Fast dot product over finite field

Jérémy Jean, Stef Graillat

▶ To cite this version:

Jérémy Jean, Stef Graillat. Fast dot product over finite field. 2010. hal-00450888

HAL Id: hal-00450888
https://hal.science/hal-00450888

Preprint submitted on 27 Jan 2010

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.

Fast dot product over finite field

Jérémy JEAN
UPMC/CNRS LIP6, PEQUAN Team
104, avenue du Président Kennedy
75016 Paris (France)
Jeremy.Jean@pequan.lip6.fr

Stef GRAILLAT
UPMC/CNRS LIP6, PEQUAN Team
104, avenue du Président Kennedy
75016 Paris (France)
Stef.Graillat@lip6.fr

ABSTRACT

Finite fields are widely used in numerous areas like cryptography, error-correcting codes or computer algebra. Dot products are ubiquitous in all computations especially when dealing with linear algebra. Developing fast libraries for computing dot products in finite fields is a key tool to tackle various problems in scientific computing. In this paper, our aim is to present several possibilities to use fast floating-point units for computing dot products in finite fields. The main concern is then to properly manage rounding errors that may appear during the computation. To solve this problem, we use error-free transformations (EFT). Using these EFT on recent processors (with an FMA), we show that it is possible to deal with large finite fields. We also compare our approach with Residue Number Systems (RNS), which is a modular approach. The RNS approach is presented using either integer arithmetic or floating-point arithmetic. Numerical experiments make it possible to compare the performances of these different approaches.

Keywords

Finite field, floating-point arithmetic, error-free transformations, FMA, RNS, GMP

1. INTRODUCTION

Let $p \geq 3$ be a prime, and $(a_i), (b_i)$ two vectors of N scalars of $\mathbb{Z}/p\mathbb{Z}$. We want to compute the dot product of a and b in $\mathbb{Z}/p\mathbb{Z}$,

$$a \cdot b = \sum_{i=1}^{N} a_i\, b_i \pmod{p}.$$

To do so, we will be using two different approaches relying on two different arithmetics. The first one uses floating-point arithmetic as described in the IEEE 754 standard [1] whereas the second one extends the classic integer arithmetic into a Residue Number System (RNS) to compute with smaller values.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

Because of physical limitations, one can not compute easily on arbitrary large values. Consequently, we set a limit on the order of the field $\mathbb{Z}/p\mathbb{Z}$. In [2], J.-G. Dumas introduced a similar idea but the general hypothesis was not strictly the same. He assumed the prime p of the field to be such that $\lambda(p-1)^2 < 2^M$, for $\lambda \in \mathbb{N}^*$ and M the size of the mantissa. This was used in the FFPACK and FFLAS libraries [3, 4, 5]. In our paper, we relax the square in this hypothesis, and consequently extend the range for the choice of primes.

The paper is organized as follows. In Section 2, we define the arithmetics used in dot product computation methods that are described in Section 3. Particularly, floating-point arithmetic requires the knowledge of basic notions of rounding (see 2.1.3) and some results we will be using in this paper: Sterbenz's subtraction (see 2.1.4) and error-free transformations (see 2.1.5). An introduction to RNS computation is presented in Section 2.2. This is nothing than a modular computation followed by a reconstruction with the Chinese Remainder Theorem. But it makes it possible to use numbers of small size and so use the integer units of the processor. In Section 3, we describe the different algorithms used to perform a dot product on finite field. Finally, in section 4, we give experimental timing results of all presented methods. The reference algorithm we will be comparing to is based on GMP [6], an open-source multi-precision library.

2. VARIOUS ARITHMETICS

2.1 Floating-point arithmetic

We use floating-point numbers in double precision to represent $\mathbb{Z}/p\mathbb{Z}$ integers. Denoting M the size of the double precision mantissa (53 bits according to the IEEE 754 standard), we limit p to

$$p - 1 < 2^{M-1}. \tag{1}$$

Any integer of the finite field could then be represented exactly by a floating-point number. The term $M-1$ is necessary rather than just M to be able to sum exactly at least two integers in the field without introducing a rounding error. In the sequel, we will assume the rounding mode to be directed *toward zero*. This is needed to ensure the error to be positive in applications of error-free transformations (see 2.1.5).

2.1.1 Notations

We denote by $\mathbb{F}$ the set of all floating-point numbers in the chosen precision. The operation $\circ \in \{+, -, *, /\}$ on

$a, b \in \mathbb{F}$ equals $a \circ b$ in $\mathbb{R}$ and is rounded to $\mathbf{fl}(a \circ b)$ in $\mathbb{F}$ (assume $b \neq 0$ if $\circ = /$). Application $\mathbf{fl}$ can thus be seen as a non-injective application on $\mathbb{R}$ into $\mathbb{F}$. For $x \in \mathbb{F}$, $\mathbf{ufp}(x)$ will be the unit in the first place of x and $\mathbf{ulp}(x)$ the unit in the last place of x [8]. We give proper definitions about **ulp** and **ufp** in the next section. We will refer to machine precision as $\mathbf{u} = 2^{-M+1}$, because we chose rounding toward zero.

2.1.2 Definitions

Definition 1. For floating-point numbers a and b and operation $\circ \in \{+, -, \times, /\}$, let $c = a \circ b$ exactly (assuming $b \neq 0$ if $\circ = /$). Let x and y be consecutive floating-point numbers with the same sign as c such that:

$$|x| \leq |c| < |y|.$$

Then, the floating-point arithmetic has the *faithful* property if $\mathbf{fl}(a \circ b) = x$ whenever $c = x$ and $\mathbf{fl}(a \circ b)$ is either x or y whenever $c \neq x$.

