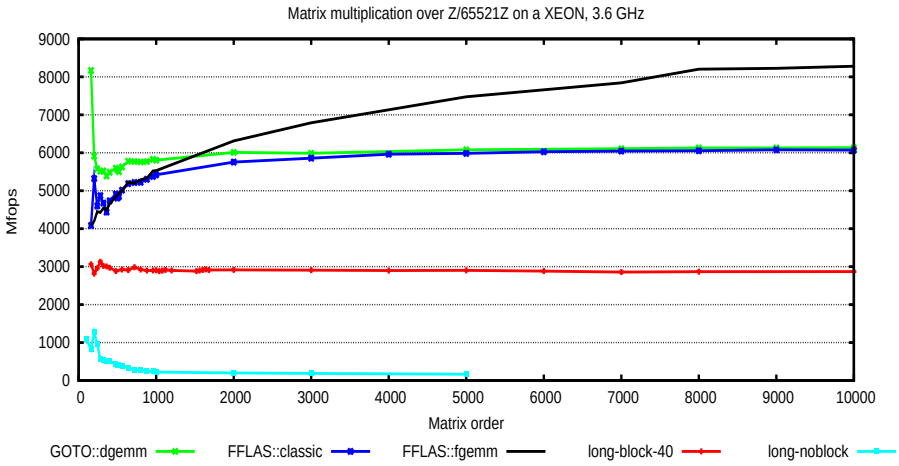

Figure 1: Blocking classical matrix multiplication, on a Xeon, 3.6GHz.

Figure 1 shows the advantage of this method (`FFLAS::classic`) compared to two other implementations: the naive algorithm (`long-noblock`), and a hand-made cache tuned implementation, based on block decomposition of the input matrices, so that each block product could be performed locally in the L2 cache memory (`long-block-40`, for a block dimension 40). The graph compares the computation speed in millions of field operations per seconds (Mfops) for different matrix orders. As a comparison we also provide the computation speed of the equivalent numerical BLAS routine `dgemm`. This approach improves on the efficiency of the two other methods over a finite field and the overhead of the modular reductions is limited. Finally, the (`FFLAS::fgemm`) implementation is the most efficient thanks to the combination of numerical computations and a fast matrix multiplication algorithm which is discussed in the next section.

3.2 Winograd fast algorithm

The third feature of this kernel is the use of a fast matrix multiplication algorithm. We will focus on Winograd's variant [24, algorithm 12.1] of Strassen's

hal-00018223, version 4 - 14 Jan 2009

algorithm [45]. We denote by $\mathrm{MM}(n)$ the dominant term of the arithmetic complexity of the matrix multiplication. The value of $\mathrm{MM}(n)$ thus reflects the choice of algorithm, e.g. $\mathrm{MM}(n) = 2n^3$ 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 $\mathrm{MM}(n)$, it is thus 3 for the classical algorithm, $\log_2(7) \approx 2.807354922$ for the Strassen-Winograd variant, and the best known exponent is about 2.375477 by [7].

In [30] Winograd's variant is discarded for numerical computations because of its bad stability and despite its better running time. In [35] aggregation-cancellation techniques of [36] are also compared. They also give better stability than the Winograd variant but worse running time. For exact computation, stability is no longer an issue and Winograd's faster variant is thus preferred.

3.2.1 A Cascade structure

Asymptotically, this algorithm improves on the number of arithmetic operations required for matrix multiplication from $\mathrm{MM}(n) = 2n^3$ to $\mathrm{MM}(n) = 6n^{2.8074}$. But for a given n, the total number of arithmetic operations can be reduced by switching after a few recursive levels of Winograd's algorithm to the classic algorithm. Table 1 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 1: 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 take advantage of the cache optimization of 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 which one recursive level is as fast the classic algorithm. Then the number of levels l is given by the formula

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

hal-00018223, version 4 - 14 Jan 2009

hal-00018223, version 4 - 14 Jan 2009

3.2.2 Schedule of the algorithm

We based our implementation of Winograd's algorithm on two different schedules. For the operation $C \leftarrow A \times B$ we use that of [12, Fig. 1] and for the extended $C \leftarrow \alpha A \times B + \beta C$, that of [32, Fig. 6] that we recall in table 2. More details about tasks scheduling and memory efficient variants of Winograd's algorithm can be found in [21].

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

Table 2: Schedule for operation $C \leftarrow \alpha A \times B + \beta C$ with 3 temporaries

3.2.3 Control of the overflow

Since Winograd's algorithms will be used with delayed modular reductions, one has to ensure that any intermediate computation will fit in the underlying fixed-size integer representation being used. Indeed, intermediate values can become large in this algorithm, and the former bound for the dot-product no-longer holds.

The main result of this section is that, in the worst case, the largest intermediate computation occurs during the recursive computation of the sixth recursive product P_6 (see appendix A). This result generalizes [17, 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 $m_A \leq a_{i,j} \leq M_A$, $m_B \leq b_{i,j} \leq M_B$ and $m_C \leq c_{i,j} \leq M_C$. 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.

Using a positive integer representation of the prime field elements (integers between 0 and $p-1$), the following corollary holds:

Corollary 3.2 (Positive modular representation)**.** *Using the same notations, with $a_{i,j}, b_{i,j}, c_{i,j}, \beta \in [0 \dots p-1]$, we have*

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

Instead, using a balanced representation (integers between $-\frac{p-1}{2}$ and $\frac{p-1}{2}$), this bound can be improved:

Corollary 3.3 (Balanced modular representation)**.** *Using the same notations with $a_{i,j}, b_{i,j}, c_{i,j}, \beta \in [-\frac{p-1}{2} \dots \frac{p-1}{2}]$, we have*

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

Corollary 3.4. *One can compute l recursive levels of Winograd algorithm without modular reduction over integers of γ bits as long as $k < k_{Winograd}$ where*

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

for a positive modular representation and

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

for a balanced modular representation.

3.3 Timings and comparison with numerical routines

This section presents experiments of our implementation of the matrix multiplication kernel described above.

The experiments use two different BLAS library: the automatically tuned BLAS ATLAS [46], and the BLAS by Kazushige Goto [28] refered to as GOTO. We used the `gcc` compiler version 4.1 on the Xeon machine and the `icc` compiler version 9.0 on the Itanium. We recall that `dgemm` refers to the BLAS matrix multiplication routine over double precision floating point numbers. Similarly, we named our routine over a word-size finite field `fgemm`.

The tables 3 and 4 report timings obtained for both exact and numeric matrix multiplication. First the comparison shows that the exact computation over a word size finite field (modulo 65521 on these tables) can reach a similar range of efficiency as the numerical computation. For increasing matrix dimensions, the exact computation becomes even more efficient (see also figure 1), thanks

hal-00018223, version 4 - 14 Jan 2009

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	0.38s	2.73s	8.59s	36.34s	95.21	134.03s	190.21s	258.08
	dgemm	0.37s	2.98s	10.02s	46.10s	126.38s	188.97s	267.83s	368.30s
	$\frac{fgemm}{dgemm}$	**1.02**	**0.92**	**0.86**	**0.79**	**0.75**	**0.71**	**0.71**	**0.70**
GOTO	fgemm	0.36s	2.53s	7.95s	33.44s	87.46s	124.86s	177.25s	238.00s
	dgemm	0.34s	2.65s	8.90s	41.01s	112.31s	167.20s	237.16s	325.62s
	$\frac{fgemm}{dgemm}$	**1.05**	**0.96**	**0.89**	**0.82**	**0.78**	**0.75**	**0.75**	**0.73**

Table 3: Comparison between `fgemm` and `dgemm` on a Xeon, 3.6GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	fgemm	0.46s	3.22s	10.14s	42.28s	110.64s	163.53s	225.08s	296.56s
	dgemm	0.45s	3.49s	11.45s	53.12s	144.45s	215.53s	305.21s	419.00s
	$\frac{fgemm}{dgemm}$	**1.01**	**0.92**	**0.89**	**0.80**	**0.77**	**0.76**	**0.74**	**0.71**
GOTO	fgemm	0.43s	2.99s	9.35s	39.21s	104.07s	152.12s	209.22s	277.32s
	dgemm	0.40s	3.18s	10.61s	48.88s	133.75s	200.11s	283.94s	390.37s
	$\frac{fgemm}{dgemm}$	**1.06**	**0.94**	**0.88**	**0.80**	**0.78**	**0.76**	**0.74**	**0.71**

Table 4: Comparison between `fgemm` and `dgemm` on Itanium2, 1.3GHz

to the use of Winograd's algorithm (improvement factor between 13% and 29% for dimension 10 000).

