ISSN 0361-7688, Programming and Computer Software, 2007, Vol. 33, No. 2, pp. 80–86. © Pleiades Publishing, Ltd., 2007.
Original Russian Text © E.V. Zima, A.M. Stewart, 2007, published in Programmirovanie, 2007, Vol. 33, No. 2.

Cunningham Numbers in Modular Arithmetic

E. V. Zima[a] and A. M. Stewart[b]

[a] *Physics and Computer Science Department, Wilfrid Laurier University, 75 University Avenue West, Waterloo, Ontario, Canada N2L 3C5*
E-mail: ezima@wlu.ca

[b] *Symbolic Computation Group, University of Waterloo, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1*
E-mail: am2steward@uwaterloo.ca

Received July 1, 2006

Abstract—The paper considers methods for modular arithmetic acceleration, based on a specific moduli selection method. Special attention is paid to the moduli of the form $2^n - 1$ and $2^n + 1$. Different schemes of choice of these types of moduli and algorithms for conversion of arbitrary precision integers into the modular representation and back are considered. Results of experimental implementation of the described algorithms in the GMP system are discussed.

DOI: 10.1134/S0361768807020053

1. INTRODUCTION

Modular algorithms present a popular tool for computation acceleration in different areas of computer algebra. They are successfully used for calculation of the greatest common divisors of polynomials, polynomial factorization, in problems of symbolic linear algebra, for symbolic integration and summation. Modular approach to symbolic computations has several advantageous features. One of them is the possibility to control the size of intermediate results, which is often impossible in the case of computations with arbitrary precision integer or rational numbers. This implies faster and more efficient implementations of symbolic algorithms.

Conversion to a modular representation is also a popular technique to accelerate the basic arbitrary precision arithmetic of computer algebra systems. This paper considers methods of moduli selection and modular arithmetic implementation that reduce time for performing basic arithmetic operations and time to convert arbitrary precision integers to the modular representation and back. If modulus m is not chosen in a specific way, division operation may be required to obtain the residue, which is a rather expensive operation, even if the modulus is representable by a standard machine word (64 or 32 bits). The simplest way to avoid division is to choose the modulus of the form $2^n - 1$ [1]. The case of modulus of the form $2^n + 1$ is of interest as well.

The numbers of the form $b^n \pm 1$ are called Cunningham numbers with the base b.[1] The book [2] contains tables of Cunningham numbers factorizations and their history. For computer implementation of modular arithmetic, the case of $b = 2$ presents a particular interest. Sets of Cunningham numbers of the forms $2^n + 1$ and $2^n - 1$ are denoted as 2+ and 2–, respectively [3].

2. BACKGROUND INFORMATION

The idea of modular calculations consists in selecting positive integers $m_1, m_2, \ldots, m_k$, referred to as moduli; replacing the initial integer data by residues modulo m_i; and performing a series of identical calculations modulo m_i (for every $i = 1, \ldots, k$) instead of required calculations with long integer numbers. On the final stage, the result needs to be reconstructed from the residues. The possibility of reconstruction is based on the Chinese remainder theorem [1] if, for example, all the moduli are pairwise coprime and all the intermediate and final results do not exceed $m_1 m_2 \ldots m_k$. Though it is not necessary, we will further assume that all the moduli are pairwise coprime and the minimum nonnegative system of moduli is used; i.e., all the residues modulo m_i are taken from the set $0, 1, \ldots, m_i - 1$.

The choice of specific values of m_i can influence significantly the time complexity of both calculations modulo m_i and reconstruction of the result. For example, after multiplication of numbers, in the general case, division by m_i may be needed to obtain the residue value. At the sane time, if $m_i = 2^n - 1$, then calculation of the residue reduces to shift and addition operations that are much simpler.

