Efficient GPU Implementation of Multiple-Precision Addition based on Residue Arithmetic

Konstantin Isupov[1]
Department of Electronic Computing Machines
Vyatka State University
Kirov, Russia 610000

Vladimir Knyazkov[2]
Research Institute of Fundamental and Applied Studies
Penza State University
Penza, Russia 440026

***Abstract*—In this work, the residue number system (RNS) is applied for efficient addition of multiple-precision integers using graphics processing units (GPUs) that support the Compute Unified Device Architecture (CUDA) platform. The RNS allows calculations with the digits of a multiple-precision number to be performed in an element-wise fashion, without the overhead of communication between them, which is especially useful for massively parallel architectures such as the GPU architecture. The paper discusses two multiple-precision integer algorithms. The first algorithm relies on *if-else* statements to test the signs of the operands. In turn, the second algorithm uses radix complement RNS arithmetic to handle negative numbers. While the first algorithm is more straightforward, the second one avoids branch divergence among threads that concurrently compute different elements of a multiple-precision array. As a result, the second algorithm shows significantly better performance compared to the first algorithm. Both algorithms running on an NVIDIA RTX 2080 Ti GPU are faster than the multi-core GNU MP implementation running on an Intel Xeon 4100 processor.**

***Keywords*—Multiple-precision algorithm; integer arithmetic; residue number system; GPU; CUDA**

I. Introduction

Multiple-precision integer arithmetic, which provides operations with numbers that consist of more than 32 or 64 bits, is an important and often indispensable method for solving scientific and engineering problems that are difficult to solve using the standard numerical precision. The most notable application of multiple-precision integer arithmetic is cryptography, where the level of security depends on the length of the keys [1], [2]. Multiple precision is also required in computer algebra (symbolic computation) systems, which operate with mathematical expressions instead of fixed-precision integer and floating-point numbers [3]. The intermediate data produced during a computation may be very large, and multiple-precision arithmetic is required to prevent overflow. Another problem requiring computations with very large integers and of practical interest in polymer physics is counting Hamiltonian cycles on two- and three-dimensional lattices, triangular grid graph, and other structures [4]. Multiple-precision arithmetic is becoming more and more in demand as the scale of computations increases.

There are several approaches for implementing multiple-precision arithmetic. One of them is special software libraries that emulate operations with large numbers using standard fixed-precision operations. Some of the well-known libraries for central processors (CPUs) include the GNU MP Bignum Library (GMP) [5], the Library for doing Number Theory (NTL) [6], and thr Fast Library for Number Theory (FLINT) [7]. There are also works devoted to the implementation of integer arithmetic operations with arbitrary/multiple precision on GPUs [8], [9], [10], [11], [12].

A higher level of arithmetic precision is also supported in a number of programming languages, e.g., Python (the built-in *int* type), Ruby (the built-in *Bignum* type), Perl (*Math::BigInt*), Java (the *BigInteger* class), Haskell (the *Integer* datatype), and C# (*BigInteger*). Another actual approach is to develop hardware accelerators that support integer and floating-point computations with multiple precision [13], [14], [15].

Previous research in [8], [9], [13], and [15] use the traditional way of representing multiple-precision numbers, according to which a number is represented as an array of weighted digits in some base, and the digits themselves are machine-precision numbers [16]. The need for carry propagation under this number representation is one of the main bottleneck of efficient multiple-precision algorithms.

This paper deals with another type of multiple-precision arithmetic, which is based on the residue number system (RNS) [17], [18]. In the RNS, a number is represented by its residues relative to a set of moduli. The moduli are mutually independent, so multiple-precision integer operations such as addition, subtraction, and multiplication are replaced by groups of reduced-precision operations with residues performed in an element-wise fashion and without the overhead of manipulating carry information between the residues.

Recently, a new software library has been developed for efficient residue number system computations on CPU and GPU architectures. The library is called GRNS and is freely available for download at https://github.com/kisupov/grns. GRNS is designed for arbitrary moduli sets with large dynamic ranges that far exceed the usual word length of computers, up to several thousand bits. In addition to a number of optimized non-modular RNS operations such as magnitude comparison and division, GRNS implements multiple-precision integer arithmetic. This paper considers two multiple-precision addition algorithms implemented in GRNS. Along with multiplication, addition and subtraction is key operations for many computational algorithms, e.g., fast Fourier transform. Multiple-precision addition is usually considered to be faster and easier than multiplication. However, in the case of RNS, signed addition is more difficult than multiplication as it requires determining the sign of the result, which is a time-consuming

operation for RNS.

Both of our multiple-precision addition algorithms use an interval floating-point evaluation technique for efficient RNS sign determination [19]. However, the first algorithm relies on *if-else* statements to test the signs of the operands, while the second one uses the radix complement RNS notation for negative numbers. It is shown that the second algorithm is better suited for implementation on massively parallel GPU architectures than the first algorithm.

The rest of this paper is organized as follows. Section II provides the background on RNS arithmetic. Section III describes the RNS-based format of multiple-precision integer numbers. Multiple-precision addition algorithms are presented in Section IV. Performance comparison results are given in Section V, and Section VI concludes the paper.