These experiments also show the advantage of relying on a generic interface for numerical BLAS: the exact computation will directly take advantage of the improvements of the best numerical routine. This appears when comparing GOTO and ATLAS on these two target architecture, where GOTO is about 10% faster.

4 Triangular system solving with matrix right/left hand side

We now discuss the implementation of solvers for triangular systems with matrix right hand side (or equivalently left hand side). The resolution of such systems plays a central role in many linear algebra problems, e.g. it is the second main operation in block Gaussian elimination after matrix multiplication as will be recalled in section 5.1. This operation is commonly named `trsm` in the BLAS convention. In the following, we will consider without loss of generality the resolution of an upper triangular system with matrix right hand side, i.e. the

hal-00018223, version 4 - 14 Jan 2009

operation $B \leftarrow U^{-1}B$, where U is $m \times m$ upper triangular and B is $m \times n$.

Following the approach of the BLAS numerical routine, our implementation is based on a block recursive algorithm to reduce the computation to matrix multiplications.

Now similarly to our approach with matrix multiplication, the design of our implementation also focuses on delaying the modular reductions as much as possible. As will be shown in section 4.2, delaying the whole resolution leads to a quick growth in the size of coefficients. Therefore we also present in section 4.3 another way of delaying these modular reductions. We lastly present how to combine these two techniques within a multi-cascade algorithm.

4.1 The block recursive algorithm

Algorithm `trsm` recalls the block recursive algorithm.

Algorithm 1: `trsm` (A, B)

Data: $A \in \mathbb{Z}/p\mathbb{Z}^{m\times m}$, $B \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$.
Result: $X \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$ such that $AX = B$.
begin
  if $m = 1$ **then**
    $X := A_{1,1}^{-1} \times B$
  else
    `/* splitting matrices into two blocks of sizes` $\lfloor \frac{m}{2} \rfloor$ `and` $\lceil \frac{m}{2} \rceil$

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

`*/`

    X_2 :=`trsm` (A_3, B_2)
    $B_1 := B_1 - A_2X_2$
    X_1 :=`trsm` (A_1, B_1)
end

Lemma 4.1. *Algorithm* `trsm` *is correct and the leading term of its arithmetic complexity over* $\mathbb{Z}/p\mathbb{Z}$ *is*

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

This complexity is m^2n *using classic matrix multiplication.*

hal-00018223, version 4 - 14 Jan 2009

Proof. Extending the previous notation MM (n), we denote by MM (m,k,n) the cost of multiplying a $m \times k$ by a $k \times n$ matrices. The cost function $\mathrm{TRSM}(m,n)$ satisfies the following equation:

$$\mathrm{TRSM}(m,n) = 2\mathrm{TRSM}(\frac{m}{2},n) + \mathrm{MM}(\frac{m}{2},\frac{m}{2},n).$$

Let $t = \log_2(m)$. Although the algorithm works for any n, we restrict the complexity analysis to the case where $m \leq n$ for the sake of simplicity. We then have:

$$\begin{aligned}\mathrm{TRSM}(m,n) &= 2\mathrm{TRSM}(\frac{m}{2},n) + \frac{1}{2^{\omega-1}}\left\lceil\frac{n}{m}\right\rceil \mathrm{MM}(m)\\ &= 2^t\mathrm{TRSM}(1,n) + \frac{1}{2^{\omega-1}}\left\lceil\frac{n}{m}\right\rceil \mathrm{MM}(m)\frac{1-\left(\frac{2}{2^{\omega-1}}\right)^t}{1-\frac{2}{2^{\omega-1}}}.\end{aligned}$$

As $\mathrm{TRSM}(1,n) = 2n$ and $\left(2^{\omega-1}\right)^t = m^{\omega-1}$, we obtain the expected complexity $\mathrm{TRSM}(m,n) = \frac{1}{2^{\omega-1}-2}\left\lceil\frac{n}{m}\right\rceil \mathrm{MM}(m) + \mathcal{O}(m^2+mn)$. □

4.2 Delaying reductions globally

As for matrix multiplication, the delayed computation relies on the fact that ring operations over the finite field can be replaced by ring operations over $\mathbb{Z}$ using the ring homomorphisms described in section 2.1.2. However, triangular system resolutions involve, in the general case, field operations: the divisions by the diagonal elements of the triangular matrix. Therefore this technique is only valid with unit diagonal matrices.

In the general case, the triangular matrix is made unit diagonal by the following factorization: $A = DU$, where D is diagonal and U is unit diagonal upper triangular. Then the system $UX = D^{-1}B$ only involves ring operations and can be solved over $\mathbb{Z}$. This normalization leads to an additional cost of $O(mn)$ arithmetic operations (see [18] for more details).

Now the integer computation with a fixed sized arithmetic (e.g. the floating point arithmetic) is exact as long as all intermediate results of the computation do not exceed the bit capacity of the representation. Therefore we now propose bounds on the values computed by the algorithm over $\mathbb{Z}$.

Theorem 4.2. *Let $T \in \mathbb{Z}^{n\times n}$ be a unit diagonal upper triangular matrix and $b \in \mathbb{Z}^n$, with $m \leq T_{i,j} \leq M$ and $m \leq b_i \leq M$ and $m \leq 0 \leq M$. Let $x = (x_i)_{i\in[1\ldots n]} \in \mathbb{Z}^n$ be the solution of the system $Tx = b$. Then $\forall\ k \in [0\ldots n-1]$:*

$$\begin{cases} -u_k \leq x_{n-k} \leq v_k & \text{for } k \text{ even,}\\ -v_k \leq x_{n-k} \leq u_k & \text{for } k \text{ odd}\end{cases}$$

with

$$\begin{cases} u_k = \frac{M-m}{2}(M+1)^k - \frac{M+m}{2}(M-1)^k,\\ v_k = \frac{M-m}{2}(M+1)^k + \frac{M+m}{2}(M-1)^k.\end{cases}$$

hal-00018223, version 4 - 14 Jan 2009

Proof. First note the following relations:

$$\forall k \begin{cases} u_k & \leq & v_k \\ -mu_k & \leq & Mv_k \\ -mv_k & \leq & Mu_k \end{cases}$$

The third one comes from

$$Mu_k + mv_k = \frac{M^2 - m^2}{2}((M+1)^k - (M-1)^k) \geq 0.$$

The proof is now an induction on k, following the system resolution order. The initial case $k = 0$ correspond to the first step: $x_n = b_n$, leading to

$$-u_0 = m \leq x_n \leq M = v_0.$$

Suppose now that the inequalities hold for $k \in [0 \dots l]$ and prove them for $k = l + 1$. If l is odd, $l + 1$ is even.

$$\begin{aligned}
x_{n-l-1} &= b_{n-l-1} - \sum_{j=n-l}^{n} T_{n-l-1,j} x_j \\
&\leq M + \sum_{i=0}^{\frac{l-1}{2}} \max(Mu_{2i}, -mv_{2i}) + \max(Mv_{2i+1}, -mu_{2i+1}) \\
&\leq M \left(1 + \sum_{i=0}^{\frac{l-1}{2}} u_{2i} + v_{2i+1} \right) \\
&\leq M \left(1 + \sum_{i=0}^{\frac{l-1}{2}} \frac{M-m}{2}(M+2)(M+1)^{2i} + \frac{M+m}{2}(M-2)(M-1)^{2i} \right) \\
&\leq M \left(1 + \frac{M-m}{2}(M+2)\frac{(M+1)^{l+1}-1}{(M+1)^2-1} + \frac{M+m}{2}(M-2)\frac{(M-1)^{l+1}-1}{(M-1)^2-1} \right) \\
&\leq \frac{M-m}{2}(M+1)^{l+1} + \frac{M+m}{2}(M-1)^{l+1} = v_{l+1}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
x_{n-l-1} &\geq m - \sum_{i=0}^{\frac{l-1}{2}} \max(Mv_{2i}, -mu_{2i}) + \max(Mu_{2i+1}, -mv_{2i+1}) \\
&\geq m - M \sum_{i=0}^{\frac{l-1}{2}} v_{2i} + u_{2i+1} \\
&\geq m - M \sum_{i=0}^{\frac{l-1}{2}} \frac{M-m}{2}(M+2)(M+1)^{2i} - \frac{M+m}{2}(M-2)(M-1)^{2i} \\
&\geq m - M \left(\frac{M-m}{2}(M+2)\frac{(M+1)^{l+1}-1}{(M+1)^2-1} - \frac{M+m}{2}(M-2)\frac{(M-1)^{l+1}-1}{(M-1)^2-1} \right) \\
&\geq \frac{M-m}{2}(M+1)^{l+1} - \frac{M+m}{2}(M-1)^{l+1} = u_{l+1}.
\end{aligned}$$

hal-00018223, version 4 - 14 Jan 2009

For l even, a similar proof leads to

$$-v_{l+1} \leq x_{n-l-1} \leq u_{l+1}.$$

□

Corollary 4.3. *Using the notation of theorem 4.2,*

$$|x| \leq \frac{M-m}{2}(M+1)^{n-1} + \frac{M+m}{2}(M-1)^{n-1}.$$

Moreover this bound is optimal.

