A

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 [**?**; **?**; **?**; **?**]. 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 [**?**], FLINT [**?**], FFLAS-FPACK [**?**] or Magma [**?**].

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 are interested in 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 ones, 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 techniques 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, but their practical performance remains somewhat lagging. Our algorithm offers an alternative to these techniques, 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 floating point matrix multiplication.

2. PREAMBLE ON POLYNOMIAL MATRIX MULTIPLICATION

We start by discussing two tangentially related questions, both concerning the cost of some computations with polynomial matrices of small degree.

Let us first make a general comment on our multiplication algorithms. Suppose we want to evaluate a R-bilinear map $U \times V \to W$, for some free R-modules U, V, W with bases $\boldsymbol{u} = (u_1, \ldots, u_r)$, $\boldsymbol{v} = (v_1, \ldots, v_s)$ and $\boldsymbol{w} = (w_1, \ldots, w_t)$. To do so, we will naturally use a *bilinear algorithm*. Such an algorithm $\boldsymbol{\mathcal{L}} = (\mathcal{L}, \mathcal{L}', \mathcal{L}'')$ is given by three sequences of linear forms

— ρ linear forms $\mathcal{L} = (L_1, \ldots, L_\rho)$ in U^*;
— ρ linear forms $\mathcal{L}' = (L'_1, \ldots, L'_\rho)$ in V^*;
— t linear forms $\mathcal{L}'' = (L''_1, \ldots, L''_t)$ defined over R^ρ,

such that $\phi(f, g)$ is obtained as $\sum_{1 \le i \le t} L''_i\big(L_1(f)L'_1(g), \ldots, L_\rho(f)L'_\rho(g)\big)w_i$ for all f, g in $U \times V$. The integer ρ is called the *rank* of $\boldsymbol{\mathcal{L}}$. We will not discuss in any detail bilinear algorithms for matrix multiplication, but we will review some existing algorithms for polynomial multiplication in low degree.

We start with a straightforward problem: given two polynomial matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, with $\boldsymbol{A} = [A_{i,j}]_{1 \le i \le m, 1 \le j \le n}$ and $\boldsymbol{B} = [B_{j,k}]_{1 \le j \le n, 1 \le k \le p}$, compute the product $\boldsymbol{AB}$. We assume that all entries of A and B are in $\mathbb{Z}[x]_\kappa$, which denotes the set of polynomials in $\mathbb{Z}[x]$ of degree less than κ, for some fixed integer κ. We are interested here in the case where κ is a small constant: in the application we have in mind, κ will be at most 3.

In the case $\kappa = 1$, we are simply doing matrix multiplication over $\mathbb{Z}$; by seeing the inputs as polynomials with matrix coefficients, we now explain how to reduce this problem to linear algebra over $\mathbb{Z}$ for any value of κ. We start from a bilinear algorithm $\boldsymbol{\mathcal{L}} = (\mathcal{L}, \mathcal{L}', \mathcal{L}'')$ for polynomial multiplication $\mathbb{Z}[x]_\kappa \times \mathbb{Z}[x]_\kappa \to \mathbb{Z}[x]_{2\kappa-1}$. For small values of κ, to compute the product of two such polynomials A and B, we will rely on the following choices:

— For $\kappa = 2$, with $A = a_0 + a_1x$ and $B = b_0 + b_1x$, the smallest possible rank is $\rho = 2$, using for instance Karatsuba's algorithm. In this case, we can take $\mathcal{L} = (a_0, a_1, a_0 + a_1)$, $\mathcal{L}' = (b_0, b_1, b_0 + b_1)$ and $\mathcal{L}'' = (c_0, c_1 - c_0 - c_2, c_2)$. It is customary to explain this algorithm as an evaluation / interpolation scheme that multiplies A and B modulo x, $x-1$ and $x-\infty$.
— For $\kappa = 3$, with $A = a_0 + a_1x + a_2x^2$ and $B = b_0 + b_1x + b_2x^2$, the smallest possible rank is 5, but we will use an algorithm of rank $\rho = 6$; it is given by the following linear forms:
  — $\mathcal{L} = (a_0, a_1, a_2, a_0 - a_2, a_0 + a_1 + a_2, a_0 + a_1 - a_2)$
  — $\mathcal{L}' = (b_0, b_1, b_2, b_0 - b_2, b_0 + b_1 + b_2, b_0 + b_1 - b_2)$
  — $\mathcal{L}'' = (c_0, -(c_0 + c_1 + c_2) + (c_4 + c_5)/2, (c_0 + c_1 + c_2) - c_3, -(c_0 + c_2) + c_3 + (c_4 - c_5)/2, c_2)$.
  It corresponds to multiplying A and B modulo x, $x - 1$, $x^2 + 1$, and $x - \infty$.