We then define the unit in the last place (**ulp**) and unit in the first place (**ufp**) of floating-point numbers. Unit in the last place is the gap between two very close floating-point numbers. To be exact: $\mathbf{ulp}(x)$ is the gap between the two finite floating-point numbers closest to the value x, even if x is one of them (Kahan's definition) [8].

As for the unit in the first place, we define it as follows:

$$\forall x \in \mathbb{R}, \quad \mathbf{ufp}(x) = \begin{cases} 0 & \text{if} \quad x = 0 \\ 2^{\lfloor \log_2 |x| \rfloor} & \text{if} \quad x \neq 0 \end{cases} \tag{2}$$

For $x \in \mathbb{R}$, the value $\mathbf{ufp}(x)$ denotes the weight of the most significant bit in the representation of x.

2.1.3 Rounding

Floating-point numbers of $\mathbb{F}$ form a subset of $\mathbb{R}$ and because of physical limitations, one can not represent an infinite amount of numbers like $\mathbb{R}$. What we can do though, is imagine $\mathbb{F}$ as a discrete set such that every real number x would have a faithful representation in $\mathbb{F}$. Clearly, $\mathbb{F}$ inherits from the classical ordering of $\mathbb{R}$ and leaving apart underflow and overflow particular situations, we have:

$$\forall x \in \mathbb{R}, \quad \exists (a, b) \in \mathbb{F}^2 \quad \text{s.t.} \quad a \leq x < b. \tag{3}$$

Following the IEEE 754 standard for double precision, our floating-point arithmetic is *faithful* (Definition 1), so that one needs to choose between a or b for the floating-point image $\mathbf{fl}(x)$ of x. Action of choosing is called *rounding* and introduces errors in computations.

In following Figure 1, rounding mode toward zero $\square(x)$ and to the nearest $\circ(x)$ are shown for a positive real number x. Note that in our case, rounding toward zero on positive numbers leads to a positive round-off term.

Generally speaking, any operations on floating-point numbers may introduce rounding errors. In the two next sections, we present a case where a subtraction can be done *exactly* and particular methods known as *error-free transformations* that make it possible to exactly compute the rounding error.

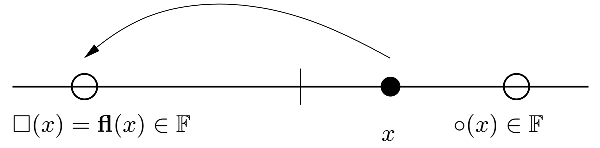

Figure 1: Rounding modes for $x \in \mathbb{R}^+$.

2.1.4 Sterbenz's result

Sterbenz suggested in [11] a particular case of subtraction where no error is introduced.

Theorem 1. If a and b are floating-point numbers of $\mathbb{F}$ such that:

$$\frac{b}{2} \leq a \leq 2b,$$

then the result computed by $\mathbf{fl}(a - b)$ is exact. That is:

$$\mathbf{fl}(a - b) = a - b.$$

Proof 1. Without loss of generality, assume $a > b > 0$. This implies $\mathbf{ulp}(a) \geq \mathbf{ulp}(b)$, so both a and b are in $\mathbf{ulp}(b)\mathbb{N}$, and so is $a - b$. The hypothesis of the theorem states $a \leq 2b$ so that $a - b \leq b$, hence $a - b$ is in $\mathbf{ulp}(b)\mathbb{N}$ but not larger than b. As a consequence, $a - b$ happens to be a floating-point number and $\mathbf{fl}(a - b) = a - b$. $\square$

In others cases, when numbers are *too far* from each other, one can not compute exactly their difference.

2.1.5 Error-free transformations

When it comes to multiply two floating-point numbers, a leak of precision arises naturally: the $2M$-bit product can not be stored as a single M-bit floating-point number. One though may use two of them, to virtually extend the precision. The product is thus not store in one piece, but as an unevaluated sum of two floating-point numbers. We refer to so called methods as *error-free transformations*, which can be extended to more complex operations than the product, as we develop in the following.

The following algorithm applies when computing a product of two positive floating-point numbers. We make use of a special instruction denoted *FMA* (*fused multiply-add*) to evaluate *exactly* the round-off term of the floating-point product. This operation had been included in the IEEE 754 standard in 2008 and performs

$$\text{FMA}(a, b, c) = a \times b + c$$

with only one rounding.

Algorithm 1 — TwoProduct

Require: $a, b \in \mathbb{F}$ such that $a, b \geq 0$
Ensure: $x \in \mathbb{F}$ and $y \in \mathbb{F}$ such that $ab = x + y$
  $x \leftarrow \mathbf{fl}(ab)$
  $y \leftarrow \text{FMA}(a, b, -x)$
  return (x, y)

Theorem 2. Let x and y be the result of **TwoProduct** applied to a and b.

We have: $ab = x + y$, $x = \mathbf{fl}(ab)$, $0 \leq x \leq ab$, $0 \leq y < \mathbf{u}.\mathbf{ufp}(x)$ and $0 \leq y < \mathbf{u}x$.

Proof 2. [1] Because a and b are floating-point numbers, each with M significant bits at most, their mathematical product ab requires at most $2M$ bits. When rounding ab down toward zero, one truncates the last M digits, so that the floating-point result $\mathbf{fl}(ab)$ and the exact mathematical difference $ab-\mathbf{fl}(ab)$ can be represented by two floating-point numbers with M digits each. Hence, the exact mathematical difference requires only M bits. Thus, $ab-\mathbf{fl}(ab)$ happens to be a floating-point number with at most M bits, which rounding to M bits leaves intact. Consequently, FMA produces the exact result $ab-\mathbf{fl}(ab)$:

$$y = \mathrm{FMA}\left(a, b, -\mathbf{fl}(ab)\right) = ab - \mathbf{fl}(ab).$$

Therefore, we exactly have:

$$\begin{aligned} ab &= \mathbf{fl}(ab) + (ab - \mathbf{fl}(ab)) \\ &= x + y \end{aligned}$$

The chosen mode of rounding by truncation implies the floating-point result to be at most the exact result: $\mathbf{fl}(ab) \le ab$, thus $0 \le x \le ab$. Because y is the round-off error in the rounding of the floating operation $x = \mathbf{fl}(ab)$, the truncation imposes: $y < \mathbf{u}.\mathbf{ufp}(x)$ and $\mathbf{ufp}(ab) \le \mathbf{fl}(ab) \le ab$, hence:

$$0 \le y < \mathbf{u}.\mathbf{ufp}(x) \le \mathbf{u}.\mathbf{ufp}(ab) \le \mathbf{u}x.$$

We now present an other error-free transformation related to euclidean division by a power of two.

