computation

MDPI

Article

High-Performance Computation in Residue Number System Using Floating-Point Arithmetic

Konstantin Isupov

Department of Electronic Computing Machines, Vyatka State University, 610000 Kirov, Russia; ks_isupov@vyatsu.ru

Abstract: Residue number system (RNS) is known for its parallel arithmetic and has been used in recent decades in various important applications, from digital signal processing and deep neural networks to cryptography and high-precision computation. However, comparison, sign identification, overflow detection, and division are still hard to implement in RNS. For such operations, most of the methods proposed in the literature only support small dynamic ranges (up to several tens of bits), so they are only suitable for low-precision applications. We recently proposed a method that supports arbitrary moduli sets with cryptographically sized dynamic ranges, up to several thousands of bits. The practical interest of our method compared to existing methods is that it relies only on very fast standard floating-point operations, so it is suitable for multiple-precision applications and can be efficiently implemented on many general-purpose platforms that support IEEE 754 arithmetic. In this paper, we make further improvements to this method and demonstrate that it can successfully be applied to implement efficient data-parallel primitives operating in the RNS domain, namely finding the maximum element of an array of RNS numbers on graphics processing units. Our experimental results on an NVIDIA RTX 2080 GPU show that for random residues and a 128-moduli set with 2048-bit dynamic range, the proposed implementation reduces the running time by a factor of 39 and the memory consumption by a factor of 13 compared to an implementation based on mixed-radix conversion.

Keywords: residue number system; digital arithmetic; high-performance computing; data-parallel primitives; graphics processing units

Citation: Isupov, K. High-Performance Computation in Residue Number System Using Floating-Point Arithmetic. *Computation* **2021**, *9*, 9. https://doi.org/10.3390/computation9020009

Received: 19 November 2020
Accepted: 19 January 2021
Published: 21 January 2021

1. Introduction

The emergence of new highly parallel architectures has increased interest in fast, carry-free, and energy-efficient computer arithmetic techniques. One such technique is the residue number system (RNS), which has received a lot of attention over recent years [1–3]. The RNS is of interest to scientists dealing with computationally intensive applications as it provides efficient highly parallelizable arithmetic operations. This number coding system is defined in terms of pairwise coprime integers called moduli, and a large weighted number is converted into several smaller numbers called residues, which are obtained as the remainders when the given number is divided by the moduli. A useful feature is that the residues are mutually independent, and for addition, subtraction and multiplication, instead of big word length (multiple-precision) operations, we can perform several small word length operations on these residues without carry propagation between them [1].

The RNS has been used in several applications, namely homomorphic encryption [4,5], cloud computing [6], stochastic computing [7], motion estimation [8], energy-efficient digital signal processing [9], high-precision linear algebra [10], blockchain [11], pseudo-random number generation [12], and deep neural networks [13]. Interest in RNS is currently growing due to the widespread adoption of massively parallel computing platforms such as graphics processing units (GPUs).

However, some operations are still difficult to implement in RNS, such as magnitude comparison, sign estimation, scaling, and division. Various methods have been

proposed in the literature to overcome this problem and perform the above operations in an efficient manner. Many of the existing methods are designed for special moduli sets like $\{2^n + 1, 2^n, 2^n - 1\}$ [14], $\{2^{n+k}, 2^n - 1, 2^n + 1, 2^{n\pm1} - 1\}$ [15], $\{2^k, 2^p - 1, 2^p + 1\}$ and $\{2^k, 2^p - 1, 2^{p-1} - 1\}$ [16]. On the other hand, there are methods for arbitrary moduli sets [17–20].

In a recent paper [21], we have presented a method for implementing difficult RNS operations via computing a finite precision floating-point interval that localizes the fractional value of an RNS representation. Such an interval is called a floating-point interval evaluation, or simply an interval evaluation. The method deserves attention for three reasons. First, it is intended for arbitrary moduli sets with large dynamic ranges significantly exceeding the usual word length of computers. Dynamic ranges consisting of hundreds and even thousands of bits are in demand in many modern applications, primarily in cryptography and high-precision arithmetic. Second, the method leads to efficient software implementations using general-purpose computing platforms, since it only requires very fast standard (finite precision) floating-point operations, and most computations can be performed in a parallel manner. Third, it is a fairly versatile method suitable for computing a wide range of fundamental operations that are problematic in RNS, including

- magnitude comparison;
- sign identification;
- dynamic range overflow detection;
- general division and scaling.

A key component of this method is an accurate algorithm that computes the floating-point interval evaluation for a number in RNS representation; see Algorithm 1 in [21]. In order to obtain the result with the desired accuracy using only finite precision operations, this algorithm performs the iterative procedure, which in some cases is the most expensive part of the algorithm.

In this paper, we continue our research on the application of finite precision floating-point intervals to implement high-performance RNS algorithms. The contribution of this paper is four-fold:

1. In Section 3, we provide proofs of some important properties of the interval evaluation algorithm that were not included in the previous article [21].
2. In Section 4, we present an improved version of the interval evaluation algorithm that reduces the number of iterations required to achieve the desired accuracy.
3. In Section 5, we demonstrate that our method can successfully be applied to implement efficient data-parallel primitives in the RNS arithmetic, namely we use it to find the maximum element of an array of RNS numbers on GPUs.
4. In Section 6, we present new numerical results showing the performance of the method on various moduli sets, from a moderately small 4-moduli set with 64-bit dynamic range to a huge 256-moduli set with 4096-bit dynamic range.

