Finite Field Linear Algebra Subroutines

Jean-Guillaume Dumas — Thierry Gautier — Clément Pernet

Laboratoire de Modélisation et Calcul,
50 av. des Mathématiques.
B.P. 53 X, 38041 Grenoble, France.

APACHE group, ID-ENSIMAG Montbonnot.
ZIRST - 51, av. Jean Kuntzmann,
38330 Montbonnot Saint-Martin, France.

Jean-Guillaume.Dumas@imag.fr — Thierry.Gautier@inrialpes.fr — Clement.Pernet@imag.fr

ABSTRACT

In this paper we study different implementations of finite field arithmetic, essential foundation of computer algebra. We focus on Galois fields of word size cardinality at most, with any characteristic. Classical representations as machine integers, floating point numbers, polynomials and Zech logarithms are compared. Furthermore, very efficient implementations of finite field dot products, matrix-vector products and matrix-matrix products (namely the symbolic equivalent of level 1, 2 and 3 BLAS) are presented. Our implementations have many symbolic linear algebra applications: symbolic triangularization, system solving, exact determinant computation, matrix normal form are such examples.

Keywords

Galois Finite fields; BLAS level 1-2-3; Winograd's symbolic matrix multiplication

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 [25], integer system solving [3], integer matrix normal forms [7] or integer determinant [16]. Finite fields are also of intrinsic use, especially in cryptology (for instance for large integer factorization [18], discrete logarithm computations [20]) 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 subroutines 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 almost as fast as numerical matrix multiplication** (only 5% loss) when using word size finite fields.

Our aim is *not* to rebuild some specialized routines for each field instance. Instead, the main idea is to use a very efficient and automatically tuned numerical library as a kernel (namely ATLAS [24]) and to make some conversions in order to perform an *exact* matrix multiplication (i. e. *without any loss of precision*). The performances will be reached by performing as few conversions as possible. Moreover, when dealing with symbolic computations, fast matrix multiplication algorithms, such as Strassen's or Winograd's variant [8], do not suffer from instability problems. Therefore their implementation can only focus on high efficiency.

The source code for the routines is available on our web page: `www-lmc.imag.fr/lmc-mosaic/Jean-Guillaume.Dumas/FFLAS`.

The paper is organized as follows. Section 2 deals with the choice of data structures to represent elements of a finite field and with different ways to implement the basic arithmetic operations. Then section 3 present efficient ways to *generically* implement dot product, matrix-vector product and matrix multiplication over prime fields (finite fields with prime cardinality), including a study of fast matrix multiplication. Section 4 extends the results obtained for matrix multiplication to any finite field, showing effective gain for many finite fields with non-prime cardinality.

2. Prime field arithmetic

In this section, we briefly present 3 different possible implementations for prime fields of size and characteristic a prime number p: $\mathbb{Z}/_{p}\mathbb{Z}$ (also denoted by $\mathbb{Z}_p$).

Our objective is to speed up the computation of the basic linear algebra subroutines by testing the best implementation for this kind of algorithms. In theory, some deep knowledge of the architecture is mandatory to provide the best implementation. For instance, data movement between the hierarchies of memory inside a modern computer are to be

taken into account. Two kinds of optimization can be done: the first one is to try to program our implementation for a specific processor at a very low level; the second one is to try to program our implementation at a high level with a well suited abstraction of the computer. The memory hierarchy can then be taken care of. This is achieved by blocking or by reusing existing good implementations, such as ATLAS.

The first general technique used in our implementations is to try to reduce number of costly instructions (such as integer division) by grouping instructions. For instance computing the axpy operation $r = a * b + c$ modulo p could involve only one division (instead of two) if the intermediate result fit in a machine integer. This technique is also used at a higher level for dot products and matrix-vector products (see next section).

The second general technique is based on using block algorithms such as matrix product. The benefits of using those algorithms are to increase the locality of computation. Indeed, some data is fetch from the main memory to the cache of the processor and can therefore be reused faster as long as it remains in the cache [9]. This technique was used through the ATLAS library with our floating point representation.

Our choice is to make algorithmic optimizations on high level subroutines and try to reuse existing high performance implementations. We thus provide a good portability together with high performances.

2.1 Implementation with division

The classical way to implement finite field arithmetic is of course to use a system integer division by p after every operation. The library ALP [19], for instance, implements that kind of operation. This library will be denoted by *ALP*.

One can also add some tests to avoid many of the divisions. Another optimization is to implement the field addition by subtracting p to the result whenever needed instead of performing a division. We have implemented those optimizations. This implementation will be denoted by *Zpz*.

The maximum values for p are those for which multiplication cannot overflow. For instance, with 32-bits, p must be lower than 2^{16}.

2.2 Floating point implementation

Another option is to use floating point numbers. The NTL library [23] implements the field multiplication this way: $a * b \mod p = a * b - \lfloor a * b * \frac{1}{p} \rfloor * p$ where $\frac{1}{p}$ is precomputed at a double precision. This implementation will be denoted by *NTL*. In this case the NTL library stores operands as machine word integer and performs conversions between integer and floating point representations.

Here also, the maximum values for p are those for which multiplication cannot overflow.

2.3 Zech logarithm representation

Finally a representation by indexes can be used. Indeed, the multiplicative group of the invertible elements in $\mathbb{Z}/_{p}\mathbb{Z}$ is cyclic. Therefore there exist at least one generator g such that all its powers are all the non-zero elements. The idea is then to store only the exponent as an index and perform all the operations on indexes. Then absolutely no system division is needed. For instance the field multiplication is just an index addition: $g^i * g^j = g^{i+j}$. Similarly the field addition is a combination of an index subtraction, a table look-up, and an index addition: $g^i + g^j = g^i * (1 + g^{j-i})$. We then need three precomputed tables, each one of size p. One table is needed for each way conversions (between indexes and elements) and one for the successors (in order to easily compute r such that $g^r = 1 + g^k$). For instance, MAGMA or AXIOM implements such a representation which is called the Zech representation [2, section 3.3], [10]. We have implemented our own version of this implementation and, as it is actually usable for any Galois field (not only when the cardinality is prime, see section 2.5), we will denote it by *GFq*.

Here, as we do only index additions, the theoretical maximal value for q is $2^{(m-1)}$, when m bits are available. However, the need of 3 tables of size q impose a memory limit: for $q = 2^{26}$, the space required to store the tables is already $768 Mbytes$. However, in this paper, we do not use memory reduction tricks that would slow down the computations [13].