Suggested in [7] and quoted in [10] by S. Rump, this algorithm splits a floating-point number into two non-overlapping others.

Algorithm 2 — **ExtractScalar**

Require: $a \in \mathbb{N} \cap \mathbb{F}$, and $\sigma = 2^k, k \in \mathbb{N}, \sigma \ge a$
Ensure: $x \in \mathbb{N} \cap \mathbb{F}$, $y \in \mathbb{N} \cap \mathbb{F}$ such that $a = x + y$
  $q \leftarrow \mathbf{fl}(\sigma + a)$
  $x \leftarrow \mathbf{fl}(q - \sigma)$
  $y \leftarrow \mathbf{fl}(a - x)$
  return (x, y)

Theorem 3. Let x and y be the result of **ExtractScalar** applied to $a \in \mathbb{N} \cap \mathbb{F}$ and $\sigma \in \mathbb{F}, \sigma = 2^k, k \ge M$. We have:

$$a = x + y, \qquad 0 \le y < \mathbf{u}\,\sigma, \quad 0 \le x \le a, \quad x \in \mathbf{u}\sigma\mathbb{N}.$$

Proof 3. The idea behind this splitting method is to use the rounding mechanism of the floating-point unit. Set to be toward zero, the rounding behaves the same way as a *truncation.* In terms of bits, the M-bit string a is divided in two strings s_1 and s_2 which do not overlap such that the concatenation $s_1 + s_2$ equals a. As subparts of a, both bit-strings s_1 and s_2 are in $\mathbb{F}$.

If $a = 0$, the result $x = y = 0$ is obvious. We assume then $0 < a \le \sigma$. Like in the algorithm, let $q = \mathbf{fl}(a+\sigma)$. We have

$$\sigma \le q \le a + \sigma \le 2\sigma,$$

so that:

$$\frac{\sigma}{2} \le q \le 2\sigma.$$

[1]This proof is largely inspired from the proof in [9].

By Sterbenz's result (Theorem 1), we get:

$$\mathbf{fl}(q - \sigma) = q - \sigma. \tag{4}$$

Hence, $x = q - \sigma$ exactly. Let δ be the error of truncation in the first summation:

$$a + \sigma = q + \delta.$$

Note that in this equality $\delta \ge 0$ because rounding is directed toward zero.

$$q = \mathbf{fl}(a + \sigma) \le a + \sigma. \tag{5}$$

Moreover, the sum of the two positive numbers a and σ leads to an error δ, which is a floating-point number as well: $\delta \in \mathbb{N} \cap \mathbb{F}$. With this,

$$a - x = a - q + \sigma = \delta,$$

and thus:

$$\mathbf{fl}(a - x) = \mathbf{fl}(\delta) = \delta.$$

This last equality means:

$$y = \mathbf{fl}(a - x) = a - x \in \mathbb{N} \cap \mathbb{F}.$$

All in all, $a = x + y$ with $x, y \in \mathbb{N} \cap \mathbb{F}$. With (4) and (5), we have: $q - \sigma \le a$, which means: $x \le a$. Since $q \ge \sigma$, x is positive. As $a \ge x$, we have: $y = a - x \ge 0$. Moreover, the rounding of the first sum forces: $\delta < \mathbf{u}.\mathbf{ufp}(a + \sigma)$, which leads to $\delta < \mathbf{u}\sigma$. Then, $0 \le y < \mathbf{u}\sigma$.

Finally, we know that $q \in \mathbf{u}.\mathbf{ufp}(q)\,\mathbb{N}$, and by $q \ge \sigma$, we have $\mathbf{ufp}(q) \ge \sigma$, so that $q \in \mathbf{u}\sigma\mathbb{N}$. In a same way, $\sigma \in \mathbf{u}\sigma\mathbb{N}$, so that $q - \sigma \in \mathbf{u}\sigma\mathbb{N}$. From (4), if follows that $x \in \mathbf{u}\sigma\mathbb{N}$. □

2.2 Residue Number System arithmetic

We will now refer to Residue Number System as RNS. A RNS can be defined as a set of n constant integers

$$\{m_1, m_2, \ldots, m_n\}$$

referred to as moduli. We assume the n integers m_i, with $i = 1, \ldots, n$ to be coprime. Let us denote $M = m_1 \cdots m_n$ the product of all the moduli. Any arbitrary number X smaller than M can be uniquely represented by n integers $\{x_1, x_2, \ldots, x_n\}$ such that

$$x_i = X \pmod{m_i}, \quad \text{for } i = 1, \ldots n.$$

We will sometimes note $x_i = |X|_{m_i}$. It is then quite easy to compute addition, subtraction and multiplication with RNS numbers. Indeed, for addition, $Z = X \pm Y \pmod{M}$ can be done as

$$z_i = x_i \pm y_i \pmod{m_i}, \quad \text{for } i = 1, \ldots n.$$

For multiplication, it is accomplished in a similar manner by computing $Z = XY \pmod{M}$ with

$$z_i = x_i y_i \pmod{m_i}, \quad \text{for } i = 1, \ldots n.$$

Given an integer X less than M, it is easy to construct its RNS counterpart using euclidean division. We will also need the converse : given a RNS number, find its integer counterpart. This can be done via the following formula (known as Chinese Remainder Theorem or CRT),

$$X = \sum_{i=1}^{n} a_i M_i |X|_{m_i}, \tag{6}$$

where $M_i = M/m_i$ and a_i are such that

$$a_i M_i = 1 \pmod{m_i}, \quad \text{for } i = 1, \dots n.$$

The a_i are computed using the Euclid's extended algorithm. This way of computing X from its RNS counterpart is not the more efficient way but is sufficient if needed only once during the computation.

