Simultaneous conversions with the Residue Number System using linear algebra

Javad Doliskani, 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.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm design and analysis; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computations in finite fields.*

General Terms: Algorithms, Experimentation, Performance.

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]. High-performance kernels for linear algebra or polynomial arithmetic are also available in the context of computations over the integers or finite fields, through libraries or systems such as NTL [Shoup 1995], FLINT [Hart 2010], FFLAS-FPACK [Linbox-Team 2015] or Magma [Bosma et al. 1997].

In this paper, we are interested in the latter kind of computation, in the context of multiple precision arithmetic: 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 we consider a basic operation such as the multiplication of these matrices or polynomials. There exist multiple applications to this fundamental operation; we illustrate this in the last section of this paper with a discussion of polynomial factorization.

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 this paper, we present new techniques for residue number systems that are applicable in a wide range of situations. In many cases, the bottlenecks in such an approach is 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. Our algorithm offers an alternative to this approach, for 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 its complexity is inferior to that of asymptotically fast methods, our algorithm behaves extremely well in practice, as it allows us to rely on high-performance libraries for matrix multiplication.

(**todo:** Outline of the paper, acknowledgements.)

2. PRELIMINARIES AND NOTATION

We denote resp. by $(a \operatorname{rem} b)$ and $(a \operatorname{quo} b)$ the remainder and quotient of the Euclidean division of $a \in \mathbb{Z}$ by $b \in \mathbb{N}$, with $0 \leqslant (a \operatorname{rem} b) < b$; we extend the notation $(a \operatorname{rem} b)$ to any rational $a = n/d \in \mathbb{Q}$ such that $\gcd(d, b) = 1$ to be the unique α in $\{0, \dots, b-1\}$ such that $\alpha d = n \bmod b$. We call informally a *pseudo-reduction* of a modulo p the computation of b such that $a = b \bmod p$ and b "not too big" compared to p. In practice, we often will have that $b = O(p^2)$.

We shall follow a bit complexity model in this paper, in the sense that our complexity statements are given in terms of number of bit operations. In some cases, such as polynomial multiplication, it is more convenient to state the complexity based on the number of operations in the coefficient field. These complexity statements remain the same in the bit complexity model when the coefficient field has a constant element's size, i.e. a constant number of machine words.

We let $\mathsf{I}(n)$ be such that two integers a, b of sizes $|a|, |b| < 2^n$ can be multiplied in $\mathsf{I}(n)$ bit operations, assuming the base-2 expansions of a and b are given as input. The best known bound for $\mathsf{I}(n)$ is $n \log(n) 2^{O(\log^*(n))}$ uses Fast Fourier Transform and is due to Fürer [**?**] (see [**?**; **?**] for further refinments). The arithmetic complexity of multiplying two polynomials f, g of degrees $\deg(f), \deg(g) \leq n$ is denoted by $\mathsf{M}(n)$. This means given a field K and polynomials f, g of degree n over K, fg can be computed in $\mathsf{M}(n)$ arithmetic operations in K. The best known bound for $\mathsf{M}(n)$ is $O(n \log(n) \log\log(n))$ again using Fast Fourier Transform [Schönhage and Strassen 1971; Cantor and Kaltofen 1991]. We assume that I and M satisfy the super-linearity assumptions of [Gathen and Gerhard 2013, Chapter 8].

**Notation $\mathsf{MM}(r, s)$ **

3. THE RESIDUE NUMBER SYSTEM

The Residue Number System (RNS) is a non-positional number system that allows us to represent a finite subset of integers. Let $m_1, m_2, \dots, m_s \in \mathbb{N}$ be pairwise coprime integers, and let $M = m_1 \times m_2 \times \dots \times m_s$. Then any integer $a \in [0, \dots, M-1]$ can be uniquely determined by its residues $([a]_{m_1}, [a]_{m_2}, \dots, [a]_{m_s})$. The uniqueness of this residual representation is ensured by the the ring isomorphism

$$\mathbb{Z}/M\mathbb{Z} \simeq \mathbb{Z}/m_1\mathbb{Z} \times \cdots \times \mathbb{Z}/m_s\mathbb{Z}. \tag{1}$$

This isomorphism is often called the Chinese Remainder Theorem (CRT) [Gathen and Gerhard 2003, Section 5.4], and this residual representation is called the Residue Number System. Assume the m_i are fixed, and denote $[a]_{m_i}$ by $[a]_i$ so that $([a]_1, [a]_2, \dots, [a]_s)_{\mathrm{RNS}}$ is the representation of the integer $a \in \mathbb{N}$.