Once $\boldsymbol{\mathcal{L}}$ is chosen, to compute the matrix product $\boldsymbol{AB}$, we apply $\mathcal{L}$ to all entries of $\boldsymbol{A}$, $\mathcal{L}'$ to all entries of $\boldsymbol{B}$, and perform ρ matrix multiplications between the corresponding scalar matrices (where ρ is the rank of $\boldsymbol{\mathcal{L}}$). Finally, we recover $\boldsymbol{AB}$ by applying $\mathcal{L}''$ to all the scalar matrices we obtained in the previous step.

For our second problem, consider a family of polynomials $\boldsymbol{B} = (B_1, \ldots, B_s)$ in $\mathbb{Z}[x]_n$, all with degree less than n, together with r families of polynomials $\boldsymbol{A}_1, \ldots, \boldsymbol{A}_r$, with $\boldsymbol{A}_i = (A_{i,1}, \ldots, A_{i,s})$, all polynomials $A_{i,j}$ being in $\mathbb{Z}[x]_\kappa$. As before, we assume that κ is a small constant (typically, we may have $\kappa \in \{1, 2, 3\}$). Our goal is to compute the products

$$C_i = \boldsymbol{A}_i \cdot \boldsymbol{B} = \sum_{j=1}^{s} A_{i,j} B_j, \quad 1 \leq i \leq r.$$

We now explain how to reduce this problem to matrix multiplication over $\mathbb{Z}$.

The basic idea is well-known: our problem can be seen as matrix product in sizes $(r \times s)$ and $(s \times 1)$, with entries in respectively $\mathbb{Z}[x]_\kappa$ and $\mathbb{Z}[x]_n$. Assuming for simplicity that κ divides n, we can split the right-hand side into a matrix of size $(s \times n/\kappa)$, with entries in $\mathbb{Z}[x]_\kappa$. After doing the corresponding $(r \times s)$ by $(s \times n/\kappa)$ product, the recombination of the entries to recover the output of our original problem takes linear time.

For $\kappa = 1$, we are simply multiplying integer matrices of respective sizes (r, s) and (s, n). In general, since we are multiplying polynomial matrices with entries in $\mathbb{Z}[x]_\kappa$, we can apply the algorithm given previously. After choosing a bilinear algorithm $\boldsymbol{\mathcal{L}}$ of rank ρ for multiplication $\mathbb{Z}[x]_\kappa \times \mathbb{Z}[x]_\kappa \to \mathbb{Z}[x]_{2\kappa-1}$, the bulk of the computation is ρ matrix products in sizes (r, s) by $(s, n/\kappa)$. (**todo:** Unify notation for the size of matrices)

3. CONVERSIONS TO AND FROM RESIDUE NUMBER SYSTEMS

3.1. Definitions

Three quantities, β, γ and δ will be used throughout.

— The default representation of positive integers will use a positional number system for a fixed radix 2^β: any positive integer $a \in \mathbb{N}$ can be uniquely written as $a = \sum_{i=0}^{n-1} a_i 2^{\beta i}$ for some a_i's in $\{0, \ldots, 2^\beta - 1\}$ and with $a_{n-1} \neq 0$; a typical choice is $\beta = 64$.
— Let $m_1, m_2, \ldots, m_s$ be pairwise coprime positive integers, 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 $[a]_i = a \bmod m_i$. We assume that the moduli m_i defining this residue number system satisfy $m_i < 2^\gamma$ for all i; possible values for γ are $\gamma = 16$, 20, or even 60.
— We will have to multiply matrices, which will be done using BLAS matrix multiplication; this imposes restrictions on the size of the operands. Explicitly, we will let δ be such that matrix multiplication in any size is feasible for integer matrices with entries in $\{0, \ldots, 2^\delta - 1\}$. Possible value for δ are 16 or 20. (**todo:** the size of the product shoul be bounded. Will be fixed later.)
For simplicity of notation, we will assume that δ divides γ, and we let $\kappa = \gamma/\delta$.

We assume that any positive (?) integer a written in basis 2^x, for x in $\{\alpha, \beta, \gamma\}$, can be written in basis 2^y, for any y in $\{\alpha, \beta, \gamma\}$, in linear time $O(\log(a))$.

In order to benefit from the RNS representation, one needs to be able to convert back and forth between the positional and residual number systems. In the following two subsections, we discuss approaches for converting integers from the classic positional system to the residue number system, and conversely, using linear algebra.

3.2. Conversion to RNS

We start with the conversion of integers in their positional representation to their residue number system representation. Our input is a sequence of positive integers $a_1, \ldots, a_r$, with $a_i < 2^L$ for all i, together with moduli $m_1, \ldots, m_s$ as above, all m_j's satisfying $m_j < 2^\gamma$. Our goal is to compute $a_i \bmod m_j$, for all i, j.

Our first step is to write all inputs in radix 2^δ. Precisely, we will write $a_i = A_i(2^\delta, 2^\gamma)$, for some polynomials A_i in $\mathbb{Z}[x, y]$ of degree less than κ in x and L/γ in y (**todo:** rounding?), and with non-negative coefficients less than 2^δ. Explicitly, we have $A_i = \sum_{0 \leq k < L/\gamma} A_{i,k}(x) y^k$.