II. Background on RNS Arithmetic

An RNS is specified by a set of n pairwise prime moduli $\{m_0, m_1, \ldots, m_{n-1}\}$. The dynamic range of the RNS is $M = m_0 \cdot m_1 \cdots m_{n-1}$. The mapping of an integer X into the RNS is defined to be the n-tuple $(x_0, x_1, \ldots, x_{n-1})$, where $x_i = |X|_{m_i}$ is the smallest non-negative remainder when X is divided by m_i, that is, $x_i = X \bmod m_i$. Within the RNS there is a unique representation of all integers in the range from 0 to $M - 1$. Namely, the Chinese Remainder Theorem (CRT) states that [18]

$$|X|_M = \left| \sum_{i=0}^{n-1} M_i \left| x_i w_i \right|_{m_i} \right|_M , \tag{1}$$

where $M_i = M/m_i$, and $w_i = \left| M_i^{-1} \right|_{m_i}$ is the modulo m_i multiplicative inverse of M_i.

Since the RNS moduli are independent of each other, arithmetic operations such as addition, subtraction, and multiplication can be computed efficiently. If X, Y, and Z have RNS representations given by $(x_0, x_1, \ldots, x_{n-1})$, $(y_0, y_1, \ldots, y_{n-1})$, $(z_0, z_1, \ldots, z_{n-1})$, then denoting $\circ$ to represent $+$, $-$, or $\times$, the RNS version of the $Z = X \circ Y$, satisfies

$$\begin{aligned} Z =& (z_0, z_1, \ldots, z_{n-1}) \\ =& (|x_0 \circ y_0|_{m_0}, |x_1 \circ y_1|_{m_1}, \ldots, |x_{n-1} \circ y_{n-1}|_{m_{n-1}}) \end{aligned} \tag{2}$$

provided that $Z \in [0, M-1]$. Thus the ith RNS digit, namely z_i, is defined in terms of $|x_i \circ y_i|_{m_i}$ only. That is, no carry information need be communicated between residue digits, and the overhead of manipulating carry information in more traditional, weighted-number systems can be avoided [20].

The disadvantage of RNS is the high complexity of estimating the magnitude of a number, which is required to perform number comparison, sign calculation, overflow checking, division, and some other operations. The classic technique to perform these operations is based on the CRT formula (1) and consists in computing the binary representations of numbers with their subsequent analysis. However, in large dynamic ranges (e.g., a few thousand bits) this technique becomes slow. Other methods for evaluating the magnitude of residue numbers are based on the mixed-radix conversion (MRC) process [21]. But these methods are often also ineffective since they require a lot of arithmetic operations with residues or the use of unacceptably large lookup tables.

An alternative method for implementing time-consuming operations in the RNS is based on computing the floating-point interval evaluation of the fractional representation of an RNS number [19]. This method is designed to be fast on modern general-purpose computing platforms that support efficient finite precision floating-point arithmetic operations such as IEEE 754 operations. For a given RNS number $X = (x_0, x_1, \ldots, x_{n-1})$, the floating-point interval evaluation is an interval defined by its lower and upper bounds (endpoints) $\underline{X/M}$ and $\overline{X/M}$ that are finite precision floating-point numbers satisfying $\underline{X/M} \le X/M \le \overline{X/M}$. The floating-point interval evaluation is denoted by $I(X/M) = [\underline{X/M}, \overline{X/M}]$.

Thus, $I(X/M)$ provides information about the range of changes in the fractional representation (also called relative value) of an RNS number. This information may not be sufficient to restore the binary representation, but it can be efficiently used to perform other difficult operations in RNS, e.g., magnitude comparison, sign detection, and division.

The most important benefit of this method is that computation of $I(X/M)$ requires only standard arithmetic operations, and no residue-to-binary conversion is required. For a given RNS representation $(x_0, x_1, \ldots, x_{n-1})$, the calculation of the bounds of $I(X/M)$ is performed on average in linear and logarithmic time for sequential and parallel cases, respectively. Furthermore, the following arithmetic operations are defined:

$$\begin{aligned} I(X/M) + I(Y/M) &= \left[\underline{X/M} \mathbin{\triangledown} \underline{Y/M},\ \overline{X/M} \mathbin{\triangleq} \overline{Y/M} \right], \\ I(X/M) - I(Y/M) &= \left[\underline{X/M} \mathbin{\triangledown} \overline{Y/M},\ \overline{X/M} \mathbin{\triangleq} \underline{Y/M} \right], \\ I(X/M) \times I(Y/M) &= \left[\underline{X/M} \mathbin{\bigtriangledown} \underline{Y/M} \mathbin{\triangledown} W,\ \overline{X/M} \mathbin{\bigtriangleup} \overline{Y/M} \mathbin{\triangleq} V \right], \\ I(X/M) \div I(Y/M) &= \left[\underline{X/M} \mathbin{\bigtriangledown} V \mathbin{\triangledown} \overline{Y/M},\ \overline{X/M} \mathbin{\bigtriangleup} W \mathbin{\triangleq} \underline{Y/M} \right]. \end{aligned} \tag{3}$$

In these interval formulas, the following notation are used:

- $\triangledown$, $\bigtriangledown$, $\bigtriangledown$ and $\triangledown$ stand for the floating-point operations of addition, subtraction, multiplication, and division, performed with rounding downwards;
- $\triangleq$, $\bigtriangleup$, $\bigtriangleup$ and $\triangleq$ stand for the floating-point operations of addition, subtraction, multiplication, and division, performed with rounding upwards;
- V is the greatest floating-point number that is less than or equal to $1/M$;
- W is the least floating-point number greater than or equal to $1/M$.