In the RNS we can perform addition and multiplication independently on each residual. Let $a, b, c, d \in \mathbb{N}$ such that $c \equiv a + b \bmod M$ and $d \equiv a \times b \bmod M$ then

$$([c]_1, [c]_2, \dots, [c]_s)_{\mathrm{RNS}} = ([a]_1 + [b]_1 \bmod m_1, \dots, [a]_s + [b]_s \bmod m_s)_{\mathrm{RNS}} \tag{2}$$
$$([d]_1, [d]_2, \dots, [d]_s)_{\mathrm{RNS}} = ([a]_1 \times [b]_1 \bmod m_1, \dots, [a]_s \times [b]_s \bmod m_s)_{\mathrm{RNS}} \tag{3}$$

It is obvious from equations 2 and 3 that addition and multiplication in RNS require $O(s)$ operations on the residuals. One may note that division in RNS is possible in the same way (individually on each residual) only when the division is exact.

In order to benefit from RNS representation, one often need to convert back and forth between the classic positional number system and RNS. Of course, these conversions are costly and must be avoided when possible. However, when the number of operations in RNS is high enough, these conversions can be neglected and the RNS approach yields the most efficient solution.

4. CONVERTING ONE INTEGER BETWEEN CLASSICAL AND RESIDUE NUMBER SYSTEMS

In this section, we give an overview of two approaches for converting integers between classical and residue number systems: the naive approach, and the quasi-linear approach.

Cost of both rem and quo to have cost of exact division

We will recall in this paragraph the cost of computing $a \operatorname{rem} p$ together with $a \operatorname{quo} p$ for various sizes of a and p. First, we can suppose w.l.o.g. that $a \leq 0$ and $a \leq p$. Let $a \in \mathbb{Z}$ and $p \in \mathbb{N}$ such that $|a| < 2^n$ and $p < 2^m$.

(1) $n = \Theta(m)$. Algo of [Barrett 1986; Montgomery 1985] when $0 \leq a < \beta^2$, $\beta/2 < p < \beta$ for some β (typically β is a power of two) [Brent and Zimmermann 2010][Section 2.4.1]. Cost $O(\mathsf{I}(\log \beta))$. –

$$O\left(\mathsf{I}(m)\right) \tag{4}$$

(2) $n \gg \Theta(m)$ Is established in 3.1 - Algorithm 1. Mentionned in [Brent and Zimmermann 2010][Section 2.4.1]

$$O\left(\frac{n}{m}\mathsf{I}(m)\right) \tag{5}$$

(3) $n - m \ll \Theta(m)$ Will be useful to finish the Euclidean division when we have a pseudo-reduction. One reference could be [Giorgi et al. 2013][Algorithm 2].

$$O\left(\mathsf{I}(n-m)\frac{m}{n-m}\right) \tag{6}$$

(4) Fast version of 5. Established in 3.2. $C(s) = O(\frac{\mathsf{I}(n/m)\log(n/m)}{n/m}\mathsf{I}(\log m))$??.

4.1. Summary of complexities

In the following table, we recall all the complexities for conversions with RNS representation. We assume that the RNS basis is given as $(m_1, \ldots, m_s)$ such that each $m_i < 2^t$ and integers to convert have n-bits with $n = \Theta(st)$.

	conversion from RNS	conversion to RNS
naive approach	$\mathcal{O}(s^2\mathsf{M}(t))$	$\mathcal{O}(s^2\mathsf{M}(t))$
fast approach	$\mathcal{O}(\mathsf{M}(s)\log s\ \mathsf{M}(t))$	$\mathcal{O}(\mathsf{M}(s)\log s\ \mathsf{M}(t))$

5. SIMULTANEOUS RNS CONVERSIONS

The algorithms of Section 4 enable us to convert back and forth between individual integers and their representations in a given residue number system. In this section, we discuss the problem of performing these conversions simultaneously between a given set of integers $a_1, \ldots, a_r \in \mathbb{Z}$ and their representations in a given residue number system $m_1, \ldots, m_s$. Assume for the entire section that $m_1, \ldots, m_s < \beta$ and $a_j < \beta^n$.

5.1. Direct approach

The simplest approach is to apply the conversions of Section 4 to each a_i separately. Of course, the precomputations of these algorithms are independent of the input integer. So, they are done once and for all. The complexity of simultaneous conversions is then $O(rs^2\mathsf{I}(\log\beta))$ for the naive approach, and $O(r\mathsf{I}(s)\log s\mathsf{I}(\log\beta))$ for the quasi-linear approach.