On the other hand, for any modulus m_j, and any positive integer k, let us write the expansion of $2^{\gamma k} \bmod m_j$ in radix 2^δ as

$$\begin{aligned} 2^{\gamma k} \bmod m_j &= b_{j,k,0} + b_{j,k,1} 2^\delta + \cdots + b_{j,k,\kappa-1} 2^{(\kappa-1)\delta} \\ &= B_{j,k}(2^\delta), \end{aligned}$$

where $B_{j,k}$ is the polynomial $B_{j,k} = b_{j,k,0} + b_{j,k,1} x + \cdots + b_{j,k,\kappa-1} x^{\kappa-1} \in \mathbb{Z}[x]$. As a result, we obtain the equality

$$a_i \bmod m_j = C_{i,j}(2^\delta) \bmod m_j,$$

where $C_{i,j}$ is the polynomial $\sum_{0 \le k < L/\gamma} A_{i,k} B_{j,k}$. Hence, we first compute the polynomials $C_{i,j}$, as deducing $a_i \bmod m_j$ is then straightforward using the above expression. We are thus reduced to performing a matrix product in size $(r \times L/\gamma)$ by $(L/\gamma, s)$, with polynomial entries of degree less than κ and coefficients in $\{0, \ldots, 2^\gamma - 1\}$. To illustrate the construction, we discuss two useful cases:

— Suppose that all moduli m_j are bounded by 2^{16}; in that case, we have $\gamma = \delta = 16$ and $\kappa = 1$. In other words, we are doing scalar matrix multiplication, and the right-hand matrix $\boldsymbol{B} = [B_{i,j}]$ is an analogue of a Vandermonde matrix, with entries that are simply the values $2^{16k} \bmod m_j$.
— Consider now the case where all moduli m_j are bounded by 2^{60}. In that case, we take $\delta = 20$ and thus $\kappa = 3$. We are thus left to perform a matrix product with entries that are polynomials of degree at most 2 and with coefficients that are at msot 2^{20}.

3.3. Conversion from RNS

We next consider the converse operation, Chinese Remaindering. Let a be an integer given by its RNS representation $([a]_1, \ldots, [a]_s)$ with base $(m_1, \ldots, m_s)$. The unique integer a in $\{0, \ldots, M-1\}$ satisfying all congruences $[a]_i = a \bmod m_i$ is

$$a = \sum_{i=1}^{s} \left([a]_i u_i \bmod m_i\right) \lambda_i \bmod M, \tag{1}$$

where we write $\lambda_i = M/m_i$ and $u_i = 1/\lambda_i \bmod m_i$ for all i.

In what follows, we are interested in the reconstruction of *several* such as, say $a_1, \ldots, a_r$, from their respective residues $([a_i]_1, \ldots, [a_i]_s)$, for $i = 1, \ldots, r$. In this case, M, all λ_is and u_is can be precomputed once and for all. We perform an operation we will call *simultaneous pseudo-reconstructions* as follows. For $i = 1, \ldots, r$ and $j = 1, \ldots, s$, let $v_{i,j} = [a_i]_j u_j \bmod m_j$ and define

$$b_i = \sum_{j=1}^{s} v_{i,j} \lambda_j,$$

so that $a_i = b_i \bmod M$, and $b_i < Ms$; we call b_i the *pseudo-reconstruction* of $([a_i]_1, \ldots, [a_i]_s)$ modulo $m_1, \ldots, m_s$ (**todo:** better to integrate u_j into M_j). Once we know b_i, we deduce a itself using one Euclidean division. (**todo:** say a bit more).

As we did for the previous conversion, we will use matrix multiplication to compute all b_is, which means that all operands must be written in base 2^δ, as

$$v_{i,j} = \sum_{k=0}^{\kappa-1} v_{i,j,k} 2^{k\delta} \quad \text{and} \quad \lambda_j = \sum_{\ell=0}^{L/\delta-1} \lambda_{j,\ell} 2^{\ell\delta}.$$

These two integers can then be identified with the values of

$$V_{i,j} = \sum_{k=0}^{\kappa-1} v_{i,j,k} x^k \quad \text{and} \quad \Lambda_j = \sum_{\ell=0}^{L/\delta-1} \lambda_{j,\ell} x^\ell$$

at 2^δ. Hence, we will compute the polynomials $B_1, \ldots, B_r \in \mathbb{Z}[x]$ defined as

$$B_i = \sum_{j=1}^{s} V_{i,j} \Lambda_j$$

before deducing their value at 2^δ.

4. APPLICATION

The following graph gives timings for the multiplication of polynomials with 12000-bit coefficients using either NTL, or a modification of it based on the algorithm above. For such degree and bit-size ranges, NTL uses an algorithm based on Chinese Remaindering, reducing the input modulo several FFT primes of about 60 bits. We replaced the built-in reduction (that uses standard algorithms based on subproduct trees) by an implementation of the algorithms described above.