HAL
open science

Simultaneous conversions with the Residue Number System using linear algebra

Javad Doliskani, Pascal Giorgi, Romain Lebreton, Eric Schost

To cite this version:

Javad Doliskani, Pascal Giorgi, Romain Lebreton, Eric Schost. Simultaneous conversions with the Residue Number System using linear algebra. 2016. lirmm-01415472v1

HAL Id: lirmm-01415472

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01415472v1

Preprint submitted on 13 Dec 2016 (v1), last revised 9 Feb 2018 (v2)

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.

Copyright

A

Simultaneous conversions with the Residue Number System using linear algebra

Javad Doliskani, Institute for Quantum Computing, University of Waterloo
Pascal Giorgi, LIRMM CNRS - University of Montpellier
Romain Lebreton, LIRMM CNRS - University of Montpellier
Eric Schost, University of Waterloo

We present an algorithm for simultaneous conversion between a given set of integers and their Residue Number System representations based on linear algebra. We provide a highly optimized implementation of the algorithm that exploits the computational features of modern processors. The main application of our algorithm is matrix multiplication over integers. Our speed-up of the conversions to and from the Residue Number System significantly improves the overall running time of matrix multiplication.

CCS Concepts: •**Mathematics of computing** → **Mathematical software performance; Computations in finite fields; Computations on matrices;** •**Theory of computation** → **Design and analysis of algorithms;**

1. INTRODUCTION

There currently exist a variety of high-performance libraries for linear algebra or polynomial transforms using floating point arithmetic [Whaley et al. 2001; Goto and van de Geijn 2008; Frigo and Johnson 2005; Püschel et al. 2005], or computations over the integers or finite fields [Shoup 1995; Bosma et al. 1997; Dumas et al. 2008; Hart 2010].

In this paper, we are interested in the latter kind of computation, specifically in the context of multi-precision arithmetic. Suppose for instance we work with matrices or polynomials with coefficients that are multi-precision integers, or lie in a finite ring $\mathbb{Z}/N\mathbb{Z}$, for some large N, and consider a basic operation such as the multiplication of these matrices or polynomials (there exist multiple applications to this fundamental operation; some of them such as polynomial factorization are illustrated in Section 4.3).

To perform a matrix or polynomial multiplication in such a context, several possibilities exist. A first approach consists in applying known algorithms, such as Strassen's, Karatsuba's, ... directly over our coefficient ring, relying *in fine* on fast algorithms for multiplication of multi-precision integers. Another large class of algorithms relies on *modular techniques*, or *Residue Number Systems*, computing the required result modulo many small primes before recovering the output by Chinese Remaindering. One should not expect either of these approaches to be superior in all circumstances. For instance, in extreme cases such as the product of matrices of size 1 or 2 with large integer entries, the modular approach highlighted above is most likely not competitive with a direct implementation. On the other hand, for the product of larger matrices or polynomials, Residue Number Systems often perform better than direct implementations, and as such, they are used in libraries or systems such as NTL, FFLAS-FFPACK, Magma, ...

In many cases, the bottlenecks in such an approach are the reduction of the inputs modulo many small primes, and the reconstruction of the output from its modular images by means of Chinese Remaindering; by contrast, operations modulo the small primes are often quite efficient. Algorithms of quasi-linear complexity have been known for long for both modular reduction and Chinese Remaindering [Gathen and Gerhard 2013, Chapter 10], based on so-called subproduct tree techniques; however, their practical performance remains somewhat lagging. In this paper, we propose an alternative to these algorithms, dedicated to those cases where we have several coefficients to convert; it relies on matrix multiplication to perform these tasks, with matrices that are integer analogues of Vandermonde matrices and their inverses. As a result, while their asymptotic complexity is inferior to that of asymptotically

fast methods, our algorithms behave extremely well in practice, as they allow us to rely on high-performance libraries for matrix multiplication.

Finally we give a glimpse at the implications in terms of performance that our improved matrix multiplication algorithm can yield to many algorithms. We will focus on the problems of modular composition, power projection, minimal polynomial and factorization. By plugging our multi-precision integer matrix multiplication algorithm into NTL existing implementations, we are able to show substantial performance gain.

Organization of the paper. We first give an overview of the Residue Number System and its classical algorithms in Section 2. In Section 3, we discuss algorithms for converting simultaneously a given set of integers to their Residue Number System representation, and vice versa. We contribute by giving a new algorithm using linear algebra in Sections 3.1 and 3.2, and we report on our implementation inside FFLAS-FFPACK in Section 3.3. We then use these results to implement an efficient integer matrix multiplication algorithm in Section 4 and report on direct performance improvement of sample applications in Section 4.3. Finally, we extend our algorithms to larger moduli in Section 5 and we report on our implementation in Section 5.5.

Acknowledgment. This work has been supported by the French ANR under the grants HPAC (ANR-11-BS02-013), CATREL (ANR-12-BS02-001), NSERC and the Canada Research Chairs program.

2. PRELIMINARIES

2.1. Basic results

Our computation model is that of [Gathen and Gerhard 2013, Chapter 2] or [Brent and Zimmermann 2010, Chapter 1.1]: we fix a base β, that can typically be a power of two such as $2, 2^{16}$ or 2^{32}, and we assume that all non-zero integers are represented on this basis: such an integer a is represented by coefficients $(a_0, \ldots, a_{s-1})$, with all a_i in $\{0, \ldots, \beta - 1\}$ and a_{s-1} non-zero, together with a bit of sign, such that $a = \pm(a_0 + a_1\beta + \cdots + a_{s-1}\beta^{s-1})$. The coefficients a_i are referred to as *words*; the integer s is called the *length* of a, and denoted by $\lambda(a)$.

Our complexity statements are given in terms of number of operations on words, counting all operations at unit cost (for the details of what operations are used, see [Gathen and Gerhard 2013, Chapter 2]). We let $\mathsf{I} : \mathbb{N} \to \mathbb{N}$ be such that two integers a, b of length at most n can be multiplied in $\mathsf{I}(n)$ word operations, assuming the base-β expansions of a and b are given as input; we assume that I satisfies the super-linearity assumptions of [Gathen and Gerhard 2013, Chapter 8]. One can take $\mathsf{I}(n)$ in $n \log(n) 2^{\mathcal{O}(\log^*(n))}$ using Fast Fourier Transform techniques [Fürer 2007].

We will often use the fact that given integers a, b of lengths respectively n and t, with $t \leqslant n$, one can compute the product ab using $\mathcal{O}(n\,\mathsf{I}(t)/t)$ word operations: without loss of generality, assume that a and b are positive and write a in base β^t as $a = a_0 + a_1\beta^t + \cdots + a_\ell\beta^{\ell t}$, with $\ell \in \Theta(n/t)$, so that $ab = (a_0 b) + (a_1 b)2^t + \cdots + (a_\ell b)2^{\ell t}$. Writing a as above requires no arithmetic operation; the coefficients a_i are themselves written in base β. All coefficients $a_i b$ can be computed in time $\mathcal{O}(\ell\,\mathsf{I}(t)) = \mathcal{O}(n\,\mathsf{I}(t)/t)$. Since $a_i b < \beta^{2t}$ holds for all i, going from the above expansion to the base-β expansion of ab takes time $\mathcal{O}(n)$, which is $\mathcal{O}(n\,\mathsf{I}(t)/t)$.

For $m \in \mathbb{N}_{>0}$, we write $\mathbb{Z}/m\mathbb{Z}$ for the ring of integers modulo m. We denote by respectively $(a \text{ rem } m)$ and $(a \text{ quo } m)$ the remainder and quotient of the Euclidean division of $a \in \mathbb{Z}$ by $m \in \mathbb{N}_{>0}$, with $0 \leqslant (a \text{ rem } m) < m$; we extend the notation $(a \text{ rem } m)$ to any rational $a = n/d \in \mathbb{Q}$ such that $\gcd(d, m) = 1$ to be the unique b in $\{0, \ldots, m-1\}$ such that $bd \equiv n \pmod{m}$.

Let $a \in \mathbb{Z}$ and $m \in \mathbb{N}_{>0}$ be of respective lengths at most n and t, with $t \leqslant n$. Then, one can compute the quotient (a quo m) and the remainder (a rem m) using $\mathcal{O}(\mathsf{I}(n))$ word operations; the algorithm goes back to [Cook 1966], see also [Aho et al. 1974, Section 8.2] and [Brent and Zimmermann 2010, Section 2.4.1]. In particular, arithmetic operations $(+, -, \times)$ in $\mathbb{Z}/m\mathbb{Z}$ can all be done using $\mathcal{O}(\mathsf{I}(t))$ word operations; inversion modulo m can be done using $\mathcal{O}(\mathsf{I}(t)\log(t))$ word operations.

More refined estimates for Euclidean division are available in some particular cases:

— Consider first the case where $n \gg t$. Given a and m as above, we can compute both (a quo m) and (a rem m) using

$$\mathcal{O}\left(\frac{\mathsf{I}(t)}{t} n\right) \tag{1}$$

word operations. To do so, write a in base β^t as $a = a_0 + a_1\beta^t + \cdots + a_\ell\beta^{\ell t}$, with $\ell \in \Theta(n/t)$; this does not involve any operations, since a is given by its base-β expansion. Define $q_{\ell+1} = r_{\ell+1} = 0$, and, for $i = \ell, \ldots, 0$, $r_i = (\beta^t r_{i+1} + a_i)$ rem m and $q_i = (\beta^t r_{i+1} + a_i)$ quo m. This computes $r_0 = (a \text{ rem } m)$, and we get $q = (a \text{ quo } m)$ as $q = q_0 + q_1\beta^t + \cdots + q_\ell\beta^{\ell t}$ (from which its base-β expansion follows without any further work). The total time of this calculation is as in (1).

