Analysis of RNS and BDK MonPro Algorithms

Pradeep Shekhar Katta and Shaman Bhat
{pradeepshekharkatta,shamanbhat}@umail.ucsb.edu
Department of Electrical and Computer Engineering
University of California Santa Barbara

June 16 , 2017

Abstract

Residue Number System (RNS) is used in parallelizing operations like addition, subtraction, multiplication and exact division. We study and investigate the performance of RNSMonPro and BDKMonPro algorithms which are based on RNS arithmetic. We first describe the theory behind the algorithms and identify the key differences. Then we assess the algorithms based on the execution time. It is expected that BDKMonPro algorithm reduces complexity and performs better than RNSMonPro.

1 Introduction

There exist many different algorithms that are used to carry out modular multiplication on very large numbers. These algorithms exploit Montgomery's modular multiplication and redundant radix number systems. Compared to these algorithms, Residue Number System (RNS) offers parallel and carry free operation which has gained significance [3]. The RNS decomposes a large integer into a set of smaller integers. A large computation can be broken down into a series of smaller calculations that are highly pliable to parallelism. This is useful for implementing Montgomery Multiplication which is widely used in most cryptographic applications.

In modular arithmetic computation, Montgomery Multiplication is used to perform fast modular multiplication. This operation can be performed in RNS domain as it consists of one addition and two multiplications. This paper analyzes two variants of RNS algorithms - RNSMonPro and BDKMonPro. RNSMonPro involves conversion between RNS domain to a weighted radix and back. BDKMonPro on the other hand is also planted on the classical Montgomery Multiplication with the difference that it uses two residue number systems. We have implemented and analyzed the aforementioned two variants of RNS. We have tested and compared the execution time of various modules and the algorithms as a whole. The source code for the above implementation can be found at `https://github.com/pradeepshekhar/RNS_BDK`.

2 Montgomery Multiplication

Montgomery multiplication was introduced by the American mathematician Peter Montgomery. It computes the product $c = a \cdot b \cdot r^{-1} \pmod{n}$ for two integers a and b and an arbitrary modulus n. Here, r is chosen to be 2^k such that $2^{k-1} < n, 2^k$. This requires that n be odd, which is often the case in cryptography. MonPro comprises of two multiplications instead of one and we do not explicitly need r^{-1} but it requires n' which is computed using extended Euclidean algorithm (EEA) as follows:

$$(s, t) \leftarrow EEA(r, n) \Rightarrow s \cdot r + t \cdot n = 1$$

where $r^{-1} = s \pmod{n}$ and $n' = -t$.

function Montgomery(a, b)
input: a, b, n, r, n'
output: $u = a \cdot b \cdot r^{-1} \bmod n$
1: $t \leftarrow a \cdot b$
2: $m \leftarrow t \cdot n' \pmod{r}$
3: $u \leftarrow (t + m \cdot n)/r$
4: **if** $u \geq n$ **then** $u \leftarrow u - n$
5: **return** u

3 Residue Number System

RNS allows the speed up of computation by distributing large dynamic range computation over small modular rings [4]. In RNS, large integers are represented using a set of remainders with respect to a set of relatively prime moduli m_i. The sum, difference and product operations can be performed on the remainders with respect to corresponding moduli. The Conversion from RNS to weighted radix is directly based on the chinese remainder theorem (CRT).

3.1 Chinese Remainder Theorem

Given the remainders $r_1, r_2, ..., r_k$, the RNS representation of a number x using moduli $[m_i]_{i=1}^k$, we can compute x using

$$x = \sum_{i=1}^{k} r_i \cdot c_i \cdot n_i \pmod{n}$$

where $n_i = n/m_i$, $n = m_1 \cdot m_2 \cdots m_k$ and $c_i = n_i^{-1} \pmod{m_i}$

The above theorem uses a summation computation that can be implemented using the Mixed Radix Conversion (MRC) Algorithm that avoids multi-precision arithmetic until the last phase.