Note that, in term of implementation, simultaneous reductions using the naive algorithms can benefit from vectorized (SIMD) instructions to lower the constant, and can be easily parallelized. On the other hand, simultaneous reductions using the quasi-linear approach do not benefit from SIMD instructions (or at least not straightforwardly).

5.2. Linear Algebra approach

Conversion to RNS. The conversion to RNS is split up to two phases: first we reduce the computation of pseudo-reductions of $a_i \bmod m_\ell$ to the reductions of $\beta^j \operatorname{rem} m_\ell$ for $0 \leqslant j < n$ and $1 \leqslant \ell \leqslant s$ using one matrix multiplication. Then we turn the pseudo-reductions into reductions in negligible time. We can precompute the values $r_{j,\ell} := 2^{\beta j} \operatorname{rem} m_\ell$, which do not depend on the a_i, once and for all. The costs of the precomputation is $O(sn\mathsf{I}(\log(\beta)))$. It is done by incrementally computing the powers β^j once, and then reducing them modulo the m_ℓ. The simultaneous pseudo-reductions are done as follows. We first write the expansion in base β of a_i:

$$a_i = \sum_{j=0}^{n-1} c_{i,j} 2^{\beta j}, \qquad 1 \leqslant i \leqslant r.$$

Let $a_{i,\ell} := \sum_{j=0}^{n-1} c_{i,j} r_{j,\ell}$ so that $a_{i,\ell} \equiv a_i \bmod m_\ell$. The value $a_{i,\ell}$ is bounded by $n\beta^2$, since $a_i < \beta^n$ by assumption. We say that $a_{i,\ell}$ is a pseudo-reduction of a_i modulo p_ℓ. The values $a_{i,\ell}$ can be computed using linear algebra : $(a_{i,\ell}) \in \mathcal{M}_{r\times s}(\mathbb{Z})$ is the product of $(c_{i,j}) \in \mathcal{M}_{r\times n}(\mathbb{Z})$ and $(r_{j,\ell}) \in \mathcal{M}_{n\times s}(\mathbb{Z})$. If we write $a_{i,\ell} = [a_i]_{m_\ell}$, and $r_{j,\ell} = [2^{\beta j}]_{m_\ell}$ the pseudo-reduction is done using the following matrix multiplication.

$$\begin{bmatrix} [a_1]_{m_1} & \cdots & [a_1]_{m_s} \\ [a_2]_{m_1} & \cdots & [a_2]_{m_s} \\ \vdots & \ddots & \vdots \\ [a_r]_{m_1} & \cdots & [a_r]_{m_s} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{r1} & c_{r2} & \cdots & c_{rn} \end{bmatrix} \times \begin{bmatrix} 1 & 1 & \cdots & 1 \\ [2^\beta]_{m_1} & [2^\beta]_{m_2} & \cdots & [2^\beta]_{m_s} \\ \vdots & \vdots & \ddots & \vdots \\ [2^{\beta(n-1)}]_{m_1} & [2^{\beta(n-1)}]_{m_2} & \cdots & [2^{\beta(n-1)}]_{m_s} \end{bmatrix}$$

This product costs $O(\mathsf{MM}(r,n,s)\mathsf{I}(\log\beta))$ where $\mathsf{MM}(,,)$ is the complexity of non-square matrix multiplication. In the most common case we have $a_j < m_1 \cdots m_s$, which means $n = O(s)$. In that case, the above product can be split into r/s products of square matrices of dimension s. The complexity then reduces to $O(rs^{\omega-1}\mathsf{I}(\log(\beta)))$. Now, the cost of computing the remainder $(a \operatorname{rem} m)$, when $a < n\beta^2$ and $m < \beta$, is $O(\mathsf{I}(\log(\beta)))$. Therefore, our final step to compute the simultaneous reductions costs $O(rs\mathsf{I}(\log(\beta)))$.