— Finally, if we suppose more precisely that $\beta^{t-1} \leqslant m < \beta^t$ holds, in cases where $n - t \leqslant t$, the remainder (a rem m) can be computed in time

$$\mathcal{O}\left(\frac{\mathsf{I}(n-t)}{n-t} t\right). \tag{2}$$

Indeed, in this case, the quotient $q = (a \text{ quo } m)$ can actually be computed in $\mathcal{O}(\mathsf{I}(n-t))$ word operations, since $\lambda(q)$ is in $\mathcal{O}(n-t)$ (see [Giorgi et al. 2013, Algorithm 2]). Once q is known, one can compute qm in time $\mathcal{O}(t\,\mathsf{I}(n-t)/(n-t))$, as explained above; deducing (a rem m) takes another $\mathcal{O}(t)$ word operations, whence the cost claimed above .

We also let $\mathsf{MM} : \mathbb{N}^3 \to \mathbb{N}$ be such that one can do integer matrix multiplication in size $(a, b) \times (b, c)$ using $\mathsf{MM}(a, b, c)$ operations $(+, -, \times)$ in $\mathbb{Z}$, with an algorithm that uses constants bounded independently of a, b, c. The latter requirement implies that for any $m > 0$, with m of length t, one can do matrix multiplication in size $(a, b) \times (b, c)$ over $\mathbb{Z}/m\mathbb{Z}$ using $\mathcal{O}(\mathsf{MM}(a, b, c)\,\mathsf{I}(t))$ word operations. This also implies that given integer matrices of respective sizes (a, b) and (b, c), if the entries of the product are bounded by β^t in absolute value, one can compute their product using $\mathcal{O}(\mathsf{MM}(a, b, c)\,\mathsf{I}(t))$ word operations, by computing it modulo an integer of length $\Theta(t)$.

Let ω be such that we can multiply $b \times b$ integer matrices within $\mathcal{O}(b^\omega)$ operations in $\mathbb{Z}$, under the same assumption on the constants used in the algorithm as above; the best known bound on ω is $\omega < 2.3729$ [Coppersmith and Winograd 1990; Stothers 2010; Vassilevska Williams 2012; Le Gall 2014]. So $\mathsf{MM}(b, b, b) = \mathcal{O}(b^\omega)$ and $\mathsf{MM}(a, a, b) = a^{\omega-1}b$ when $a \leqslant b$ by partitioning into square blocks of size a.

2.2. The Residue Number System

The Residue Number System (RNS) is a non-positional number system that allows one to represent finite sets of integers. Let $m_1, m_2, \ldots, m_s \in \mathbb{N}$ be pairwise distinct primes, and let

$$M = m_1 m_2 \cdots m_s.$$

Then any integer $a \in \{0, \ldots, M-1\}$ is uniquely determined by its residues $([a]_1, [a]_2, \ldots, [a]_s)$, with $[a]_i = (a \text{ rem } m_i)$; this residual representation is called a Residue Number System.

Operations such as addition and multiplication are straightforward in the residual representation; however, conversions between the binary representation of a and its residual one are potentially costly. For the following discussion, we assume that all m_i's have length at most t; in particular, any a in $\{0, \ldots, M-1\}$ has length at most st.

The conversion to the RNS representation amounts to taking a as above and computing all remainders (a rem m_i). Using (1), we can compute each of them in $\mathcal{O}(s\,\mathsf{I}(t))$ word operations, for a total time in $\mathcal{O}(s^2\mathsf{I}(t))$. A divide-and-conquer approach [Borodin and Moenck 1974] leads to an improved estimate of $\mathcal{O}(\mathsf{I}(st)\log(s))$ word operations (see also [Gathen and Gerhard 2013, Theorem 10.24]).

The inverse operation is Chinese Remaindering. Let a be an integer in $\{0, \ldots, M-1\}$ given by its RNS representation $([a]_1, \ldots, [a]_s)$; then, a is equal to

$$\left[\sum_{i=1}^{s} ([a]_i u_i \text{ rem } m_i)\, M_i\right] \text{ rem } M, \tag{3}$$

where we write $M_i = M/m_i$ and $u_i = (1/M_i \text{ rem } m_i)$ for all i.

From this expression, one can readily deduce an algorithm of cost $\mathcal{O}(s^2\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t))$ to compute a. One first computes M in the direct manner, by multiplying 1 by successively $m_1, \ldots, m_s$; the total time is $\mathcal{O}(s^2\,\mathsf{I}(t))$ word operations. From this, all M_i's can be deduced in the same asymptotic time, since we saw in the previous section that we can compute (M quo m_i) in time $\mathcal{O}(s\,\mathsf{I}(t))$ for all i. One can further compute all remainders (M_i rem m_i) in time $\mathcal{O}(s^2\,\mathsf{I}(t))$ as well, by the same mechanism. Deducing all u_i's takes $\mathcal{O}(s\,\mathsf{I}(t)\log(t))$ word operations, since we do s modular inversions with inputs of length at most t. One can then deduce a by simply computing all summands in (3) and adding them (subtracting M whenever a sum exceeds this value), for another $\mathcal{O}(s^2\,\mathsf{I}(t))$ word operations. The total time is thus $\mathcal{O}(s^2\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t))$ word operations.

To do better, consider the mapping

$$\varphi : (v_1, \ldots, v_s) \mapsto \sum_{i=1}^{s} v_i M_i \text{ rem } M,$$

with v_i in $\{0, \ldots, m_i - 1\}$ for all i. Using a divide-and-conquer approach, $\varphi(v_1, \ldots, v_s)$ can be computed in $\mathcal{O}(\mathsf{I}(st)\log(s))$ word operations using [Gathen and Gerhard 2013, Algorithm 10.20] (taking care to reduce all results, whenever required). In particular, (M_i rem m_i) can be computed as ($\varphi(1, \ldots, 1)$ rem m_i), so all of them can be deduced in time $\mathcal{O}(\mathsf{I}(st)\log(s))$; as above, we deduce all u_i's using an extra $\mathcal{O}(s\,\mathsf{I}(t)\log(t))$ operations. Once they are known, the computation of $a = \varphi(a_1u_1, \ldots, a_su_s)$ from its residuals is reduced to s modular multiplications of length t, and an application of φ; the overall cost is thus $\mathcal{O}(\mathsf{I}(st)\log(s) + s\,\mathsf{I}(t)\log(t))$.

Note that in term of implementation, simultaneous reductions using the naive algorithms can benefit from vectorized (SIMD) instructions to save constant factors, and can be easily parallelized. On the other hand, simultaneous reductions using the quasi-linear approach does not benefit from SIMD instructions in a straightforward manner.

3. ALGORITHMS FOR SIMULTANEOUS CONVERSIONS

In this section, we present new algorithms for conversions to and from a Residue Number System, with a focus on the case where the moduli $(m_1, \ldots, m_s)$ are fixed and several conversions are needed. In this section, we suppose that all m_i's have length at most $t \geq 1$, which implies that $s < \beta^t$. As above, we also let $M = m_1m_2\cdots m_s$; in particular, any integer a in $\{0, \ldots, M-1\}$ has length at most st.

Using the divide-and-conquer algorithms mentioned in the previous section, converting a vector $(a_1, \ldots, a_r) \in \{0, \ldots, M-1\}^r$ to its RNS representation can be done in

$\mathcal{O}(r\,\mathsf{I}(st)\log(s))$ word operations. For the converse operation, Chinese Remaindering, the cost $\mathcal{O}(s\,\mathsf{I}(t)\log(t))$ of computing all modular inverses u_i needs only be incurred once, so that the cost of Chinese Remaindering for r inputs is $\mathcal{O}(r\,\mathsf{I}(st)\log(s) + s\,\mathsf{I}(t)\log(t))$. Up to logarithmic factors, both results are quasi-linear in the size rst of the input and output.

The algorithms in this section are not as efficient in terms of asymptotic complexity, but we will see in further sections that they behave very well in practice. They are inspired by analogue computations for polynomials: in the polynomial case, multi-point evaluation and interpolation are linear maps, which can be performed by plain matrix-vector products. In our context, we will thus use matrices that mimic Vandermonde matrices and their inverses, which essentially reduce conversions to matrix-vector products. Simultaneous conversions reduce to matrix-matrix products, which brings an asymptotic complexity improvement with respect to many matrix-vector products.

To describe the algorithms, we need a further parameter, written t': for conversions to and from the RNS, we will write the multi-precision inputs and outputs in base $\beta^{t'}$. We will always assume that $t' \leqslant t$ holds, and we accordingly let $s' = \lceil st/t' \rceil \geqslant s$ be an upper bound on the length of all elements of $\{0, \ldots, M-1\}$ written in base $\beta^{t'}$. We will see that choosing $t' \simeq t$ allow us to minimize the cost of the algorithms, at least in theory; for practical purposes, we will see in Section 4.2 that we actually choose $t' < t$.