Conversion to modular representation is advantageous when heavy calculations are to be performed in the ring of integers, which require many multiplications, additions, and subtractions. Modular representation gives no advantages for addition and subtraction operations, but it can accelerate multiplication considerably. Let $M(n)$ be the time complexity of multiplica-

[1] Note that any positive integer X is a Cunningham number, for example, with the base $X \pm 1$, that is why a usual constraint is $b \in \{2, 3, 5, 6, 7, 10, 11, 12\}$.

tion of two integer numbers of bit length n, and let moduli $m_1, \ldots, m_k$ be chosen to have approximately the same bit length equal to $\frac{2n}{k}$. Multiplication of two integers of bit length n in modular representation reduces to multiplying k pairs of numbers of length $\frac{2n}{k}$ and has the time complexity of $kM\left(\frac{2n}{k}\right)$. For sufficiently large k, this considerably speeds up not only elementary multiplication algorithm for integers with $M(n) \in \Theta(n^2)$ but also fast Karatsuba ($M(n) \in \Theta(n^{\log_2 3})$) or Schönhage-Shtrassen ($M(n) \in \Theta(n\log n\log\log n)$) algorithms, if, of course, we do not take into account the time needed for conversion into modular representation and back.

Note that even addition and subtraction operations can be accelerated if parallel machine is used, since calculations for different moduli can be performed simultaneously.

Let us consider factors affecting the choice of the moduli while implementing modular arithmetic on computer. It is very simple to perform calculations modulo 2^n. One should simply ignore any overflow of the n-bit grid. Unfortunately, numbers of type 2^n, $n > 1$, are not coprime. Using Cunningham numbers as moduli is attractive, since performing calculations modulo $2^n \pm 1$ is almost as simple as modulo 2^n, and there are sufficiently many coprime integers among Cunningham numbers.

As it is widely known [1], $\mathrm{GCD}(2^n - 1, 2^m - 1) = 2^{\mathrm{GCD}(n, m)} - 1$. Consequently, numbers $2^n - 1$ and $2^m - 1$ are coprime if and only if n and m and coprime. A similar (and even simpler) fact for numbers of type $2^n + 1$ is less known. Let $\nu_2(x)$ denote the maximum degree of 2 that is contained in x. Numbers $2^m + 1$ and $2^n + 1$ are coprime if and only if $\nu_2(m) \neq \nu_2(n)$.

This is a corollary of a more general result: for any positive integers $a, n, m, a > 1$ [4],

$$\mathrm{GCD}(a^m + 1, a^n + 1) = \begin{cases} a^{\mathrm{GCD}(m, n)} + 1, & \text{if } \nu_2(m) = \nu_2(n) \\ 1, & \text{if } \nu_2(m) \neq \nu_2(n) \\ & \text{and } a \text{ is even} \\ 2, & \text{if } \nu_2(m) \neq \nu_2(n) \\ & \text{and } a \text{ is odd.} \end{cases}$$

It is interesting that the latter result is not usually mentioned in monographs, such as [1], and textbooks on number theory. In the book [5] devoted to Fermat numbers [5] (which appeared 11 years later after publication [4]), only a particular case of this result is proven:

$$\mathrm{GCD}(2^m + 1, 2^{mn} + 1) = \begin{cases} 1, & \text{if } n \text{ is even} \\ 2^m + 1, & \text{if } n \text{ is odd.} \end{cases}$$

If the Cunningham numbers of type 2– or 2+ are used, the choice of the modulus is essentially the choice of the exponent. These exponents should satisfy a simple rule depending on the type of the number we work with. For numbers of type 2–, the exponents must be coprime. For numbers of type 2+, binary notation of the exponents should not end with the same number of zeros. Different strategies of exponent selection in the case of numbers of type 2–, including the simplest strategy of choosing only prime numbers as exponents, are discussed in [1, 6].

For numbers of type 2+, the simplest scheme of modulus choice is based on "shift": the first exponent a is chosen in an arbitrary way; every consecutive exponent is obtained from the previous one by multiplying by 2 (which corresponds to the one bit left shift in binary notation). This scheme generates moduli $2^a + 1$, $2^{2a} + 1$, $2^{4a} + 1$, …, $2^{2^i a} + 1$. If a is chosen to be equal to 1, then the moduli are consecutive Fermat numbers $2^1 + 1, 2^2 + 1, 2^4 + 1, \ldots, 2^{2^i} + 1, \ldots$.