Proof. The sequence (v_k) is increasing and always greater than (u_k). Thus $\forall\ k \in [0 \dots n-1]\ |x_{n-k}| \leq\ u_k \leq v_k \leq v_{n-1}$.

Now the vector $x = (x_i)_{i\in[1\dots n]} \in \mathbb{Z}^n$ such that $\forall\ k \in [0 \dots n-1]\ |x_{n-k}| = v_k$ satisfies the system $Tx = b$ with

$$T = \begin{bmatrix} \ddots & \ddots & \ddots & \ddots & \ddots \\ & 1 & M & m & M \\ & & 1 & M & m \\ & & & 1 & M \\ & & & & 1 \end{bmatrix}, b = \begin{bmatrix} \vdots \\ m \\ M \\ m \\ M \end{bmatrix}$$

Therefore the bound is reached. □

The following corollaries apply this result to the positive and balanced modular representations.

Corollary 4.4 (Positive modular representation)**.** *For $1 \leq i, j \leq n$, if $T_{i,j}, b_i \in [0 \dots p-1]$, then*

$$|x| \leq \frac{p-1}{2}(p^{n-1} + (p-1)^{n-1}).$$

Corollary 4.5 (Balanced modular representation)**.** *For $1 \leq i, j \leq n$, if $T_{i,j}, b_i \in [-\frac{p-1}{2} \dots \frac{p-1}{2}]$, then*

$$|x| \leq \frac{p-1}{2}\left(\frac{p+1}{2}\right)^{n-1}.$$

Remark 4.6. *The balanced modular representation improves the bound by a factor of 2^{n-1}.*

As a consequence, one can solve a unit diagonal triangular system of dimension n using arithmetic operations with integers stored on γ bits if

$$\frac{p-1}{2}(p^{n-1} + (p-1)^{n-1}) < 2^{\gamma} \tag{4}$$

for a positive representation and

$$\frac{p-1}{2}\left(\frac{p+1}{2}\right)^{n} < 2^{\gamma} \tag{5}$$

hal-00018223, version 4 - 14 Jan 2009

for a balanced representation.

For instance, using the `double` floating point representation (53 bits of mantissa) the maximal dimension of the system is 34 (resp. 52) for a positive (resp. balanced) representation of $\mathbb{Z}_3$. For larger fields, this maximal dimension becomes quickly very small: with $p = 1001$, $n \leq 5$ (resp. $n \leq 6$) for a positive (resp. balanced) representation.

In the following, we will denote by $t_{\text{del}}(p, \gamma)$ the maximum dimension for the resolution with delayed modular reductions. This dimension is small, and this approach can therefore only be used as a terminal case of the recursive block algorithm. This first cascade algorithm is characterized by the threshold t_{del}. For efficiency, we used in our implementation the BLAS routine `trsm` to perform the delayed computation over $\mathbb{Z}$. Despite the small dimension of the blocks, we will see in section 4.4 that this approach can slightly improve the efficiency of the computation when the finite field is small.

4.3 Delaying reductions in the update phase only

The block recursive algorithm consists in several matrix multiplications of different dimensions. In most cases, the matrix multiplications are done over $\mathbb{Z}$ with a modular reduction on the result only. But part of these result matrices will be accumulated to other matrix multiplications in later computations. Therefore these intermediate modular reductions could be delayed even more by allowing to accumulate these results over $\mathbb{Z}$ as much as possible.

This technique can be applied within the former cascade algorithm, to produce a double cascade structure. The key idea is to split the matrices at two levels as shown on figure 2: a fine grain splitting with the dimension t_{del} of the pre-

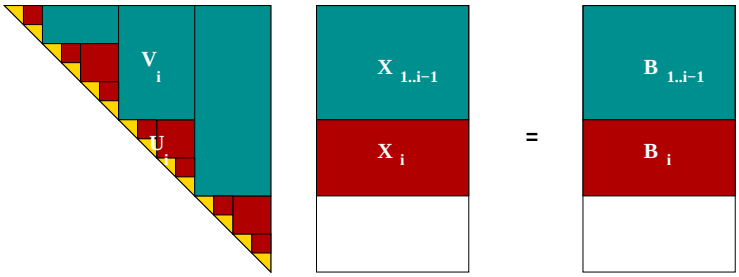

Figure 2: Splitting for the double cascade `trsm` algorithm

vious section, and a coarse grain splitting with the dimension t_{update} such that all recursive calls of dimension lower than t_{update} can let the matrix multiplication updates accumulate without modular reductions. Choosing $t_{\text{update}} = k_{\text{Winograd}}$ (from corrolary 3.4) will ensure this property. To adjust together the dimensions of the two block decompositions, we set $t_{\text{split}} = \lfloor t_{\text{Winograd}}/t_{\text{del}} \rfloor \, t_{\text{del}}$.

Algorithm 2 is a loop on every block of column dimension t_{update}. For each of them, the triangular system is solved using algorithm 3 and the update is

hal-00018223, version 4 - 14 Jan 2009

Algorithm 2: `trsm-rec-BLAS-delayed` :

Data: $A \in \mathbb{Z}/p\mathbb{Z}^{m\times m}$, $B \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$
Result: $X \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$ s.t. $AX = B$
begin
  Compute t_{del} from equation (4 or 5)
  Compute t_{Winograd} from corrolary (3.4)
  $t_{\text{split}} = \lfloor t_{\text{Winograd}}/t_{\text{del}} \rfloor \, t_{\text{del}}$
  foreach *block column of A of dimension $m \times t_{split}$ of the form* $\begin{bmatrix} V_i \\ U_i \\ 0 \end{bmatrix}$
  do
    $X_i =$ `trsm-partial-delayed`(U_i, B_i)
    $X_i = X_i \mod p$
    $B_{1\ldots i-1} = B_{1\ldots i-1} - V_i X_i$
    $B_{1\ldots i-1} = B_{1\ldots i-1} \mod p$
  return X
end

Algorithm 3: `trsm-partial-delayed`

Data: $A \in \mathbb{Z}/p\mathbb{Z}^{m\times m}$, $B \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$, m must be lower than t_{update}
Result: $X \in \mathbb{Z}/p\mathbb{Z}^{m\times n}$ s.t. $AX = B$
begin
  if $m \leq n_{del}$ **then**
    $B = B \mod p$
    $X =$ `dtrsm`(A, B) ; `/* the BLAS routine */`
    $X = X \mod p$
  else
    `/* (splitting of the matrix into blocks of dimension` $\lfloor \frac{m}{2} \rfloor$ `and` $\lceil \frac{m}{2} \rceil$`) */`

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

    $X_2 :=$ `trsm-partial-delayed`(A_3, B_2)
    $B_1 := B_1 - A_2 X_2$; `/* without modular reduction */`
    $X_1 :=$ `trsm-partial-delayed`(A_1, B_1)
  return X
end

hal-00018223, version 4 - 14 Jan 2009

performed by a matrix multiplication over $\mathbb{Z}$ followed by a modular reduction. Algorithm 3 is simply the cascade algorithm of the previous section: the block recursive algorithm 1 with the fully delayed algorithm as a terminal case. The matrix multiplication updates are performed over $\mathbb{Z}$ without any reduction of the result, since the threshold t_{update} allows to accumulate them.