3.1. Conversion to RNS

Let $a = (a_1, \ldots, a_r)$ be a vector of r integers, all in $\{0, \ldots, M-1\}$. The conversion to its residual representation is split up into three phases:

(1) First, we compute all $[\beta^{it'}]_j = (\beta^{it'} \text{ rem } m_j)$ for $1 \leqslant i < s'$, $1 \leqslant j \leqslant s$ and gather them into the matrix

$$B = \begin{bmatrix} 1 & [\beta^{t'}]_1 & [\beta^{2t'}]_1 & \ldots & [\beta^{(s'-1)t'}]_1 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & [\beta^{t'}]_s & [\beta^{2t'}]_s & \ldots & [\beta^{(s'-1)t'}]_s \end{bmatrix} \in \mathcal{M}_{s \times s'}(\mathbb{Z}).$$

Because $t' \leqslant t$ and all m_i have length at most t, computing each row of B takes $\mathcal{O}(s'\,\mathsf{I}(t))$ word operations, by successive multiplications, for a total of $\mathcal{O}(ss'\,\mathsf{I}(t))$ for the whole matrix. Note that this computation is independent of the numbers a_i to reduce and so can be precomputed.

(2) Then, we use this matrix to compute a *pseudo-reduction* of $(a_1, \ldots, a_r)$ modulo the m_i's. For this purpose, we write the expansion of each a_i in base $\beta^{t'}$ as

$$a_i = \sum_{j=0}^{s'-1} a_{i,j} \beta^{jt'} \quad \text{for } 1 \leqslant i \leqslant r.$$

We then gather these coefficients into the matrix

$$C = \begin{bmatrix} a_{1,0} & a_{2,0} & a_{3,0} & \ldots & a_{r,0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{1,s'-1} & a_{2,s'-1} & a_{3,s'-1} & \ldots & a_{r,s'-1} \end{bmatrix} \in \mathcal{M}_{s' \times r}(\mathbb{Z}).$$

From the definition of matrices B and C we have

$$(BC)_{i,j} \equiv a_j \pmod{m_i} \tag{4}$$

and since $s' \leqslant st < \beta^t \cdot \beta^t$

$$0 \leqslant (BC)_{i,j} < s' m_i \beta^{t'} < \beta^{4t}. \tag{5}$$

Computing the entries of C is done by rewriting a_i from its β-expansion to $\beta^{t'}$-expansion, which takes no arithmetic operation. The bound above on the entries of the product BC shows that computing it takes $\mathcal{O}(\mathsf{MM}(s, s', r)\,\mathsf{I}(t))$ word operations (see Section 2.1).

(3) The last step consists in reducing the i-th row of the matrix product BC modulo m_i, for $1 \leqslant i \leqslant r$. Since all entries of BC have length at most $4t$, this costs $\mathcal{O}(rs\,\mathsf{I}(t))$ word operations.

The overall cost is thus $\mathcal{O}(\mathsf{MM}(s, s', r)\,\mathsf{I}(t))$ word operations. As announced above, choosing $t' = t$, or equivalently $s' = s$, minimizes this cost, making it $\mathcal{O}(\mathsf{MM}(s, s, r)\,\mathsf{I}(t))$. This amounts to $\mathcal{O}(rs^{\omega-1}\mathsf{I}(t))$ when $r \geqslant s$.

3.2. Conversion from RNS

Given residues $(a_{1,1}, \ldots, a_{1,s}), \ldots, (a_{r,1}, \ldots, a_{r,s})$, we now want to reconstruct $(a_1, \ldots, a_r)$ in $\{0, \ldots, M-1\}^r$ such that $a_{i,j} = (a_i \text{ rem } m_j)$ holds for $1 \leqslant i \leqslant r$ and $1 \leqslant j \leqslant s$. As in Subsection 2.2, we write $M_j = M/m_j$ and $u_j = (1/M_j \text{ rem } m_j)$ for $1 \leqslant j \leqslant s$.

We first compute *pseudo-reconstructions*

$$\ell_i := \sum_{j=1}^{s} \gamma_{i,j} M_j, \quad \text{with} \quad \gamma_{i,j} = (a_{i,j} u_j) \text{ rem } m_j,$$

for $1 \leqslant i \leqslant r$, so that $a_i = (\ell_i \text{ rem } M)$ holds for all i, and $\ell_i < sM < s\beta^{st}$. In a second stage, we reduce them modulo M. We can decompose our algorithm in 5 steps:

(1) We saw in Subsection 2.2 that $M_1, \ldots, M_s$ and $u_1, \ldots, u_s$ can be computed using $\mathcal{O}(s^2\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t))$ word operations (which is quasi-linear in the size $s^2 t$ of $M_1, \ldots, M_s$). This is independent of the a_i's.

(2) Computing all $\gamma_{i,j}$, for $1 \leqslant i \leqslant r$ and $1 \leqslant j \leqslant s$ takes another $\mathcal{O}(rs\,\mathsf{I}(t))$ operations.

(3) For $1 \leqslant j \leqslant s$, write M_j in base $\beta^{t'}$ as

$$M_j = \sum_{k=0}^{s'-1} \mu_{j,k} \beta^{kt'},$$

so that we can write ℓ_i as

$$\ell_i = \sum_{j=1}^{s} \gamma_{i,j} M_j = \sum_{j=1}^{s} \gamma_{i,j} \sum_{k=0}^{s'-1} \mu_{j,k} \beta^{kt'} = \sum_{k=0}^{s'-1} d_{i,k} \beta^{kt'},$$

with

$$d_{i,k} = \sum_{j=1}^{s} \gamma_{i,j}\, \mu_{j,k}.$$

These latter coefficients can be computed through the following matrix multiplication

$$\begin{bmatrix} d_{1,0} & d_{1,1} & \cdots & d_{1,s'-1} \\ d_{2,0} & d_{2,1} & \cdots & d_{2,s'-1} \\ \vdots & \vdots & & \vdots \\ d_{r,0} & d_{r,1} & \cdots & d_{r,s'-1} \end{bmatrix} = \begin{bmatrix} \gamma_{1,1} & \gamma_{1,2} & \cdots & \gamma_{1,s} \\ \gamma_{2,1} & \gamma_{2,2} & \cdots & \gamma_{2,s} \\ \vdots & \vdots & & \vdots \\ \gamma_{r,1} & \gamma_{r,2} & \cdots & \gamma_{r,s} \end{bmatrix} \begin{bmatrix} \mu_{1,0} & \mu_{1,1} & \cdots & \mu_{1,s'-1} \\ \mu_{2,0} & \mu_{2,1} & \cdots & \mu_{2,s'-1} \\ \vdots & \vdots & & \vdots \\ \mu_{s,0} & \mu_{s,1} & \cdots & \mu_{s,s'-1} \end{bmatrix}, \tag{6}$$

using $\mathcal{O}(\mathsf{MM}(r, s, s')\,\mathsf{I}(t))$ word operations.

(4) From the matrix $(d_{i,j})$, we get all $\ell_i = \sum_{k=0}^{s'-1} d_{i,k}\beta^{kt'}$. Note that $(d_{i,j})_{0\leqslant k\leqslant s'-1}$ are not the exact coefficients of the expansion of ℓ_i in base $\beta^{t'}$; however, since they are all at most $s\beta^t\beta^{t'} \leqslant \beta^{3t}$, we can still recover the base-$\beta^{t'}$ expansion of all ℓ_i's in linear time $\mathcal{O}(rs't)$. From this, their expansion in base β follows without any further work.

(5) The final step of the reconstruction consists in reducing all ℓ_i's modulo M. This step is relatively cheap, since the ℓ_i's are "almost" reduced. Indeed, since all ℓ_i's are at most sM, Equation (2) shows that this last step costs $\mathcal{O}(\log_\beta(M)\,\mathsf{I}(\log_\beta(s))/\log_\beta(s))$ per ℓ_i. By assumption $x \mapsto \mathsf{I}(x)/x$ is non-decreasing, and since we have $s < \beta^t$ and $\log_\beta(M) \leqslant st$, the latter cost is $\mathcal{O}(s\,\mathsf{I}(t))$ per ℓ_i, for a total cost of $\mathcal{O}(rs\,\mathsf{I}(t))$ word operations.

Altogether, the cost of this algorithm is $\mathcal{O}(\mathsf{MM}(r,s,s')\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t))$ word operations, where the second term accounts for inversions modulo respectively $m_1,\ldots,m_s$. As for the first conversion, choosing $t' = t$, and thus $s' = s$, gives us the minimal value, $\mathcal{O}(\mathsf{MM}(r,s,s)\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t))$.

3.3. Implementation

We have implemented our algorithm in C++ as a part of the FFLAS-FFPACK library [FFLAS-FFPACK-Team 2016], which is a member of the LinBox project [LinBox-Team 2016; Dumas et al. 2002].

Our main purpose in designing our algorithm was to improve performances for multi-precision integer matrix multiplication for a wide range of dimensions and bitsizes. For general dimensions and bitsizes, it is worthwhile to reduce one matrix multiplication with large entries to many matrix multiplications with machine-word entries using multi-modular reductions.