2.4 Elementary operations performances

We now compare these implementations operation by operation (AXPY is the combination in one call of a multiplication and an addition: $r = a * x + y$; AXPYIN is the same operation in place: $r+ = a*x$). We count the number of millions of field operations per second: *Mop/s*. This measure represents the speed of the algorithm on a specific machine, so it also gives an idea of the performance of each implementation when compared with the processor's frequency. In order to compare the possible implementations, we have compiled them all with the same compiler (gcc version 2.95.3 20010315) and the same optimization flags.

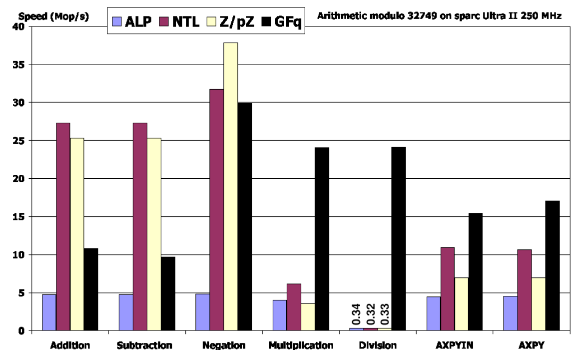

Figure 1: Elementary operation implementation comparison, modulo 32749

In figures 1 and 2, ALP and NTL denotes their implementations, *Zpz* is our classical implementation, quite similar to ALP, but with more tests in order to avoid as many divisions as possible, and *GFq* is our implementation using Zech representation. On figure 1, with an half word size field, ALP has the same performances for any basic operation.

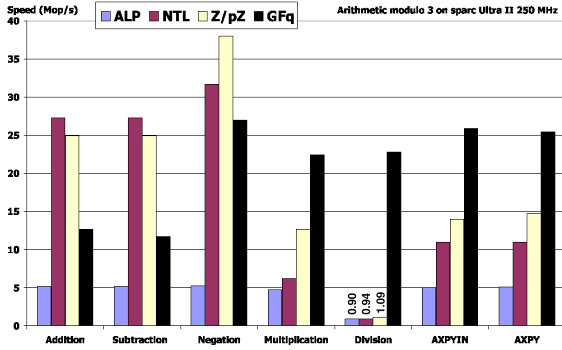

Figure 2: Elementary operation implementation comparison, modulo 3

Of course this is due to the system call to integer division. Then for the multiplication: between NTL and *Zpz* we see two things. On the one hand NTL's multiplication is better than *Zpz* when the modulus is big: on ultras, floating point arithmetic is faster than integer's and this compensates the casts. But on the other hand, when the modulus is very small, classical `unsigned int` division beats NTL back (this is unfortunately true only for very small primes like 2, 3, 5 and 7).

But the most impressive result on this figures is the fact that Zech division and multiplication are very fast, and Zech addition and subtraction are not so slow. We can thus show a speed-up of nearly 3 for the AXPY operation. For a very small modulus, when table look-up fits better in the cache, we see on figure 2 that this speed-up can grow up to a factor of 5 !

2.5 Extension to any word size Galois fields

A classical way to build extension of prime fields is to find an irreducible polynomial Q over $\mathbb{Z}/_{p}\mathbb{Z}$ of degree k. Then $\mathbb{Z}/_{p}\mathbb{Z}[X]/Q$ is a field, namely the Galois field of cardinality p^k, $\texttt{GF}(p^k)$. The implementation of operations in such a field is then just classical polynomial arithmetic with polynomial division by Q.

Now, as long as p^k does not exceed word size, there is a faster way. Pick Q as a primitive polynomial instead. A primitive polynomial has the special property that its roots are generators of the field $\texttt{GF}(p^k)$. There exists tables of such polynomials [12] and, anyway, about $\frac{1}{12k}$ of the polynomials of degree k in $\mathbb{Z}/_{p}\mathbb{Z}$ are primitive [5, proposition 3.3.7]. Therefore a random search is sufficient to find one easily. Then, like in the prime field case, one has just to compute the X^i mod Q in $\mathbb{Z}/_{p}\mathbb{Z}[X]$ to generate the field and the conversion tables between indexes and field elements. Then, the implementation of the arithmetic operators is *exactly* the Zech implementation, much faster than polynomial arithmetic.

3. Prime field BLAS

The other way to accelerate finite field performances is to specialize the high level algorithms. In order to not rewrite every single one, it is desirable to dispose of some basic linear algebra subroutines. This section is devoted to the construction of efficient dot product, matrix-vector product and matrix-matrix multiplication over prime fields. All the experiments presented in this section have been conducted on an Intel Pentium III, 735 MHz and a DEC alpha 500 MHz, so as to prove the portability of the routines, and their independence towards the cache size. The figures 5 & 6, by showing a similar global behaviour of the routines, highlight this portability. In both cases, we used gcc version 3.0.2 20010922 with the adequate optimisation flags for each architecture.

3.1 Dot product

The first idea to specialize dot product is to use a classical representation and make several multiplications and additions before performing the division. Indeed, one needs to perform a division only when the intermediate result is able to overflow. If the available mantissa is of m bits and the modulo is p, this happens at worst every λ multiplications where λ verifies the following condition:

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

We tested this idea on a vector of size 512 with 32 bits (`unsigned long` on PIII, `int` on alpha) and 64 bits (`unsigned long long` on PIII, `unsigned long` on alpha).

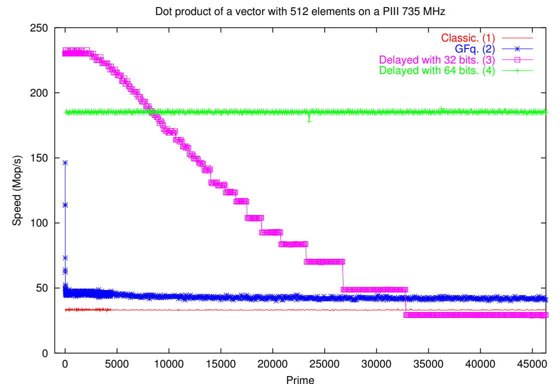

Figure 3: Improvement of dot product by delayed division, on a PIII

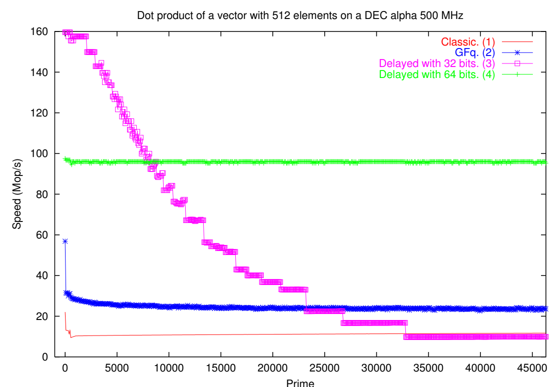

Figure 4: Improvement of dot product by delayed division, on an alpha

On figures 3 & 4, we can see that with this size of vector, when the prime is very small, only one division is performed for the whole dot product computation, thus yielding peak performances around 230 Mop/s.