Conversion from RNS. We perform *simultaneous pseudo-reconstructions* as follows. Recall that we denote $M = \Pi_{i=1}^{s} m_i$ and $M_i = (M/m_i) \operatorname{rem} m_i$ for $1 \leqslant i \leqslant s$. Let $l_i := \sum_{j=1}^{s} a_{i,j} M_j [M_j^{-1}]_{m_j}$ so that $a_i = l_i \bmod M$ (see Formula (**??**)) and $l_i < M\beta$. We say that l_i are pseudo-reconstructions of $(a_{i,\ell})$ modulo $m_1, \ldots, m_s$. Let us precompute $M_\ell[M_\ell^{-1}]_{m_\ell}$ for all $1 \leqslant \ell \leqslant s$. If we have precomputed the M_ℓ before (*e.g.* during the reduction step), then we can compute $[M_\ell]_{m_\ell}$ from M_ℓ in time $O(s\mathsf{I}(\log\beta))$, then $[M_\ell^{-1}]_{m_\ell}$ in time $O(\mathsf{I}(\log\beta)\log\log\beta)$ and finally $M_\ell[M_\ell^{-1}]_{m_\ell}$ in time $O(s\mathsf{I}(\log\beta))$. When β is a constant, this comes down to $O(s)$ bit operations. Now we write $M_\ell[M_\ell^{-1}]_{m_\ell}$ in base β as

$$M_j[M_j^{-1}]_{m_j} = \sum_{k=0}^{s-1} e_{j,k}\beta^k,$$

and then we can write l_i as

$$l_i = \sum_{j=1}^{s} a_{i,j} M_j[M_j^{-1}]_{m_j} = \sum_{j=1}^{s} a_{i,j} \sum_{k=0}^{s-1} e_{j,k}\beta^k = \sum_{k=0}^{s-1} \left(\sum_{j=1}^{s} a_{i,j} \cdot e_{j,k} \right) \beta^k.$$

The last inner sum can be computed using the following matrix multiplication

$$\begin{bmatrix} d_{11} & d_{12} & \cdots & d_{1s} \\ d_{21} & d_{22} & \cdots & d_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ d_{r1} & d_{r2} & \cdots & d_{rs} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1s} \\ a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{r1} & a_{r2} & \cdots & a_{rs} \end{bmatrix} \times \begin{bmatrix} e_{11} & e_{12} & \cdots & e_{1s} \\ e_{21} & e_{22} & \cdots & e_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ e_{s1} & e_{r2} & \cdots & e_{ss} \end{bmatrix}$$

This product can be computed in $O\left(rs^{\omega-1}\mathsf{I}(\log(\beta))\right)$ bit operations. From the matrix (d_{ij}) we get $l_i = \sum_{k=0}^{s-1} d_{i,j}\beta^k$. Note that $(d_{i,j})$ are not the exact coefficients of the β-expansion of l_i. But we can still compute the l_is in time $O(rs\mathsf{I}(\log\beta))$ since $d_{i,j} \leqslant s\beta^2$ and $s \leqslant \beta$ by assumption. The final step of the reconstruction consists in reducing l_i modulo M. This step is relatively cheap since l_i is almost reduced. Using $s < \beta$ and $l_i = O(s\beta^{s+2}) = O(\beta^{s+2})$ and $M = O(\beta^s)$, the last reduction step costs $O\left(s\mathsf{I}(\log\beta)\right)$ per l_i. Thus a total cost of $O(rs\mathsf{I}(\log\beta))$.

6. MATRIX MULTIPLICATION WITH MULTI-PRECISION INTEGER ENTRIES

In this section, we show how two matrices with multi-precision integer entries can be multiplied efficiently. The idea is to reduce to multiplication of matrices with small entries, which can then be performed using highly efficient architecture-aware algorithms.

Let $A, B \in \mathbb{Z}^{n\times n}$ be integer matrices such that $\|AB\|_\infty < 2^k$, i.e. the entry size of the product is at most k bits, for a positive integer k. Computing the product using the naive algorithm takes $O(n^\omega \mathsf{I}(k))$. This neither give the best complexity nor the best practical result as we will see later on. To compute AB efficiently we proceed as follows. We first generate enough primes $m_1, m_2, \ldots, m_s$ with $m_i < \beta$ such that $\|AB\|_\infty < M = m_1 m_2 \cdots m_s$. The number s of the primes is obviously bounded by k, so we have $s = O(k)$. Next, we compute $A_i = A \bmod m_i$ for all $1 \leq i \leq s$, and similarly for B_i. This is done using the algorithms in Section 5. Now the matrices A_i, B_i have small entries, and the products A_iB_i can be computed efficiently. Finally, we reconstruct AB from A_iB_i again using the algorithms in Section 5. Algorithm 1 summarizes this approach.