2. Background

2.1. Residue Number System

We fix a set of n positive relatively prime integers $\{m_1, m_2, \ldots, m_n\}$ called the moduli set. The number m_i from this set is called a modulus, and the inequality $m_i > 1$ should hold for all $i \in \{1, 2, \ldots, n\}$. We define M as the product of all the m_i's. Throughout this paper, we assume that operations modulo m_i are inexpensive, in contrast to modulo M operations, which can take a long time. Consider an integer X. For each modulus in $\{m_1, m_2, \ldots, m_n\}$, we have $x_i = X \bmod m_i$, i.e., x_i is the smallest non-negative remainder of X modulo m_i; this is also written $|X|_{m_i}$, and x_i is called the ith residue of X. Thus, an integer X in the RNS is represented by the n-tuple of residues

$$X = (x_1, x_2, \ldots, x_n).$$

It has been proved that if $0 \leq X < M$, then the number X is one-to-one corresponding to the RNS representation [22]. An interesting and useful feature of RNS is that addition, subtraction and multiplication are computed in an element-wise fashion. If X, Y, and Z have RNS representations given by $(x_1, x_2, \ldots, x_n)$, $(y_1, y_2, \ldots, y_n)$, and $(z_1, z_2, \ldots, z_n)$, then for $\circ \in \{+, -, \times\}$ we have

$$Z = (|x_1 \circ y_1|_{m_1}, |x_2 \circ y_2|_{m_2}, \ldots, |x_n \circ y_n|_{m_n}).$$

That is, the ith RNS digit, namely z_i, is defined in terms of $|x_i \circ y_i|_{m_i}$ only, and no carry information need be communicated between residue digits [23]. As a result, we have very high speed concurrent (parallel) operations, which makes the RNS attractive for modern high-performance applications.

We can convert an RNS representation back to its integer form using the following well-known formula:

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

where $M_i = M/m_i$ and w_i is the modulo m_i multiplicative inverse of M_i. Both M_i and w_i are the RNS constants that follow from the Chinese Remainder Theorem (CRT) [2].

2.2. Implementing Difficult RNS Operations Using Finite Precision Floating-Point Intervals

Here we give a brief overview of a method for implementing difficult *non-modular* RNS operations (e.g., comparison, sign determination, overflow detection, and division) using limited precision floating-point arithmetic [21]. A straightforward approach is based on the CRT Formula (1). However, the sum modulo M causes a major implementation problem because M is generally a very large and arbitrary integer [24]. When M consists of hundreds or thousands of bits, the CRT-based method is very slow and inefficient. Instead, the paper [21] uses a fractional version of the CRT, which operates with a fractional representation of an RNS number [25].

Let an integer X be represented in the RNS by the residues $(x_1, x_2, \ldots, x_n)$. The fractional representation of X is calculated as follows:

$$\frac{X}{M} = \left| \sum_{i=1}^{n} \frac{|x_i w_i|_{m_i}}{m_i} \right|_1, \tag{2}$$

where the notation $|\,|_1$ denotes that the integer part of the expression is discarded and only the fractional part is retained. Unlike (1), the fractional version of the CRT defined by formula (2) does not require the time consuming modulo M operation.

However, X/M is actually a rational number and difficulties arise when we attempt to compute X/M using limited precision arithmetic. To address these difficulties without increasing the precision, ref. [21] suggests evaluating X/M using floating-point interval arithmetic.

Definition 1 ([21]). *Let $X = (x_1, x_2, \ldots, x_n)$ be an RNS number in the range from 0 to $M-1$ and let X/M be the exact fractional representation of X. The floating-point interval evaluation of X, denoted by $I(X/M) = [\underline{X/M}, \overline{X/M}]$, is an interval defined by its lower and upper bounds $\underline{X/M}$ and $\overline{X/M}$ that are finite precision floating-point numbers satisfying $\underline{X/M} \leq X/M \leq \overline{X/M}$.*

Thus, $I(X/M)$ provides information about the range of possible values for the exact fractional representation of an RNS number. This information may not be sufficient to restore the binary form of the number, but it can be efficiently used to perform other hard RNS operations such as magnitude comparison, sign detection, scaling, and division.

2.3. Highly Accurate Computation of $I(X/M)$

Consider an RNS representation $X = (x_1, x_2, \ldots, x_n)$, which can take any value in the range 0 to $M-1$. To compute $I(X/M)$ such that $\delta I(X/M) < \varepsilon$, where ε is a given accuracy parameter and $\delta I(X/M)$ denotes the difference between the upper and lower bounds of $I(X/M)$ divided by X/M, we can use Algorithm 1 of [21]. The general structure of this algorithm is shown in Figure 1.