Interval formulas (3) are useful in that they do not limit the possible values of the result interval in the range of $[0, 1)$. This allows for easy overflow detection or sign identification despite the cyclical (modulo M) nature of RNS arithmetic.

Using interval evaluations, new algorithms have been proposed in [19] to efficiently implement several difficult RNS operations, such as number comparison and general division.

III. Number Representation

The format for multiple-precision integers is shown in Fig. 1. A multiple-precision integer x consists of a sign s, a significand X composed of n significand digits (x_0 to x_{n-1}), and an interval floating-point evaluation of the significand $I(X/M) = [\underline{X/M}, \overline{X/M}]$. The sign is interpreted in the same way as in two's complement representation: the sign is equal

to zero when x is positive and one when it is negative. The significand expresses the absolute value of x and is represented in the RNS with the moduli set $\{m_0, m_1, \ldots, m_{n-1}\}$. The significand digits (residues) are represented as ordinary two's complement integers.

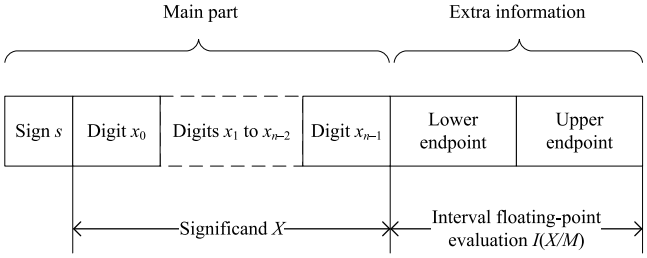


Fig. 1. Multiple-Precision Integer Format.

The size of the moduli set n specifies the number of digits in the significand. If the product of all RNS moduli is M, then the precision of x is equal to $\lfloor \log_2 M \rfloor$ bits. Thus, changing the size of the moduli set allows one to achieve arbitrary precision.

The following notation is used to denote a multiple-precision integer in the described number format:

$$x = \langle s, X, I(X/M) \rangle. \tag{4}$$

The value of a multiple-precision integer of the form (4) can be computed using the CRT formula:

$$x = (-1)^s \times \left| \sum_{i=0}^{n-1} M_i \left| x_i w_i \right|_{m_i} \right|_M. \tag{5}$$

The interval evaluation is included in the number representation as additional information, that is, $\underline{X/M}$ and $\overline{X/M}$ are stored in system memory along with other fields of the multiple-precision integer. This provides efficient comparison, sign computation and overflow detection, allowing one to calculate $I(X/M)$ in $O(1)$ time using the formulas (3). Recently, this approach has been successfully used in the context of multiple-precision floating-point arithmetic based on the residue number system [22].

In order to be able to use interval evaluations for virtually any (arbitrarily large) value of M without worrying about underflow, $\underline{X/M}$ and $\overline{X/M}$ are represented as binary floating-point numbers with an extended exponent range, that is, have the form

$$f \times 2^e, \tag{6}$$

where f is a regular floating-point number (IEEE 754), and i is a two's complement integer. This extended-range representation is not intended to improve the level of numerical precision or accuracy, but it does ensure that there is no overflow or underflow when dealing with extremely large or small values.

IV. Multiple-Precision Integer Addition

In this section, two algorithms for signed multiple-precision integer addition are presented. A naive implementation is presented first and then an improved one. Step-by-step examples are also provided for both implementations.

A. Useful Notation

For given $a \in \{0, 1\}$ and $X = (x_0, x_1, \ldots, x_{n-1})$, the paper [22] introduces a function $B[X, a]$ such that

$$B[X, a] = \begin{cases} \underline{X/M}, & \text{for } a = 0, \\ \overline{X/M}, & \text{for } a = 1. \end{cases} \tag{7}$$

That is, the lower bound of $I(X/M)$ is denoted by $B[X, 0]$, while the upper one is denoted by $B[X, 1]$. Using this notation, we have $I(X/M) = [B[X, 0], B[X, 1]]$. This notation is useful in that it allows one to dynamically specify the bound to be accessed. This notation is used in the rest of the present paper.

B. Note on Overflow Detection

For the set of RNS moduli $\{m_0, m_1, \ldots, m_{n-1}\}$, the largest representable integer is $(M–1)$, and the result of an arithmetic operation should belong to the interval $[0, M–1]$ if we want to obtain its valid representation in the RNS. Otherwise, the result will be reduced modulo M, and this event is classified as an integer overflow. The GRNS library implements efficient overflow detection using the floating-point interval evaluations, but that is beyond the scope of this paper.