Another ("block") strategy consists in generating exponents of the same bit length, which gives a series of moduli that is better balanced in lengths. First, the number of moduli b is chosen, and the exponents $e_1, e_2, \ldots, e_b$ are generated using recurrence $e_b = 2^{b+1} - 1$, $e_k = e_{k+1} - 2^{b-k-1}$, $k = b - 1, b - 2, \ldots, 1$. Thus, binary notation of the exponent e_i ($i = 1, \ldots, b$) ends with $(b - i)$ zeros following 1, which ensures coprimality of the corresponding moduli. For example, for $b = 4$, the following four moduli with 4-bit exponents will be generated: 8, 12, 14, and 15.

We assume that the size of initial data and all intermediate results can be estimated a priori, and, thus, it is possible to choose a priori the number of moduli b ensuring correct reconstruction of the result from residues. If one needs to add a modulus in the process of calculations, this presents no difficulty in the case of using the "shift" scheme. In the case of using the "block" scheme, it is possible to add moduli from the next block with the exponents of bit length $2b$ ending with $b + 1, b + 2, \ldots$ zeros. Both schemes allow efficient generation of moduli and avoid verification of exponent coprimality.

One of the obvious disadvantages of the scheme based of the "shift" is the imbalance of the moduli in

length. However, this scheme has some attractive properties as well. Let us consider moduli of the type

$$m_i = 2^{a2^i} + 1, \quad i = 0, 1, \ldots, k, \tag{1}$$

with arbitrary positive integer a and the products

$$M_i = \prod_{j=0}^{i-1} m_j = \prod_{j=0}^{i-1} (2^{a2^j} + 1), \quad i = 1, 2, \ldots, k. \tag{2}$$

Proposition 1.

$$M_i^{-1} \bmod m_i = 2^{a2^i - 1} - 2^{a-1} + 1, \quad i = 1, 2, \ldots, k. \tag{3}$$

Proof. Note that

$$M_i = \sum_{j=0}^{2^i - 1} 2^{aj}, \quad i = 1, 2, \ldots. \tag{4}$$

Consider

$$s_i = 2^{a2^i - 1} - 2^{a-1} + 1,$$
$$t_i = -(M_i - 1)/2 = \left(\sum_{j=1}^{2^i - 1} 2^{aj-1} \right), \quad i = 1, 2, \ldots. \tag{5}$$

It is easy to check that

$$M_i s_i + m_i t_i = 1, \quad i = 1, 2, \ldots. \tag{6}$$

Indeed, substituting L for 2^i in (4) and (5), we can take advantage of the geometric summation formula. The first term in the left-hand side of (6) is equal to

$$\left(\frac{(2^a)^L - 1}{2^a - 1} \right) (2^{aL-1} - 2^{a-1} + 1),$$

and the second term is equal to

$$-\left(1/2 \frac{(2^a)^L - 2^a}{2^a - 1} \right) (2^{aL} + 1).$$

Expanding the first term, we obtain

$$1/2 \frac{(2^{aL})^2 - 2^{aL}2^a + 2^{aL} + 2^a - 2}{2^a - 1},$$

and expanding the second term, we get

$$-1/2 \frac{(2^{aL})^2 + 2^{aL} - 2^{aL}2^a - 2^a}{2^a - 1}.$$

Adding the last two expressions together, we obtain

$$\frac{22^a - 2}{2(2^a - 1)} = 1.$$

Taking into consideration coprimality of M_i and m_i in (6), we obtain the required result.

The numbers $M_i^{-1} \bmod m_i$ are used in the algorithms of reconstruction from residues [1]. In the "shift" scheme of modulus choice, it is not necessary to calculate and store these values, since they are uniquely determined by the value of a and the modulus number i. As will be shown below, some variants of the Chinese algorithm become totally free of multiplications, since multiplication by $(M_i^{-1} \bmod m_i)$ is implemented by two shift operations, addition, and subtraction. In the case of $a = 1$, only one shift operation is needed.