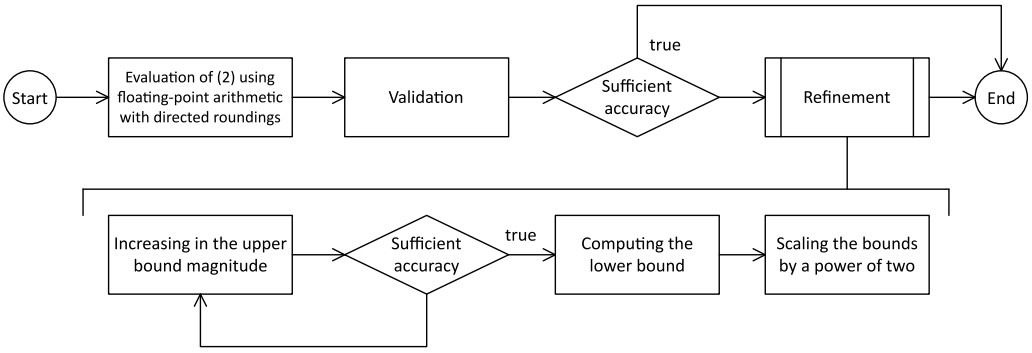


Figure 1. Accurate computation of the floating-point interval evaluation for an RNS representation using finite precision arithmetic. For a complete description and proof of the correctness of the algorithm, see [21].

The algorithm consists of the following main stages:

1. *Initial computation.* In this stage, evaluation of (2) using floating-point interval arithmetic is performed. We first calculate $u_i = |x_i w_i|_{m_i}$ for all $i \in \{1, 2, \ldots, n\}$. Then, $S_L(X)$ and $S_U(X)$ are computed by evaluating $\sum_{i=1}^{n} u_i/m_i$ in finite precision arithmetic with downwardly directed rounding and upwardly directed rounding, respectively. The algorithm ends if both $S_L(X)$ and $S_U(X)$ are equal to zero, since in this case X is also zero; otherwise (at least one of $S_L(X)$ and $S_U(X)$ is not zero) the initial values of $\underline{X/M}$ and $\overline{X/M}$ are obtained by discarding the integer parts and keeping only the fractional parts in $S_L(X)$ and $S_U(X)$.
2. *Validation.* In this stage, the algorithm checks the condition $\lfloor S_L(X) \rfloor = \lfloor S_U(X) \rfloor$. If it is true, then $\underline{X/M}$ and $\overline{X/M}$ are calculated correctly. Otherwise (a rare case), one of the bounds is adjusted by calling the mixed-radix conversion (MRC) procedure.
3. *Checking the accuracy.* The algorithm operates with the predefined constant

$$\psi = 4\mathbf{u}n \log_2 n(1 + \varepsilon/2)/\varepsilon,$$

where $\mathbf{u} = 2^{1-p}$ assuming p-bit floating-point arithmetic ($p = 24$ and 53 for binary32 and binary64, respectively). The value of ψ may seem strange at first glance; however, it has been proved in [21] that if $\overline{X/M}$ is greater than or equal to ψ, then $\delta I(X/M) < \varepsilon$. If this is the case, then the algorithm ends; otherwise, the refinement stage is performed. One should ensure that $\psi \leq 1/4$ (note that this condition is very weak and is satisfied for virtually all values of n and ε).
4. *Refinement.* The ultimate goal of this stage is to determine K such that $b_U = \overline{2^K X/M} \geq \psi$. We set $K \leftarrow 0$ and do the following iterative computation:

$$\begin{aligned} u_i &\leftarrow \left| u_i \cdot |2^k|_{m_i} \right|_{m_i} \quad \text{for all } i \in \{1, 2, \ldots, n\}, \\ b_U &\leftarrow \left| \mathrm{fl}_{\triangle}\left(\textstyle\sum_{i=1}^{n} u_i/m_i\right) \right|_1, \\ K &\leftarrow K + k, \end{aligned} \tag{3}$$

where $k = \lfloor \log_2(1/2\psi) \rfloor$ and the initial values of the u_i's are computed earlier (at the first stage of the algorithm). All the $|2^k|_{m_i}$'s are constants that can be preprocessed. The symbol $\mathrm{fl}_{\triangle}(\cdot)$ denotes the result of a finite precision floating-point computation

performed with upwardly directed rounding. The computation of (3) is repeated until $b_U \geq \psi$. Once the iterations are finished, the following is computed:

$$b_L \leftarrow \left|\mathrm{fl}_{\triangledown}\left(\sum_{i=1}^{n} u_i/m_i\right)\right|_1, \tag{4}$$

where the u_i's are those computed in the iterative procedure and $\mathrm{fl}_{\triangledown}(\cdot)$ denotes the result of a finite precision floating-point computation performed with downwardly directed rounding. Finally, we compute the endpoints of $I(X/M)$ as follows:

$$\begin{aligned} \underline{X/M} &\leftarrow b_L/2^K, \\ \overline{X/M} &\leftarrow b_U/2^K. \end{aligned} \tag{5}$$

We note that in the described algorithm, pairwise summation is used to compute the sum of a set of floating-point numbers in finite precision arithmetic.

3. Properties of the Interval Evaluation Algorithm

The following theorem gives the maximum number of iterations in the refinement stage of the described algorithm.