Algorithm 1: Multi-Modular matrix multiplication

```
Input: $A, B$, and precomputed values related to $m_i$, $1 \leq i \leq k$
Output: $AB$
for $i = 1$ to $k$ do
    $A_i \leftarrow A \bmod m_i$
    $B_i \leftarrow B \bmod m_i$
for $i = 1$ to $k$ do
    $C_i \leftarrow A_iB_i$
Reconstruct $AB$ from $C_i$
return $AB$
```

Here, we do $O(k)$ matrix-matrix multiplications with small entries at the cost of $O(n^\omega k)$. Adding the costs of reduction and reconstruction, we get the final complexities of $O(n^\omega k + n^2k^2)$ for the naive RNS, $O(n^\omega k + n^2\mathsf{I}(k)\log(k))$ for quasi-linear RNS, and $O(n^\omega k + n^2k^{\omega-1})$ for the linear algebra RNS.

The above technique of reducing to matrices of small integers is not unique to the RNS approach. Other multi-precision integer multiplication approaches such as the ones in [Schönhage and Strassen 1971] and [Pollard 1971] can be used, in much the same spirit. In [Schönhage and Strassen 1971], the input integer is recursively split into vectors of smaller chunks, and then FFT conversions are used to multiply those vectors. Adapting the algorithm to matrices, one requires $O(k\log(k))$ multiplications of matrices with small entries, and n^2 applications of FFT each costing $O(k\log(k)\log\log(k))$ operations. This gives the overall complexity of $O(n^\omega k + n^2k\log(k)\log\log(k))$ for multi-precision integer matrix multiplication.

To adapt the algorithm in [Pollard 1971] we consider matrices of polynomials $\mathcal{A}, \mathcal{B} \in \mathbb{Z}[X]^{n\times n}$ with the following conditions:

(1) $\|\mathcal{A}_{ij}\|, \|\mathcal{B}_{ij}\| < 2^\beta$, i.e. polynomial entries have coefficients smaller than 2^β.
(2) $\mathcal{A}(2^\beta) = A, \mathcal{B}(2^\beta) = B$, i.e. evaluating the polynomial entries gives the input.
(3) $\deg(\mathcal{A}), \deg(\mathcal{B}) < d = O(k)$, i.e. polynomial entries have degrees smaller than d.

We also let $M = m_1m_2\cdots m_t$ with primes $m_i < 2^{O(1)}$ such that $d2^{2\beta+\log n} < M < 2^{2\beta+\log n+\log k}$. Then AB can be computed using t products of polynomials matrices over $\mathbb{Z}[X]/m_i\mathbb{Z}[X]$ and CRT. This requires d multiplications of matrices of small integers, and n^2 polynomial multiplications. The overall complexity is therefore $tO(n^\omega d + n^2\mathsf{M}(d))$. We have $t = O(\log n + \log k)$. Since $d = O(k)$ and $d < m_i$ the complexity simplifies to $O(n^\omega kC + n^2k\log kC)$ where $C = O(\log n + \log k)$. Table I summarizes the asymptotic complexities of the mentioned matrix multiplication algorithms.

Table I: Integer matrix multiplication complexities

Algorithm	**Complexity**
Naive algorithm	$O(n^\omega \mathsf{I}(k))$
Multi-modular + naive CRT	$O(n^\omega k \quad +n^2k^2)$
Multi-modular + fast CRT [Borodin and Moenck 1974]	$O(n^\omega k \quad +n^2\mathsf{I}(k)\log(k))$
Kronecker + multi-mod FFT [Pollard 1971]	$O(n^\omega kC \quad +n^2k\log(k)C)$
[Schönhage and Strassen 1971]	$O(n^\omega k \quad +n^2k\log(k)\log\log(k))$
Multi-modular + linear algebra CRT	$O(n^\omega k \quad +n^2k^{\omega-1})$

7. IMPLEMENTATION

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

	From RNS			To RNS		
bitsize	**FLINT**	**Mathemagix**	**FFLAS**	**FLINT**	**Mathemagix**	**FFLAS**
2^7	0.4	0.3	0.1	0.1	0.2	0.1
2^8	1.1	0.5	0.2	0.4	0.6	0.1
2^9	2.5	0.9	0.3	1.1	1.5	0.2
2^{10}	5.6	2.1	0.6	2.9	3.5	0.5
2^{11}	11.8	6.2	1.5	7.0	8.3	1.4
2^{12}	29.0	19.0	4.3	16.7	22.9	4.0
2^{13}	66.1	68.3	14.5	40.7	64.5	13.5
2^{14}	168.4	229.4	52.0	112.9	220.8	49.6
2^{15}	534.0	1174.5	235.4	361.0	928.7	217.1
2^{16}	1150.6	3985.1	747.6	977.5	3016.7	711.2
2^{17}	2677.2	13194.2	2355.6	2516.7	10392.9	2156.7