A more general result is valid for an arbitrary numerical system with an even base B. Consider moduli of type $m_i = B^{2^i} + 1$ $(i = 0, 1, \ldots, k)$ and products $M_i = \prod_{j=0}^{i-1} m_j = \prod_{j=0}^{i-1} (B^{2^j} + 1)$ $(i = 1, 2, \ldots, k)$.

Proposition 2.

$$M_i^{-1} \bmod m_i = \frac{B^{2^i} - B + 2}{2}, \quad i = 1, 2, \ldots, k.$$

Proof. Noting that $M_i = \sum_{j=0}^{2^i - 1} B^j$, $i = 1, 2, \ldots$, we set

$$s_i = \frac{B^{2^i} - B + 2}{2}, \quad t_i = -\frac{M_i - 1}{2}, \quad i = 1, 2, \ldots.$$

It is easy to check that $M_i s_i + m_i t_i = 1$, $i = 1, 2, \ldots$. Taking into consideration coprimality of m_i and M_i, we obtain the required result.

This result can be used for implementation of multiplication-free modular arithmetic algorithms similar to those considered below in computer algebra systems that use integer number representation with bases 10 or 10^l. Just as for GMP system, it is not necessary to calculate and store numbers $M_i^{-1} \bmod m_i$, and the complexity of multiplication by these numbers is proportional to the bit length of the result.

3. MODULAR ARITHMETIC WITH CUNNINGHAM NUMBERS AS MODULI

Let m, u, and v be positive integers such that $0 \le u, v < m$. In calculating $u \pm v \bmod m$, an additional subtraction (if $u \pm v \ge m$) or addition (if $u \pm v < 0$) may be needed. In order to calculate $u \times v \bmod m$, a division with remainder may be needed (if $u \times v \ge m$). Bit complexity of division with remainder has an order of magnitude of $M(\log_2 m)$.

If modulus m is of type $2^n - 1$, then the addition ($\oplus$), subtraction ($\ominus$), and multiplication ($\otimes$) operations can be defined as [1]

$$u \oplus v = ((u + v) \bmod 2^n) + [u + v \geq 2^n], \quad (7)$$

$$u \ominus v = ((u - v) \bmod 2^n) - [u < v], \quad (8)$$

$$u \otimes v = (uv \bmod 2^n) \oplus \lfloor uv/2^n \rfloor. \quad (9)$$

Here, just as in [1], $[B]$ denotes characteristic function of a Boolean expression B: $(B \Rightarrow 1; 0)$. In fact, in (7), we have ordinary addition modulo 2^n, and, in the case of overflow, the obtained result is increased by 1 (note, that repeated overflow cannot occur in this case).[2]

In (9), ordinary multiplication is performed; then, n lower bits are added to the upper n bits using the operation defined in (7). Thus, potential division by m with remainder is replaced by addition and shifts, which are operations of bit complexity $\Theta(\log_2 m)$. This speeds up basic operations not only in the case of very large moduli m but even for moduli of the length of one machine word.

Similarly, if modulus m is of type $2^n + 1$, then addition ($\oplus$), subtraction ($\ominus$), and multiplication ($\otimes$) can be defined as follows:

$$u \oplus v = ((u + v) \bmod 2^n) - [u + v > 2^n], \quad (10)$$

$$u \ominus v = ((u - v) \bmod 2^n) + [u < v], \quad (11)$$

$$u \otimes v = (uv \bmod 2^n) \ominus \lfloor uv/2^n \rfloor. \quad (12)$$

Thus, the use of the Cunningham numbers speeds up basic operations of modular arithmetic.

4. CONVERSION INTO MODULAR REPRESENTATION

Operation of bit substring extraction from the binary representation of a positive integer will be frequently used below. Consider a function BITS(a, m, n) that returns a positive integer consisting of the binary digits of a with positions from m to n. Thus, if $a = \sum_{i=0}^{l} a_i 2^i$ ($a_i \in \{0, 1\}$) and $m \leq n \leq l$, then

$$\mathrm{BITS}(a, m, n) = \sum_{i=0}^{n-m} a_{m+i} 2^i.$$

In order to obtain the remainder of division of a by $2^n - 1$ in the case of $a < 2^{2n}$, we can use the same method as in (9):

[2] Note that, in this version of modular arithmetic, the number 0 can be also represented as $2^n - 1$, and this does not violate correctness of the result.