On the one hand, this speed is then decreasing when the prime increases as more divisions have to be performed. The steps that one can see on curve (3) reflects the switch between one division every λ multiplication and one division only every $\lambda + 1$ multiplication. On the other hand, when used with 64 bits, the speed does not drop and remains around 185 Mop/s. This is because the smallest vector size for which 2 divisions are mandatory is bigger than 10^8 !

Note that for the primes bigger than 33000, the delayed method is slower than the classical one. This difference is due to the additional tests needed to ascertain wether a division has to be done or not.

3.2 Matrix-vector product

The idea here is to compute n dot products with delayed division. The performances are identical to those obtained for dot products. This algorithm performs n^2 arithmetic operations for n^2 data fetch from main memory, thus no blocking techniques could be used to improve locality and decrease the computation time.

3.3 Matrix multiplication

Here also, one could use delayed division in order to reach performances close to 200 Mop/s. Another idea is to take benefit of numerical libraries. Indeed, sufficiently well tuned numerical routines for matrix multiplication (BLAS level 3) can reach 600 Mop/s on a Pentium III 735 MHz [4]. Therefore, the goal of this section is to approach these performances. We chose ATLAS [24]. It is optimized by an AEOS (Automated Empirical Optimization of Software) for the machine it has been installed on, thus the BLAS are portable as well as very efficient.

3.3.1 Using numerical BLAS

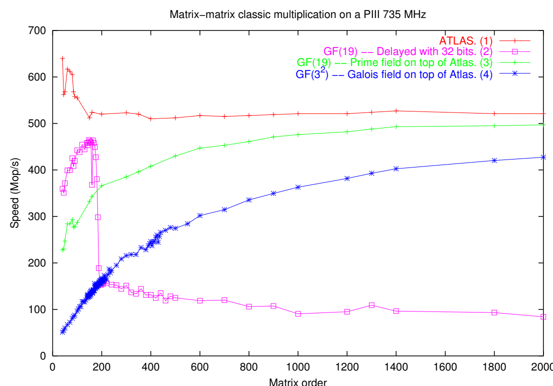

Figure 5: Speed comparison between matrix-matrix multiplications on a Pentium III 735 MHz.

The main idea is to convert the matrices from a finite field representation to a double precision representation. Then a call to the *numerical* BLAS is done and the result is converted back. The performance improvement of the BLAS is really worth these $3n^2$ conversions as shown on figures 5 and 6. However, we require a larger amount of memory in order to be able to perform the matrix conversions. This not a problem on recent architectures but, on the alpha, this prevents us from computing multiplication on matrices of size bigger than 1500.

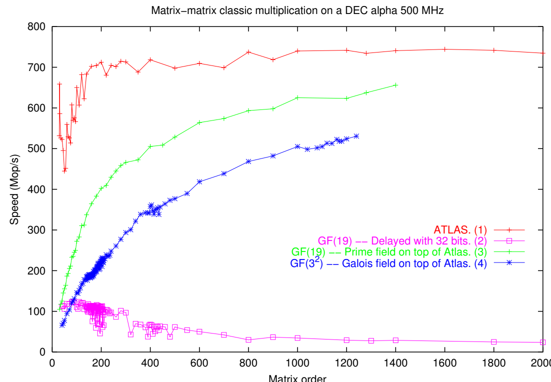

Figure 6: Speed comparison between matrix-matrix multiplications on a DEC alpha 500 MHz.

The fields chosen in the following sections are small ($\texttt{GF}(19)$ and $\texttt{GF}(3^2)$) so that the memory used for their tabulated representation do not interfere with the memory needed by the matrix algorithms. Of course, as we showed in section 2, the implementation gives the same timing results for bigger fields as long as enough memory is available.

Moreover, even though computed numerically, the results can still be *exact* as long as the computation over `double` does not overflow the 53 bits of the mantissa. Here also, if the two matrices $A \in \mathbb{Z}/_{\mathrm{p}}\mathbb{Z}^{M\times K}$ and $B \in \mathbb{Z}/_{\mathrm{p}}\mathbb{Z}^{K\times N}$ are to be multiplied, it is sufficient that $K(p-1)^2 < 2^{53}$, as long as the classical algorithm is used. For matrices which fit into the actual random access memories, this is always the case when the prime has less than 16 bits. Otherwise, the matrix is recursively cut in four blocks until the block size satifies the condition. A block algorithm is then applied over the base field.

In next section, we focus on the overflow bound when another kind of algorithm is used.

3.3.2 Winograd's Fast Matrix Multiplication

As our matrix-matrix multiplication is over finite fields, we do not suffer of any stability problem. Therefore, we only need to establish the best recursion level for the given entries and to control the overflow. With adapted data structures (e.g. recursive blocked data formats [11]), the results can then be of a very good acceleration of timing performances as demonstrated in [21]: for matrices of sizes close to 3000×3000, 19% of gain is expected. This is achieved with 3 Winograd's recursive steps using ATLAS, accelerating from around 500 Mop/s to an equivalent of more than 600 Mop/s on a PIII, 735 MHz. Figure 7 shows the gain obtained with several recursion levels.

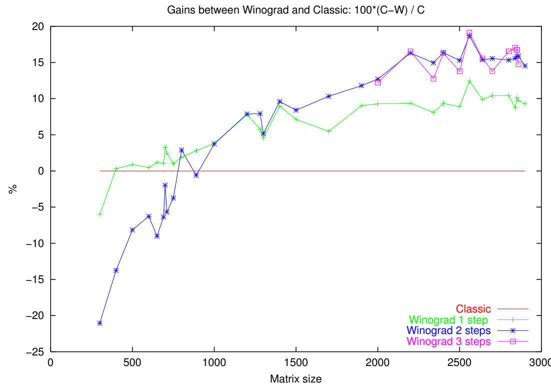

Figure 7: Winograd's fast matrix multiplication performances, on a Pentium III 735 MHz

Once again, the overflow control is made by insuring that every operation over `double` does remains within the 53 bits of mantissa.

Let $A \in \mathbb{Z}/_{p}\mathbb{Z}^{M \times K}$, $B \in \mathbb{Z}/_{p}\mathbb{Z}^{K \times N}$, be the two matrices to be multiplied. A classic multiplication requires $M \times N$ dot products of size K. For each one, the intermediate values are smaller than the biggest possible result, $K \times (p-1)^2$. But the Winograd's algorithm uses others intermediate computations which can go beyond this bound. More precisely, we have the following theorem giving us an optimal upper bound on the intermediate values of Winograd's algorithm.