3. DOT PRODUCT COMPUTATION

3.1 Delayed-division

Theorem 4. Strengthening the general hypothesis (1) on p and assuming there exists $\lambda \geq 1$ such that:

$$\lambda(p-1) < 2^{M-1}, \tag{7}$$

there exists an algorithm computing the dot product of two vectors of $\mathbb{Z}/p\mathbb{Z}$ of size N using only N/λ reductions in $\mathbb{Z}/p\mathbb{Z}$. We will refer to this algorithm as λ-algorithm.

Principle of the algorithm. The actual algorithm we implemented is available in pseudo code in the Appendix of this paper. The basic idea of this algorithm relies on avoiding reductions. By introducing the λ parameter, we may accumulate λ times elements of $\mathbb{Z}/p\mathbb{Z}$ without invoking a costly reduction. For a double product $a_i\, b_i$, the use of the error-free transformation **TwoProduct** gives us an unevaluated sum $h_i + r_i$, which equals exactly this double product (Figure 2).

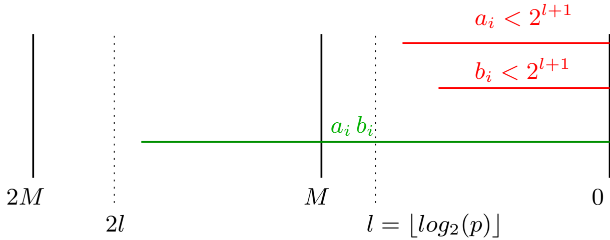

Figure 2: TwoProduct on a_i and b_i

With considerations on **ulp** and **ufp**, we introduce a variable $l = \lfloor \log_2(p) \rfloor$, such that $p < 2^{l+1}$ and:

$$\forall x \in [0, 2^{l+1}-1] \cap \mathbb{F}, \quad \begin{cases} 0 \leq x \leq 2^l & \Longrightarrow x \in \mathbb{Z}/p\mathbb{Z} \\ 2^l < x < 2^{l+1} & \Longrightarrow x - 2^l \in \mathbb{Z}/p\mathbb{Z} \end{cases}$$

which gives us a way to characterize elements of $\mathbb{Z}/p\mathbb{Z}$ (Figure 3). Once we only have elements of this field, we can sum them together using the property resulting from λ.

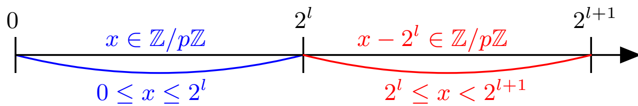

Figure 3: TwoProduct on a_i and b_i

Proof 4. The main interest in assumption of Lemma (4) lies in delaying reductions modulo p, which are heavy computations for the processor. Under hypothesis (7), one can add λ integers of $\mathbb{Z}/p\mathbb{Z}$ without exceeding 2^M. So, rather than dividing by p at each step in the dot product, one can delay the reduction and divide only N/λ times.

Let us set:

$$l = \lfloor \log_2(p) \rfloor \quad \text{and} \quad \mathbf{u} = 2^{-M+1}.$$

Note that by definition of l, we have: $l = \log_2(\mathbf{ufp}(p))$. Then, $2^l = \mathbf{ufp}(p)$. Since p is a prime greater than 3, we have

$$2^l < p < 2^{l+1}, \tag{8}$$

which means, in terms of bits: $\mathbf{ufp}(p) < p < 2.\mathbf{ufp}(p)$. In the field of prime characteristic p, the upper bound for elements is $p-1$, and from the previous inequality, we have (p is odd):

$$2^{2l} < (p-1)^2 < 2^{l+2}(2^l - 1).$$

We will know distinguish two cases regarding the value on the modulo p, whether p^2 can be represented in *one* floating-point number or not. Let i such that $1 \leq i \leq N$.

Case 1: $2l > M$
The product $a_i\, b_i$ may exceed M bits because one float is not enough to represent p^2, but two would. Applying the error-free transformation **TwoProduct**, one can deduce (Theorem 2): $a_i\, b_i = h_i + r_i$, with

$$h_i = \mathbf{fl}(a_i\, b_i), \quad 0 \leq h_i \leq a_i\, b_i, \quad 0 \leq r_i < \mathbf{u}.\mathbf{ufp}(h_i).$$

Note that because rounding mode had been chosen to be toward zero, all numbers are positive. That's why h_i is less than (or equal to) the exact value $a_i\, b_i$.