Algorithm 1 Simple-Reduce-Minus(a, n)—calculates $a \bmod (2^n - 1)$.

Require: $0 \leq a < 2^{2n}$
Ensure: $0 \leq a \leq 2^n - 1$
1: $t \leftarrow$ BITS(a, n, $2n - 1$)
2: $a \leftarrow$ BITS(a, 0, $n - 1$)
3: $a \leftarrow a + t$
4: $t \leftarrow$ BITS(a, n, n)
5: $a \leftarrow$ BITS(a, 0, $n - 1$) + t

If the bit length of the number a is greater than $2n$, some more operations need to be performed (see [1]). Let $|a|$ be the bit length of number a. Then,

$$a = x_t 2^{nt} + x_{t-1} 2^{n(t-1)} + \ldots + x_1 2^n + x_0,$$

where $t = \lceil |a|/n \rceil$ and $0 \leq x_i < 2^n$, $i = 0, 1, \ldots, t$. Using the fact that $2^n \equiv 1 \bmod 2^n - 1$, we obtain $a \equiv x_t + x_{t-1} + \ldots + x_1 + x_0 \bmod (2^n - 1)$. Thus, in order to calculate $a \bmod (2^n - 1)$ one should partition binary representation of number a into n-bit blocks and sum up the obtained numbers modulo $(2^n - 1)$. The variant of this algorithm from [1] is presented below.

Algorithm 2 Theoretic-Reduce-Minus(a, n)—calculates $a \bmod (2^n - 1)$.

Require: $0 \leq a$
Ensure: The returned number r satisfies the condition $0 \leq r \leq 2^n - 1$
1: $r \leftarrow 0$
2: $b \leftarrow (|a| - 1)/n + 1$
3: **for** $i = 0$ **to** $b - 1$
4: $\quad r \leftarrow r +$ BITS(a, $i \cdot n$, $(i + 1) \cdot n - 1$)
5: **end for**
6: **if** $r \geq 2^n$ **then**
7: $\quad$ **return** THEORETIC-REDUCE-MINUS(r, n)
8: **else**
9: $\quad$ **return** r
10: **end if**

The bit complexity of this algorithm is $\Theta(|a|)$ (which is better than complexity of division of a $|a|$-bit number by an n-bit number) assuming that $n \in \Theta(1)$, and BITS(a, m, n) has complexity $\Theta(n - m)$. Unfortunately, in the case of the GMP package, the latter assumption is not valid. In order to extract a bit substring of length l from an integer a in GMP, $\Theta(|a|)$ operations are needed, in spite of repeated requests from users to improve this situation [7]. Thus, the bit complexity of the implementation of this algorithm in GMP is $\Theta(|a|^2/n)$.

Nevertheless, obtaining the remainder with complexity $\Theta(|a|)$ is possible by using the "divide-and-conquer" strategy, as shown in the following algorithm:

Algorithm 3 DC-Reduce-Minus(a, n)—calculates $a \bmod (2^n - 1)$.

Require: $0 \le a$
Ensure: $0 \le a \le 2^n - 1$
1: **while** $a \ge 2^n$ **do**
2: $b \leftarrow (|a| - 1)/n + 1$
3: $b \leftarrow b/2$
4: $t \leftarrow \mathrm{BITS}(a, bn, 2bn - 1)$
5: $a \leftarrow \mathrm{BITS}(a, 0, bn - 1)$
6: $a \leftarrow a + t$
7: **end while**
8: **return** a

This algorithm uses the fact that, for any positive integers x and y, number $2^x - 1$ divides $2^{xy} - 1$. On each iteration, the binary representation of the current value of a is partitioned into two blocks of length divisible by n, which are then summed up. Congruence $\mathrm{BITS}(a, 0, bn - 1) + \mathrm{BITS}(a, bn, 2bn - 1) \equiv a \bmod (2^n - 1)$ is conserved.