Theorem 1. *The refinement stage of the described algorithm for computing* $I(X/M) = [\underline{X/M}, \overline{X/M}]$ *is performed in no more than* $\lceil (\log_2 \psi M)/k \rceil$ *iterations.*

Proof. The iterative procedure ends at the jth iteration if $b_U \geq \psi$. On the other hand, when applying upwardly directed rounding, b_U cannot be less than $2^{jk} X/M$, provided that $\lfloor \sum_{i=1}^{n} u_i/m_i \rfloor = \lfloor \mathrm{fl}_{\triangle}\left(\sum_{i=1}^{n} u_i/m_i\right) \rfloor$. Thus, the number of iterations depends on the magnitude of X: the smaller the magnitude, the more iterations are needed. For $X = 1$, the loop termination condition is satisfied when $2^{jk}/M \geq \psi$. The result follows by expressing j in terms of M, ψ, and k and requiring it to be an integer. □

One way to reduce the number of iterations is to increase the refinement factor k. However, the following theorem says that this can lead to an incorrect result.

Theorem 2. *To guarantee the correct result of calculating* $I(X/M) = [\underline{X/M}, \overline{X/M}]$, *the refinement factor k must not be greater than* $\lfloor \log_2(1/2\psi) \rfloor$.

Proof. The proof is based on two remarks:

1. As shown in [21], if X is too close to M, namely if $1 - X/M \leq 2\mathbf{u}n \log_2 n$ then $\overline{X/M}$ may not be computed correctly. Consequently, X should be less than $M(1 - 2\mathbf{u}n \log_2 n)$ if we want to calculate the correct value of $\overline{X/M}$.
2. Calculation of (3) at the $(i-1)$th refining iteration can be considered the same as calculating (2) with rounding upwards for some number X_{i-1}. Accordingly, calculating (3) at the ith iteration can be considered the same as calculating (2) with rounding upwards for the number $X_i = 2^k X_{i-1}$.

We denote the result of the $(i-1)$th iteration by $b_U^{(i-1)}$, and the result of the ith iteration by $b_U^{(i)}$. The ith iteration is performed when $b_U^{(i-1)} < \psi$, whence it follows that

$$X_{i-1}/M < \psi \Rightarrow X_{i-1} < \psi M \Rightarrow X_i < 2^k \psi M.$$

First, assume $k = \lfloor \log_2(1/2\psi) \rfloor$. In this case, X_i is less than $2^{\lfloor \log_2(1/2\psi) \rfloor} \psi M$, and since $2^{\lfloor \log_2(1/2\psi) \rfloor} \leq 1/2\psi$ then X_i is less than $M/2$. Thus, if $\mathbf{u}n \log_2 n \leq 1/4$, then $b_U^{(i)}$ will be computed correctly (see the first remark above).

Now consider the case when $k = \lfloor \log_2(1/2\psi) \rfloor + 1$. For this setting, we have $X_i < 2 \cdot 2^{\lfloor \log_2(1/2\psi) \rfloor} \psi M$, so we can only guarantee that X_i is less than M, but not that it is less than $M(1 - 2\mathbf{u}n \log_2 n)$. Thus, $b_U^{(i)}$ may not be computed correctly. □

4. Proposed Improvement

4.1. Description

We propose an improvement to the algorithm of [21] described above by modifying each iteration of the refinement loop as follows:

$$\begin{aligned} r &\leftarrow \max\{-(\lceil \log_2 b_U \rceil + 1),\ k\}, \\ u_i &\leftarrow \left| u_i \cdot |2^r|_{m_i} \right|_{m_i} \quad \text{for all } i \in \{1, 2, \ldots, n\}, \\ b_U &\leftarrow \left| \mathrm{fl}_{\triangle}\left(\textstyle\sum_{i=1}^{n} u_i / m_i\right) \right|_1, \\ K &\leftarrow K + r. \end{aligned} \tag{6}$$

Thus, we make the refinement factor r dependent on the value of b_U, but not less than $k = \lfloor \log_2(1/2\psi) \rfloor$, and before starting the iterations, we assign the value of $\overline{X/M}$ computed at the first stage of the algorithm to b_U. Once the iterations are finished, b_L is computed according to (4), and the desired endpoints of $I(X/M)$ are obtained according to (5). The proposed modification improves the performance of the algorithm by reducing the number of iterations required to achieve the desired accuracy.

Summarizing, we present an improved accurate algorithm for computing the floating-point interval evaluation of an RNS number in Algorithm 1. The algorithm takes as input an integer $X \in [0, M-1]$ represented by the residues $(x_1, x_2, \ldots, x_n)$ relative to the moduli set $\{m_1, m_2, \ldots, m_n\}$ and produces as output an interval $I(X/M) = [\underline{X/M}, \overline{X/M}]$ such that $\underline{X/M} \le X/M \le \overline{X/M}$ and $\delta I(X/M) < \varepsilon$, where ε is a given accuracy parameter and $\delta I(X/M) = (\overline{X/M} - \underline{X/M}) \ / \ X/M$.