THEOREM 3.1. *Let $A \in \mathbb{Z}^{M \times K}$ and $B \in \mathbb{Z}^{K \times N}$ where $0 \leq a_{i,j} < p$ and $0 \leq b_{i,j} < p$. Then, each value z involved in the computation of $A \times B$ using l recursion levels of Winograd's algorithm satisfies the following :*

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

Moreover, this bound is optimal.

This theorem is proved in appendix A. The idea is to prove by a double induction that the maximum possible values are always located in the same intermediate result. Then matrices for which some intermediate values reach the bound are produced.

The theorem shows that the loss in bits is linear on the number of recursion levels: when using Winograd with 1 to 4 levels of recursion the primes used must be smaller by respectively 0.5, 1.33, 2.31, and 3.36 bits. When considering that 4 levels of recursion seems to be sufficient for dense matrices of sizes up to 3500×3500, this is a minor loss: indeed, with $K = 3500$ and a 53 bits mantissa, the biggest possible prime with classical multiplication is 1604191. This drops to 156749 for a biggest prime when Winograd's is used.

3.3.3 *Blocking classical matrix multiplication*

We saw in section 3.3.1, that delaying the modulus computation when performing matrix multiplication is quite fast as long as matrices remains in the cache. Therefore, cutting big matrices into small blocks would allow us to keep the blocks in the cache as long as they are used and perform page faults only when dealing with other blocks. This way we can avoid the drop of performances (starting at a matrix size of around 200 for the PIII). This is what is done by the numerical BLAS. As we said before the problem is then tuning and portability. ATLAS gives an answer to those problems for numerical routines. Well, an alternative to our wrapping approach would be to rewrite ATLAS, but for finite fields. To see if this is of some use, we have implemented the cutting of matrices into small blocks and tried our implementations for different block sizes (32, 40, 48, 56, 64, 80 and 92). We have implemented this cutting only for matrices of size a multiple of the cutting. We show now that in fact there is no real need for blocking when compared to the wrapping of BLAS.

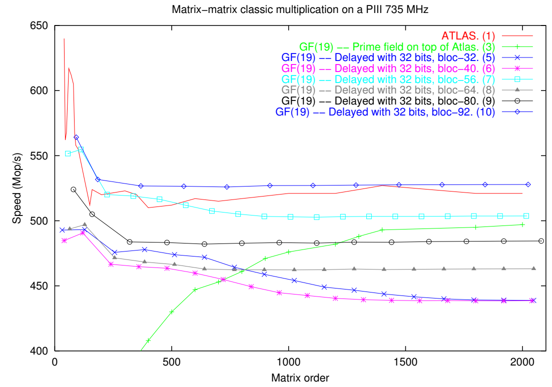

Figure 8: Blocking classical matrix multiplication, on a Pentium III 735 MHz.

Indeed, figure 8, shows two things. (1) This can be quite hard to find the best blocking parameter. (2) On Pentiums, there is some gain when small blocks are used. This is only a slight gain. Furthermore, when focusing on figure 9, we see that integer arithmetic on alpha's is far below floating point arithmetic. There, even blocking does not compete with our wrapping of BLAS.

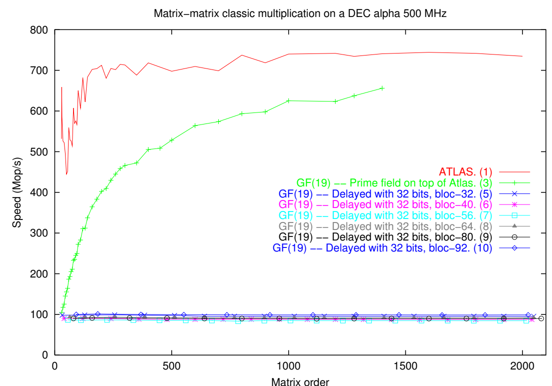

Figure 9: Blocking classical matrix multiplication, on a DEC alpha 500 MHz.

4. Extension to some small non-prime Galois fields

We want now to use our ATLAS based implementation with non-prime fields. The requirements are to be able to produce a coherent representation of $\texttt{GF}(p^k)$ with double precision numbers.

4.1 A $q-$adic representation

Following a new idea, by B. D. Saunders [22], we go back to the polynomial arithmetic but in a $q-$adic way, with q a sufficiently big prime or power of a single prime.

Suppose that $a = \sum_{i=0}^{k-1} \alpha_i X^i$ and $b = \sum_{i=0}^{k-1} \beta_i X^i$ are two elements of $\texttt{GF}(p^k)$ represented by $\mathbb{Z}/_{p}\mathbb{Z}[X]/Q$ as in section 2.5. One can perform the polynomial multiplication ab via $q-$adic numbers. Indeed, by setting $\tilde{a} = \sum_{i=0}^{k-1} \alpha_i q^i$ and $\tilde{b} = \sum_{i=0}^{k-1} \beta_i q^i$, the product is computed in the following maner (we suppose that $\alpha_i = \beta_i = 0$ for $i > k-1$):

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

Now if q is big enough, the coefficient of q^i will not exceed q. In this case, it is possible to evaluate a and b as floating point numbers, compute the product of these evaluations, and convert back to finite field element, via a $q-$adic reconstruction, a division by p and a division by Q:

Algorithm 4.1. *Dot product over Galois fields via $q-$adic conversions to floating point numbers*

Input : – a field $\texttt{GF}(p^k)$ represented as polynomials mod p and mod Q, for Q a degree k irreducible polynomial over $\mathbb{Z}/_{p}\mathbb{Z}$.
– Two vectors v_1 and v_2 of n elements of $\texttt{GF}(p^k)$ each, as polynomials.
– a prime power q.
Output : – $R \in \texttt{GF}(p^k)$, with $R = v_1^T.v_2$.

Polynomial to $q-$adic conversion

1 : Set $\widetilde{v_1}$ and $\widetilde{v_2}$ to the floating point vectors of the evaluations at q of the elements of v_1 and v_2.
{Using Horner's formula, for instance}

One computation

2 : Compute $\tilde{r} = \widetilde{v_1}^t \widetilde{v_2}$

Building the solution

3 : $\tilde{r} = \sum_{l=0}^{2k-2} \widetilde{\gamma_l} q^l$.
{Using radix conversion, see [8, Algorithm 9.14] for instance}
4 : For each l, set $\gamma_l = \widetilde{\gamma_l} \mod p$
5 : set $R = \sum_{l=0}^{2k-2} \gamma_l X^l \mod Q$

Theorem 4.2. *Let m be the number of available mantissa bits within the machine floating point numbers. If*

$$q > nk(p-1)^2 \text{ and } (2k-1)\log_2(q) < m,$$

then Algorithm 4.1 is correct.