Algorithms 1 and 2 can be modified in order to calculate remainders modulo $2^n + 1$ in the same way as (10)–(12) are modifications of (7)–(9) using the equality $2^n \equiv -1 \pmod{2^n + 1}$ instead of $2^n \equiv 1 \pmod{2^n - 1}$. In order to calculate the remainder for $|a| > 2n$ with costs $\Theta(|a|)$, one can use the fact that $2^n + 1$ divides $2^{2n} - 1$ and calculate the remainder modulo $2^{2n} - 1$, which can be then easily reduced modulo $2^n + 1$, as shown in the following algorithm:

Algorithm 4 DC-Reduce-Plus(a, n)—calculates $a \bmod 2^n + 1$.

Require: $0 \le a$
Ensure: $0 \le a \le 2^n + 1$
1: DC-REDUCE-MINUS(a, $2n$)
2: $t \leftarrow \mathrm{BITS}(a, n, 2n - 1)$
3: $a \leftarrow \mathrm{BITS}(a, 0, n - 1)$
4: $a \leftarrow a - t$
5: **if** $a < 0$ **then**
6: $a \leftarrow a + 2^n + 1$
7: **end if**
8: **return** a

5. RECONSTRUCTION OF RESULT FROM RESIDUES

Below we present the Garner algorithm, one of the popular algorithms for reconstructing a positive integer from the residues.

Algorithm 5 Garner(r, m)

Require: $\forall i \in [0, N - 1]: 0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N - 1$ **do**
3: $M \leftarrow M m_{i-1}$
4: $M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a_0 \leftarrow r_0$
7: **for** $i = 1$ **to** $N - 1$ **do**
8: $t \leftarrow a_{i-1}$
9: **for** $j = i - 2$ **downto** 0 **do**
10: $t \leftarrow t m_j$
11: $t \leftarrow t + a_j$
12: **end for**
13: $t \leftarrow r_i - t$
14: $a_i \leftarrow t M_i \bmod m_i$
15: **end for**
16: $a \leftarrow a_{N-1}$
17: **for** $i = N - 1$ **downto** 0
18: $a \leftarrow a m_i$
19: $a \leftarrow a + a_i$
20: **end for**
21: **return** a

The prove of correctness of this algorithm can be found in standard textbooks on computer algebra [1, 8]; therefore, we will only give a brief comment to the algorithm and discuss optimizations brought by using the Cunningham numbers.

In lines 1–5 of the algorithm, intermediate values $\left(\prod_{j=0}^{i-1} m_j\right)^{-1} \bmod m_i$ are calculated. Note that these values can be calculated once and then be used every time we need to reconstruct the result obtained in the same system of moduli. In lines 6–15, coefficients of the mixed radix representation are calculated, i.e., numbers $a_0, a_1, \ldots, a_{N-1}$ such that

$$X = a_0 + a_1 m_0 + a_2 m_0 m_1 + \ldots + a_{N-1} m_0 m_1 \ldots m_{N-2},$$
$$0 \le a_i < m_i, \quad 0 \le X < m_0 m_1 \ldots m_{N-1},$$

and $X \equiv r_i \pmod{m_i}$, $i = 0, 1, \ldots, N - 1$. In lines 16–20, the value of X is calculated using Horner's method.

If Cunningham numbers of types 2+ or 2– are used as the moduli, multiplying by modulus m_i (lines 3, 10,

18) and calculating the remainder of division by m_i (lines 4 and 14) are replaced by shifts, additions, and subtractions. If the scheme with the choice of moduli of type 2+ based on "shifts" is used, there is no longer need in calculating and storing the intermediate values in lines 1–5, and multiplication by such a value (line 14) is replaced by shifts and additions-subtractions with the use of (3). Thus, we obtain a multiplication-free implementation of the Garner algorithm.