Without loss, assume $r_i > 0$. If r_i is null, then the product has been done without error and the following holds. Hence, from previous results,

$$0 < h_i \leq a_i\, b_i \leq (p-1)^2.$$

Since $h_i < 2^{2l+2}$, we get $\mathbf{ufp}(h_i) \leq 2l+1$, so that:

$$0 < h_i < 2^{l+2}(2^l - 1), \tag{9}$$

$$0 < r_i < 2^{2l-M+2}. \tag{10}$$

The general hypothesis (1) on p sets $l < M-1$, and then $2l < l + M - 1$. This leads to the necessary condition to apply **ExtractScalar** to h_i with parameter $\sigma = 2^{l+M}$:

$$2l + 2 \leq l + M. \tag{11}$$

We then have: $h_i \leq \sigma$ and **ExtractScalar** on (σ, h_i) gives $h_i = \alpha_i + \beta_i$ with:

$$0 \leq \beta_i < 2^{l+1}, \; 0 \leq \alpha_i < 2^{l+2}(2^l - 1), \; \alpha_i \in 2^{l+1}\mathbb{N}. \tag{12}$$

So:

$$0 \leq \alpha_i\, 2^{-(l+1)} < 2^{l+1} - 2, \tag{13}$$

$$\text{and:} \quad 0 \leq \beta_i < 2^{l+1}. \tag{14}$$

Splitting interval $[0, 2^{l+1}]$ into two halves, $[0, 2^l - 1]$ and $[2^l, 2^{l+1} - 1]$, (13) leads to two cases for α_i:

- If $2^l \leq \alpha_i\, 2^{-(l+1)} < 2^{l+1} - 2$, then :

  $$0 \leq \alpha_i\, 2^{-(l+1)} - 2^l < 2^l - 2 < 2^l,$$

  so that (8) leads to: $\alpha_i\, 2^{-(l+1)} - 2^l \in \mathbb{Z}/p\mathbb{Z}$.

- Otherwise, if $\alpha_i\, 2^{-(l+1)}$ lies in the first half, that is to say: $0 \le \alpha_i\, 2^{-(l+1)} < 2^l$, we have: $\alpha_i\, 2^{-(l+1)} \in \mathbb{Z}/p\mathbb{Z}$.

Equally for β_i, (14) gives:

- If β_i lies in the second half, $2^l \le \beta_i < 2^{l+1}$, we then have: $0 \le \beta_i - 2^l < 2^l$, so that: $\beta_i - 2^l \in \mathbb{Z}/p\mathbb{Z}$.
- Otherwise $0 \le \beta_i < 2^l$, and (8) gives: $\beta_i \in \mathbb{Z}/p\mathbb{Z}$.

In the same way with r_i, (10) and (11), it leads to: $0 < r_i < 2^l$. Thus, the result is directly given by (8): $r_i \in \mathbb{Z}/p\mathbb{Z}$.

All in all, we get integers in $\mathbb{Z}/p\mathbb{Z}$ one can sum together exactly λ times without reducing modulo p, thanks to hypothesis (7). Let n_α and n_β the number of 2^l corrections needed for α and β respectively. These values are defined by:

$$n_\alpha = \#\{\alpha_i \,/\, 2^{2l+1} \le \alpha_i < 2^{2l+2}\},$$
$$\text{and:}\quad n_\beta = \#\{\beta_i \,/\, 2^l \le \beta_i \le 2^{l+1}\}.$$

In the finite field $\mathbb{Z}/p\mathbb{Z}$, we then have the dot product:

$$\begin{aligned} a \cdot b &= \sum_{i=1}^{N} \alpha_i + \sum_{i=1}^{N} \beta_i + \sum_{i=1}^{N} r_i \\ &= \sum_{n_\alpha} (\alpha_i - 2^l) + \sum_{N-n_\alpha} \alpha_i + \sum_{n_\beta} (\beta_i - 2^l) + \sum_{N-n_\beta} \beta_i \\ &\quad + \sum_{N} r_i + (n_\alpha + n_\beta)\, 2^l \end{aligned}$$

Each of first five sums can be computed exactly by groups of size λ. One reduction is needed once the number of terms accumulated in each sum had reached λ.

Case 2: $2l \le M$
Here, the result of the product $a_i\, b_i$ does not need more than M bits to be computed exactly. The previous case where $2l \ge M$ still holds, even if the error-free transformation for the product is useless, since it produces $r_i = 0$.

This remark may be the starting point for a slightly different usage of **ExtractScalar**: when called at most three times with correct parameters, we can show that, for some size of vectors, there exists an algorithm computing the dot product using no reduction in the main loop.

3.2 Binary RNS basis with integers

The aim is here to use a RNS basis to compute the dot product $a \cdot b$. We assume that the coefficients a_i, b_i are integers in $[0, p-1]$. As a consequence, the dot product $a \cdot b$ considered as an integer satisfies

$$0 \le a \cdot b \le N(p-1)^2.$$

We choose a RNS basis $\{m_1, m_2, m_3, m_4\}$ such that

$$N(p-1)^2 < m_1 m_2 m_3 m_4.$$

We can compute $a \cdot b$ in the RNS basis. We will in fact get four numbers r_1, r_2, r_3, r_4 such that

$$\begin{aligned} a \cdot b &= r_1 \pmod{m_1} \\ &= r_2 \pmod{m_2} \\ &= r_3 \pmod{m_3} \\ &= r_4 \pmod{m_4}. \end{aligned}$$

Using formula (6), we can find $0 \le r < m_1 m_2 m_3 m_4$ such that

$$a \cdot b = r \pmod{m_1 m_2 m_3 m_4}. \tag{15}$$

Because of the choice of the m_i, we have

$$0 \le r < m_1 m_2 m_3 m_4,$$
$$\text{and } 0 \le a \cdot b < m_1 m_2 m_3 m_4$$

As a consequence, we have

$$|a \cdot b - r| < m_1 m_2 m_3 m_4.$$

From (15), it follows that $m_1 m_2 m_3 m_4$ divides $a \cdot b - r$. As a consequence, $a \cdot b = r$ in $\mathbb{Z}$. It is then easy to obtain the result in $\mathbb{Z}/p\mathbb{Z}$. Indeed, we have

$$a \cdot b = r \pmod{p}.$$

To perform the computation of the dot product in the most effective way, we use the four coprime integers

$$\mathcal{B} = \{2^n, 2^n + 1, 2^n - 1, 2^{n-1} - 1\} \tag{16}$$

as basis for RNS, with $n = 32$. Considering this word-size, internal computations are achieved in 64-bit integer precision. Reductions in the four-component basis use following equalities:

$$|2^n|_{2^n+1} = |2^n + 1 - 1|_{2^n+1} = -1, \tag{17}$$
$$|2^n|_{2^n-1} = |2^n - 1 + 1|_{2^n-1} = +1, \tag{18}$$
$$|2^n|_{2^{n-1}-1} = |2(2^{n-1} - 1) + 2|_{2^{n-1}-1} = +2. \tag{19}$$