Algorithm 1 Computing the floating-point interval evaluation for an RNS number

1: $u_i = \left| x_i w_i \right|_{m_i}$ for all $i \in \{1, 2, \ldots, n\}$
2: $S_L(X) \leftarrow \mathrm{fl}_{\triangledown}\left(\sum_{i=1}^{n} u_i / m_i\right)$
3: $S_U(X) \leftarrow \mathrm{fl}_{\triangle}\left(\sum_{i=1}^{n} u_i / m_i\right)$
4: **if** $S_L(X) = S_U(X) = 0$ **then**
5: **return** $\underline{X/M} \leftarrow 0$ and $\overline{X/M} \leftarrow 0$
6: **end if**
7: $\underline{X/M} \leftarrow |S_L(X)|_1$
8: $\overline{X/M} \leftarrow |S_U(X)|_1$
9: **if** $\lfloor S_L(X) \rfloor \neq \lfloor S_U(X) \rfloor$ **then**
10: Compute the mixed-radix representation of X and test the most significant mixed-radix digit, $\bar{x}_n$: if $\bar{x}_n \neq 0$, then set $\overline{X/M} \leftarrow \overline{\frac{M-1}{M}}$; otherwise set $\underline{X/M} \leftarrow \underline{1/M}$
11: **end if**
12: **if** $\overline{X/M} \ge \psi$ **then**
13: **return** $\underline{X/M}$ and $\overline{X/M}$
14: **end if**
15: $b_U \leftarrow \overline{X/M}$
16: $K \leftarrow 0$
17: **while** $b_U < \psi$ **do**
18: $r \leftarrow \max\{-(\lceil \log_2 b_U \rceil + 1),\ k\}$
19: $u_i \leftarrow \left| u_i \cdot |2^r|_{m_i} \right|_{m_i}$ for all $i \in \{1, 2, \ldots, n\}$
20: $b_U \leftarrow \left| \mathrm{fl}_{\triangle}\left(\sum_{i=1}^{n} u_i / m_i\right) \right|_1$
21: $K \leftarrow K + r$
22: **end while**
23: $b_L \leftarrow \left| \mathrm{fl}_{\triangledown}\left(\sum_{i=1}^{n} u_i / m_i\right) \right|_1$
24: $\underline{X/M} \leftarrow b_L / 2^K$
25: $\overline{X/M} \leftarrow b_U / 2^K$
26: **return** $\underline{X/M}$ and $\overline{X/M}$

The correctness statement of the original algorithm has been proved in [21], so we only must prove that the proposed modification does not violate the correctness of the algorithm. The following theorem establishes this fact.

Theorem 3. *The proposed modification does not violate the correctness of the algorithm for computing the floating-point interval evaluation.*

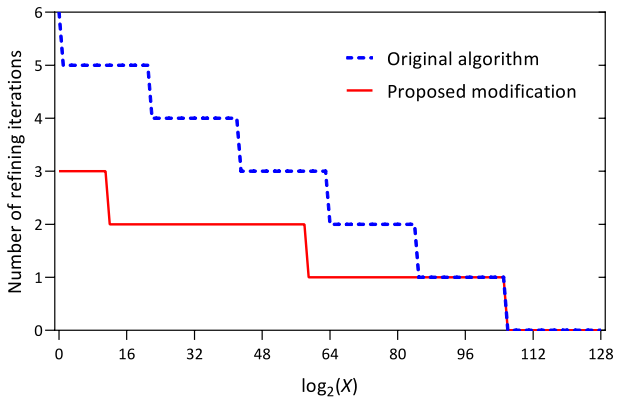
Proof. Denote again the result of the $(i-1)$th iteration by $b_U^{(i-1)}$, and the result of the ith iteration by $b_U^{(i)}$. Assuming that $b_U^{(i-1)}$ is calculated properly, it is only necessary to prove that $b_U^{(i)}$ is also calculated properly. The proof is as follows. Calculation of (6) at the ith iteration can be considered the same as calculating (2) with upwardly directed rounding for the input $X_i = 2^r X_{i-1}$. Please note that X_{i-1}/M cannot exceed $b_U^{(i-1)}$. According to the first remark from the proof of Theorem 2 above, X_i should be less than $M(1 - 2\mathbf{u}n\log_2 n)$. Assuming $r = -(\lceil \log_2 b_U^{(i-1)} \rceil + 1)$ (we are not interested in the case $r = k$), 2^r cannot exceed $2^{-(\log_2 b_U^{(i-1)}+1)}$ and thus X_i cannot exceed $2^{-(\log_2 b_U^{(i-1)}+1)} X_{i-1}$. Simplifying, we have $X_i \leq X_{i-1}/2b_U^{(i-1)}$. Finally, since $1/b_U^{(i-1)} \leq M/X_{i-1}$, then $X_i \leq M/2$, so if $\mathbf{u}n\log_2 n \leq 1/4$, then it is guaranteed that $b_U^{(i)}$ is computed properly. □

To implement the proposed improved algorithm, all the $|2^r|_{m_i}$'s should be preprocessed and stored in a lookup table of size no more than n by $\lfloor \log_2 M \rfloor$. We note that this memory overhead is not actually significant. In fact, let each RNS modulus consists of 32 bits and $n = 512$. Then M has a bit size of about 16 thousand bits (huge dynamic range), and the total size of the lookup table is 32 MB. Moreover, the table of powers of two is in demand in various RNS applications.