4.4 Experiments

We now compare three implementations of the `trsm` routine over a word size finite field:

Pure recursive (`Pure-Rec`): Simply algorithm 1,

Recursive-BLAS (`Rec-BLAS`): The cascade algorithm formed by the recursive algorithm and the BLAS routine `dtrsm` as a terminal case. It differs from algorithm 3 by the fact that the matrix multiplication $B_1 := B_1 - A_2X_2$ is always followed by a modular reduction.

Recursive-BLAS-Delayed (`Rec-BLAS-Delayed`): algorihtm 2.

We compare these three variants over finite fields with different cardinalities, so as to make the parameters t_{del} and t_{update} vary as in the following table:

p	$\lceil \log_2 p \rceil$	t_{del}	t_{update}
5	3	23	2 147 483 642
1 048 583	20	2	8190
8 388 617	23	2	126

In the experiments of figure 3, the matrix B is square ($m = n$). One can first notice the gain provided by the use of the first cascade with the delayed `dtrsm` routine by comparing the curves `rec-BLAS` and `pure-rec` for $p = 5$. This advantage shrinks when the characteristic gets larger, since $t_{\text{del}} = 2$ for $p = 1\,048\,583$ or $p = 8\,388\,61$.

Now the introduction of the coarse grain splitting, delaying the reductions in the update phase improves by up to 500 Mfops the computation speed. This gain is similar for $p = 5$ and $p = 1\,048\,583$ since in both cases $n < t_{\text{update}}$ and there is therefore no modular reduction between the matrix multiplications.

Lastly for $p = 8\,388\,617$, the speed drops down since more reductions are required. The variants `pure-rec` and `rec-BLAS` are penalized by their dichotomic splitting, creating too many modular reductions after each matrix multiplication. Now `rec-BLAS-delayed` has the best efficiency since the double cascade structure minimizes the number of reductions.

We now give a comparison of this implementation with the equivalent routine of the original BLAS `dtrsm`. As for matrix multiplication in section 3.3, we compare the routines according to two different BLAS implementations (i.e. ATLAS and GOTO) and two different architectures. Nevertheless, we do not present the results with ATLAS on Xeon architecture due to the surprisingly poor efficiency of ATLAS `dtrsm` during our tests. In the following, `ftrsm` denotes

hal-00018223, version 4 - 14 Jan 2009

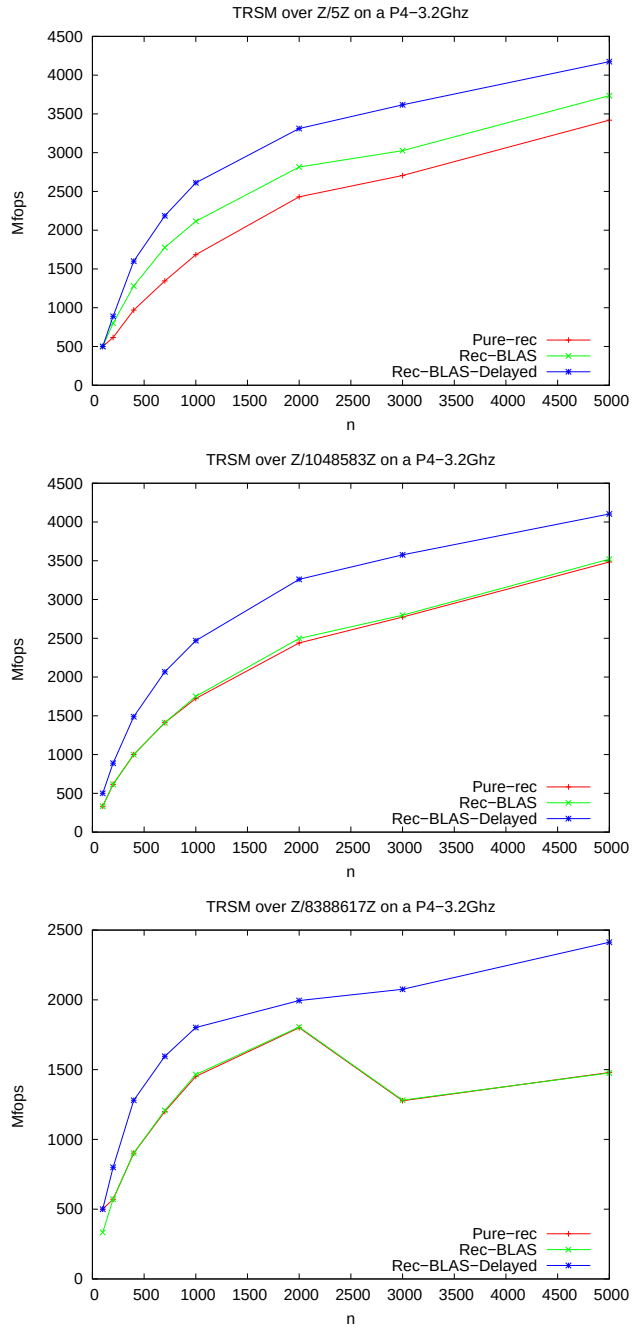

Figure 3: Comparison of the `trsm` variants for $p = 5, 1\,048\,583, 8\,388\,617$, on a Pentium4-3,2Ghz-1Go

hal-00018223, version 4 - 14 Jan 2009

the `trsm` routine over 16-bits prime field (i.e. $\mathbb{Z}_{65521}$) using the `ZpZ-double` implementation.

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	ftrsm	0.37s	1.93s	5.73s	23.63s	62.50s	91.67s	121.84s	166.74s
GOTO	ftrsm	0.25s	1.66s	5.08s	21.47s	55.95s	80.77s	111.57s	150.81s
	dtrsm	0.17s	1.35s	4.50s	20.64s	56.19s	83.85s	119.18s	163.33s
	$\frac{ftrsm}{dtrsm}$	**1.47**	**1.23**	**1.13**	**1.04**	**1.00**	**0.96**	**0.94**	**0.92**

Table 5: Timings of triangular solver with matrix hand side on a Xeon, 3.6GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	ftrsm	0.34s	2.28s	7.11s	30.26s	77.43s	112.01s	158.00s	214.31s
	dtrsm	0.26s	1.95s	6.37s	28.60s	76.44s	113.78s	161.19s	219.31s
	$\frac{ftrsm}{dtrsm}$	**1.31**	**1.17**	**1.12**	**1.06**	**1.01**	**0.98**	**0.98**	**0.98**
GOTO	ftrsm	0.30s	2.00s	6.23s	26.67s	68.22s	104.32s	137.96s	192.37s
	dtrsm	0.21s	1.61s	5.36s	24.59s	67.35s	100.42s	142.43s	195.79s
	$\frac{ftrsm}{dtrsm}$	**1.43**	**1.24**	**1.16**	**1.08**	**1.01**	**1.04**	**0.97**	**0.98**

Table 6: Timings of triangular solver with matrix hand side on Itanium2, 1.3GHz

Tables 5 and 6 show that our implementation of exact `trsm` solving is not far from numerical performances. Moreover, on our Xeon architecture, with GOTO BLAS, we are able to achieve even better performances than numerical solving for matrices of dimension greater than 7 000.

The good performance of our implementation is mostly achieved with the efficient reduction to fast matrix multiplication and the double cascade structure. Figure 4 shows the ratio of the computation time of our `trsm` compared with matrix multiplication routine. According to lemma 4.1, this ratio is 1/2 with $\omega = 3$ and 2/3 with $\omega = \log_2 7$. In practice, our implementation only performs a few recursive calls of Winograd's algorithm, and the ratio appears to be between 0.5 and 0.666 as soon as the dimension is large enough, showing the good efficiency of the reduction to matrix multiplication.