3.1.1 Mixed Radix Conversion

MRC involves computation of the mixed radix representation of a number using it's RNS representation. Our implementation of MRC can be summarized as follows:

1. Calculate the inverses c_{ij} for $1 \leq i < j \leq k$, using EEA.

$$c_{ij} = m_j^{-1} \pmod{m_i}$$

   where m_i's are the moduli.

2. Given the remainders $(r_1, r_2, ..., r_k)$, form the lower triangular r matrix as given below. It's diagonal elements represent the mixed radix coefficients.

   - First column, $r_{i1} = r_i$ for i = 1,2,...,k
   - Compute j^{th} column using the $(j-1)^{th}$ column and c_{ij}'s

$$r_{ij} = (r_{i,j-1} - r_{j-1,j-1}) \cdot c_{i,j-1} \pmod{m_i}$$

3. The integer x is then determined using the diagonal entries of r matrix as

$$c = r_{11} + r_{22} \cdot m_1 + r_{33} \cdot m_1 \cdot m_2 + ... + r_{kk} \cdot m_1 \cdot m_2 \cdots m_{k-1}$$

   This is the only step that requires multi-precision arithmetic.

3.2 Classical RNS Montgomery Algorithm

The vector set $M = \{m_1, m_2, .., m_k\}$ is chosen such that $m = \prod_{i=1}^{k} m_i$ where m_i's are relatively prime with $n^2 < m$ so that overflow during multiplication can be avoided within RNS [1].

A denotes the RNS representation of a number a of dimension k where $A = (A_1, A_2, ..., A_k)$ and $A_i = a \pmod{m_i}$. A is computed from a as $A = RNS(a)$ and conversely a is computed from A as $a = CRT(A)$. Here, CRT uses the MRC algorithm. Below we show the setup steps involved in our implementation of the above algorithm.

3.2.1 Setup

1. Given two integers a, b and moduli n, we determine Montgomery parameters r and n' using EEA as discussed in Section 2.

2. We find the RNS representations A, B, N, R, N' using the RNS function on a, b, n, r, n' respectively.

3. Compute R^{-1} - the vector $(R_1^{-1} \bmod m_1, R_2^{-1} \bmod m_2, ..., R_k^{-1} \bmod m_k)$

3.2.2 RNSMonPro Algorithm

Input: A, B, N, R^{-1}
Output: T : RNS representation of $t = a \cdot b \cdot r^{-1} \bmod n$
1: $T' \leftarrow A \cdot B$
$T \leftarrow T' \cdot N'$
2: $t \leftarrow \text{CRT}(T)$
3: $q \leftarrow t \pmod r$
4: $Q \leftarrow \text{RNS}(q)$
5: $T \leftarrow (T' + Q \cdot N) \cdot R^{-1}$

Steps 1 and 5 in the above algorithm can be performed in parallel on k processors. Steps 2 and 4 require conversions between RNS and weighted radix since step 3 is performed in weighted radix.

3.3 BDK Montgomery Algorithm

The arduous calculations involved in conversion and multi-precision arithmetic can be averted in BDK algorithm. It uses two residue number systems defined as

$$M = (m_1, m_2, ..., m_k) \text{ where } m = m_1 m_2 ... m_k$$
$$P = (p_1, p_2, ..., p_k) \text{ where } p = p_1 p_2 ... p_k$$

Some conditions that must be satisfied include $gcd(m, p) = 1$ and $gcd(n, m) = 1$ and $2n < m < p$. The BDK algorithm evaluates $t = a \cdot b \cdot m^{-1} \pmod n$. Here $r = m = m_1 m_2 ... m_k$. It is important to note here that r is not equal to 2^k, normally seen in other variants of Montgomery multiplication. The reason behind this decision is to circumvent conversion between RNS and weighted radix.