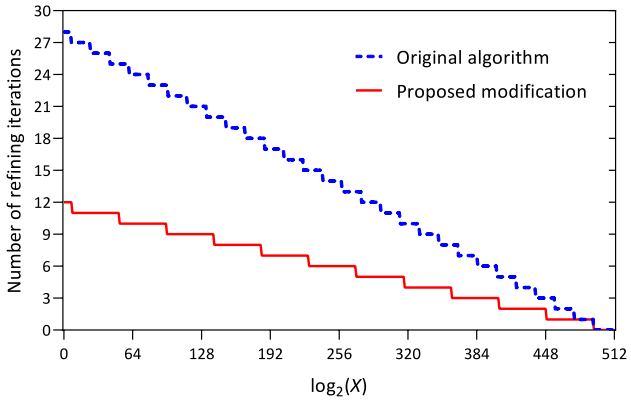
4.2. Demonstration

To demonstrate the benefits of the proposed modification, we have counted the number of iterations required to compute the interval evaluation in residue number systems with four different moduli sets. The first set consists of 8 moduli and provides a 128-bit dynamic range. The second set consists of 32 moduli and provides a 512-bit dynamic range. The third set consists of 64 moduli and provides a 1024-bit dynamic range. The fourth set consists of 256 moduli and provides a 4096-bit dynamic range.

The results are reported in Figures 2 and 3, where the algorithm from [21] is labeled as "Original algorithm" and the modified algorithm (Algorithm 1) as "Proposed modification". The inputs are represented by powers of two from 2^0 to $2^{\lfloor \log_2 M \rfloor}$, and the x-axis on the plots denotes the binary logarithm of the RNS number for which the interval evaluation is computed. The y-axis denotes the number of refining iterations required to compute the interval evaluation with accuracy $\varepsilon = 10^{-7}$.

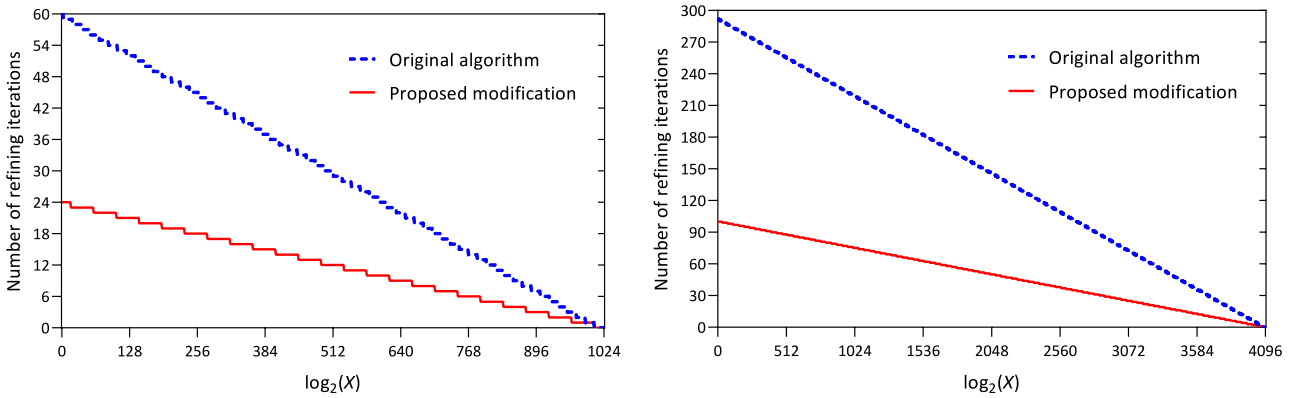


(**a**) 8-moduli set (128 bits)



(**b**) 32-moduli set (512 bits)

Figure 2. Number of iterations required to compute the interval evaluation of an RNS number with accuracy $\varepsilon = 10^{-7}$ in 8- and 32-moduli sets using double precision floating-point arithmetic.



(**a**) 64-moduli set (1024 bits) (**b**) 256-moduli set (4096 bits)

Figure 3. Number of iterations required to compute the interval evaluation of an RNS number with accuracy $\varepsilon = 10^{-7}$ in 64- and 256-moduli sets using double precision floating-point arithmetic.

In this demonstration, all calculations were done in standard floating-point arithmetic (double precision). The plots show that for small inputs, the proposed modification reduces the number of iterations by almost 3 times, and increasing the size of the moduli set increases the advantage.

5. Application: Finding the Maximum Element of an Array of RNS Numbers

Reduction is a widely used data-parallel primitive in high-performance computing and is part of many important algorithms such as least squares and MapReduce [26]. For an array of N elements $\{X_1, X_2, \ldots, X_N\}$, applying the reduction operator $\oplus$ gives a single value $X^* = X_1 \oplus X_2 \oplus \cdots \oplus X_N$. The reduction operator is a binary associative (and often commutative) operator such as $+$, $\times$, MIN, MAX, logical AND, logical OR. In this section, we applied the considered interval evaluation method to implement one operation of the parallel reduction primitive over an array of RNS numbers, namely MAX, which consists of finding the maximum element in the array. Our implementation is intended for GPUs supporting the Compute Unified Device Architecture (CUDA) [27].