5 Finite Field Matrix Factorizations

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 a reduction to matrix multiplication [45]. Nevertheless, our main concern here is the singularity of

hal-00018223, version 4 - 14 Jan 2009

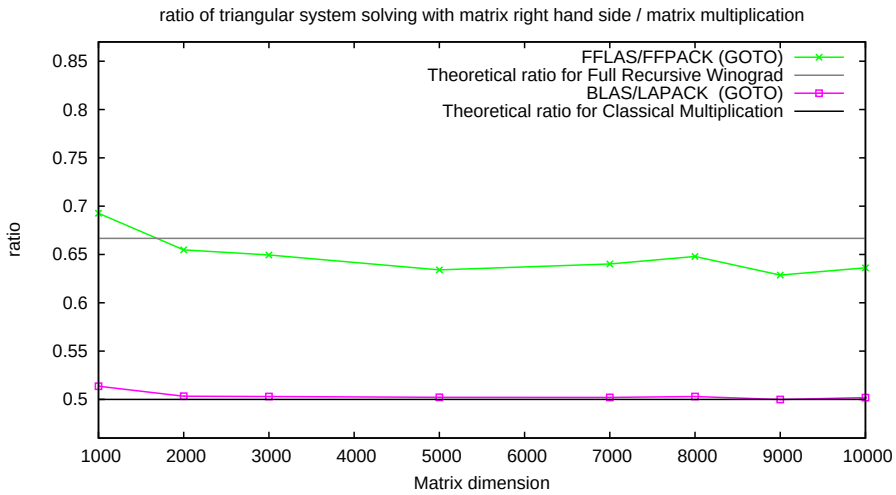

Figure 4: Comparing triangular system solving with matrix multiplication on a Xeon, 3.6GHz

the matrices since we want to derive efficient algorithms for most problems (e.g. rank or nullspace). One approach there is then to use 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.

5.1 Triangularizations

The classical block LDU or LUP factorizations (see [1]) can not be used due to their restriction to non-singular case. Instead one would rather use the LQUP factorization of [33]. We here propose a fully in-place variant and analyze its behaviour.

The LQUP factorization is a generalization of the well known block LUP factorization for the singular case [5]. Let A be a $m \times n$ matrix, we want to compute the quadruple $< L, Q, U, P >$ such that $A = LQUP$. The matrix `L` is lower triangular, `P` and `Q` are permutation matrices and `U` is a rank r upper triangular matrix with its r first rows non-zero.

The algorithm with best known complexity computing this factorization uses a divide and conquer approach and reduces to matrix multiplication [33]. 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 half. After some row permutations, It thus gives the T, Y and

hal-00018223, version 4 - 14 Jan 2009

L_1 blocks of figure 5, together with some row permutations stored in Q. Then, after some column permutations ($[XZ] = [A_{21}A_{22}]P$), the algorithm computes G such that $GT = X$ via `trsm`, replaces X by zeroes and eventually updates $Z = Z - GY$. The third step is a recursive call on Z, followed by an update of Q. We let the readers refer e.g. to [3, (2.7c)] for further details.

Furthermore, our implementation of LQUP also uses the trick proposed in [18, §4.2], namely storing L in its compressed form $\tilde{L}$.

This triangularization is thus fully in-place.

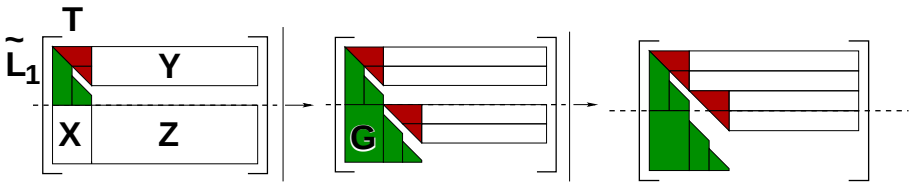

Figure 5: Principle of the LQUP factorization

Lemma 5.1. *The dominant term of the time complexity of algorithm LQUP with $m \leq n$ is*

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

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

Proof. Lemma 2.2 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 $\mathrm{LQUP}(m,n) = \alpha m^\omega + \beta n m^{\omega-1} = \mathrm{LQUP}(m/2,n) + \mathrm{TRSM}(m/2,r) + R(m/2,r,n-r) + \mathrm{LQUP}(m/2,n-r)$, where r is the rank of the first $m/2$ rows. This gives $\alpha m^\omega + \beta n m^{\omega-1} = \alpha(m/2)^\omega + \beta n (m/2)^{\omega-1} + \frac{1}{2^{\omega-1}-2}\left\lceil \frac{m}{2r} \right\rceil \mathrm{MM}(r) + \left\lceil \frac{m(n-r)}{2r^2} \right\rceil \mathrm{MM}(r) + \alpha(m/2)^\omega + \beta(n-r)(m/2)^{\omega-1}$. With $m \leq n$, the latter is maximal for $r = m/2$, and then, writing $\mathrm{MM}(x) = C_\omega x^\omega$, 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 $\alpha = 2\frac{\alpha}{2^\omega} - \frac{\beta}{2^\omega} - C_\omega \frac{2^\omega-6}{2^\omega(2^\omega-4)}$. Solving for β and α gives the announced terms. □

5.2 Performance 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 efficiency 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.

hal-00018223, version 4 - 14 Jan 2009

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	0.32s	1.84s	4.89s	19.34s	48.94s	73.86s	97.50s	131.11s
	dgetrf	0.17s	1.19s	3.83s	16.90s	45.32s	67.44s	94.83s	130.15s
	$\frac{lqup}{dgetrf}$	**1.88**	**1.55**	**1.28**	**1.14**	**1.08**	**1.10**	**1.03**	**1.01**
GOTO	lqup	0.25s	1.52s	4.47s	17.93s	44.54s	67.88s	89.63s	119.65s
	dgetrf	0.15s	1.03s	3.33s	14.84s	39.58s	58.61s	82.89s	113.47s
	$\frac{lqup}{dgetrf}$	**1.67**	**1.48**	**1.34**	**1.21**	**1.13**	**1.16**	**1.08**	**1.05**

Table 7: Performance of matrix triangularization (for $\mathbb{Z}/65521\mathbb{Z}$ and floats) on a Xeon, 3.6GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	lqup	0.38s	2.20s	6.36s	25.22s	61.64s	89.74s	127.43s	163.68s
	dgetrf	0.20s	1.47s	4.61s	20.26s	53.57s	79.37s	111.66s	152.42s
	$\frac{lqup}{dgetrf}$	**1.85**	**1.50**	**1.38**	**1.25**	**1.15**	**1.13**	**1.14**	**1.07**
GOTO	lqup	0.34s	2.00s	5.81s	23.11s	56.80s	83.90s	113.66s	150.82s
	dgetrf	0.16s	1.17s	3.80s	17.07s	46.18s	69.00s	97.56s	134.01s
	$\frac{lqup}{dgetrf}$	**2.21**	**1.72**	**1.53**	**1.35**	**1.23**	**1.22**	**1.16**	**1.13**

Table 8: Performance of matrix triangularization (for $\mathbb{Z}/65521\mathbb{Z}$ and floats) on Itanium2-1.3GHz

Tables 7 and 8 show efficiency obtained with our exact triangularization based on fast matrix multiplication and the one obtained with numerical computation. There, "dgetrf" computes a floating point LU factorization of a general $m \times n$ matrix using partial pivoting with row interchanges. Exact computation is done in the prime field of integers modulo 65521. We are now mostly able to reach the speed of numerical computations. 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 Xeon 3.6GHz architecture. This is only 5% slower than the best numerical computation.

We could have expected that our speed would have been even better than numerical approach since we take advantage of Strassen-Winograd's multiplication while numerical computations are 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 due to the recursive structure of the algorithm. Then, the number of Winograd calls is at least one less than within matrix multiplication routines. In our tests, it appears that we only use 3 calls on our Xeon architecture and 1 call on the Itanium2 architecture according to matrix multiplication threshold. This explains the better performance on the

hal-00018223, version 4 - 14 Jan 2009

Xeon compared to numerical routines than the Itanium2 architecture.

Note also that in order to take even more into account data locality one can develop a version of LQUP where blocks are maintained as square as possible. Indeed, as soon as the RAM is full, data locality becomes more important than memory saves. The TURBO method [22] addresses this issue. A first implementation of TURBO has been studied in [18, §4.5] and it reveals to be the fastest for large matrices, despite its bigger memory demand [18, Figure 6]. This is advocating further uses of recursive blocked data formats and of more recursive levels of TURBO.

5.3 Comparison with the multiplication

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 $\frac{2}{5}$. Figure 6 shows that the experimental behavior of the factorization is not very far from this theoretical ratio.