Using those formulas, one performs really fast reductions using logical operations embedded in the processor. For example, let us consider the forward conversion of an integer X into its residues in $\mathcal{B}$ defined by (16). The first residue modulo 2^{32} is simply obtained by keeping the 32 less significant bits of X, which is exactly $|X|_{2^{32}}$. Writing the Euclidean division of X by 2^{32}, one gets:

$$X = q \times 2^{32} + r, \quad \text{with:} \quad r = |X|_{2^{32}}.$$

The three others residues derive from (17), (18) and (19):

$$|X|_{2^n+1} = |q\, 2^{32} + r|_{2^n+1} = |r - q|_{2^n+1}, \tag{20}$$
$$|X|_{2^n-1} = |q\, 2^{32} + r|_{2^n-1} = |r + q|_{2^n-1}, \tag{21}$$
$$|X|_{2^{n-1}-1} = |q\, 2^{32} + r|_{2^{n-1}-1} = |2q + r|_{2^{n-1}-1}. \tag{22}$$

This particular basis enables us to reach good speedups in comparison to classical bases where four primes are used. In this ordinary case, one would then have to perform divisions each time a forward conversion is needed.

To reconstruct the result using the CRT, we use the GMP library. This is necessary because in the formula (6), we need to compute the product of three numbers of 32 bits, which may lead to a number of 96 bits. But the use of GMP has a negligible cost in the algorithm since it is used once at the end.

A slightly different approach we present now consists in using floating-point numbers to represent the four residues.

3.3 Binary RNS basis for floating-point arithmetic

The idea behind a RNS basis in floating-point arithmetic is the performance. Indeed, the use of floating-point rather

than integer makes it possible to use BLAS (Basic Linear Algebra Subprograms[2]) routines. These routines are highly efficient but work only with floating-point numbers (and not integers). Even if it is not very important for dot products (with a mild size), the use of BLAS is very important when dealing with matrix-vector products or matrix-matrix products.

We use the same basis as (16) but with $n = \lfloor M/2 \rfloor$, where M is the size of floating-point mantissa in double precision. This is needed to allow exact floating-point multiplication. An important difference with the integer case relies on reductions: one can not here use logical instructions to perform division modulo 2^n. As in the prime basis, one has to divide four times at each step to compute residues in RNS basis.

4. EXPERIMENTAL RESULTS

In this section, we present our experimental results for the previously detailed algorithms. Both take two input parameters: the size N of vectors and the prime characteristic p of the field. Measured performances in all the following refer to time of computation and do not take into account memory-related operations.

Environment used for benchmarks was an Intel Itanium2 1.5 GHz processor with a FMA instruction in its floating-point unit. Along with algorithm comparisons, we compiled all twice, using both `gcc` (the GNU Compiler Collection) and `icc` (Intel C Compiler), since the latter is well-suited for the Intel Itanium family.

First off, concerning the floating-point λ-algorithm, we show how the value λ affects the performances. Numerically, λ represents *where* p lies in the floating-point range: λ is small for big p and big for small p. Computationally, it traduces how many iterations we can make without reducing partial results. On Figure 4 (linear scale) and Figure 5 (logarithmic scale), one can clearly see the values for λ when p is increasing in the interval $[0, 2^{52}]$, whereas GMP-based algorithm runs in constant time for a fixed N.

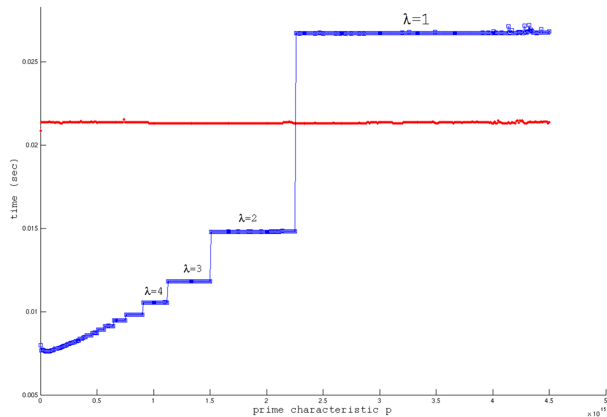

Figure 4: Timing comparisons between GMP (steady) and λ-algorithm (increasing steps for decreasing λ) for $p \in [0, 2^{52}]$ and $N = 100000$.

[2]http://www.netlib.org/blas/

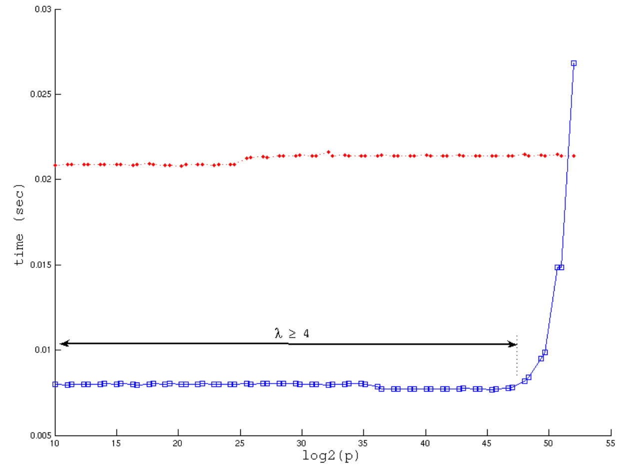

Figure 5: Timing comparisons between GMP (steady) and λ-algorithm (increasing curve) for $\log_2(p) \in [1, 52]$ and $N = 100000$.