3.3.1 Setup

1. Given c, d and modulus o; C, D, O are calculated in base M; $\mathbb{C}, \mathbb{D}, \mathbb{O}$ are calculated in base P.
2. Find O^{-1} in base M using EEA.
3. Determine $-C \pmod M$.
4. Calculate $\mathbb{M}^{-1}$ - the vector ($m^{-1} \pmod{p_1}$,$m^{-1} \pmod{p_2}$,...,$m^{-1} \pmod{p_k}$).
5. Precompute $\mathbb{M}_{ji}$ and $\mathbb{T}_{ji}$, the product terms used in the Step 3 of MRC.
   - $\mathbb{M}_{ji} = m_1 m_2 ... m_j \pmod{p_i}$ for $j = 1, 2...k-1$
   - $\mathbb{P}_{ji} = p_1 p_2 ... p_j \pmod{m_i}$ for $j = 1, 2...k-1$

3.3.2 BDKMonPro Algorithm

Input: C, D, O: Basis M
$\mathbb{C}, \mathbb{D}, \mathbb{O}$,$\mathbb{M}^{-1}$: Basis P
Output: T,$\mathbb{T}$: Basis M and P
1: $S \leftarrow (-C \cdot D) \cdot O^{-1}$
2: $\mathbb{S} \leftarrow$ BasisConversion(S)
3: $\mathbb{T} \leftarrow (\mathbb{C} \cdot \mathbb{D} + \mathbb{S} \cdot \mathbb{O}\ )\ \mathbb{M}^{-1}$
4: $T \leftarrow$ BasisConversion($\mathbb{T}$)

Computation of S and $\mathbb{T}$ are performed in RNS bases M and P respectively. Steps 2 and 4 are used to execute the basis conversion function i.e, the MRC algorithm [2]

3.3.3 Basis Conversion

The BDK algorithm requires computation of $c \cdot d + s \cdot o \geq m$ which cannot be represented in M. Hence a second residue system P is used such that $p > m$ [4]. So we need conversion between bases. The following steps elucidate basis conversion from base M to P.

1. Given $(S_1, S_2, ..., S_k)$, representation of s in M, the MRC representation ($\mathbf{S}_1$,$\mathbf{S}_2$,...,$\mathbf{S}_k$) is determined as explained in Section 3.1.1.

2. ($\mathbb{S}_1$,$\mathbb{S}_2$,...,$\mathbb{S}_k$), representation of s in P is then calculated using

$$\mathbb{S}_j = \mathbf{S}_1 + \mathbf{S}_2 \cdot \mathbb{M}_{1j} + \mathbf{S}_3 \cdot \mathbb{M}_{2j} + \cdots + \mathbf{S}_k \cdot \mathbb{M}_{k-1,j}$$

   where $\mathbf{M}_{ji}$'s are calculated during setup. k processors execute $k-1$ multiplications and $k-1$ additions in parallel to find the k elements of $\mathbb{S}$.

4 Analysis

The RNSMonPro and BDKMonPro algorithms were implemented in **C** language on a portable computer with Intel Core i7 2.6 GHz processor running a 64-bit Ubuntu 16 operating system. They were analyzed by calculating the execution time using the "time.h" library in C. All the values reported were averaged over 10 executions for consistency. The **NextPrime**[·] function in Mathematica was used to generate the moduli.

4.1 Timing Procedure

In RNSMonPro, as steps 1 and 5 are performed in parallel on k processors, the time taken for execution in each moduli is calculated and averaged to get the actual execution time. The execution time of steps 2,3 and 4 is added to get the total execution time of the algorithm.

In BDKMonPro, all the steps are performed in parallel on k processors. Hence the actual execution time is calculated by averaging the total time taken by each processor.

In both cases, the time taken for setup is not included as it is not part of the actual algorithm.

4.2 Timing Comparison

For comparing execution times, the following values were used as inputs to the algorithms:

- $a = c = 19, b = d = 21, n = 29, k = 5$(number of moduli)

As the execution time for an iteration is very low, the algorithms were executed several times(n=1000000,100000,10000) and execution time per iteration is reported. From Fig. 1, it can be clearly seen that BDKMonPro is faster than RNSMonPro in all the cases.