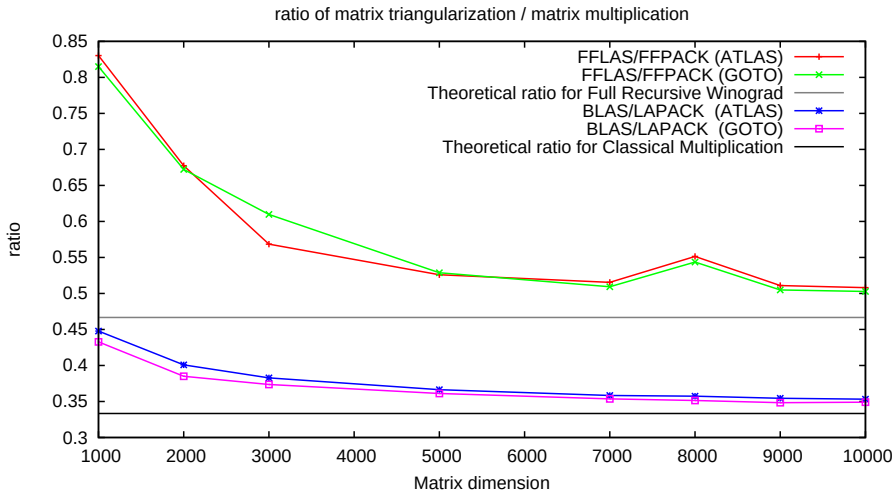

Figure 6: Comparing matrix triangularization with matrix multiplication on a Xeon, 3.6GHz

6 Applications

In this section, we use our matrix multiplication, matrix factorization and matrix solvers as basic routines to perform other linear algebra routines. For instance,

hal-00018223, version 4 - 14 Jan 2009

from the two routines (i.e. LQUP and `trsm`), one can also directly derive several other algorithms, e.g.:

- The **rank** is the number of non-zero rows in U.
- The **determinant** is the product of the diagonal elements of U (stopping whenever a zero is encountered).

In the following, we first give the theoretical complexities with explicit constant terms. These constants depend on the kind of matrix multiplication used (fast or classical). In order to validate our approach we then compare this theoretical ratios to some experimental ones.

6.1 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 $NS(m;n) = LQUP(m;n) + \mathrm{TRSM}(m; n-m)$. which gives

$$NS(m;n) = (\left\lceil \frac{n}{m} \right\rceil \frac{2}{2^{\omega-1}-2} - \frac{1}{2^{\omega}-2})\mathrm{MM}(m) \tag{6}$$

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.

6.2 Triangular multiplications

6.2.1 Triangular matrix multiplication

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

$$TRMM(n) = \frac{1}{2^{\omega-1}-2}\mathrm{MM}(n). \tag{7}$$

The latter is n^3 with classical multiplication.

6.2.2 Upper-lower Triangular matrix multiplication

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

$$\begin{bmatrix} A_1 & A_2 \\ & A_4 \end{bmatrix} \times \begin{bmatrix} B_1 & \\ B_3 & B_4 \end{bmatrix} = \begin{bmatrix} A_1 B_1 + A_2 B_3 & A_2 B_4 \\ A_4 B_3 & A_4 B_4 \end{bmatrix}$$

hal-00018223, version 4 - 14 Jan 2009

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

$$\mathrm{UTLT}(n) = \frac{2^\omega}{(2^\omega - 4)(2^\omega - 2)}\mathrm{MM}(n). \tag{8}$$

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

6.2.3 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 $\mathrm{UTUT}(n) = 2\mathrm{UTUT}(n/2) + 2TRMM(n/2)$, which yields

$$\mathrm{UTUT}(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)}\mathrm{MM}(n). \tag{9}$$

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

6.3 Squaring

6.3.1 $A \times A^T$

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 ADA^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 $AAT(n) = 4AAT(n/2) + 2\mathrm{MM}(n/2) + 3\mathrm{ADD}(n/2) + 2(n/2)^2$

Ignoring the cost of the three additions and the diagonal multiplications, this yields

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

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

$$AAT(m; n) = \left\lceil \frac{n}{m} \right\rceil \frac{2}{2^\omega - 4}\mathrm{MM}(m). \tag{11}$$

hal-00018223, version 4 - 14 Jan 2009

hal-00018223, version 4 - 14 Jan 2009

6.3.2 Symmetric case

When A is already symmetric, and if the diagonal is unitary, the constant factor decreases. 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 calls is a call to a non symmetric AA^T. Therefore the cost is now: $SymAAT(n) = 2SymAAT(n/2) + AAT(n/2) + 2\mathrm{MM}(n/2)$, once again ignoring n^2. This yields

$$SymAAT(n) = \frac{2(2^\omega - 3)}{(2^\omega - 4)(2^\omega - 2)}\mathrm{MM}(n). \tag{12}$$

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

6.3.3 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

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

Thus it requires two recursive calls, 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

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

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

6.4 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 $L_1 D_1 G^T = A_2$; an AAT to compute GD_1G^T and a recursive call to compute $L_2 D_2 L_2^T = A_4 - GD_1G^T$. The cost is thus $LDLT(n) = 2LDLT(n/2) + \mathrm{TRSM}(n/2) + AAT(n/2)$, which yields

$$LDLT(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)}\mathrm{MM}(n). \tag{14}$$

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

6.5 Matrix inverse

6.5.1 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 $\mathrm{INVT}(n) = 2\mathrm{INVT}(n/2) + 2TRMM(n/2)$ which yields

$$\mathrm{INVT}(n) = \frac{2}{2^\omega - 2} TRMM(n) = \frac{4}{(2^\omega - 4)(2^\omega - 2)} \mathrm{MM}(n). \tag{15}$$

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

6.5.2 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 $INV(n) = LQUP(n) + \mathrm{INVT}(n) + \mathrm{TRSM}(n)$. This gives

$$INV(n) = \frac{3 \times 2^\omega}{(2^\omega - 4)(2^\omega - 2)} \mathrm{MM}(n). \tag{16}$$

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

6.5.3 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 $SymINV(n) = LDLT(n) + \mathrm{INVT}(n) + \mathrm{LTL}(n)$ which yields

$$SymINV(n) = \frac{12}{(2^\omega - 2)(2^\omega - 4)} \mathrm{MM}(n). \tag{17}$$

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

6.5.4 Full-rank Moore-Penrose pseudo-inverse

A is a rectangular full rank $m \times n$ matrix. We 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. [42] 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:

$$MPINV(m; n) = AAT(m; n) + LDLT(m) + 2\mathrm{TRSM}(m; n).$$

hal-00018223, version 4 - 14 Jan 2009

The cost is then

$$MPINV(m;n) = \left(\left\lceil \frac{n}{m} \right\rceil \frac{6}{2^\omega - 4} + \frac{4}{(2^\omega - 2)(2^\omega - 4)} \right) \mathrm{MM}(m) \qquad (18)$$

The latter is $3m^2n + \frac{1}{3}m^3$ with classical multiplication. This correspond e.g. to the normal equations numerical resolution [27, algorithm 5.3.1].

6.5.5 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 $LQUP$ 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 [39, (7)] as follows:

$$A^\dagger = \begin{bmatrix} I \\ Y^T U_1^{-T} \end{bmatrix} \left((L_1 + L_1^{-T} G^T G)(U_1 + YY^T U_1^{-1}) \right)^{-1} [I | L_1^{-T} G^T]. \qquad (19)$$

We note $W = (L_1 + L_1^{-T} G^T G)(U_1 + YY^T U_1^{-1})$. 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, $A^\dagger = \begin{bmatrix} W^{-1} & L_w^{-1} H \\ Z U_w^{-1} & ZH \end{bmatrix}$. 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.

$$\begin{aligned} MPINV_r(m;n) = \mathrm{LQUP}(m;n) + AAT(r;m-r) + AAT(r;n-r) + 3\mathrm{TRSM}(r,m-r) \\ + 3\mathrm{TRSM}(r,n-r) + \mathrm{MM}(r) + \mathrm{LQUP}(r) + 2\mathrm{UTUT}(r) + 2\mathrm{INVT}(r) + \mathrm{UTLT}(r) \\ + R(n-r;r;m-r) \qquad (20) \end{aligned}$$

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 [8] as faster than SVD or QR or iterative methods would require $3m^2n + 2r^2m + 3r^3$ flops.