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

Typically, $m = 53$ for 64-bits double precision; in that case, the biggest implementable fields are $\texttt{GF}(2^8)$, $\texttt{GF}(3^6)$, $\texttt{GF}(7^4)$, $\texttt{GF}(23^3)$ and $\texttt{GF}(317^2)$, with $n = 1$. Table 1 in appendix B gives the biggest possible block size for different fields, and the associated prime for the $q-$adic decomposition. Of course a highest block order of 1 is of absolutely no interest in practice. However, on the Pentium, for instance, as soon as the size approaches 200, the conversion cost will be compensated by the cache effects. We can see this for curve (4) on figure 5.

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

4.2 Implementation optimizations

For the performances, a first naïve implementation would only give limited speed-up as the conversion cost is then very expensive. However, some simple optimizations can be done:

(1) The prime power q can be chosen to be a power of 2. Then the Horner like evaluation of the polynomials at q (line 1 of algorithm 4.1) is just a left shift. One can then compute this shift with exponent manipulations in floating point arithmetic and use then native C++ << operator as soon as values are within the 32 bits range, or use the native C++ << on 64 bits when available. This choice also speeds up the radix reconversion (line 3 of algorithm 4.1).

(2) Some *sparse* primitive polynomials modulo p can be chosen to build $\texttt{GF}(p^k)$. Then the division (line 5 of algorithm 4.1) can be simplified. The idea is to consider primitive trinomials, or when generating is not the bottleneck, irreducible binomials. Indeed at least one of those exists for nearly every prime field [1, Theorem 1]. In this case, we suppose that $Q = X^k + \beta X^t + \alpha$ is primitive over $\mathbb{Z}/_{p}\mathbb{Z}$

with $t < \frac{k}{2}$ (this is not a restriction since whenever Q is primitive, its reciprocal is primitive also and no primitive polynomial is self reciprocal [17, Theorem 3.13]). Now, as we are working in $\mathbb{Z}/_p\mathbb{Z}[X]/Q$, we want to compute the division of $W = \sum_{l=0}^{2k-2} \gamma_l X^l$ by Q. Therefore, we first split W into its higher and lower parts: $W = HX^k + L = H(-\beta X^t - \alpha) + L \mod Q$. We then pursue by splitting H also into its higher and lower parts: $H = H_h X^{k-t} + H_l$, leading to $W = -\beta H_h X^k - \beta H_l X^t - \alpha H + L$ which is also $W = \beta^2 H_h X^t + \alpha\beta H_h - \beta H_l X^t - \alpha H + L \mod Q$. We conclude by using the fact that W is of degree at most $2k-2$, so that H_h is of degree less than $t-1$. This proves that the expression above is of degree strictly less than k and that it is exactly the remainder of the division of W by Q. A careful counting of the operations, taking advantage of the respective degrees of the terms, would also show that this computation requires less than $5k$ field operations. That way, the division can be replaced by the equivalent of two and a half simple polynomial subtractions ! This speed up is even better when β is zero, because then the division is only one subtraction: $-\alpha H + L$.

As shown on figures 5 and 6, with those optimization we can reach very high peak performances, quite close to those obtained with prime fields, namely 420 Mop/s on the PIII, 735 MHz, and more than 500 Mop/s on the DEC alpha 500 MHz !

5. Conclusion

We have achieved the goal of approaching the speed of numerical routines but for finite fields. To reach these performances one could use blocks that fit the cache dimensions of a specific machine. We proved that this is not mandatory. By a simple wrapping of a portable and efficient library for numerical BLAS, one can reach the same performances. Therefore, long range efficiency is guaranteed, as opposed to every day tuning, with maybe only 1 or 2 % loss.

Moreover, we have designed and implemented a generic version of Winograd's fast matrix multiplication. We are able to attain speed-ups of 25% for any underlying arithmetic: for instance, on a PIII 735 MHz, we were able to go from 500 Millions of finite field arithmetic operations per second for classical matrix multiplication to an equivalent of more than 600 Mop/s, and for numerical computations we were able to go from 600 Millions of floating point operations per second to an equivalent of more than 800 Mop/s (see [21]).

There remains to wrap the other BLAS routines. For instance, by using specially adapted sequential ([15]) or parallel ([6]) block LU symbolic algorithms we will surely be able to attain very good performances also for triangularizations.

Now for the applications. Many algorithms have been designed to use matrix multiplication in order to be able to prove an optimal theoretical complexity, but in practice those algorithms are not used. An example is the fast computation of determinants by way of Kaltofen-Villard's algorithm [16]. We believe that with our kernel, those optimal algorithms can also be the most efficient.

6. REFERENCES

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

[2] W. Bosma, J. J. Cannon, and A. Steel. Lattices of compatibly embedded finite fields. *Journal of Symbolic Computations*, 24(3–4):351–370, Sept.–Oct. 1997. Computational algebra and number theory (London, 1993).

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

[4] J. J. Dongarra, J. D. Croz, S. Hammarling, and I. Duff. A set of level 3 Basic Linear Algebra Subprograms. *Transactions on Mathematical Software*, 16(1):1–17, Mar. 1990. `www.acm.org/pubs/toc/Abstracts/0098-3500/-79170.html`.

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

[6] J.-G. Dumas and J.-L. Roch. A fast parallel block algorithm for exact triangularization of rectangular matrices. In *SPAA'01. Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures, Kreta, Greece.*, pages 324–325, July 2001.

[7] J.-G. Dumas, B. D. Saunders, and G. Villard. On efficient sparse integer matrix Smith normal form computations. *Journal of Symbolic Computations*, 32(1/2):71–99, July–Aug. 2001.

[8] J. v. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 1999.

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

[10] J. Grabmeier and A. Scheerhorn. Finite fields in AXIOM. Technical Report TR7/92 (ATR/5) (NP2522), The Numerical Algorithm Group, Downer's Grove, IL, USA and Oxford, UK, 1992.

[11] 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.

[12] T. Hansen and G. L. Mullen. Primitive polynomials over finite fields. *Mathematics of Computation*, 59(200):639–643, S47–S50, Oct. 1992.

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

[14] S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm for matrix multiplication. In ACM, editor, *Supercomputing '96 Conference Proceedings: November 17–22, Pittsburgh, PA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. ACM Press and IEEE Computer Society Press. www.supercomp.org/sc96/proceedings/SC96PROC/-JACOBSON/INDEX.HTM.

[15] O. H. Ibarra, S. Moran, and R. Hui. A generalization of the fast LUP matrix decomposition algorithm and applications. *Journal of Algorithms*, 3(1):45–56, Mar. 1982.