5.1. Approach

Let $\{X_1, X_2, \ldots, X_N\}$ be an array of N integers in RNS representation and we want to find the largest element of this array, $X^* = \max\{X_1, X_2, \ldots, X_N\}$. To simplify our presentation, we are considering unsigned numbers, i.e., each X_i varies in the range of 0 to $M - 1$. Our approach to finding X^* is illustrated in Figure 4. The computation is decomposed into two stages:

1. For a given array $\{X_1, X_2, \ldots, X_N\}$, the array $\{I(X_1/M), I(X_2/M), \ldots, I(X_N/M)\}$ is calculated. Each interval evaluation is coupled with the corresponding index of the RNS representation in the input array (*idx* in Figure 4), so knowing the interval evaluation, we can always fetch the original RNS representation.
2. The reduction tree is built over the array of interval evaluations to obtain the maximum RNS representation. The basic brick of the reduction is comparing the magnitude of two numbers in the RNS. For two given numbers $X = (x_1, x_2, \ldots, x_n)$ and $Y = (y_1, y_2, \ldots, y_n)$, the magnitude comparison is performed as follows [21]:
   - if $\underline{X/M} > \overline{Y/M}$ then $X > Y$;
   - if $\overline{X/M} < \underline{Y/M}$ then $X < Y$;
   - if $x_i = y_i$ for all $i \in \{1, 2, \ldots, n\}$, then $X = Y$;
   - if neither case is true, then the mixed-radix representations of X and Y are calculated and compared component-wise to produce the final result.

Thus, after the array of interval evaluations is computed, each RNS comparison is performed very quickly, namely in $O(1)$ time, and the input RNS array is accessed only in corner cases, when the numbers being compared are equal or nearly equal to each other and the accuracy of their interval evaluations is insufficient. On the other hand, the presented approach also reduces the memory footprint of intermediate computations, since not RNS numbers are stored on the reduction layers, but only their interval evaluations, and each interval evaluation has a fixed size, regardless of the size of the moduli set and dynamic range.

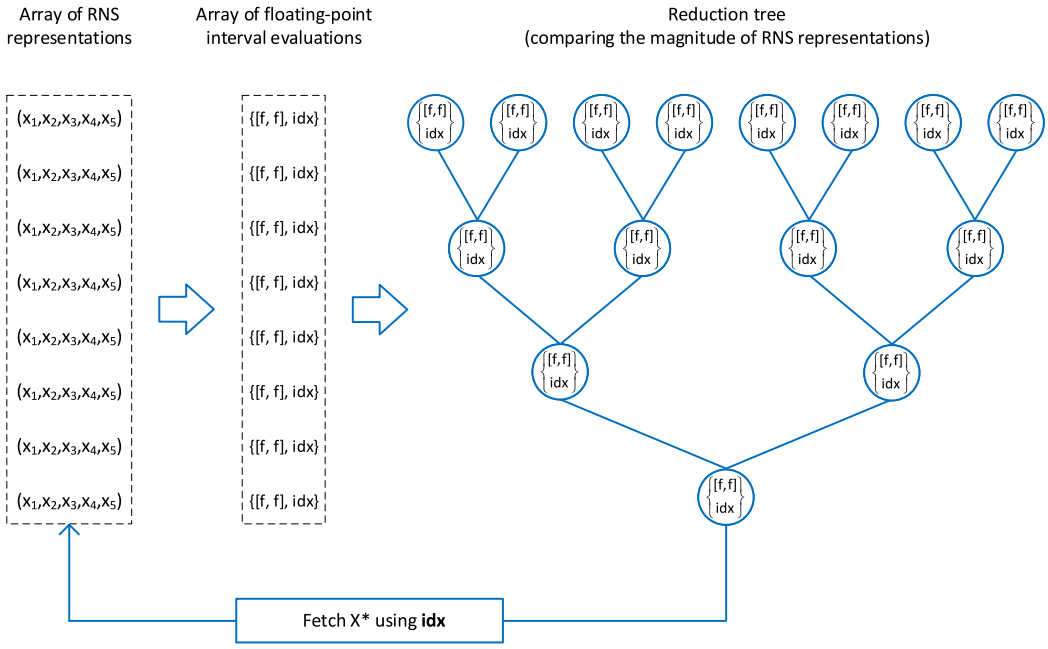


Figure 4. Calculating the maximum element of an array of RNS numbers using floating-point interval evaluations. Each interval evaluation is denoted as $[f, f]$, and idx is the index of the corresponding RNS number in the input array.

5.2. CUDA Implementation

In CUDA, programs that run on the GPU are called kernels. Each kernel launches a grid of parallel threads, and within the grid, GPU threads are grouped into thread blocks. Threads belonging to the same block can communicate through shared memory or shuffle instructions, while global communication between thread blocks is done by sequential kernel launches or using device memory atomic operations.

In what follows, we denote by *Arr* the pointer to the global memory for accessing the input RNS array and by *Eval* the pointer to the global memory for accessing the array of floating-point interval evaluations. Each element of the *Eval* array is an instance of a structure that consists of three fields:

- *low* is the lower bound of the interval evaluation;
- *upp* is the upper bound of the interval evaluation;
- *idx* is the index of the corresponding RNS representation in the *Arr* array.