Algorithm 1 Multiple-precision integer addition

if $s_x = s_y$ **then**
  $s_z \leftarrow s_x$
  for each $i \in \{0, 1, \ldots, n-1\}$ **do**
    $z_i \leftarrow |x_i + y_i|_{m_i}$
  end for
  $B[Z, 0] \leftarrow B[X, 0] \triangledown B[Y, 0]$
  $B[Z, 1] \leftarrow B[X, 1] \triangle B[Y, 1]$
else if $B[X, 0] \geq B[Y, 1]$ **then**
  $s_z \leftarrow s_x$
  for each $i \in \{0, 1, \ldots, n-1\}$ **do**
    $z_i \leftarrow |x_i - y_i|_{m_i}$
  end for
  $B[Z, 0] \leftarrow B[X, 0] \triangledown B[Y, 1]$
  $B[Z, 1] \leftarrow B[X, 1] \triangle B[Y, 0]$
else if $B[Y, 0] \geq B[X, 1]$ **then**
  $s_z \leftarrow s_y$
  for each $i \in \{0, 1, \ldots, n-1\}$ **do**
    $z_i \leftarrow |y_i - x_i|_{m_i}$
  end for
  $B[Z, 0] \leftarrow B[Y, 0] \triangledown B[X, 1]$
  $B[Z, 1] \leftarrow B[Y, 1] \triangle B[X, 0]$
else
  Use mixed-radix conversion to compare the magnitude of X and Y. If $X \geq Y$, subtract Y from X and take $s_z \leftarrow s_x$; otherwise, subtract X from Y and take $s_z \leftarrow s_y$; In any case, $I(Z/M)$ should be recalculated.
end if

C. Algorithm 1 (Naive Implementation)

1) Description: Algorithm 1 takes two multiple-precision integers x and y represented as $x = \langle s_x, X, I(X/M)\rangle$ and $y = \langle s_y, X, I(X/M)\rangle$, and outputs the sum $z = x + y$ represented as $z = \langle s_z, Z, I(Z/M)\rangle$. This algorithm analyzes the signs of the numbers, and if they are the same, then RNS addition of the significands is performed; otherwise, RNS subtraction is performed. The sign of the result is computed by comparing the magnitude of $X = (x_0, x_1, \ldots, x_{n-1})$ and $Y = (y_0, y_1, \ldots, y_{n-1})$ using the floating-point interval evaluations.

2) Illustration: Consider the moduli set $\{7, 9, 11, 13\}$ with the moduli product $M = 9009$. Suppose we are given two integers of the form (3),

$$x = \langle 0, (5, 7, 5, 8), [0.416, 0.420]\rangle,$$
$$y = \langle 1, (3, 7, 6, 4), [0.444, 0.448]\rangle,$$

and we want to find $z = x + y$. Since $B[Y, 0]$ (0.444) is greater than $B[X, 1]$ (0.420), steps 16 to 21 of the algorithm are performed. They are presented in Table I.

TABLE I. EXAMPLE OF ALGORITHM 1

Step no.	Calculations
16	$s_z = 1$
17–19	$Z = (3, 7, 6, 4) - (5, 7, 5, 8) = (5, 0, 1, 9)$
20	$B[Z, 0] = 0.444 \bigtriangledown 0.420 = 0.024$
21	$B[Z, 1] = 0.448 \bigtriangleup 0.416 = 0.032$

The computed result is $z = \langle 1, (5, 0, 1, 9), [0.024, 0.032]\rangle$. We check this result by converting it to decimal: $z = -243$. In fact, $x = 3778$ and $y = -4021$.

3) Drawback: The main disadvantage of Algorithm 1 is that checking the signs of the operands via conditionals (*if–else* statements) results in branch divergence among threads that concurrently compute different elements of a multiple-precision array. This may be normal for modern multi-core processors with good branch prediction accuracy, but this is a problem for SIMT (single instruction, multiple threads) architectures such as GPUs, where many threads run in lock-step.

For example, a CUDA-compliant GPU consists of an array of streaming multiprocessors (SMs), each of which contains multiple streaming processors. Although each SM can run one or more different instructions, conditionals can greatly decrease performance inside an SM, as each branch of each conditional must be evaluated. Long code paths in a conditional can cause a 2-fold slowdown for each conditional within a warp (a group of 32 threads) and a 2^N slowdown for N nested conditionals. A maximum 32-time slowdown can occur when each thread in a warp executes a separate condition [23].

This bottleneck is illustrated in Fig. 2, which contains a flowchart of Algorithm 1. In the figure, 14 threads concurrently compute 14 multiple-precision additions on a system that follows the SIMT execution model. The right side of the figure shows threads running at once.