We have implemented our algorithm using C++ as a part of FFLAS-FFPAC library [Linbox-Team 2015]. In our experiments, we have chosen $\beta = 16$, i.e. a two-byte machine word. As mentioned in Section 6, we reduce multi-precision matrix multiplication to machine-word matrix multiplication. To reach peak performances of CPUs for the machine-word case we have used the standard interface BLAS [Dongarra et al. 1988]. It is a reference implementation that has many optimized versions such as ATLAS [Whaley et al. 2001], MKL [Intel 2007], GotoBlas/OpenBlas [Goto and Van De Geijn 2008]. Through these implementations one can exploit architecture- level optimizations such as

— blocking: data re-use in the CPU cache as much as possible
— packing: data rearrangement for contiguous accesses and low cache misses
— SIMD: simultaneous data manipulation using vector operations.

The provided machine-word matrix multiplications use the standard 64-bit double floating point entries. Since we want exact computations, we can only use the 53-bit mantissa part. Therefore, given matrices of dimension n we choose the moduli m_i such that $n(m_i - 1)^2 < 2^{53}$. This makes sure the exact result of the matrix multiplication fits in a double-precision floating point. For handling multi-precision operations we have used GMP [GMP-Team 2015]. The GMP large integer data structure is so that 2^{β}-adic conversions are done by simply casting to `unit16_t*` pointers.

To turn pseudo-reductions to reduction it suffices to subtract entries by $-q_i m_i$ with $|q_i| < 2^{2\beta + \log k}$. For this, we have implemented an SIMD version of the Barrett reduction [Barrett 1986]. To minimize cache misses we store modular matrices contiguous to each other in form strides as shown in following.

$A \bmod m_1$	$A \bmod m_2$	$\cdots$	$A \bmod m_k$

We have compared our results with FLINT [Hart 2010], and Mathemagix [van der Hoeven et al. 2012] libraries both of which implement fast multi-precision integer matrix multiplication. FLINT provides two implementations: the naive algorithm with some hand-tuned inline integer arithmetic, and the multi-modular algorithm + fast CRT. The `fmpz_mat_mul` method in FLINT automatically switches between these two algorithms based on a heuristic crossover point. The Mathemagix library provides three implementations: the naive algorithm, the multi-modular+fast CRT algorithm, and the Kronecker+FFT algorithm. These implementations can be found in the `Algebramix` package.