Indeed, it is well known that matrix multiplication with floating point entries can almost reach the peak performance of current processors. Many optimized implementations such as ATLAS [Whaley et al. 2001], MKL [Intel 2007] or GotoBlas/OpenBlas [Goto and van de Geijn 2008] achieve this in practice. It was shown in [Dumas et al. 2008] that one can get similar performance with matrix multiplication modulo m_i whenever m_i^2 holds in a machine word. Indeed, if the moduli m_i satisfies the inequality $b(m_i-1)^2 < 2^{53}$, with b the inner dimension of the matrix multiplication, then one can get the result by first computing the matrix product over the floating point number and then reducing the result modulo m_i. Furthermore, using Strassen's subcubic algorithm [Strassen 1969] and a stronger bound on m_i one can reach even better performance [Dumas et al. 2008; FFLAS-FFPACK-Team 2016]. This approach provides the best performance per bit for modular matrix multiplication [Pernet 2015] and thus we rely on it to build our RNS conversion algorithms from Section 3.

In particular, we set the base $\beta = 2$ and $t \leqslant 26$ to ensure that all the moduli m_i satisfy $(m_i-1)^2 < 2^{53}$, which is the case whenever $m_i^2 \leqslant 2^{53}$. Our simultaneous RNS conversion algorithms also rely on machine-word matrix multiplication, see Equations (4) and (6); in order to guarantee that the results of these products fit into 53 bits, we must ensure that the inequality $s'm_i2^{t'} \leqslant 2^{53}$ holds for all i, where t' is the parameter introduced in Section 3 which satisfies $1 \leqslant t' \leqslant t \leqslant 26$.

This parameter governs how we cut multi-precision integers; indeed we use the $\beta^{t'}$-expansion of the integers $a_1,\ldots,a_r$ during reduction and of $M_1,\ldots,M_s$ during reconstruction. The GMP library [GMP-Team 2015] is used to handle all these multi-precision integers. We choose $t' = 16$ because the integer data structure of GMP is so that conversions to the $\beta^{t'}$-adic representation are done almost for free by simply casting `mp_limb_t*` pointers to `uint16_t*` pointers.

Then we choose the maximum value of t satisfying $16 \leqslant t \leqslant 26$ such that $s'2^{t+t'} \leqslant 2^{53}$. Note that in the case of integer matrix multiplication, we may have to pick t even lower

because the inner dimension of the modular matrix multiplication may be bigger than s'. Therefore in our setting, we get an upper bound on the integer bitsizes of about 189 KBytes ($st \simeq 2^{20.5}$ bits) for RNS conversions on a 64-bits machine, which is obtained for values $t = 20, s = 2^{16.2}, t' = 16, s' = 2^{16.5}$. Note that for these very large integers, our method might not be competitive with the fast divide-and-conquer approach of [Borodin and Moenck 1974].

We finish with a few implementation details. When we turn pseudo-reductions into reductions, we are dealing with integers that fit inside one machine-word. Therefore we have optimized our implementation by coding an SIMD version of the Barrett reduction [Barrett 1986] as explained in [Hoeven et al. 2016]. Finally, to minimize cache misses we store modular matrices contiguously to each other, as $((A \text{ rem } m_1), (A \text{ rem } m_2), \dots)$.

3.4. Timings

We compared our new RNS conversion algorithms with the libraries FLINT [Hart 2010] and Mathemagix [Hoeven et al. 2012], both of which implement simultaneous RNS conversion. However, these implementations do not all come with the same restrictions on the moduli size. The Mathemagix library uses 32 bits integers and limits the moduli to 31 bits, the FLINT library uses 64 bits integers and limits the moduli to 59 bits and our code represents integers using 64 bits floating point numbers and limits the moduli to 26 bits. Each library uses different RNS bases, but all of them were chosen to allow the same number of bits (*i.e.* the length of M is almost constant) in order to provide a fair comparison. Our benchmark mimics the use of RNS conversions within multi-modular matrix multiplication : therefore the inputs of the conversion to RNS have a bitsize that is about half of the bitsize of M.

Our benchmarks are done on an Intel Xeon E5-2697 2.6 GHz machine. We chose the number of elements to convert from/to RNS to be 128^2 as conversion time per element is constant above 1000 elements. In Tables I and II we report the RNS conversion time per element from an average of several runs (adding up to a total time of a few seconds). The MMX columns correspond to Mathemagix library where two implementations are available: the asymptotically fast divide-and-conquer approach and the naive one. The FLINT column corresponds to fast divide-and-conquer implementation available in FLINT version 2.5 while FFLAS corresponds to the implementation of our new algorithms with parameters $t' = 16$ and an adaptative value of $t < 27$ that is maximal for the given RNS bitsize. For both

Table I: Simultaneous conversions to RNS (time per integer in μs)

RNS bitsize	**FLINT**	**MMX (naive)**	**MMX (fast)**	**FFLAS** [speedup]
2^8	0.17	0.34	1.49	0.06 [x 2.8]
2^9	0.35	0.75	3.07	0.13 [x 2.7]
2^{10}	0.84	1.77	6.73	0.27 [x 3.1]
2^{11}	2.73	4.26	14.32	0.75 [x 3.6]
2^{12}	7.03	11.01	30.98	1.92 [x 3.7]
2^{13}	17.75	29.86	72.42	5.94 [x 3.0]
2^{14}	50.90	88.95	183.46	21.09 [x 2.4]
2^{15}	165.80	301.69	435.05	80.82 [x 2.0]
2^{16}	506.91	1055.84	1037.79	298.86 [x 1.7]
2^{17}	1530.05	3973.46	2733.15	1107.23 [x 1.4]
2^{18}	4820.63	15376.40	8049.31	4114.98 [x 1.2]
2^{19}	13326.13	59693.64	20405.06	15491.90 [none]
2^{20}	37639.48	241953.39	54298.62	55370.16 [none]

Table II: Simultaneous conversions from RNS (time per integer in μs)

RNS bitsize	**FLINT**	**MMX (naive)**	**MMX (fast)**	**FFLAS** [speedup]
2^8	0.63	0.74	3.80	0.34 [x 1.8]
2^9	1.34	1.04	7.40	0.39 [x 3.4]
2^{10}	3.12	1.86	15.53	0.72 [x 4.3]
2^{11}	6.92	4.29	30.91	1.57 [x 4.4]
2^{12}	16.79	12.18	63.53	3.94 [x 4.3]
2^{13}	40.73	43.89	139.16	12.77 [x 3.2]
2^{14}	113.19	144.57	309.88	43.13 [x 2.6]
2^{15}	316.61	502.18	687.45	161.44 [x 2.0]
2^{16}	855.48	2187.65	1502.16	609.22 [x 1.4]
2^{17}	2337.96	10356.08	3519.61	2259.84 [x 1.1]
2^{18}	7295.26	39965.23	9883.07	8283.64 [none]
2^{19}	18529.38	156155.06	22564.36	31382.81 [none]
2^{20}	48413.81	685329.45	59809.07	111899.47 [none]

tables, we add in the FFLAS column the speedup of our code against the fastest code among FLINT and Mathemagix.

One can see from Tables I and II that up to RNS bases with 200 000-bits, our new method outperforms existing implementations, even when asymptotically faster methods are used. For RNS basis with at most 2^{15} bits = 4 KBytes our implementation is at least twice faster. If we only compare our code with the naive implementation in Mathemagix, which has basically the same theoretical complexity, we have alway a speedup between 3 and 6. Recall however that our approaches of Section 3 cannot handle bitsizes greater that 2^{20} bits (this is why our benchmarks do not consider larger bitsizes); in practice, asymptotically fast methods perform better before this bitsize limit is reached.

Table III: Precomputation for RNS conversions from/to together (total time in μs)

RNS bitsize	**FLINT**	**MMX (naive)**	**MMX (fast)**	**FFLAS**
2^8	18.58	0.34	1.49	670.30
2^9	19.60	0.75	3.07	708.40
2^{10}	35.80	1.77	6.73	933.30
2^{11}	68.30	4.26	14.32	1958.60
2^{12}	148.10	11.01	30.98	5318.30
2^{13}	338.90	29.86	72.42	17568.60
2^{14}	765.10	88.95	183.46	65323.40
2^{15}	2013.70	301.69	435.05	250234.90
2^{16}	5392.80	1055.84	1037.79	987044.30
2^{17}	13984.80	3973.46	2733.15	4066830.50
2^{18}	35307.50	15376.40	8049.31	17133438.90
2^{19}	89413.10	59693.64	20405.06	69320436.10
2^{20}	243933.90	837116.30	805324.30	244552123.40

Table III provides time estimates of the precomputation phase for each implementation. As expected, one can see that our method is based on a long phase of precomputation that

make possible to save some time during the conversions. Despite this long precomputation, our method is really competitive when the number of elements to convert is sufficiently large. For example, if we use an RNS basis of 2^{15} bits, our precomputation phase needs $250ms$ while FLINT's code needs $2ms$. However, for such basis bitsize, our conversion to RNS (resp. from) needs $80\mu s$ and $161\mu s$ per element while FLINT's one needs $165\mu s$ and $316\mu s$. Assuming one needs to convert back and forth with such RNS basis, our implementation will be faster when more than 1000 integers need to be converted. Note also that the code of our precomputation phase has not been fully optimized and we should be able to lower down this threshold.