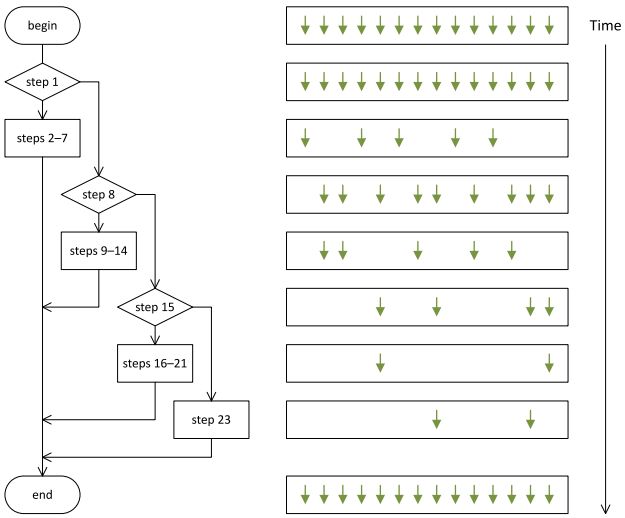


Fig. 2. Flowchart of Algorithm 1.

D. Algorithm 2 (Improved Implementation)

1) Description: Algorithm 2 shows how to avoid conditional expressions when adding multiple-precision signed integers. This algorithm is a simplified version of the multiple-precision RNS-based floating-point addition algorithm that was originally proposed in [22]. The main idea is to use the radix-complement representation of a negative number in the RNS. Recall that the precomputed constant V used in this algorithm is the greatest finite precision floating-point number that is less than or equal to $1/M$.

Algorithm 2 Multiple-precision integer addition using radix complement RNS arithmetic

1: $\alpha \leftarrow (1 - 2s_x)$
2: $\beta \leftarrow (1 - 2s_y)$
3: **for each** $i \in \{0, 1, \ldots, n-1\}$ **do**
4: $\quad z_i \leftarrow (\alpha x_i + \beta y_i) \bmod m_i$
5: **end for**
6: $B[Z, 0] \leftarrow \alpha B[X, s_x] \mathbin{\triangledown} \beta B[Y, s_y]$
7: $B[Z, 1] \leftarrow \alpha B[X, (1 - s_x)] \mathbin{\triangle} \beta B[Y, (1 - s_y)]$
8: **if** $B[Z, 0]$ and $B[Z, 1]$ have the same sign **then**
9: $\quad$ Assign the sign of $B[Z, 0]$ and $B[Z, 1]$ to s_z
10: **else**
11: $\quad$ Use mixed-radix conversion to compare X and Y:
• If $X > Y$, then assign $s_z \leftarrow s_x$.
• If $X < Y$, then assign $s_z \leftarrow s_y$.
• If $X = Y$, then assign $s_z \leftarrow 0$.
12: $\quad B[Z, s_z] \leftarrow (1 - 2s_z)V$
13: **end if**
14: **if** $s_z = 1$ **then**
15: $\quad$ **for each** $i \in \{0, 1, \ldots, n-1\}$ **do**
16: $\quad\quad z_i \leftarrow (m_i - z_i) \bmod m_i$
17: $\quad$ **end for**
18: $\quad$ Swap $B[Z, 0]$ and $B[Z, 1]$ with sign inversion, that is, set $B[Z, 0] = -B[Z, 1]$ and $B[Z, 1] = -B[Z, 0]$
19: **end if**

Fig. 3 shows a flowchart of Algorithm 2. The *if–else* statement at steps 8 to 12 cannot be eliminated, since the accuracy of $B[Z, 0]$ and $B[Z, 1]$ may be insufficient to unambiguously determine the sign of z. This ambiguity is possible due to the limited precision arithmetic used in calculating $B[Z, 0]$ and $B[Z, 1]$. However, this is actually a rare case, and it can only occur when the result is too close to zero.

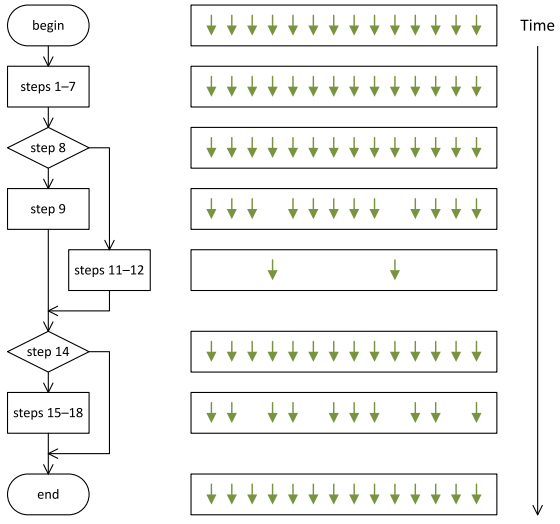


Fig. 3. Flowchart of Algorithm 2.

We note that the *if* statement at step 14 of Algorithm 2 does not cause branch divergence, since there is no the corresponding *else* statement here.

2) Illustration: In Table II, Algorithm 2 is used to compute the sum of the numbers from the previous example.

TABLE II. EXAMPLE OF ALGORITHM 2

Step no.	Calculations
1,2	$\alpha = 1 - 2 \times 0 = 1 \qquad \beta = 1 - 2 \times 1 = -1$
3–5	$z_0 = (5 - 3) \bmod 7 = 2$
	$z_1 = (7 - 7) \bmod 9 = 0$
	$z_2 = (5 - 6) \bmod 11 = 10$
	$z_3 = (8 - 4) \bmod 13 = 4$
6	$B[Z, 0] = 0.416 \mathbin{\triangledown} (-0.448) = -0.032$
7	$B[Z, 1] = 0.420 \mathbin{\triangle} (-0.444) = -0.024$
9	$s_z = 1$
15–17	$Z = (7, 9, 11, 13) - (2, 0, 10, 4) = (5, 0, 1, 9)$
18	$B[Z, 0] = 0.024$, $B[Z, 1] = 0.032$

Thus, as in the first example, the correct result is computed: $z = \langle 1, (5, 0, 1, 9), [0.024, 0.032] \rangle$.