There also exist other variants of reconstruction algorithms. The algorithm presented below uses the same auxiliary values as the Garner algorithm, but does not calculate an intermediate mixed radix representation

Algorithm 6 Simple–Reconstruction(r, m)

Require: $\forall i \in [0, N-1]$: $0 \le r_i < m_i$
Ensure: $a \equiv r_i \bmod m_i$
1: $M \leftarrow 1$
2: **for** $i = 1$ **to** $N-1$ **do**
3: $M \leftarrow Mm_{i-1}$
4: $M_i \leftarrow M^{-1} \bmod m_i$
5: **end for**
6: $a \leftarrow r_0$
7: $M \leftarrow m_0$
8: **for** $i = 1$ **to** $N-1$ **do**
9: $t \leftarrow r_i - a$
10: $t \leftarrow tM_i \bmod m_i$
11: $a \leftarrow a + tM$
12: $M \leftarrow Mm_i$
13: **end for**
14: **return** a

Like in the case of the Garner algorithm, this algorithm can be improved by using the Cunningham numbers. If the scheme of the choice of moduli of type 2+ based on shifts is used, there is either no need in calculating and storing the intermediate values in lines 1–5. The only multiplication that can not be avoided is that by M in line 11 where $M = m_0, m_1, \ldots, m_{i-1}$ in the ith iteration. However, having chosen parameter a in (1) large, we can take advantage of sparsity of the binary representation of the product of moduli (see [4]) and reduce this multiplication to a series of shifts and additions.

6. IMPLEMENTATION

In order to confirm experimentally that the Cunningham numbers can speed up integer arithmetic, all the algorithms described above were implemented in C++, as an application for the GMP program package [9]. Both standard versions of these algorithms (i.e., versions that do not take into consideration the specific modulus type) and the versions optimized for the Cunningham numbers of types 2– and 2+ were implemented. Besides, we also implemented a version of modular arithmetic that uses only prime moduli with the bit length not exceeding machine word.

In order to compare the time costs of different implementations, we used the same problem with the initial data of different bit length, namely, multiplication of two matrices with integer elements. Matrices of sizes 64×64 and 128×128 were generated randomly using tools of the GMP package. The bit length of the matrix elements varied from 16 to $12288 = \frac{3}{2} \cdot 8192$ bits.

Test runs were held on the AMD Duron 750M processor with 512 Mb RAM in the Debian GNU/Linux operating system supplied with the GMP package version 4.1.4–6. We used the GNU g++ compiler version 1:3.3.5–13. For the sake of comparison, each matrix multiplication was also run using the GMP built-in fast integer arithmetic (`mpz_class` methods). A detailed description of the implementation, as well as comparison of run times for different sizes of input data, can be found in [10]. Here, we present only several revealed observations.

• For comparatively small values of bit length of the matrices elements, the GMP built-in arithmetic is the fastest. This is not surprising, since the GMP package contains implementations of the best algorithms of integer multiplication optimized for the processor architecture and chooses the fastest code for the specific size of the multipliers.

• Modular arithmetic optimized for the Cunningham numbers is always faster than the standard modular arithmetic.

• The time for the result reconstruction is reduced considerably if the multiplication-free variant of the Garner algorithm is used: from 20% of total problem run time to 0.4% for the same input data.

• When the bit length of matrix elements increases, the implementation that uses the shift scheme generated moduli of type 2+ starts to prevail over the fast GMP arithmetic. For example, multiplication of two pseudorandom matrices of size 64×64 with the elements uniformly distributed between 0 and $2^{32768} - 1$ based on the use of the shift scheme with the first modulus $2^{65} + 1$ took 283 seconds. The same calculation by the `mpz_class` methods took 495 seconds.

• Efficiency of the shift scheme depends on the selection of the parameter a (exponent of the smallest modulus). The appropriate selection is possible when the initial data allow obtaining a good estimate of the result size.