[16] E. Kaltofen and G. Villard. On the complexity of computing determinants. In *Proceedings of the Fifth Asian Symposium on Computer Mathematics ASCM 2001*, volume 9 of *Lecture Notes Series on Computing*, pages 13–27, Singapore, Sept. 2001.

[17] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications.* Cambridge University Press, revised edition, 1994.

[18] P. L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In L. C. Guillou and J.-J. 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.

[19] B. Mourrain and H. Prieto. *The ALP reference manual: A framework for symbolic and numeric computations*, 2000. www-sop.inria.fr/saga/logiciels/ALP.

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

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

[22] B. D. Saunders. Personal communication, 2001.

[23] V. Shoup. NTL 5.2: A library for doing number theory, 2002. www.shoup.net/ntl.

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

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

APPENDIX

A. Proof of theorem 3.1

THEOREM 3.1 Let $A \in \mathbb{Z}^{M\times K}$ and $B \in \mathbb{Z}^{K\times N}$ where $0 \leq a_{i,j} < p$ and $0 \leq b_{i,j} < p$. Then, each value z involved in the computation of $A \times B$ using l recursion levels of Winograd's algorithm verifies the following :

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

Moreover, this bound is optimal.

PROOF. In this proof will start by recalling briefly the principle of Winograd's algorithm. Then we will show that the maximal possible intermediate values always appear in the same intermediate block, and prove the bound for K a multiple of a power of 2. We then extend the result to any K and end the proof by producing matrices for which some intermediate values actually reach the bound.

A.1 Winograd's variant of Strassen's algorithm

First, let's remind us briefly the principle of Winograd's algorithm: at each recursion level, the following 22 tasks (given in [8, Algorithm 12.1]) are computed, with recursive calls for the matrix products. A ground level is arbitrary chosen to stop the recursion, and then, calls to the classical matrix multiplication are made. These tasks are:

- 8 Pre-Additions:

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

- 7 Multiplications called recursively:

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

- 7 Post-Additions:

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

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

Let us note that the result of the product is still lower than $K(p-1)^2$, since it is independent of the way it is computed. We can then only focus on the intermediate computations.

These one are, after some simplifications:

$$
\begin{array}{ll}
P1 = & A_{11} \times B_{11} \\
P2 = & A_{12} \times B_{21} \\
P3 = & (A_{12} + A_{11} - A_{21} - A_{22}) \times B_{22} \\
P4 = & A_{22} \times (B_{22} + B_{11} - B_{21} - B_{12}) \\
P5 = & (A_{21} + A_{22}) \times (B_{12} - B_{11}) \\
P6 = & (A_{21} + A_{22} - A_{11}) \times (B_{22} + B_{11} - B_{12}) \\
P7 = & (A_{11} - A_{21}) \times (B_{22} - B_{12}) \\
U2 = & (A_{21} + A_{22} - A_{11}) \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} \\
U3 = & A_{22} \times (B_{22} - B_{12}) + (A_{21} + A_{22}) \times B_{11} \\
U4 = & (A_{21} + A_{22}) \times B_{22} + A_{11} \times (B_{12} - B_{22})
\end{array}
$$

We want to find which one of those 10 intermediate matrices, can yield the biggest intermediate value. We consider l recursion levels. The recursion step $j = l$ is the cutting of the $M \times K$ and $K \times N$ matrices in 4 blocks and the recursion step $(j = 0)$ is the switch to classical matrix multiplication. The work is a round-trip: we start by a cutting from $j = l$ to $j = 0$ and then perform the multiplications from $j = 0$ to $j = l$.

Notations A.1.

- $M^{j,k}_{m_A,M_A,m_B,M_B}(X)$ *is an upper bound on all the numbers involved in the computation of the intermediate result* $X = A \times B$ *with* j *Winograd's recursion levels where all the input coefficients are between* m_A *and* M_A *for A and between* m_B *and* M_B *for B. The dimensions of A and B are respectively of sizes* $m \times k$ *and* $k \times n$.

- $M^{j,k}_{m_A,M_A,m_B,M_B}$ *is an upper bound on all the* $M^{j,k}_{m_A,M_A,m_B,M_B}(X)$ *for the 10 possible* X. *Clearly* $M^{j,k}_{m_A,M_A,m_B,M_B} = \max_X M^{j,k}_{m_A,M_A,m_B,M_B}(X)$.

For instance, in the P6 product, we have

$$
M^{j,k}_{m_A,M_A,m_B,M_B}(P6) = M^{j-1,\frac{k}{2}}_{2m_A-M_A,2M_A-m_A,2m_B-M_B,2M_B-m_B}. \quad (3)
$$

We will show next that all the biggest possible intermediate values actually appear in the P6 product. We will therefore need the following results on some $2u - v$ series.

A.2 Useful considerations about $2u - v$ series

Let us consider the two series:

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

Since

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

It comes that

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

We can directly deduce the following properties:

$$
u_l \leq 0 \text{ and } v_l \geq 0 \quad (4)
$$

$$
u_l \text{ is decreasing and } v_l \text{ is increasing} \quad (5)
$$

$$
v_l > -u_l \text{ whenever } v_0 > -u_0 \quad (6)
$$

Let's define v^A and v^B two series of the v kind. All we need are u_0, v_0 for both; the associated u series can just be virtual. We set their initial values to $u^A_0 = m_A$,$v^A_0 = M_A$,$u^B_0 = m_B$ and $v^B_0 = M_B$, with m_A, M_A, m_B and M_B as in notations A.1. Let's also define $t_j = \frac{1+3^j}{2}$ and $s_j = \frac{1-3^j}{2}$, so that $t_j + s_j = 1$ and $t_j - s_j = 3^j$.

Note that we have the following property:

$$
(2M_A - m_A)t_j + (2m_A - M_A)s_j = M_A t_{j+1} + m_A s_{j+1} = v^A_{j+1} \quad (7)
$$

We are now ready to prove by induction that P6 contains the maximal possible values.

A.3 Main induction, for $K = \lambda 2^l$

We consider the following induction hypothesis IH_j, where the induction is on j, the number of Winograd's recursion levels:

If $m_A \leq 0 \leq M_A$, $m_B \leq 0 \leq M_B$, $|m_A| \leq |M_A|$ and $|m_B| \leq |M_B|$ we have

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

• When $j = 0$, the classic matrix multiplication is used, so we have

$$
\begin{aligned}
M^{0,k}_{m_A,M_A,m_B,M_B} &= k \cdot max(|m_A|, |M_A|) \cdot max(|m_B|, |M_B|) \\
&= k M_A M_B
\end{aligned}
$$

And IH_0 is true.