V. PERFORMANCE COMPARISON RESULTS

This section gives comparative results of the presented multiple-precision integer addition algorithms. In the experiments, we used an GeForce RTX 2080 Ti graphics card that has 11 GB of GDDR6 memory, 4352 CUDA cores, and Compute Capability 7.5. This GPU was installed on a machine with an Intel Xeon 4100/8.25M S2066 OEM processor running Ubuntu 18.04.5 LTS, CUDA 10.2 and NVIDIA Driver 450.51.06 were used. The source code was compiled using the nvcc compiler with the *-O3* and *-Xcompiler=-fopenmp* options.

A. Methodology

The parameters of the experiments are shown in Table III. Each dataset was composed of two multiple-precision integer arrays of the same length, and the performance was evaluated for element-by-element addition of the arrays. The performance was measured in the number of multiple-precision arithmetic operations (additions) per second. For comparison purposes, the performance of the GNU MP library was also measured on 4 CPU cores. In the experiments, we considered only the computation time, so the measurements do not include neither the data transfer time nor the time of converting data into internal multiple-precision representations.

TABLE III. EXPERIMENTAL PARAMETERS

Parameter	Value
Size of the RNS moduli set, n	from 8 to 256
Bit width of each modulus	32
Precision in bits, p	from 128 to 4096
Dataset size	1,000,000
Datasets	*Dataset-1*: pseudo-random integers in the range 0 to $(M - 1)/2$
	Dataset-2: pseudo-random integers in the range $(1 - M)/2$ to 0
	Dataset-3: pseudo-random integers in the range $(1 - M)/2$ to $(M - 1)/2$

For each precision p, a corresponding set of RNS moduli was generated such that

$$\lfloor \log_2 M \rfloor \geq p, \tag{8}$$

where M is the product of all the moduli in the set. Table IV shows the relationship between the precision and moduli sets used in the experiments.

TABLE IV. RELATIONSHIP BETWEEN THE PRECISION AND MODULI SETS USED IN THE EXPERIMENTS

Precision, p	Size of moduli set, n	Dynamic range, M (approx.)
128	8	3.4864747615962733744449E+38
256	16	1.1828692372765598929 56E+77
512	32	1.3817508674984534848 69E+154
1024	64	1.8349720826501144353 87E+308
2048	128	3.2674938937887830734 05E+616
4096	256	1.1137168375511667691 74E+1233

The moduli sets were generated using Algorithm 3. This algorithm takes as input the smallest odd modulus m_0, the size of the desired moduli set n, and produces an increasing sequence of $n-1$ consecutive odd integers $m_1, m_2, \ldots, m_{n-1}$ that are coprime to each other and also coprime to m_0. The value of m_0 is selected by trial and error until the condition (8) is satisfied. The used tool for generating moduli sets is freely available at https://github.com/kisupov/rns-moduli-generator.

For the CUDA implementations of the presented multiple-precision addition algorithms, 32 threads per each thread block were used, and the total number of blocks was calculated as follows:

$$nBlocks = \left\lfloor \frac{N}{nThreads} \right\rfloor + K, \tag{9}$$

where N is the size of the dataset (1,000,000), $nThreads = 32$, and K is defined as

$$K = \begin{cases} 1, & \text{if } N \bmod nThreads > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

The GNU MP library implementation have been accelerated using the OpenMP library.

B. Results

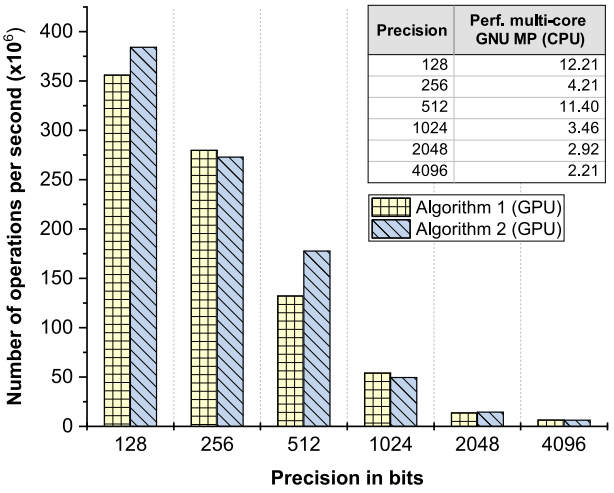
In the first experiment, the input arrays were filled with pseudo-random non-negative integers ranging from 0 to $(M-1)/2$, where M is the product of all the RNS moduli. The performance results are shown in Fig. 4.

In the second experiment, the input arrays were filled with pseudo-random non-positive integers ranging from $(1-M)/2$ to 0. The results are reported in Fig. 5.