We present results for two very different sizes of input vectors, $N \in \{512, 40000\}$, when the prime characteristic p lies in both `float` or `double` ranges, that is $[0, 2^{23}]$ and $[0, 2^{52}]$ respectively. We distinguish those two ranges for RNS `double` method to be applicable. This method requires indeed a floating-point product to be exact.

Values in tables are timing ratios between reference and measured algorithms: if ratio is greater than 1, then the measured one is faster; and slower if less than 1. When p lies in the interval $[0, 2^{52}]$, ratios are really different whether p is small or big, that is whether λ is big or small, respectively. The best performances are reached when p is small, since many divisions are avoided. In tables, we show minimal and maximal values (min/max) of the staircase (similar to Figure 4). There is no need to make this distinction when p lies in the smaller interval $[0, 2^{23}]$ since λ is not small enough regarding the size N of vectors. In this range, λ would be bigger than 2^{29}, so that differences in timings are negligible.

Reference	Algorithm	`gcc`	`icc`
GMP	λ-algorithm	0.30/**1.23**	0.28/**1.17**
	RNS (binary basis)	1.90	**4.13**
	RNS (prime basis)	0.14	0.97
	RNS (`double`)	–	–

Table 1: Ratio of time computations for prime characteristic $p \in [0, 2^{52}]$ and size of input vectors $N = 512$.

Minimal ratios for λ-algorithm are generally smaller than 1, since they are reached for $\lambda = 1$, which leads to a reduction at each step (Table 1). For $\lambda \geq 2$, reductions generally do not occur that often, so that a speedup occurs.

Huge differences between RNS with binary basis and RNS with prime basis are explained by reductions. In the binary case, moduli are *almost* power of two so that we use logical instruction AND to perform all reductions in 32 bits (section 3.2). These instructions speed up a lot the implementation in comparison with the slow division instruction of the prime

moduli in the prime basis. This is even more true when working on a dedicated machine with a special compiler. Here, `icc` makes it run a lot faster than `gcc` (Table 2).

Reference	Algorithm	`gcc`	`icc`
GMP	λ-algorithm	**1.15**	1.05
	RNS (binary basis)	1.77	**3.83**
	RNS (prime basis)	0.14	0.90
	RNS (`double`)	0.31	0.33
RNS (`double`)	λ-algorithm	**3.73**	3.16

Table 2: Ratio of time computations for prime characteristic $p \in [0, 2^{23}]$ and size of input vectors $N = 512$.

In Table 2, we also compared λ-algorithm and RNS working with `double` floating-point numbers. In that case, residue arithmetic is not efficient, even with the binary basis, since there is no logical instruction in floating-point units (FPU). Consequently, reductions cost a lot and make the whole computation relatively slow.

During benchmarking, we also evaluate results on different sizes of vectors. For $N = 2048$ for instance, results were not that different from the 512 case. For very large vectors, with 40000, 100000 or 500000 elements in particular, our presented algorithms get genuine speedups. In `double` precision when computing with 40000-element vectors (Table 3), one gets a speedup of 2 for the λ-algorithm and more than 7 for the RNS one with the binary basis.

Reference	Algorithm	`gcc`	`icc`
GMP	λ-algorithm	0.54/**2.01**	0.49/1.67
	RNS (binary basis)	3.42	**7.30**
	RNS (prime basis)	0.26	1.63
	RNS (`double`)	–	–

Table 3: Ratio of time computations for prime characteristic $p \in [0, 2^{52}]$ and size of input vectors $N = 40000$.

This difference holds when computations are performed in the smaller field (Table 4). In that case, again and for the same reasons as detailed before, λ-algorithm is three times faster than the RNS one using floating-point unit.

Reference	Algorithm	`gcc`	`icc`
GMP	λ-algorithm	**1.92**	1.81
	RNS (binary basis)	3.57	**8.10**
	RNS (prime basis)	0.26	1.81
	RNS (`double`)	0.60	0.66
RNS (`double`)	λ-algorithm	**3.21**	2.74

Table 4: Ratio of time computations for prime characteristic $p \in [0, 2^{23}]$ and size of input vectors $N = 40000$.

5. CONCLUSIONS AND FUTURE WORK

We have shown different ways to compute dot product in a prime finite field making use of floating-point and Residue Number System (RNS) arithmetics. For that matter, critical operations are reductions in the field. The first idea relies on the λ parameter of section 3.1, which allows us to avoid systematic reduction. Great performances are reached for big λ when the prime characteristic p of the field may be up to 2^{52}. The second idea introduces the RNS to perform computation on smaller values on a binary basis. In that special case, reductions are almost costless since moduli are power of two.

The conclusion of this work forks into two main lines regarding which arithmetic one considers. Floating-point numbers happen to be a widely studied area and we showed in this paper a way of using them to perform *integer* computations. Although this is not the major area of application, one can manage them correctly to compute in high order fields. The other side of the conclusion is directly linked to integer numbers, since it introduces residue arithmetic. We use two bases to perform computations: the classic one with prime numbers, and a particular one with four coprimes quasi power of two. This latter RNS usage happens to be very fast since reductions are almost costless.

We are currently working on a port of our algorihms on GPU (Graphics Processing Unit). With the λ-algorithm, this implies a pretreatment of data to get them distribute in groups of proper size to be able to sum them together exactly. As for RNS computations, we might compute the four components at the same time. The more serious drawback of GPU is the lack of efficiency in double precision. It would then be interesting to adapt RNS with floating-point arithmetic and compare performances.