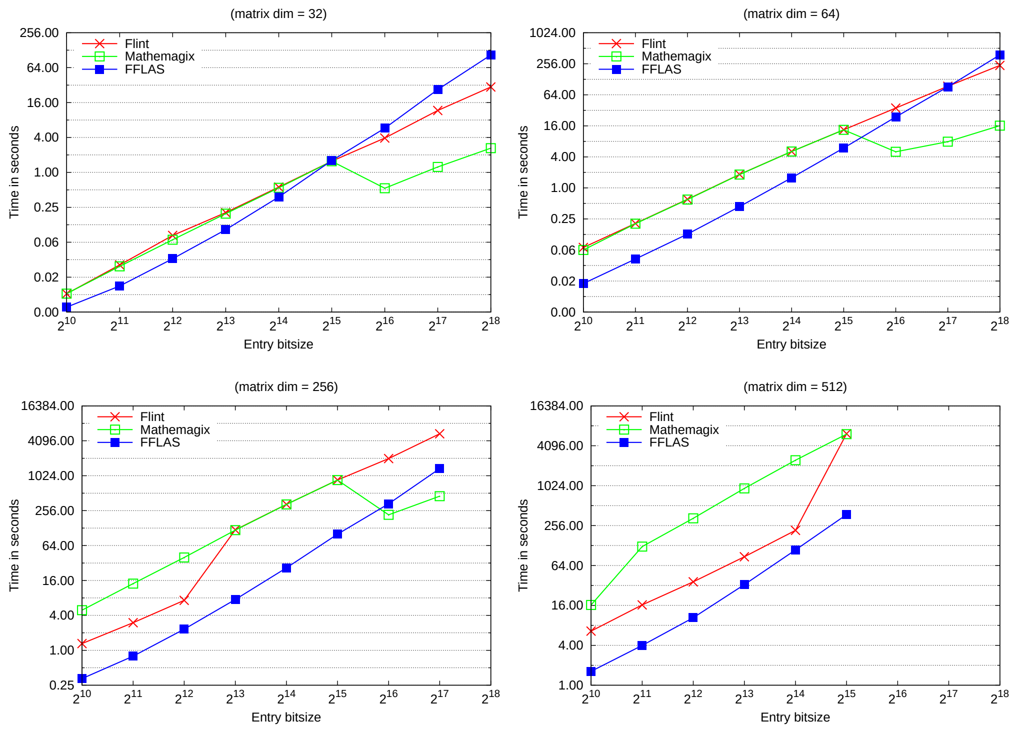


Fig. 1: Multi-precision integer matrix multiplication

Our benchmarks are done on an Intel Xeon E5-2697 2.6 GHz machine. As seen in Figure 1 our linear algebra implementation starts to outperform other implementations as the dimensions of the input matrices increase. Due to large entry sizes higher dimensional matrices become intractable. For tractable matrices, when the entry sizes are not to large compared to the dimension n, our algorithm offer very good performances. For example, Table III shows timings for a higher dimension and lower entry sizes.

Table III: 1024×1204 integer matrix multiplication

bitsize	**FLINT**	**Mathemagix**	**FFLAS**
4096	184s	2536s	54s
8192	427s	7562s	156s

8. EXAMPLE APPLICATIONS

Matrix multiplication is a fundamental operation for many computational problems. In particular, integer matrix multiplication comes to play when one deals with finite fields. Prime finite fields, denoted by $\mathbb{F}_p$ for a given prime p, are usually represented using the set of integers $\{0, 1, \ldots, p-1\}$. Then general finite fields are represented using polynomials over this set of integers. In this section, we briefly review some important operations in finite field arithmetic, and show how our linear algebra implementations of these operations result in significant speed-ups over conventional implementations.

Modular polynomial 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 $f(g) \bmod h$. The upper-bound cost of this operations is denoted by $\mathsf{C}(n)$ counted as operations over $\mathbb{F}_p$. The well-known algorithm of Brent and Kung [Brent and Kung 1978], which is implemented by most computer algebra systems, gives the bound $\mathsf{C}(n) = O(n^{(\omega+1)/2})$. The bottleneck of their algorithm can be written as a matrix multiplication. More precisely, for $k = \lceil\sqrt{n+1}\rceil$, $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$, and $q(x) = g_0 + g_1 x + \cdots + g_{n-1}x^{n-1}$ we should compute the matrix multiplication

$$\begin{bmatrix} f_0^{(0)} & f_0^{(1)} & \cdots & f_0^{(k-1)} \\ f_1^{(0)} & f_1^{(1)} & \cdots & f_1^{(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ f_n^{(0)} & f_n^{(1)} & \cdots & f_n^{(k-1)} \end{bmatrix} \begin{bmatrix} g_0 & g_k & \cdots & g_{(k-1)k} \\ g_1 & g_{k+1} & \cdots & g_{(k-1)k+1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1} & g_{2k-1} & \cdots & g_{k^2-1} \end{bmatrix}$$

where $f_i^{(j)}$ is the i-th coefficient of the power f^j. We have implemented modular polynomials composition using our matrix multiplication in C++. Figure 2a compares our implementation to the `CompMod` method of NTL library [Shoup et al. 2015] on an Intel(R) Core(TM) i7-4790, 3.60GHz machine.

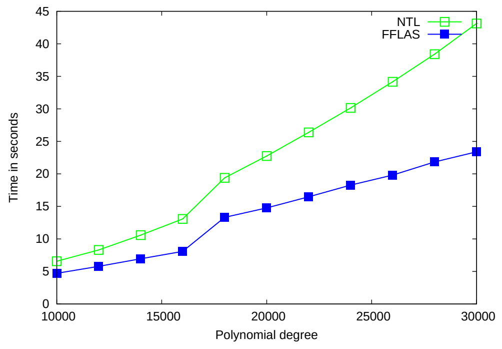
Power projection. Let f be a polynomial of degree n over $\mathbb{F}_p$, and let $g \in F = \mathbb{F}_p[x]/(f)$. For vectors $v, u \in \mathbb{F}_p^n$ denote by $\langle v, u\rangle$ their inner product. Coefficients of polynomials over F can be considered as vectors in $\mathbb{F}_p^n$. The power projection problem is computing the sequence

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

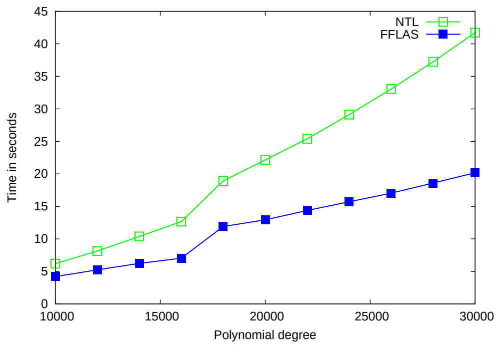
for a given integer $m > 0$. The best known algorithm for power projection is due to [Shoup 1999]. The dominant part of the algorithm can be formulated as a matrix multiplication similar to the one for modular polynomial composition. Similar to modular composition, we have implemented power projection using our matrix multiplication. The implementation is in C++ on an Intel(R) Core(TM) i7-4790, 3.60GHz machine. Figure 2b compares implementation to the `ProjectPowers` method of NTL.