6.5.6 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 Xeon architecture according to the bad behavior of "dtrsm" function which is the main routine used by LAPACK for triangular

hal-00018223, version 4 - 14 Jan 2009

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.

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	tri. inv	0.11s	0.70s	2.17s	9.21s	24.21s	35.53s	49.95s	68.26s
GOTO	tri. inv	0.10s	0.62s	1.90s	8.00s	20.97s	30.77s	43.38s	58.98
	dtrtri	0.18s	1.04s	2.90s	10.97s	26.85s	38.57s	52.93s	70.95s
	$\frac{tri.inv}{dtrtri}$	**0.56**	**0.60**	**0.66**	**0.73**	**0.78**	**0.80**	**0.82**	**0.83**

Table 9: Timings of triangular matrix inversion on a Xeon, 3.6GHz

	n	*1000*	*2000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	tri. inv	0.19s	1.03s	3.02s	11.91s	31.71s	44.43s	61.37s	82.55s
	dtrtri	0.08s	0.58s	2.55s	11.39s	30.50s	44.52s	63.34s	85.19s
	$\frac{tri.inv}{dtrtri}$	**2.25**	**1.77**	**1.18**	**1.05**	**1.04**	**1.00**	**0.97**	**0.97**
GOTO	tri. inv	0.15s	0.85s	2.47s	10.10s	26.10s	38.29s	53.65s	72.74s
	dtrtri	0.08s	0.61s	1.96s	8.77s	23.68s	35.73s	49.84s	69.10s
	$\frac{tri.inv}{dtrtri}$	**1.90**	**1.40**	**1.26**	**1.15**	**1.10**	**1.07**	**1.08**	**1.05**

Table 10: Timings of triangular matrix inversion on Itanium2, 1.3GHz

Tables 9 and 10 illustrate the performances of our exact triangular matrix inversion regarding performances of LAPACK routine "dtrtri". Results show that our exact computations tend to catch up with the numerical ones and even outperform them on Itanium2 with ATLAS for large matrices (dimension greater than 8000).

One can notice that the implementation of triangular matrix inversion provided by GOTO is quite efficient compare to ATLAS, and thus lead our exact computation to be more efficient but not better than numerical ones. Here again, this demonstrates that exact triangular matrix inversion over finite field is not much more costly than its numerical counterpart.

Now, Tables 11 and 12 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

hal-00018223, version 4 - 14 Jan 2009

	n	*1000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	inverse	0.75s	13.57s	54.52s	141.19s	206.26s	285.19s	385.35s
	dgetrf+dgetri	0.69s	16.94s	80.83s	222.07s	368.66s	531.29s	761.28s
	$\frac{inverse}{dgetrf+dgetri}$	**1.09**	**0.80**	**0.67**	**0.64**	**0.56**	**0.54**	**0.51**
GOTO	inverse	0.63s	11.82s	48.56s	125.30s	179.17s	256.12s	343.91s
	dgetrf+dgetri	0.55s	13.02s	58.36s	159.21s	232.30s	328.55s	450.46s
	$\frac{inverse}{dgetrf+dgetri}$	**1.15**	**0.91**	**0.83**	**0.79**	**0.77**	**0.78**	**0.76**

Table 11: Timings of matrix inversion on a Xeon, 3.6GHz

	n	*1000*	*3000*	*5000*	*7000*	*8000*	*9000*	*10000*
ATLAS	inverse	1.01s	17.27s	69.24s	173.21s	256.67s	353.02s	483.08s
	dgetrf+dgetri	0.60s	14.29s	66.08s	184.74s	276.09s	393.62s	541.37s
	$\frac{inverse}{dgetrf+dgetri}$	**1.67**	**1.21**	**1.05**	**0.94**	**0.93**	**0.90**	**0.89**
GOTO	inverse	0.85s	14.92s	61.00s	153.78s	226.68s	313.84s	422.78s
	dgetrf+dgetri	0.47s	11.45s	51.33s	139.00s	207.36s	293.02s	402.72s
	$\frac{inverse}{dgetrf+dgetri}$	**1.80**	**1.30**	**1.19**	**1.11**	**1.09**	**1.07**	**1.05**

Table 12: Timings of matrix inversion on Itanium2, 1.3GHz

multiplication. Figures 7 and 8 show the correlation between matrix inversion performances and matrix multiplication performances; triangular and dense case are studied.

According to section 6.5.1, 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.

From section 6.5.2 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 algorithms involved in this application. In particular it involves three different reductions 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

hal-00018223, version 4 - 14 Jan 2009

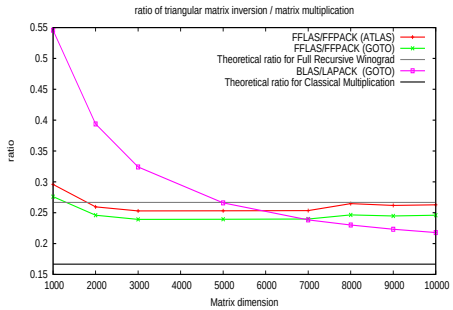

Figure 7: Comparing triangular matrix inversion with matrix multiplication on a Xeon, 3.6GHz

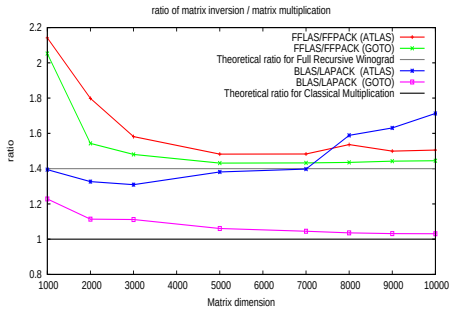

Figure 8: Comparing matrix inversion with matrix multiplication on a Xeon, 3.6GHz

role in performances as already demonstrated by ATLAS software with optimized BLAS [46]. 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.

7 Conclusions

We have achieved the goal of approaching the efficiency of the numerical linear algebra library but for word-size prime fields. We showed that exact computation can benefit from Winograd fast matrix multiplication algorithm and then even leads to outperform the efficiency of the well known BLAS and LAPACK libraries.

This performance is 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 problems (e.g. null-space, generalized inverse, etc.). 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 efficiency. By reusing BLAS library this has been proven to be almost useless for matrix multiplication in [17] and we think we proved here that this is not mandatory also for any dense linear algebra routine. Therefore, using recursive block algorithms, efficient numerical BLAS and fast matrix multiplication algorithms one can approach the numerical performance or even surpass them over some finite fields. Moreover, long range efficiency and portability are warranted as opposed to every day tuning. Except for small matrices where the conversions increase slightly the running time, and

hal-00018223, version 4 - 14 Jan 2009

hal-00018223, version 4 - 14 Jan 2009

except for the LQUP transform, we have shown that all our exact routines can be faster than their numerical counterparts.

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 (e.g. bounds 4.5 and 3.3). Those optimal bounds enable further uses of the BLAS routines.

Further developments include:

• The main 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. Here, a design using the finite field directly might improve the speed.

• More generally, a Self-adapting Software [10] would allow to provide hybrid implementations with best empirical thresholds.

• 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 then be an interesting way to provide similar results for larger finite fields.

• Finally, extending the out of core versions by more recursive data format and the building of 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 [19] for triangular system solving.

A Proof of theorem 3.1

Consider the natural block decomposition

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix},$$

where A_{11} and B_{11} have respectively dimension $m/2 \times k/2$ and $k/2 \times n/2$.

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 one of the intermediaite products. 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:

$$u_l \leq 0 \text{ and } v_l \geq 0 \tag{21}$$

$$u_l \text{ is decreasing and } v_l \text{ is increasing} \tag{22}$$

$$v_l > -u_l \text{ if } v_0 > -u_0 \tag{23}$$

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:

$$(2M_A - m_A)t_j + (2m_A - M_A)s_j = M_A t_{j+1} + m_A s_{j+1} = v_{j+1}^A \tag{24}$$

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 serie $(b_l)_{l>0}$ is increasing since (22).