4. APPLICATION TO INTEGER MATRIX MULTIPLICATION

An obvious application for our algorithms is integer matrix multiplication, and by extension matrix multiplication over rings of the form $\mathbb{Z}/N\mathbb{Z}$ for large values of N. Several computer algebra systems or libraries already rely on Residue Number Systems for this kind of matrix multiplication, such as Magma [Bosma et al. 1997], FFLAS-FFPACK [FFLAS-FFPACK-Team 2016] or Flint [Hart 2010]; in these systems, conversions to and from the RNS are typically done using divide-and-conquer techniques. In this section, after reviewing such algorithms, we give details on our implementation in FFLAS-FFPACK, and compare its performance to other software; we conclude the section with a practical illustration of how computations with polynomials over finite fields can benefit from our results.

4.1. Overview of the algorithm

Let A, B be matrices in respectively $\mathcal{M}_{a\times b}(\mathbb{Z})$ and $\mathcal{M}_{b\times c}(\mathbb{Z})$, such that all entries in the product AB have β-expansion of length at most k, for some positive integer k. Computing the product AB in a direct manner takes $\mathcal{O}(\mathsf{MM}(a,b,c)\,\mathsf{I}(k))$ bit operations as seen in Section 2.1.

Using multi-modular techniques, we proceed as follows. We first generate primes $m_1, m_2, \ldots, m_s$ in an interval $\{\beta^{t-1}, \ldots, \beta^t - 1\}$, for some integer $t \leqslant k$ to be discussed below; these primes are chosen so that we have $\beta^{k+1} < M$, with $M = m_1 m_2 \cdots m_s$, as well as $M < \beta^{k+1}\beta^t$. These constraints on the m_i's imply that $st = \Theta(k)$.

We then compute $A_i = (A \text{ rem } m_i)$ and $B_i = (B \text{ rem } m_i)$ for all $1 \leqslant i \leqslant s$, using the algorithm of Section 3.1. The matrices A_i, B_i have entries at most β^t, so the products $(A_i B_i \text{ rem } m_i)$ can be computed efficiently (typically using fixed precision arithmetic). We can then reconstruct $(AB \text{ rem } M)$ from all $(A_i B_i \text{ rem } m_i)$, using the algorithm of Section 3.2; knowing $(AB \text{ rem } M)$, we can recover AB itself by subtracting M to all entries greater than $M/2$. Choosing the parameter $t' \in \{1, \ldots, t\}$ as in the previous section, the total cost of this procedure is

$$\mathcal{O}\Big(\mathsf{MM}(a,b,c)s\,\mathsf{I}(t) + \big(\mathsf{MM}(s,s',ab) + \mathsf{MM}(s,s',bc) + \mathsf{MM}(ac,s,s')\big)\,\mathsf{I}(t) + s\,\mathsf{I}(t)\log(t)\Big) \quad (7)$$

word operations, with $s' = \lceil st/t' \rceil$; here, the first term describes the cost of computing all products $(A_i B_i \text{ rem } m_i)$, and the second one the cost of the conversions to and from the RNS by means of our new algorithms.

Let us briefly discuss the choice of parameters s, t and t'. We mentioned previously that to lower the cost estimate, one should simply take $t' = t$, so that $s' = s$; let us assume that this is the case in the rest of this paragraph. Since we saw that $st = \Theta(k)$, the middle summand, which is $\Omega(s^2 t)$, is minimal for small values of s. For $s = 1$ and $t = \Theta(k)$, this is the same as the direct approach and we get the same cost estimate $\mathcal{O}(\mathsf{MM}(a,b,c)\,\mathsf{I}(k))$. On the other hand, minimizing the first and third term in the runtime expression then amounts to choosing the smallest possible value of t. The largest possible value of s for a given t is $s = \Theta(\beta^t/t)$, by the prime number theorem since β is a constant; since $st = \Theta(k)$, this *a priori* analysis leads us to choose $t = \Theta(\log(k))$ and $s = \Theta(k/\log(k))$. However, as we will

see below, in practice the choice of moduli is also in large part dictated by implementation considerations.

4.2. Implementation and timings

As mentionnned in Section 3.3, our code has been integrated in the FFLAS-FFPACK library. In order to re-use the fast modular matrix multiplication code [Dumas et al. 2008] together with our simultaneous RNS conversion, we use the following choice of parameters. We set $\beta = 2$, $t' = 16$ and choose the maximum value of t such that $b\ 2^{2t} \leqslant 2^{53}$ (modular matrix multiplication constraint) and $s'2^{t+t'} \leqslant 2^{53}$ (RNS constraint). As an example, with $t = 20$ our code is able to handle matrices up to dimension 8192 with entries up to 189 KBytes ($\simeq 2^{20.5}$ bits).

Let k denote a bound on the bitsize in our input matrices. From Equation 7, one can easily derive a crossover point where RNS conversions become dominant over the modular matrix multiplication. Assuming matrix multiplication of dimension n costs $2n^3$ operations then RNS conversions dominate as soon as $k > 16n/3$. Indeed, modular matrix multiplications cost $2n^3k/t$ while the three RNS conversions (2 conversions to RNS and 1 conversion from RNS) costs $6k^2n^2/16t$ since $t' = 16$. According to the matrix dimensions that will be used in our benchmark, our code will be always dominated by the RNS conversions and we will see its impact.

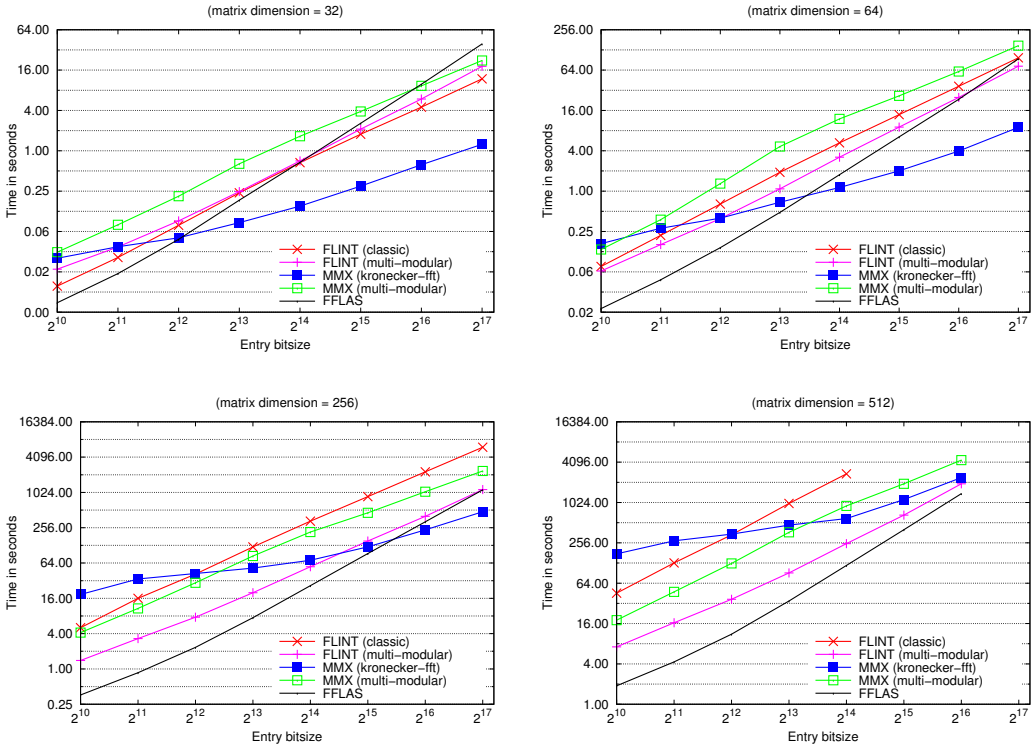


Fig. 1: Multi-precision integer matrix multiplication (time comparisons)

Our benchmarks are done on an Intel Xeon E5-2697 2.6 GHz machine. Figure 1 reports time of integer matrix multiplication for different matrix entries bitsize (abscissa of plots)

and different matrix dimensions (4 different plots). We compared our integer matrix multiplication code in FFLAS-FFPACK with the ones in FLINT [Hart 2010], and Mathemagix [Hoeven et al. 2012] :

— FLINT provides two implementations: the direct algorithm `FLINT (classic)` with some hand-tuned inline integer arithmetic, and multi-modular algorithm `FLINT (multi-modular)` using divide-and-conquer techniques for conversions to and from the RNS. The `fmpz_mat_mul` method in FLINT automatically switches between these two algorithms based on a heuristic crossover point.
— The `Algebramix` package of the Mathemagix library provides three implementations: the direct algorithm, the multi-modular one and the Kronecker+FFT algorithm. The `MMX (kronecker-fft)` reduces multi-precision integer to polynomials over single-precision integers via Kronecker subtitution, and then perform FFT on those polynomials. The `MMX (multi-modular)` plot corresponds to an hybrid multi-modular approach that either uses quadratic or fast divide-and-conquer algorithms for RNS conversions. The generic integer matrix multiplication code in Mathemagix switches between these two strategies according to hardcoded thresholds.
— The `FFLAS` entry correspond to our multi-modular implementations from Section 3.

In order to provide a fair comparison to our code, we decided to not use the generic code using threshold from FLINT and Mathemagix, and call directly the specific underlying implementations.