Note that the use of the Cunningham numbers saves memory as well. Instead of the moduli, the corresponding exponent values are stored: if the maximum size of intermediate results is equal to N, then, instead of approximately N bits, only $\log_2 N$ bits are required for

storing moduli $m_1, \ldots, m_k$. If numbers of type 2+ with the shift scheme of modulus generation are used, $\log_2 a + \log_2\log_2 N$ bits are required, since one needs to store only the smallest exponent a and number i for each modulus $2^{a2^i} + 1$. There is also no need in storing intermediate values in the algorithms of reconstruction by the residues. In the general case, these values occupy about N bits of memory.

Another feature of our implementation of modular algorithms is that new moduli can be added dynamically, as new input data become available, based on of the estimates of possible size of the arithmetic operation result. In this situation, the list of intermediate values is updated, and all calculations performed by this moment are repeated for the new modulus value. This scheme of modular arithmetic programs operation needs further development. Difficulties here are associated with the fact that it is rather difficult to estimate the result size or compare two numbers on the basis of residues and moduli values. The use of rough estimates can lead to generating redundant moduli and performing unnecessary calculations.

7. CONCLUSIONS

Modular calculations are inherently parallelizable (computations in each residue class can be performed in parallel). The use of base-2 Cunningham numbers makes it possible to exploit bit level parallelizm for reductions, as well as for reconstruction algorithms. Currently, we carry out the work on prototyping the algorithms described above for the programmable logic integrated circuits in VHDL.

We also investigate dynamic schemes of modular calculations allowing one to discard individual moduli if the new generated modulus is too large. This may help to solve some problems with non-balanced sizes of moduli of type 2+.

It is known that moduli can be chosen in such a way that all intermediate values in reconstruction algorithms are equal to ±1 [11]. For example, one can choose $m_i = \prod_{j=1}^{i-1} m_j + 1$. This not only results in moduli that are not balanced in size, but also does not allow us to avoid standard multiplication by m_i in reconstruction algorithms. From this point of view, the choice of non-balanced moduli of type 2+ appears an acceptable and practically useful compromise.

ACKNOWLEDGMENTS

The authors are grateful to Professor Stinson and Professor Storjohann (University of Waterloo) for useful discussions of preliminary materials of this paper, as well as to Professor Shallit (University of Waterloo) for providing reference to the work [4].

REFERENCES

1. Knuth, D.E., *The Art of Computer Programming*, vol. 2.
2. Brillhart, J., Lehmer, D.H., Selfridge, J.L., Tuckerman, B., and Wagstaff, S.S., Jr., Factorizations of $b^n + 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ Up to High Powers, *3rd Ed. Contemporary Mathematics Series*, vol. 22, Providence: Am. Math. Soc., 2002.
3. Wagstaff, S., *The Cunningham Project*, http://www.cerias.purdue.edu/homes/ssw/cun/index.html.
4. Cade, J.J., Kee-Wai, Lau, Pedersen, A., and Lossers, O.P., Problem E3288. Problems and Solutions, *The Am. Math. Monthly*, 1990, vol. 97, no. 4, pp. 344–345.
5. Křížek, M., Luca, F., and Somer, L., *17 Lectures on Fermat Numbers: From Number Theory to Geometry.* New York: Springer, 2001.
6. Solinas, J.A., *Generalized Mersenne Numbers.* Waterloo: Faculty of Mathematics, Univ. of Waterloo, 1999.
7. Bernstein, D.J., *Want Fast Bit Set/Extract*, Message to GMP Discussions List ⟨gmp-discuss@swox.com⟩. April 17, 2002.
8. Geddes, K.O., Czapor, S.R., and Labahn, G., *Algorithms for Computer Algebra* (6th printing), Boston: Kluwer, 1992.
9. *GNU MP 4.1*, http://www.swox.com/gmp/manual/.
10. Stewart, A. and Zima, E., Base-2 Cunningham Numbers in Modular Arithmetic, *Technical Report*, Wilfrid Laurier Univ., January 2006.
11. Szabo, N.S. and Tanaka, R.I., *Residue Arithmetic and Its Applications to Computer Technology*, McGraw-Hill, 1967.
12. Montgomery, P.L., Modular Multiplication Without Trial Division, *Math. Computation*, 1985, vol. 44, no. 170, pp. 519–521.