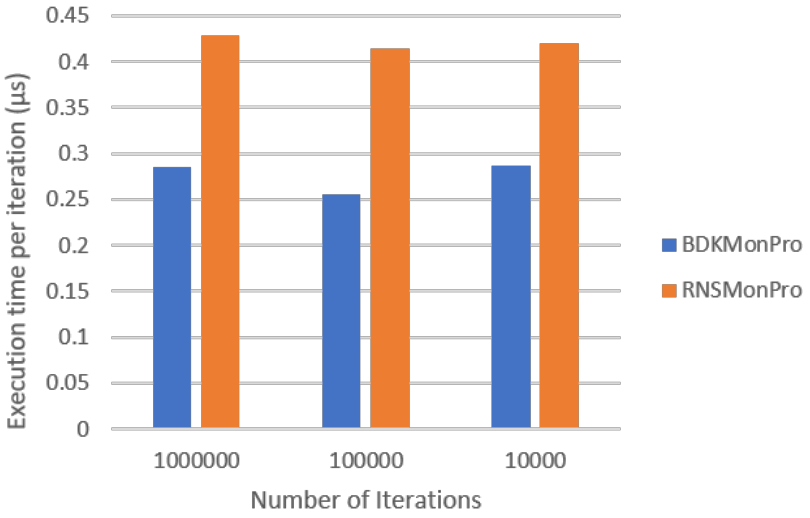

Figure 1: Timing Comparison

Set #	$a = c$	$b = d$	n	k
1	19	21	29	5
2	26386123	12379231	14527	5
3	2638451216123	871247372	5423687	5

Table 1: Input Sets

4.3 Execution time for varying inputs

In this section, we will look at execution time of the algorithms for three different input sets described in Table. 1. Larger primes were used as moduli for bigger inputs.

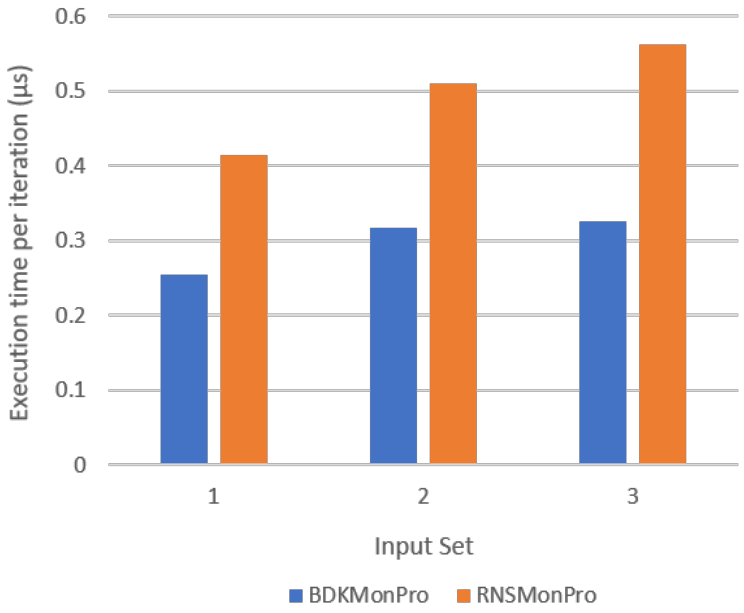

Figure 2: Execution times for different input sets

From Fig. 2, we see that the execution time increases as we use larger inputs.

5 Conclusions

We have successfully implemented the RNSMonPro and BDKMonPro algorithms. Adapting the modifications suggested by J.C. Bajard et al., we observe improvement in execution time of the BDK algorithm over RNS. BDK reduces the complexity by parallelizing the inter basis conversions. Increasing the number of iterations does not affect the execution time per iteration of the algorithms. We also gained a good understanding of the work-flow of the algorithms and were able to quantify the gain in performance.

References

[1] J-C Bajard, L-S Didier, and Peter Kornerup. An rns montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–776, 1998.

[2] J-C Bajard, L-S Didier, and Peter Kornerup. Modular multiplication and base extensions in residue number systems. In *Computer Arithmetic, 2001. Proceedings. 15th IEEE Symposium on*, pages 59–65. IEEE, 2001.

[3] Jean-Claude Bajard, Laurent-Stephane Didier, and Peter Kornerup. Montgomery modular multiplication in residue arithmetic, 2000.

[4] Jean-Claude Bajard and Thomas Plantard. Rns bases and conversions. In *Optical Science and Technology, the SPIE 49th Annual Meeting*, pages 60–69. International Society for Optics and Photonics, 2004.