6. REFERENCES

[1] IEEE standard for floating-point arithmetic. Technical report, 2008.

[2] J.-G. Dumas. Efficient dot product over word-size finite fields. In V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, editors, *Proceedings of the 7th International Workshop on Computer Algebra in Scientific Computing, CASC'2004 (St. Petersburg, Russia, July 12-19, 2004)*, pages 139–153, Garching, 2004. Institut für Informatik, Technische Universität München.

[3] J. G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation*, pages 63–74, New York, NY, USA, 2002. ACM.

[4] J.-G. Dumas, P. Giorgi, and C. Pernet. Ffpack: finite field linear algebra package. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 119–126, New York, NY, USA, 2004. ACM.

[5] J.-G. Dumas, P. Giorgi, and C. Pernet. Dense linear algebra over word-size prime fields: the fflas and ffpack packages. *ACM Trans. Math. Softw.*, 35(3):1–42, 2008.

[6] T. Granlund and al. GNU multiple precision arithmetic library 4.1.2, December 2002.

[7] C. Hecker. Let's get to the (floating) point. *Game Developer Magazine*, 1996.

[8] J.-M. Muller. On the definition of ulp(x). Technical report, École normale supérieure de Lyon - Laboratoire de l'Informatique du Parallélisme, 2005.

[9] Y. Nievergelt. Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit. *ACM Trans. Math. Softw.*, 29(1):27–48, 2003.

[10] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[11] P. H. Sterbenz. *Floating-point computation.* Prentice Hall, Englewood Cliffs, New Jersey, 1974.

APPENDIX

Algorithm 3 — Dot product computation by delaying reducing modulo p every λ

Require: $p \geq 3$ a prime, a, b two $\mathbb{Z}/p\mathbb{Z}$ vectors of size N
Ensure: The dot product $a \cdot b$ of vectors a and b in $\mathbb{Z}/p\mathbb{Z}$

  $k \leftarrow 1$
  $N_1 \leftarrow \left\lfloor \frac{N}{\lambda} \right\rfloor$
  $l \leftarrow \lfloor \log_2(p) \rfloor \qquad \sigma \leftarrow 2^{l+M+1}$
  $n_\alpha \leftarrow 0 \qquad n_\beta \leftarrow 0 \qquad n_r \leftarrow 0$
  $C \leftarrow 0 \qquad P \leftarrow 0 \qquad S \leftarrow 0 \qquad n \leftarrow 0$
  for $y = 1$ to N_1 **do**
    for $z = 1$ to λ **do**
      $[h, r] \leftarrow$ **TwoProduct**(a_k, b_k)
      $[\alpha, \beta] \leftarrow$ **ExtractScalar**(σ, h)
      if $2^l \leq \alpha 2^{-(l+1)}$ **then**
        $n_\alpha \leftarrow n_\alpha + 1$
        $\alpha \leftarrow \alpha - 2^l$
      end if
      if $2^l \leq \beta$ **then**
        $n_\beta \leftarrow n_\beta + 1$
        $\beta \leftarrow \beta - 2^l$
      end if
      $c \leftarrow \mathbf{fl}(c + \alpha)$
      $p \leftarrow \mathbf{fl}(p + \beta)$
      $s \leftarrow \mathbf{fl}(s + r)$
    end for
    $C \leftarrow C + (c \pmod p)$
    $P \leftarrow P + (p \pmod p)$
    $S \leftarrow S + (s \pmod p)$
    $c \leftarrow 0 \qquad p \leftarrow 0 \qquad s \leftarrow 0$
    $n \leftarrow n + 1$
    if $n = \lambda$ **then**
      $n \leftarrow 0$
      $C \leftarrow C \pmod p$
      $P \leftarrow P \pmod p$
      $S \leftarrow S \pmod p$
    end if
    $k \leftarrow k + 1$
  end for
  for $x = k$ to N **do**
    $[h, r] \leftarrow$ **TwoProduct**(a_k, b_k)
    $[\alpha, \beta] \leftarrow$ **ExtractScalar**(σ, h)
    if $2^l \leq \alpha 2^{-(l+1)}$ **then**
      $n_\alpha \leftarrow n_\alpha + 1$
      $\alpha \leftarrow \alpha - 2^l$
    end if
    if $2^l \leq \beta$ **then**
      $n_\beta \leftarrow n_\beta + 1$
      $\beta \leftarrow \beta - 2^l$
    end if
    $c \leftarrow \mathbf{fl}(c + \alpha)$
    $p \leftarrow \mathbf{fl}(p + \beta)$
    $s \leftarrow \mathbf{fl}(s + r)$
  end for
  $C \leftarrow C \pmod p$
  $P \leftarrow P \pmod p$
  $S \leftarrow S \pmod p$
  $C \leftarrow C + (c \pmod p)$
  $P \leftarrow P + (p \pmod p)$
  $S \leftarrow S + (s \pmod p)$
  $c \leftarrow C \pmod p$
  $p \leftarrow P \pmod p$
  $s \leftarrow S \pmod p$
  $res \leftarrow \mathbf{fl}(p + s)$
  $res \leftarrow res \pmod p$
  $res \leftarrow \mathbf{fl}(res + c)$
  $res \leftarrow res \pmod p$
  $h \leftarrow 2^l \pmod p$
  $[q, r] \leftarrow$ **TwoProduct**$(h, n_\alpha + n_\beta)$
  $q \leftarrow q \pmod p$
  $r \leftarrow r \pmod p$
  $q \leftarrow \mathbf{fl}(q + r)$
  $q \leftarrow q \pmod p$
  $res \leftarrow \mathbf{fl}(res + q)$
  $res \leftarrow res \pmod p$