Winograd's implementation, see e.g. [32, 21], uses the following intermediate computations

$$\begin{aligned}
P_1 &= A_{11} \times B_{11} \\
P_2 &= A_{12} \times B_{21} + \beta C_{11} \\
P_3 &= (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22} \\
P_4 &= A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) + \beta(C_{22} - C_{12} - C_{21}) \\
P_5 &= (A_{21} + A_{22}) \times (B_{12} - B_{11}) + \beta C_{12} \\
P_6 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12}) \\
P_7 &= (A_{11} - A_{21}) \times (B_{22} - B_{12}) + \beta(C_{22} - C_{12}) \\
C_{11} = U_1 &= P_2 + P_1 \\
U_2 &= (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} \\
U_3 &= A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} + \beta(C_{22} - C_{12}) \\
U_4 &= (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22}) + \beta C_{12} \\
C_{12} = U_5 &= U_4 + P_3 \\
C_{21} = U_6 &= U_3 - P_4 \\
C_{22} = U_7 &= U_3 + P_5
\end{aligned}$$

hal-00018223, version 4 - 14 Jan 2009

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 and U_7 always satisfy the bound. Now if the remaining 9 intermediate computations are bounded by b_l, we will be done.

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:

$$M^{j+1,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max\begin{pmatrix} M(P_1) = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,0,0} \\ M(P_2) = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B,m_C,M_C} \\ M(P_3) = M^{j,\frac{k}{2}}_{2m_A-2M_A,2M_A-2m_A,m_B,M_B,0,0} \\ M(P_4) = M^{j,\frac{k}{2}}_{m_A,M_A,2m_B-2M_B,2M_B-2m_B,m_C-2M_C,M_C-2m_C} \\ M(P_5) = M^{j,\frac{k}{2}}_{2m_A,2M_A,m_B-M_B,M_B-m_B,m_C,M_C} \\ M(P_6) = M^{j,\frac{k}{2}}_{2m_A-M_A,2M_A-m_A,2m_B-M_B,2M_B-m_B,0,0} \\ M(P_7) = M^{j,\frac{k}{2}}_{m_A-M_A,M_A-m_A,m_B-M_B,M_B-m_B,m_C-M_C,M_C-m_C} \end{pmatrix} \tag{25}$$

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

$$M^{0,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = \max\begin{pmatrix} M_AM_Bk+\beta M_C \\ -m_AM_Bk-\beta m_C \\ -M_Am_Bk-\beta m_C \end{pmatrix} \tag{26}$$

A.3 Some invariants

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

1. $0 \leq -m_A \leq M_A,\ 0 \leq -m_B \leq M_B,\ 0 \leq -m_C \leq M_C$

hal-00018223, version 4 - 14 Jan 2009

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 (25), 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

$$M^{j,k}_{m_A,M_A,m_B,M_B,m_C,M_C} = [v_j^A][v_j^B]\frac{k}{2^j}.$$

Suppose that the previous invariants are satisfied and that IH_j is true. We will prove that the maximum of (25) 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 and M_B are 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 $P_3 = (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22}$:

$$\begin{aligned} M(P_6) - M(P_3) &= v_{j+1}^A v_{j+1}^B - v_j^{A_{11}+A_{12}-A_{21}-A_{22}} v_j^{B_{22}} \\ &= v_{j+1}^A v_{j+1}^B - [(2M_A - 2m_A)t_j + (2m_A - 2M_A)s_j] v_j^B \\ &= v_{j+1}^A v_{j+1}^B - (v_{j+1}^A - m_A t_j - M_A s_j) v_j^B (24) \\ &= v_{j+1}^A [v_{j+1}^B - v_j^B] - u_j^A v_j^B \\ &\geq v_{j+1}^A [v_{j+1}^B - v_j^B] - v_{j+1}^A v_j^B (23) \\ &\geq v_{j+1}^A [v_{j+1}^B - 2v_j^B] \\ &\geq v_{j+1}^A 3^j [M_B - m_B] \geq 0 \end{aligned}$$

hal-00018223, version 4 - 14 Jan 2009

- 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,

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

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

$$\forall x \in U_2, |x| \leq \max \begin{pmatrix} (2M_A - m_A)(M_B - m_B) + 2M_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2m_AM_B \\ (-2m_A + M_A)(M_B - m_B) - 2M_Am_B \end{pmatrix} k/2^j \tag{27}$$

  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 27 simplifies into $\forall x \in U_2, |x| \leq (2M_A - m_A)(M_B - m_B) + 2M_AM_B$.

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

hal-00018223, version 4 - 14 Jan 2009

- 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

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

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:

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

- 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,

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

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 (25) and (26), 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.

hal-00018223, version 4 - 14 Jan 2009

- 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 $2^l d \leq k < 2^l(d+1)$ ($d = \lfloor \frac{k}{2^l} \rfloor$). A *dynamic peeling* technique [31] 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 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 $d \geq 1$)

hal-00018223, version 4 - 14 Jan 2009

- Let us suppose that it is satisfied for $l \geq 1$ and prove it for $l+1$:

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

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

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:

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

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:

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

hal-00018223, version 4 - 14 Jan 2009

The order of A_l and B_l is $k = 2^l$, so $\lfloor \frac{k}{2^l} \rfloor = 1$. Therefore, the computation of $A_l \times B_l$ with l recursive levels of Winograd's algorithm involves intermediate values equals to $v_l^{A_l} v_l^{B_l} \lfloor \frac{k}{2^l} \rfloor$. This proves the optimality of the bound.

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

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

[4] Morgan Brassel, Pascal Giorgi, and Clement Pernet. LUdivine: A symbolic block LU factorisation for matrices over finite fields using blas, April 2003. Poster, `http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/FFLAS/FFLAS_Download/ludivine_poster_eccad2003.ps.gz`.

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

[6] 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.

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

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

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

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

[11] 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. `http://doi.acm.org/10.1145/77626.79170`.

hal-00018223, version 4 - 14 Jan 2009

[12] 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.

[13] Jean-Guillaume Dumas, , Pascal Giorgi, and Clément Pernet. FFLAS-FFPACK: Finite field linear algebra subroutine/package. Software, `http://ciel.ccsd.cnrs.fr/ciel-00000025`, February 2006.

[14] 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.

[15] Jean-Guillaume Dumas. Q-adic transform revisited. Technical Report 0710.0510 [cs.SC], ArXiv, October 2007. http://hal.archives-ouvertes.fr/hal-00173894.

[16] 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.

[17] 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.

[18] Jean-Guillaume Dumas, Pascal Giorgi, and Clément Pernet. FFPACK: Finite field linear algebra package. In Jaime Gutierrez, editor, *Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, Santander, Spain*, pages 119–126. ACM Press, New York, July 2004.

[19] Jean-Guillaume Dumas, Clément Pernet, and Jean-Louis Roch. Adaptive triangular system solving. In *Challenges in Symbolic Computation Software*, October 2006. Dagstuhl Seminar proceedings 06271, paper 770.

[20] 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.

[21] Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. Technical report, arXiv:0707.2347v2, August 2007. `http://arxiv.org/abs/0707.2347v2`.

hal-00018223, version 4 - 14 Jan 2009

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] Kazushige Goto and Robert van de Geijn. On reducing tlb misses in matrix multiplication. Technical Report TR-2002-55, University of Texas, November 2002. FLAME working note #9.

[29] 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.

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

[31] 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. `http://doi.acm.org/10.1145/369028.369096`.

[32] Steven Huss-Lederman, Elaine M. Jacobson, Jeremy R. Johnson, Anna Tsao, and Thomas Turnbull. Strassen's algorithm for matrix multiplication : Modeling analysis, and implementation. Technical report, Center for Computing Sciences, November 1996. CCS-TR-96-17.

hal-00018223, version 4 - 14 Jan 2009

[33] 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.

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

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

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

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

[38] 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.

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

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

[41] Clément Pernet. Implementation of Winograd's matrix multiplication over finite fields using ATLAS level 3 BLAS. Technical Report RR011122, Laboratoire Informatique et Distribution, July 2001. `http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/FFLAS/FFLAS_Download/FFLAS_technical_report.ps.gz`.

[42] 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.

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

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

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

hal-00018223, version 4 - 14 Jan 2009

[46] 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. `http://www.netlib.org/utk/people/JackDongarra/PAPERS/atlas_pub.pdf`.

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

hal-00018223, version 4 - 14 Jan 2009