Finally, in the third experiment, the input arrays were filled with pseudo-random positive and negative integers ranging from $(1-M)/2$ to $(M-1)/2$. Fig. 6 demonstrates the performance results obtained in this setting.

Algorithm 3 Moduli set generation

```
t ← m_0 + 2
k ← 1
while k < n do
    p ← 1
    for i ← 1 to k do
        if gcd(m_i, t) > 1 then
            p ← 0
        end if
    end for
    if p = 1 then
        m_k ← t
        k ← k + 1
    end if
    t ← t + 2
end while
```



Precision	Perf. multi-core GNU MP (CPU)
128	12.21
256	4.21
512	11.40
1024	3.46
2048	2.92
4096	2.21

Fig. 4. Performance of Multiple-Precision Integer Addition Implementations with Non-Negative Inputs (Dataset-1).

C. Discussion

For Dataset-1 (Fig. 4), Algorithm 1 has nearly the same performance as Algorithm 2. This is because in Algorithm 1, all parallel threads follow steps 2–7 and there is no divergent execution paths. The results show that the developed CUDA functions are up to $65\times$ faster than the parallel CPU implementation using GNU MP.

In the case of Dataset-2 (Fig. 5) the performance of Algorithm 1 remains the same as in the case of Dataset-1, since there are still no branch divergence (all parallel threads follow steps 2–7). In turn, the need to restore negative results reduces the performance of Algorithm 2 by an average of $1.1\times$ compared to Dataset-1, and this performance degradation does not seem to be significant.

When using Dataset-3 (Fig. 6), branch divergence leads to an average 1.9-fold decrease in the performance of Algorithm

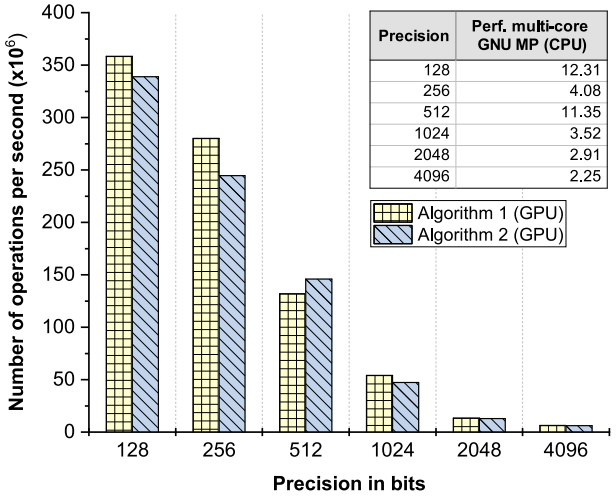


Fig. 5. Performance of Multiple-Precision Integer Addition Implementations with Non-Positive Inputs (Dataset-2).

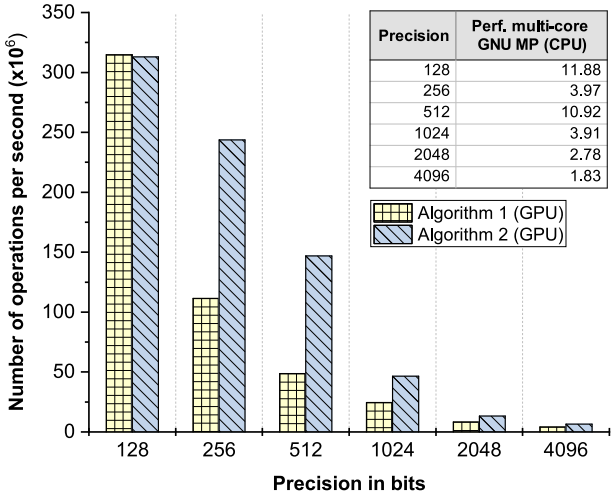


Fig. 6. Performance of Multiple-Precision Integer Addition Implementations with Mixed Positive and Negative Inputs (Dataset-3).

1 compared to Dataset-1 and Dataset-2. With 512-bit precision, the performance of Algorithm 1 is reduced by almost $3\times$ compared to Dataset-1. In turn, the performance of Algorithm 2 reduced by at most a factor of 1.2 compared to Dataset-1. The net result is that when the operands have different signs, Algorithm 2 outperforms Algorithm 1 by up to $3\times$.

A limitation of the proposed CUDA implementations is that the execution time grows linearly with increasing the precision. This happens for the following reasons:

1) Each multiple-precision addition is performed as a single thread, that is, the digits of multiple-precision numbers are calculated sequentially.
2) As the precision increases, the stride between elements in the input arrays increases accordingly and the effective GPU memory bandwidth decreases.

It should be noted that it is possible to compute all the digits (residues) of multiple-precision significands in parallel across different RNS moduli without worrying about carry propagation. This parallel arithmetic property of the RNS is employed in [22] to implement GPU-accelerated multiple-precision linear algebra kernels. Furthermore, we note that if all the digits of a multiple-precision number are computed in parallel, then the structure-of-arrays (SoA) layout with a sequential addressing scheme will provide coalesced access to the global GPU memory. Implementing digit-parallel multiple-precision integer addition is a direction for future work.