From Figure 1, one can see that our method improves performance in every case on some initial bitsize range (as our method has a super-linear complexity with respet to the bitsize, it cannot be competitive with fast methods for large bitsize); however, when the matrix dimension is increasing, the benefits of our method tend also to increase. One should note that the Kronecker method with FFT has the best asymptotic complexity in terms of integer bitsize. However, it turns out not to be the best one when matrix dimension is increasing. This is confirmed in Figure 1 where for a given bitsize value (*e.g.* $k = 2^{12}$), `MMX (kronecker-fft)` implementation is the fastest code for small matrix dimension (*e.g.* $n = 32$) while it becomes the worst for larger one (*e.g.* $n = 512$).

4.3. Applications

We conclude this section with an illustration of how improving matrix multiplication can impact further, seemingly unrelated operations. Explicitly, we recall how matrix multiplication comes to play when one deals with finite fields and how our linear algebra implementations of these operations result in significant speed-ups over conventional implementations for some important operations.

Modular composition. Given a field $\mathbb{F}_p$ and polynomials f, g, h of degrees less than n over $\mathbb{F}_p$, modular composition is the problem of computing $f(g) \bmod h$. No quasi-linear algorithm is known for this task, at least in a model where one counts operations in $\mathbb{F}_p$ at unit cost. The best known algorithm to date is due to Brent and Kung [Brent and Kung 1978], and is implemented in several computer algebra systems; it allows one to perform this operations using $\mathsf{C}(n) = \mathcal{O}(\sqrt{n}\mathsf{M}(n) + \mathsf{MM}(\sqrt{n}, \sqrt{n}, n))$ operations in $\mathbb{F}_p$, where $\mathsf{M} : \mathbb{N} \to \mathbb{N}$ is such that one can multiply polynomials of degree n over $\mathbb{F}_p$ in $\mathsf{M}(n)$ operations. One should point out that the Kedlaya-Umans algorithm reaches an almost linear cost, in a boolean model, but to our knowledge no implementation of it has been showed to be competitive with the Brent and Kung approach.

As showed in the run-time estimate, the bottleneck of this algorithm is a matrix multiplication in sizes $(\sqrt{n}, \sqrt{n})$ and $(\sqrt{n}, n)$, and can benefit from our work. Remark that the goal is here to multiply matrices with coefficients defined modulo a potentially large p; this

is done by seeing these matrices over $\mathbb{Z}$ and multiplying them as such, before reducing the result modulo p.

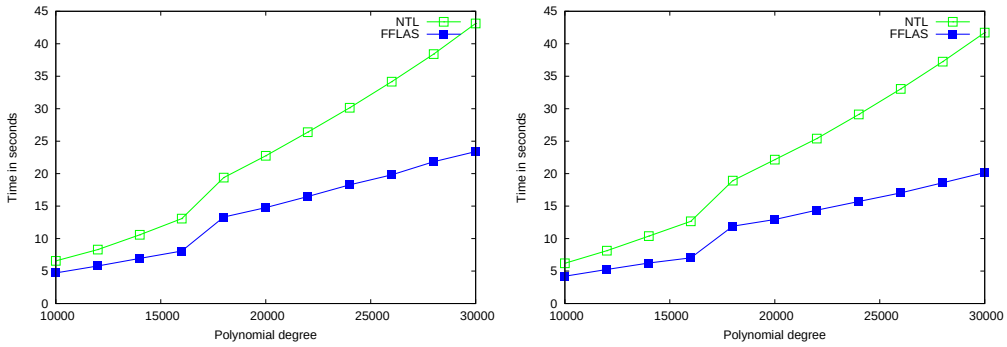
We have implemented modular polynomials composition using our matrix multiplication in C++, on the basis of Shoup's NTL [Shoup 2015], simply by modifying the existing `CompMod` method to use the fast matrix multiplication we implemented in FFLAS-FFPACK. Figure 2a compares our implementation to the preexisting one in NTL on an Intel(R) Core(TM) i7-4790, 3.60GHz machine. As we can see, this readily offers a substantial improvement for primes of 200 bits in our example.

Power projection. Let f be a polynomial of degree n over $\mathbb{F}_p$, and define $F = \mathbb{F}_p[x]/(f)$. For vectors $v, u \in \mathbb{F}_p^n$ denote by $\langle v, u\rangle$ their inner product; this extends to an inner product over F, considering elements of F as vectors in $\mathbb{F}_p^n$. The power projection problem is computing the sequence

$$\langle v, 1\rangle, \langle v, g\rangle, \ldots, \langle v, g^{m-1}\rangle$$

for given integer $m > 0$, $v \in \mathbb{F}_p^n$ and g in F. The best known algorithm for power projection is due to Shoup [Shoup 1994; Shoup 1999], with a runtime that matches that of the Brent and Kung algorithm. The dominant part of the algorithm can be formulated as a matrix multiplication similar to the one for modular composition, this time in sizes $(\sqrt{n}, n)$ and $(n, \sqrt{n})$.

Similar to modular composition, we have modified NTL's `ProjectPowers` routine to use our matrix multiplication implementation. Figure 2b compares our implementation to the built-in method, and shows improvements that very similar to the ones seen for modular composition.

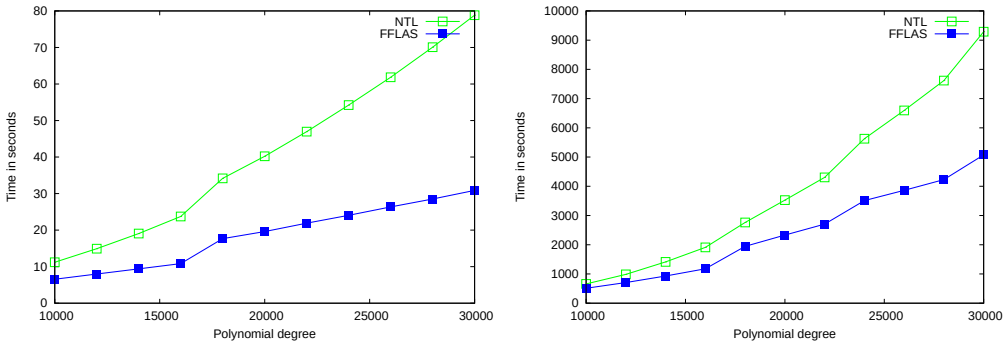


(a) Modular composition (b) Power projection

Minimal polynomial and factorization. The above two operations are key ingredients of many higher level operations in finite fields. Two such operations are *Minimal polynomial computation*, and *Polynomial factorization*. Let $f \in \mathbb{F}_p[X]$ be a polynomial of degree n, and let $a \in \mathbb{F}_p[X]/(f)$. The minimal polynomial of a over $\mathbb{F}_p$ is a monic irreducible polynomial $g \in \mathbb{F}_p[X]$ of degree less than n such that $g(a) = 0 \bmod f$. An efficient algorithm for computing minimal polynomials is presented in [Shoup 1994], using power projection as its main subroutine.

Given $f \in \mathbb{F}_p[X]$, polynomial factorization is the problem of expressing f as a product of irreducible factors. A well-known algorithm for factoring polynomials is due to [Cantor and Zassenhaus 1981]; one of the main operations in this algorithm is modular composition, as explained in [Gathen and Shoup 1992].

We have modified the minimal polynomial and factoring implementations available in NTL to use our new power projection and modular composition algorithms. Figures 2a, 2b compare the methods `MinPolyMod` and `CanZass` of NTL to their new versions on an Intel(R) Core(TM) i7-4790, 3.60GHz machine, showing significant improvements. Here the prime p is random of size 200 bits, and the polynomials are chosen with uniformly random coefficients in $\mathbb{F}_p$.



(a) Minimal polynomial computation (b) Polynomial factorization

5. EXTENDING THE ALGORITHMS FOR LARGER MODULI

5.1. Motivation

Our algorithms for simultaneous conversions of Section 3 work for primes of limited bitsize in practice. Indeed, we have seen in Section 3.3 that we use BLAS to benefit from very fast matrix multiplication and that since BLAS works on double precision floating point numbers, it limits our prime size to about 26 bits. There are enough of those primes for most application of multi-modular techniques. However, there is still an application where such limit can be too restrictive : the multiplication of polynomials with large integer coefficients.

The multi-modular strategy is classically used to multiply such polynomials, but we need to use particular primes. The most advantageous situation for multiplying large modular polynomials is when one uses DFT algorithms over a ring $\mathbb{Z}/p\mathbb{Z}$ which has 2^d-roots of unity, for 2^d larger than the degree of the product [Gathen and Gerhard 2013, Chapter 8]. The existence of 2^d-roots of unity in $\mathbb{Z}/p\mathbb{Z}$ happens if and only if 2^d divides $p-1$. We call *FFT primes* such primes p.

However, the number of FFT primes is much smaller than the number of primes. Since FFT primes are exactly primes that appear in the arithmetic progression $(1+2^d k)_{k\in\mathbb{N}}$, number theory tells us that the number of FFT primes less than x is asymptotically equivalent to the numbers of primes less than x divided by $\varphi(2^d) = 2^{d-1}$ (where φ is Euler's totient function).

In practice, assuming *e.g.* that we multiply polynomials of degree 2^{14} and we use prime's bitsize limit of Section 3 ($m_i < 2^{26}$), we have enough FFT primes to handle input of approximately $2^{12.5}$ bits (724 Bytes), which is not sufficient for some applications (for instance, in many forms of Hensel lifting algorithms, one is led to compute modulo large powers of primes).