The above two operations are main 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 $g \in F_p[X]$ of degree less than n such that $g(a) = 0 \bmod f$. An efficient algorithm for computing minimal polynomials is presented in [Shoup 1999], which uses power projection.

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 the algorithm is modular polynomial composition, as explained in [Von Zur Gathen and Shoup 1992]. We have modified the minimal polynomial and factoring implementations in NTL to use our new power projection

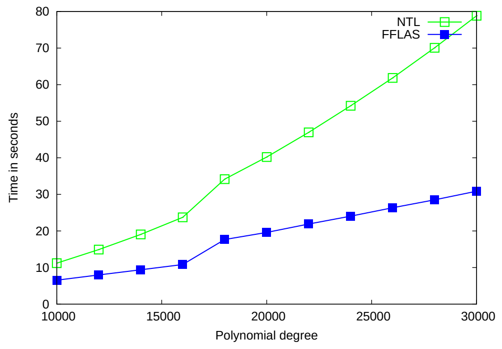


(a) Modular polynomial composition

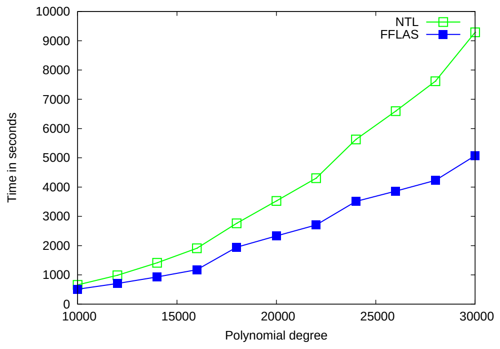


(b) Power projection

and modular composition. 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.



(a) Minimal polynomial computaion



(b) Polynomial factorization

REFERENCES

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.

D. J. Bernstein. 2008. *Algorithmic Number Theory*. Vol. 44. MSRI Publications, Chapter Fast multiplication and its applications, 325–384.

Allan Borodin and Robert 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).

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

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.

D. G. Cantor and E. Kaltofen. 1991. On Fast Multiplication of Polynomials over Arbitrary Algebras. *Acta Informatica* 28, 7 (1991), 693–701.

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

J Dongarra, S Hammarling, J Du Croz, and R Hanson. 1988. An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software* 14, 1 (1988), 1–32.

Matteo Frigo and Steven 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".

Joachim Von Zur Gathen and Jurgen Gerhard. 2003. *Modern Computer Algebra* (3 ed.). Cambridge University Press, New York, NY, USA.

Joachim Von Zur Gathen and Jurgen Gerhard. 2013. *Modern Computer Algebra* (3 ed.). Cambridge University Press, New York, NY, USA.

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

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

Kazushige Goto and Robert Van De Geijn. 2008. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)* 35, 1 (2008), 4.

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

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

Linbox-Team. 2015. Finite Field Linear Algebra Subroutines/Package. (2015). https://github.com/linbox-team/fflas-ffpack.

P. L. Montgomery. 1985. Modular Multiplication Without Trial Division. *Math. Comp.* 44, 170 (April 1985), 519–521.

John M Pollard. 1971. The fast Fourier transform in a finite field. *Mathematics of computation* 25, 114 (1971), 365–374.

Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas 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. 1995. A New Polynomial Factorization Algorithm and its Implementation. *J. Symb. Comp.* 20, 4 (1995), 363–397.

Victor Shoup. 1999. Efficient Computation of Minimal Polynomials in Algebraic Extensions of Finite Fields. In *ISSAC.* 53–58.

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

Joris van der Hoeven, Grégoire Lecerf, Bernard Mourrain, Philippe Trébuchet, Jérémy Berthomieu, Daouda Niang Diatta, and Angelos 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/.

Joachim Von Zur Gathen and Victor Shoup. 1992. Computing Frobenius maps and factoring polynomials. *Computational complexity* 2, 3 (1992), 187–224.

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