VI. Conclusion

In this paper, we have considered two multiple-precision integer addition algorithms for graphics processing units. The algorithms are based on the representation of large integers in the residue number system.

The first algorithm uses conditional operators to check the signs of the operands. However, in this case, threads that concurrently compute different elements of a multiple-precision array take divergent execution paths, which leads to an increase in the total computation time. To overcome this disadvantage, the second algorithm uses the radix-complement representation of a negative number in the RNS.

Experiments have shown that when the signs of the operands are different, the second algorithm outperforms the first one by far. In turn, both algorithms running on an NVIDIA RTX 2080 Ti GPU have shown to be faster than the multi-core GNU MP implementation on an Intel Xeon 4100 processor.

The presented implementation is part of GRNS, a library for efficient computations in the residue number system using CUDA-enabled GPUs. In the future, we plan to implement digit-parallel versions of the multiple-precision integer operations to take full advantage of the internal RNS parallelism. Furthermore, we will focus on extending the GRNS functionality and implementing real-world multiple-precision applications using this library.

Acknowledgment

This research is supported by the Ministry of Science and Higher Education of the Russian Federation, grant id RFMEFI61319X0092.

References

[1] A. Omondi, *Cryptography Arithmetic*. Springer International Publishing, 2020.

[2] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *IEEE Transactions on Computers*, vol. 64, no. 3, pp. 698–706, 2015.

[3] R. Sehgal and V. Nehra, "Symbolic computation of mathematical transforms and its application: A MATLAB computational project-based approach," *IUP Journal of Electrical & Electronics Engineering*, vol. 8, no. 1, pp. 53–76, 2015.

[4] O. Bodroža-Pantić, H. Kwong, and M. Pantić, "Some new characterizations of Hamiltonian cycles in triangular grid graphs," *Discrete Applied Mathematics*, vol. 201, pp. 1–13, 2016.

[5] "The GNU multiple precision arithmetic library," 2020. [Online]. Available: https://gmplib.org/

[6] "NTL: A library for doing number theory," 2020. [Online]. Available: https://shoup.net/ntl/

[7] "FLINT: Fast library for number theory," 2020. [Online]. Available: http://www.flintlib.org/

[8] K. Zhao and X. Chu, "GPUMP: A multiple-precision integer library for GPUs," in *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology (CIT 2010)*, Bradford, UK, 2010, pp. 1164–1168.

[9] T. Ewart, A. Hehn, and M. Troyer, "VLI – a library for high precision integer and polynomial arithmetic," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 267–278.

[10] E. Ochoa-Jiménez, L. Rivera-Zamarripa, N. Cruz-Cortés, and F. Rodríguez-Henríquez, "Implementation of RSA signatures on GPU and CPU architectures," *IEEE Access*, vol. 8, pp. 9928–9941, 2020.

[11] N. Emmart and C. C. Weems, "High precision integer multiplication with a GPU using Strassen's algorithm with multiple FFT sizes," *Parallel Processing Letters*, vol. 21, no. 3, pp. 359–375, 2011.

[12] B.-C. Chang, B.-M. Goi, R. C.-W. Phan, and W.-K. Lee, "Multiplying very large integer in GPU with Pascal architecture," in *Proceedings of the 2018 IEEE Symposium on Computer Applications Industrial Electronics (ISCAIE)*, Penang, Malaysia, 2018, pp. 401–405.

[13] K. Rudnicki, T. P. Stefański, and W. Żebrowski, "Open-source coprocessor for integer multiple precision arithmetic," *Electronics*, vol. 9, no. 7, p. article no. 1141, 2020.

[14] A. Bocco, Y. Durand, and F. De Dinechin, "SMURF: Scalar multiple-precision unum Risc-V floating-point accelerator for scientific computing," in *Proceedings of the Conference for Next Generation Arithmetic 2019*. New York, NY, USA: ACM, 2019.

[15] M. J. Schulte and E. E. Swartzlander, "A family of variable-precision interval arithmetic processors," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 387–397, 2000.

[16] R. Brent and P. Zimmermann, *Modern Computer Arithmetic*. Cambridge: Cambridge University Press, 2010.

[17] P. V. Ananda Mohan, *Residue Number Systems: Theory and Applications*. Cham: Birkhäuser, 2016.

[18] A. Omondi and B. Premkumar, *Residue Number Systems: Theory and Implementation*. London, UK: Imperial College Press, 2007.

[19] K. Isupov, "Using floating-point intervals for non-modular computations in residue number system," *IEEE Access*, vol. 8, pp. 58 603–58 619, 2020.

[20] F. J. Taylor, "Residue arithmetic a tutorial with examples," *Computer*, vol. 17, no. 5, pp. 50–62, 1984.

[21] N. S. Szabo and R. I. Tanaka, *Residue Arithmetic and its Application to Computer Technology*. New York, USA: McGraw-Hill, 1967.

[22] K. Isupov, V. Knyazkov, and A. Kuvaev, "Design and implementation of multiple-precision BLAS level 1 functions for graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 140, pp. 25–36, 2020.

[23] R. Farber, *CUDA Application Design and Development*. Boston: Morgan Kaufmann, 2011.