If we want to handle larger integer coefficients with multi-modular techniques, we could take at least two different directions. A first direction is to lift the prime size limit of our simultaneous RNS conversion; this requires us to provide efficient (comparable to BLAS) fixed precision matrix multiplication above 26-bits entries. The second direction is to use 3-primes FFT [Gathen and Gerhard 2013, Chapter 8] when we have exhausted all the possible

FFT primes. This option at least triples the asymptotic runtime per bit of the input. For the sake of completeness, we also have to consider the direct approach, *e.g.* Schönhage-Strassen algorithm [Schönhage and Strassen 1971].

In this section, we investigate the first direction and we present a variant of our algorithm that handles larger moduli. Note that already slightly larger moduli will allow one to substantially increase the possible bitsize of coefficients: with our forthcoming technique, we will be able to multiply polynomials up to degree 2^{22} and of coefficient bitsize 2^{20} using primes of 42-bits. Note that this is particularly interesting since FFT performances are almost not penalized when one uses primes up to 53 bits instead of primes of 32 bits as demonstrated in [Hoeven et al. 2016].

5.2. Enlarging moduli size in practice

Since we want moduli above BLAS limit (26-bits) but still want to perform matrix operation using BLAS, we will cut our moduli of bitsize t into κ chunks of bitsize δ. The new parameter δ represents the precision for which numerical matrix multiplications are feasible, *i.e.* $s'\beta^{\delta+t'} \leqslant 2^{53}$. In Section 3, we considered the case where $\kappa = 1$ and $\delta = t$. We will assume that $t' \leqslant \delta$ the same way we assumed $t' \leqslant t$ before. Finally, we will assume that $s' < \beta^{2\delta}$ in the following sections. Indeed, this hypothesis is verified in practice since $t' = 16$ and $\beta = 2$ implies $s'\beta^{2t'} \leqslant s'\beta^{\delta+t'} \leqslant 2^{53} < \beta^{4t'}$ and so $s' < \beta^{2t'} \leqslant \beta^{2\delta}$.

5.3. Conversion to RNS

We start by giving the algorithm, followed by its complexity analysis. As in Section 3.1, we first compute all $[\beta^{it'}]_j = (\beta^{it'} \text{ rem } m_j)$ for $1 \leqslant i < s'$, $1 \leqslant j \leqslant s$ and gather them into the matrix

$$B = \begin{bmatrix} 1 & [\beta^{t'}]_1 & [\beta^{2t'}]_1 & \dots & [\beta^{(s'-1)t'}]_1 \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & [\beta^{t'}]_s & [\beta^{2t'}]_s & \dots & [\beta^{(s'-1)t'}]_s \end{bmatrix} \in \mathcal{M}_{s\times s'}(\mathbb{Z}).$$

Then we write the β^δ-expansion of the matrix B as $B = B_0 + B_1\beta^\delta + \cdots + B_{\kappa-1}\beta^{\delta(\kappa-1)}$ and also compute the matrix C which gathers the $\beta^{t'}$-expansions of the a_j's. We rewrite Equation (4) as

$$(B_0C)_{i,j} + \cdots + \beta^{(\kappa-1)\delta}(B_{\kappa-1}C)_{i,j} \equiv a_j \pmod{m_i}. \tag{8}$$

So, we compute the left-hand side of Equation (8) and reduce it to get $a_j \bmod m_i$.

Complexity estimates. The computation of $[\beta^{(i+1)t'}]_j$ from $[\beta^{it'}]_j$ is a multiplication of integers of length 1 and κ in base β^δ, which costs $\mathcal{O}(\kappa\mathsf{I}(\delta))$, just as the reduction modulo m_j using Equation (2). So computing B reduces to $\mathcal{O}(ss'\kappa\mathsf{I}(\delta))$ arithmetic operations. As before, matrices B_k and C do not involve any arithmetic operations.

Now turning to matrix products, we get $0 \leqslant (B_\ell C)_{i,j} < s'\beta^\delta\beta^{t'} < \beta^{4\delta}$ for all ℓ. This bound is for theoretic purposes ; in practice all these matrix products do not exceed BLAS limit and can be performed efficiently. In Equation (8) all the products $(B_\ell C)$ can be computed in one matrix multiplication of cost $\mathcal{O}(\mathsf{MM}(\kappa s, s', r)\mathsf{I}(\delta))$ by stacking vertically the B_ℓ in a matrix $\bar{B} \in \mathcal{M}_{\kappa s\times s'}(\mathbb{Z})$ and computing $\bar{B}C$. Then the sum of the $(B_\ell C)$ costs $\mathcal{O}(\kappa rs\delta)$ and its reduction $\mathcal{O}(rs\kappa\mathsf{I}(\delta))$ using Equation (2).

Altogether, the cost is dominated by matrix multiplication $\mathcal{O}(\mathsf{MM}(\kappa s, s', r)\mathsf{I}(\delta))$, which matches the cost in Section 3 when $\kappa = 1$. To compare for other values of κ, notice that our cost is $\mathcal{O}(\mathsf{MM}(s, s', r)\mathsf{I}(t))$ using $\mathsf{MM}(\kappa s, s', r) = \mathcal{O}(\kappa\mathsf{MM}(s, s', r))$ and the super-linearity assumption $\kappa\mathsf{I}(\delta) \leqslant \mathsf{I}(t)$. So our new approach retain the asymptotic complexity bound while lifting the restriction on the moduli size.

5.4. Conversion from RNS

Similarly, we adapt the method from Section 3.2 to our setting $m_i < \beta^t \leqslant \beta^{\kappa\delta}$. Recall that we first compute the *pseudo-reconstructions*

$$\ell_i := \sum_{j=1}^{s} \gamma_{i,j} M_j, \quad \text{with} \quad \gamma_{i,j} = (a_{i,j} u_j \text{ rem } m_j), u_j = 1/M_j \bmod m_j$$

for $1 \leqslant j \leqslant s$ and $1 \leqslant i \leqslant r$, followed by a cheap reduction step $a_i = (\ell_i \text{ rem } M)$ with $\ell_i < sM$. Following the notations of Section 3.2, let $G = [\gamma_{i,j}] \in \mathcal{M}_{r\times s}(\mathbb{Z})$ and write $U = [\mu_{j,k}] \in \mathcal{M}_{s\times s'}(\mathbb{Z})$ the matrix of the $\beta^{t'}$-expansions of all M_j. Then $D = [d_{i,k}] = GU \in \mathcal{M}_{r\times s'}(\mathbb{Z})$ satisfies $\ell_i = \sum_{k=0}^{s'-1} d_{i,k}\beta^{kt'}$. According to our setting, $\gamma_{i,j} < m_i < \beta^{\kappa\delta}$, and one cannot use BLAS to perform this product. As before, we expand $G = G_0 + G_1\beta^\delta + \cdots + \beta^{(\kappa-1)\delta} G_{\kappa-1}$ in base β^δ so that we can compute D as

$$D = G_0 U + \beta^\delta G_1 U + \cdots + \beta^{(\kappa-1)\delta} G_{\kappa-1} U. \tag{9}$$

It remains to recover ℓ_i using $\ell_i = \sum_{k=0}^{s'-1} d_{i,k}\beta^{kt'}$ and reduce them modulo M.

Complexity estimates. As in Section 3.2, computing M_i, u_i, $\gamma_{i,j}$ and the final reductions $(\ell_i \text{ rem } M)$ costs $\mathcal{O}((r + s + \log(t))s\mathsf{I}(t))$. Now focusing on the linear algebra part, the products $G_i U$ have coefficients less than $s\beta^{\delta+t'} \leqslant s'\beta^{\delta+t'} < \beta^{4\delta}$; as in Section 5.3 all these products can be computed at cost $\mathcal{O}(\mathsf{MM}(\kappa r, s, s')\mathsf{I}(\delta))$ by stacking vertically the G_i. Then we shift and sum these products to recover D at cost $rs'\kappa\delta$ and recovering the ℓ_i in the same cost.

So we get a total cost of $\mathcal{O}(\mathsf{MM}(\kappa r, s, s')\mathsf{I}(\delta) + (r + s + \log(t))s\mathsf{I}(t))$ which boils down to $\mathcal{O}(\mathsf{MM}(r, s, s')\mathsf{I}(t) + s\mathsf{I}(t)\log(t))$ because $\mathsf{MM}(\kappa r, s, s') = \mathcal{O}(\kappa\mathsf{MM}(r, s, s'))$ and $\kappa\mathsf{I}(\delta) \leqslant \mathsf{I}(t)$. This corresponds to the same complexity of Section 3.2. Here again, our modification for RNS conversion with larger moduli does not change the asymptotic complexity bound while still allowing the use of BLAS.

5.5. Implementation and timings

Our modified RNS conversions have been implemented in the FFLAS-FFPACK package.

We restrict our implementation to $\kappa = 2$ as it offers a sufficient range of values for integer polynomial multiplication as mentioned in Section 5.1. As in Section 3.3, we chose $\beta = 2$ and $t' = 16$ to simplify Kronecker substitution. The value of δ is dynamically chosen to ensure $s'\beta^{\delta+16} \leqslant 2^{53}$. Since we chose $\kappa = 2$, this means our primes must not exceed $2^{2\delta}$. For small RNS basis of bitsize less than 2^{15} ($\simeq$ 4 KBytes), we can chose the maximum value $\delta = 26$. For larger RNS basis, we need to reduce the value of δ, e.g. with $\delta = 21$ one can use 42-bits primes and reach RNS basis of 2^{20} bits ($\simeq$ 131 KBytes).