• Let's now consider that IH_j is true, and prove IH_{j+1}

$$
M^{j+1,k}_{m_A,M_A,m_B,M_B} = \max_{P_1 \dots P_7, U_2 \dots U_4}
\begin{pmatrix}
M(P_1) & = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B} \\
M(P_2) & = M^{j,\frac{k}{2}}_{m_A,M_A,m_B,M_B} \\
M(P_3) & = M^{j,\frac{k}{2}}_{2m_A-2M_A,2M_A-2m_A,m_B,M_B} \\
M(P_4) & = M^{j,\frac{k}{2}}_{m_A,M_A,2m_B-2M_B,2M_B-2m_B} \\
M(P_5) & = M^{j,\frac{k}{2}}_{2m_A,2M_A,m_B-M_B,M_B-m_B} \\
M(P_6) & = M^{j,\frac{k}{2}}_{2m_A-M_A,2M_A-m_A,2m_B-M_B,2M_B-m_B} \\
M(P_7) & = M^{j,\frac{k}{2}}_{m_A-M_A,M_A-m_A,m_B-M_B,M_B-m_B} \\
M(U_2) & \\
M(U_3) & \\
M(U_4) &
\end{pmatrix}
$$

We will now prove that this maximum is reached by P_6.

One clearly sees that the condition on the indices m_A, M_A, m_B and M_B of the induction hypothesis relation are satisfied

by each expression inside the max parenthesis. So we can apply this relation in order to compare each $M^{j+1,k}_{m_A,M_A,m_B,M_B}(X)$ with $M^{j+1,k}_{m_A,M_A,m_B,M_B}(P_6)$. For aesthetic reasons, we will simplify the expression of $M^{j+1,k}_{m_A,M_A,m_B,M_B}(X)$ with the notation $M(X)\cdot\frac{k}{2^{j+1}}$

1. We start with $P_1 = A_{11}\times B_{11}$.

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

Now v^A and v^B are increasing and positive, therefore

$$M(P_6)-M(P_1)\geq 0$$

2. The same argument holds for $P_2 = A_{12}\times B_{21}$ also.

3. P_3 is $(A_{12}+A_{11}-A_{21}-A_{22})\times B_{22}$, so:

$$\begin{array}{rcll}(B) & = & M(P_6)-M(P_3) & \\ & = & v_{j+1}^A v_{j+1}^B - v_j^{A_{11}+A_{12}-A_{21}-A_{22}}v_j^{B_{22}} & \\ & = & v_{j+1}^A v_{j+1}^B - [(2M_A-2m_A)t_j+ & \\ & & (2m_A-2M_A)s_j]v_j^B & \\ & = & v_{j+1}^A v_{j+1}^B - (v_{j+1}^A - m_At_j - M_As_j)v_j^B & (7)\\ & = & v_{j+1}^A[v_{j+1}^B - v_j^B] + u_j^A v_j^B & \\ & \geq & v_{j+1}^A[v_{j+1}^B - v_j^B] - v_{j+1}^A v_j^B & (6)\\ & \geq & v_{j+1}^A[v_{j+1}^B - 2v_j^B] & \\ & \geq & v_{j+1}^A 3^j[M_B-m_B] & \\ & \geq & 0 & \end{array}$$

4. By applying the same argument to $P_4 = A_{22}\times(B_{22}+B_{11}-B_{21}-B_{12})$, it comes

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

5. P_5 is $(A_{21}+A_{22})\times(B_{12}-B_{11})$:

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

Both last terms being positive, we have $M(P_6)-M(P_5)\geq 0$.

6. Here, we can use P_5 to bound $P_7=(A_{11}-A_{21})\times(B_{22}-B_{12})$:

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

Lastly, the computation of $U_2=P_1+P_6$, $U_3=U_2+P_7$ and $U_4=U_2+P_7$ consists in fact in some matrix additions. That's why $M(U_i)$ must control the computation of each product involved in it *and* the result of the addition:

$$\left\{\begin{array}{l}M(U_2)=\max\{M(P_1),M(P_6),\frac{2^j}{k}\max_{U_2}|x|\}\\ M(U_3)=\max\{M(U_2),M(P_7),\frac{2^j}{k}\max_{U_3}|x|\}\\ M(U_4)=\max\{M(U_2),M(P_5),\frac{2^j}{k}\max_{U_4}|x|\}\end{array}\right.$$

We only need to prove that any x in $U_{2,3,4}$ is of value less than $M(P6)$.

7. Let's continue with U_2. Since its final expression is $(A_{21}+A_{22}-A_{11})\times(B_{22}-B_{12})+(A_{21}+A_{22})\times B_{11}$, we have :

$$\forall x\in U_2,\left\{\begin{array}{ll}x\leq & (2M_A-m_A)(M_B-m_B)+2M_AM_B\\ x\geq & \min\begin{pmatrix}(2m_A-M_A)(M_B-m_B)\\(2M_A-m_A)(m_B-M_B)\end{pmatrix}\\ & +\min\begin{pmatrix}2m_AM_B\\2M_Am_B\end{pmatrix}\end{array}\right.$$

Now

$$2m_A-M_A-(m_A-2M_A)=m_A+M_A\geq 0$$

So we have

$$|x|\leq\max\begin{pmatrix}(2M_A-m_A)(M_B-m_B)+2M_AM_B\\(-2m_A+M_A)(M_B-m_B)+2|m_A|M_B\\(-2m_A+M_A)(M_B-m_B)+2M_A|m_B|\end{pmatrix}$$

And since

$$2M_A-m_A-(-2m_A+M_A)=M_A+m_A\geq 0$$

It comes

$$\forall x\in U_2,|x|\leq(2M_A-m_A)(M_B-m_B)+2M_AM_B\quad(8)$$

There now remains to prove that

$$\begin{array}{rl}|x| & \leq M^{j,k}_{m_A,M_A,m_B,M_B}(P_6)\\ & \leq M^{j+1,k}_{m_A,M_A,m_B,M_B}(P_6)\end{array}$$

Well:

$$\begin{array}{rl} & (2M_A-m_A)(M_B-m_B)+2M_AM_B\\ & \quad-(2M_A-m_A)(2M_B-m_B)\\ = & (2M_A-m_A)(-M_B)+2M_AM_B\\ = & m_AM_B\\ & \leq 0\end{array}$$

And that's why $M(P_6)\geq M(U_2)$.