Our implementation is similar to the one in Section 3.3, and we use the same tricks to improve performances. In order to speed-up our conversions with larger primes, we always stack the matrices to compute the κ matrix product using one larger multiplication. We have seen that the complexity estimates of stacking are always better because of fast matrix multiplication algorithms. And in practice, it is often best to have larger matrices to multiply because peak performance of BLAS are attained starting from a certain matrix dimension. Furthermore, doubling a matrix dimension may offer an extra level of sub-cubic matrix multiplication in FFLAS-FFPACK.

We perform our benchmark on an Intel Xeon E5-2697 2.6GHz. As in section 4.2, we choose the number of elements to convert from/to RNS to be 128^2, and the bitsize of integer inputs are almost twice as small as the RNS basis bitsize. In table IV, we report the conversion time per element for a given RNS bitsize. As matter of comparison, we report the time of

our RNS conversions when $\kappa = 1$, corresponding to "small" prime, and also the ones from FLINT library which is the fastest available contestant with 59-bits primes.

Values reported in Table IV confirm our conclusion that performances should not asymptotically changed for different value of κ. However, for small RNS bitsize, one may remark slight difference between $\kappa = 1$ and $\kappa = 2$. For this size, the matrix multiplication is not dominant in the complexity and the constant behind second order terms roughly double the cost. Furthermore, our implementation with $\kappa = 2$ does not benefit from all SIMD vectorization code that has been done with $\kappa = 1$, explaining the variation.

Note that for larger RNS bitsize, the conversion to RNS with $\kappa = 2$ is faster then the one with $\kappa = 1$. This is of course due to larger matrix multiplication which benefit more from sub-cubic matrix multiplication of FFLAS-FFPACK. This is not true for conversion from RNS as the last step is almost twice more costly than our method with $\kappa = 1$. Finally, comparing to FLINT, we can see that our approach can improve performances up to a factor of two. For very large bitsize, the fast divide-and-conquer approach of FLINT kicks in and becomes more advantageous.

Table IV: Simultaneous RNS conversions (time per integer in μs)

	To RNS			From RNS		
RNS bitsize	**FLINT**	**FFLAS**	**FFLAS**	**FLINT**	**FFLAS**	**FFLAS**
		($\kappa = 1$)	($\kappa = 2$)		($\kappa = 1$)	($\kappa = 2$)
m_i	$< 2^{59}$	$< 2^{27}$	$< 2^{53}$	$< 2^{59}$	$< 2^{27}$	$< 2^{53}$
2^8	0.17	0.06	0.15	0.63	0.34	0.63
2^9	0.35	0.13	0.24	1.34	0.39	0.70
2^{10}	0.84	0.27	0.53	3.12	0.72	1.39
2^{11}	2.73	0.75	1.20	6.92	1.57	2.46
2^{12}	7.03	1.92	2.92	16.79	3.94	5.15
2^{13}	17.75	5.94	8.01	40.73	12.77	14.98
2^{14}	50.90	21.09	25.05	113.19	43.13	47.54
2^{15}	165.80	80.82	85.38	316.61	161.44	167.93
2^{16}	506.91	298.86	299.11	855.48	609.22	629.69
2^{17}	1530.05	1107.23	1099.52	2337.96	2259.84	2375.98
2^{18}	4820.63	4114.98	4043.68	7295.26	8283.64	8550.81
2^{19}	13326.13	15491.90	15092.94	18529.38	31382.81	33967.42
2^{20}	37639.48	55370.16	67827.24	48413.81	111899.47	121432.66

REFERENCES

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.

P. Barrett. 1986. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology, CRYPTO'86 (LNCS)*, Vol. 263. Springer, 311–326.

A. Borodin and R. Moenck. 1974. Fast modular transforms. *J. Comput. System Sci.* 8, 3 (1974), 366–386.

W. Bosma, J. Cannon, and C. Playoust. 1997. The Magma algebra system. I. The user language. *J. Symbolic Comput.* 24, 3-4 (1997), 235–265. DOI:http://dx.doi.org/10.1006/jsco.1996.0125 Computational algebra and number theory (London, 1993).

R. P. Brent and H. T. Kung. 1978. Fast algorithms for manipulating formal power series. *Journal of the Association for Computing Machinery* 25, 4 (1978), 581–595.

R. P. Brent and P. Zimmermann. 2010. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA.

D. G. Cantor and H. Zassenhaus. 1981. A new algorithm for factoring polynomials over finite fields. *Math. Comp.* (1981), 587–592.

S. Cook. 1966. *On the minimum computation time of functions*. Ph.D. Dissertation. Harvard University.

D. Coppersmith and S. Winograd. 1990. Matrix multiplication via arithmetic progressions. *J. Symb. Comp* 9, 3 (1990), 251–280.

J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, G. Villard, and others. 2002. LinBox: A generic library for exact linear algebra. In *Proceedings of the 2002 International Congress of Mathematical Software, Beijing, China*. World Scientific Pub, 40–50.

J.-G. Dumas, P. Giorgi, and C. Pernet. 2008. Dense Linear Algebra over Word-Size Prime Fields: the FFLAS and FFPACK Packages. *ACM Trans. on Mathematical Software (TOMS)* 35, 3 (2008), 1–42. DOI:http://dx.doi.org/10.1145/1391989.1391992

FFLAS-FFPACK-Team. 2016. *FFLAS-FFPACK: Finite Field Linear Algebra Subroutines / Package* (v2.2.2 ed.). http://github.com/linbox-team/fflas-ffpack.

M. Frigo and S. G. Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

M. Fürer. 2007. Faster integer multiplication. In *STOC'07*. ACM, 57–66.

J. von zur Gathen and J. Gerhard. 2013. *Modern Computer Algebra* (3 ed.). Cambridge University Press, New York, NY, USA.

J. von zur Gathen and V. Shoup. 1992. Computing Frobenius maps and factoring polynomials. *Computational complexity* 2, 3 (1992), 187–224.

P. Giorgi, L. Imbert, and T. Izard. 2013. Parallel modular multiplication on multi-core processors. In *21st IEEE Symposium on Computer Arithmetic (ARITH)*. 135–142. DOI:http://dx.doi.org/10.1109/ARITH.2013.20

GMP-Team. 2015. Multiple precision arithmetic library. (2015). https://gmplib.org/.

K. Goto and R. van de Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Trans. Math. Software* 35, 1 (2008), 4.

W. B. Hart. 2010. Fast library for number theory: an introduction. In *Mathematical Software–ICMS 2010*. Springer, 88–91. http://www.flintlib.org/.

J. van der Hoeven, G. Lecerf, B Mourrain, P. Trébuchet, J. Berthomieu, D. N. Diatta, and A. Mantzaflaris. 2012. Mathemagix: the quest of modularity and efficiency for symbolic and certified numeric computation? *ACM Communications in Computer Algebra* 45, 3/4 (2012), 186–188. http://www.mathemagix.org/.

J. van der Hoeven, G. Lecerf, and G. Quintin. 2016. Modular SIMD Arithmetic in Mathemagix. *ACM Trans. Math. Softw.* 43, 1 (Aug. 2016), 5:1–5:37. DOI:http://dx.doi.org/10.1145/2876503

MKL Intel. 2007. Intel math kernel library. (2007).

F. Le Gall. 2014. Powers of Tensors and Fast Matrix Multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC '14)*. ACM, New York, NY, USA, 296–303. DOI:http://dx.doi.org/10.1145/2608628.2608664

LinBox-Team. 2016. *LinBox: C++ library for exact, high-performance linear algebra* (v1.4.2 ed.). http://github.com/linbox-team/linbox.

Clément Pernet. 2015. Exact Linear Algebra Algorithmic: Theory and Practice. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation (ISSAC '15)*. ACM, New York, NY, USA, 17–18. DOI:http://dx.doi.org/10.1145/2755996.2756684

M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* 93, 2 (2005), 232– 275.

A. Schönhage and V. Strassen. 1971. Schnelle Multiplikation grosser Zahlen. *Computing* 7 (1971), 281–292.

V. Shoup. 1994. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation* 17, 5 (1994), 371–391.

V. Shoup. 1995. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation* 20, 4 (1995), 363–397.

V. Shoup. 1999. Efficient computation of minimal polynomials in algebraic extensions of finite fields. In *ISSAC'99*. ACM, 53–58.

V. Shoup. 2015. NTL: A library for doing number theory. (2015). http://www.shoup.net/ntl/.

A. Stothers. 2010. *On the Complexity of Matrix Multiplication*. Ph.D. Dissertation. University of Edinburgh.

V. Strassen. 1969. Gaussian elimination is not optimal. *Numer. Math.* 13, 4 (1969), 354–356. DOI:http://dx.doi.org/10.1007/BF02165411

V. Vassilevska Williams. 2012. Multiplying Matrices Faster Than Coppersmith-winograd. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing (STOC '12)*. ACM, New York, NY, USA, 887–898. DOI:http://dx.doi.org/10.1145/2213977.2214056

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput.* 27, 1 (2001), 3–35.