8. Now, we turn to $U_3=A_{22}\times(B_{22}-B_{12})+(A_{21}+A_{22})\times B_{11}$.

$$\forall x\in U_3,\left\{\begin{array}{ll}x\leq & M_A(M_B-m_B)+2M_AM_B\\ x\geq & \min\begin{pmatrix}m_A(M_B-m_B)\\M_A(m_B-M_B)\end{pmatrix}\\ & +\min\begin{pmatrix}2m_AM_B\\2M_Am_B\end{pmatrix}\end{array}\right.$$

Now

$$m_A\geq -M_A$$

So we have

$$|x| \leq \max \begin{pmatrix} M_A(M_B - m_B) + 2M_A M_B \\ M_A(M_B - m_B) + 2|m_A| M_B \\ M_A(M_B - m_B) + 2M_A|m_B| \end{pmatrix}$$

And then

$$\begin{aligned} |x| &\leq M_A(M_B - m_B) + 2M_A M_B \\ &\leq (2M_A - m_A)(M_B - m_B) + 2M_A M_B \\ &\leq M(U2) \leq M(P6) \end{aligned}$$

So that

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

9. Since the expression of U_4 is similar to the expression of U_3, the same argument can be applied to prove that $M(P_6) \geq M(U_4)$

We end the proof with:

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

so that the induction hypothesis IH_{j+1} is true.

• We then conclude, when $K = \lambda 2^l$, by replacing with $u^A_0 = u^B_0 = 0$, $v^A_0 = v^B_0 = p-1$, so that

$$M^{l,k}_{0,p-1,0,p-1} = [v^A_l][v^B_l]\frac{k}{2^l} = \left(\frac{1+3^l}{2}\right)^2 (p-1)^2 \left\lfloor \frac{K}{2^l} \right\rfloor$$

A.4 Generic case

There, we have $2^l d \leq k < 2^l(d+1)$ ($d = \lfloor \frac{K}{2^l} \rfloor$), a dynamic peeling [14] is used. It means that, at each recursion step, the even-sized left top corner block of A and B are given to the recursive algorithm, and then a fix-up is eventually computed, if one or more dimensions were odd-sized. Since this fix-up uses conventional matrix-vector, vector-vector and dot-product to compute the result of $A \times B$, all numbers involved in its computation are bounded by $k(p-1)^2$. Then we have

$$M^{l,k}_{0,p-1,0,p-1} = max(\left(\frac{1+3^l}{2}\right)^2 d \cdot 2^l(p-1)^2, k(p-1)^2)$$

Let's prove that this maximum is never reached by $k(p-1)^2$ Suppose

$$\left(\frac{1+3^l}{2}\right)^2 d \cdot 2^l(p-1)^2 < k(p-1)^2$$

Then

$$\left(\frac{1+3^l}{2}\right)^2 < 1$$

This is impossible as soon as $l \geq 1$.

Lastly

$$M^{l,k}_{0,p-1,0,p-1} = \left(\frac{1+3^l}{2}\right)^2 \left\lfloor \frac{K}{2^l} \right\rfloor (p-1)^2$$

There remains to prove that the bound is optimal. To do so, it is sufficient to produce matrices for which some intermediate values reach the bound.

A.5 Matrices with highest possible intermediate values

Since the matrices A_l and B_l induce a computation where the bound $M^{l,2^l}_{0,p-1,0,p-1} = \left(\frac{1+3^l}{2}\right)(p-1)^2$ is reached. $(A_l)_{l\in\mathbb{N}^*}$ and $(B_l)_{l\in\mathbb{N}^*}$ are defined as follows:

$$\begin{cases} A_1 = \begin{bmatrix} 0 & 0 \\ p-1 & p-1 \end{bmatrix}, & B_1 = \begin{bmatrix} p-1 & 0 \\ 0 & p-1 \end{bmatrix} \\ \\ A_{l+1} = \begin{bmatrix} \overline{A_l} & 0 \\ A_l & A_l \end{bmatrix}, & B_{l+1} = \begin{bmatrix} B_l & \overline{B_l} \\ 0 & B_l \end{bmatrix} \end{cases}$$

where $\overline{X_{i,j}} = (p-1) - X_{i,j}$

We used $P6 = (A_{21}+A_{22}-A_{11})\times(B_{22}+B_{11}-B_{12})$ to build the bound, since its computation involves the potentially greatest numbers, then we will note

$$S(A_l) = (A_l)_{2,1} + (A_l)_{2,2} - (A_l)_{1,1}$$

The definition of A_l implies

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

where $J_k \in \mathbb{Z}^{k\times k}$ and $(J_k)_{i,j} = p-1, \forall i = 1..k, j = 1..k$.

We have the identity $S(J_k) = J_{k-1}$. Therefore, by applying P_6 l times, and since S is linear, we get:

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

Then, as $S(A_1) = 2(p-1)$ and $J_1 = p-1$, we have:

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

The same argument, applied to B_l, shows that with l recursion levels, $S^l(B_l) = \frac{1+3^l}{2}(p-1)$ also.

Therefore, the computation of $A_l \times B_l$ involves at least one number whose value is $\left(\frac{1+3^l}{2}\right)^2(p-1)^2$; thus proving the optimality of the theorem.

□

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

Here are some of the galois fields implementable with our q-adic representation. In the tables, n_{max} is the biggest matrix size for which this field is usable without loss of precision and $q_b(n_{max})$ is the best prime power to use with this maximal size. Curve (4) on figures 5 (resp. 6), shows that a matrix size of more than 200 (resp. more than 100) is needed to get some speed-up. Therefore the q-adic representation is very interesting as long as n_{max} is bigger than that.

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

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

Table 1: Highest block order for some non-prime Galois fields, with a 53 bits mantissa

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

GF	n_{max}	$q_b(n_{max})$
2^2	1321119	2642239
2^3	2376	7129
2^4	140	563
2^5	27	137
2^6	8	53
2^7	4	29
2^8	2	19
2^9	1	13
3^2	330279	2642239
3^3	594	7129
3^4	35	563
3^5	6	137
3^6	2	53
3^7	1	29
5^2	82569	2642239
5^3	148	7129
5^4	8	563
5^5	1	137
7^2	36697	2642239
7^3	66	7129
7^4	3	563
11^2	13211	2642239
11^3	23	7129
11^4	1	563

GF	n_{max}	$q_b(n_{max})$
13^2	9174	2642239
13^3	16	7129
17^2	5160	2642239
17^3	9	7129
19^2	4077	2642239
19^3	7	7129
23^2	2729	2642239
23^3	4	7129
29^2	1685	2642239
29^3	3	7129
31^2	1467	2642239
31^3	2	7129
37^2	1019	2642239
37^3	1	7129
41^2	825	2642239
41^3	1	7129
47^2	624	2642239
47^3	1	7129
53^2	488	2642239
101^2	132	2642239
139^2	69	2642239
227^2	25	2642239
317^2	13	2642239
1129^2	1	2642239

Table 2: Highest block order for some non-prime Galois fields, with a 64 bits mantissa