We also use the following notations:

- N is the size of the input array;
- *gSize* is the number of thread blocks per grid;
- *bSize* is the number threads per block;
- *bx* is the block index within the grid;
- *tx* is the thread index within the block.

For an input array of RNS numbers, the calculation of the array of floating-point interval evaluations is implemented as a separate kernel. One thread computes one interval

evaluation, and many threads run concurrently, storing the results in a pre-allocated global memory buffer of the GPU. The pseudocode of the kernel (the code executed by each thread) is shown in Algorithm 2, where the device function *ComputeEval* is a sequential computation of an interval evaluation as described in Algorithm 1.

Algorithm 2 Computing an array of floating-point interval evaluations

$i \leftarrow bSize \times bx + tx$
while $i < N$ **do**
    $(Eval[i].low, Eval[i].upp) \leftarrow ComputeEval(Arr[i])$
    $Eval[i].idx \leftarrow i$
    $i \leftarrow i + gSize \times bSize$
end while

Remark 1. *We have also implemented a parallel algorithm in which n threads simultaneously compute an interval evaluation for a number represented in an n-moduli RNS, and the ith thread is assigned for modulo m_i computation. However, for our purpose (calculating an array of interval evaluations), the sequential algorithm is preferable, since it does not require communication between parallel threads. The parallel version can be used in applications that have insufficient internal parallelism to saturate the GPU.*

Once the array of interval evaluations is computed, the reduction kernel is launched, which generates and stores partial reduction results using multiple (*gSize*) thread blocks. To avoid multiple global memory accesses, fast shared memory cache is employed at the internal reduction levels. The same reduction kernel then runs again to reduce the partial results into a single result using a single thread block. The result is an interval evaluation that represent the maximum element in the input array, and the desired RNS number is retrieved from the array using the index associated with that interval evaluation.

The pseudocode of the reduction kernel is shown in Algorithm 3. In this algorithm, *S* is the size of the reduction operation, *Sh* is an array in the shared memory of each thread block, and *Pow* is the next highest power of 2 of *bSize*. Just like the *Eval* array, each element of the *Sh* array is a structure consisting of three fields, *low*, *upp* and *idx*.

Algorithm 3 Reduction of an array of floating-point interval evaluations

$i \leftarrow bSize \times bx + tx$
$Sh[tx].idx \leftarrow -1$
while $i < S$ **do**
    if $RnsCmp(Eval[i], Sh[tx], Arr) = 1$ **then**
        $Sh[tx] \leftarrow Eval[i]$
    end if
    $i \leftarrow i + gSize \times bSize$
end while
— Intra-block synchronization —
$i \leftarrow Pow/2$
while $i \geq 1$ **do**
    if $tx < i$ **and** $tx + i < bSize$ **and** $RnsCmp(Sh[tx + i], Sh[tx], Arr) = 1$ **then**
        $Sh[tx] \leftarrow Sh[tx + i]$
    end if
    $i \leftarrow i/2$
    — Intra-block synchronization —
end while
if $tx = 0$ **then**
    $Eval[bx] \leftarrow Sh[tx]$
end if

Note that in Algorithm 3, $S = N$ for the first kernel invocation and $S = gSize$ for the second one. The final result of the two kernel launches is stored in the first element of the *Eval* array, i.e., $Eval[0]$ assuming zero-based indexing. We give the pseudocode of the *RnsCmp* function in Algorithm 4.

Algorithm 4 $RnsCmp(A, B, Arr)$

if $B.idx < 0$ **or** $A.low > B.upp$ **then**
    return 1
else if $A.idx < 0$ **or** $A.upp < B.low$ **then**
    return -1
else if $Arr[A.idx]$ and $Arr[B.idx]$ are equal component-wise **then**
    return 0
else
    Compare $Arr[A.idx]$ and $Arr[B.idx]$ using mixed-radix conversion
end if

Although we have only considered the MAX operation in this paper, other reduction operations can be implemented in a quite similar manner. We also note that our approach can be straightforwardly extended to find the maximum element in an array of signed RNS numbers.

6. Results and Discussion

We present several performance results of different approaches to finding the maximum element in an array of RNS numbers on the GPU:

- *Proposed approach* is an implementation of the MAX operation as described in Section 5 using floating-point interval evaluations to compare the magnitude of RNS numbers.
- *Naive approach* is a straightforward parallel reduction using floating-point interval evaluations that consists of two kernel invocations. In contrast to the proposed variant, the naive one does not have a kernel that computes an array of interval evaluations. Instead, the computation of two interval evaluations is performed each time two RNS numbers are compared. This reduces the memory footprint but leads to more computation load.
- *Mixed-radix approach* is an implementation of the MAX operation as described in Section 5, but using the MRC procedure instead of floating-point interval evaluations to compare the magnitude of RNS numbers. We used the Szabo and Tanaka MRC algorithm [3] for this implementation.

All tests were carried out on a system running Ubuntu 20.04.1 LTS, equipped with an NVIDIA GeForce RTX 2080 video card (see Table 1), an Intel Core i5 7500 CPU, and 16 GB of DDR4 memory. We used CUDA Toolkit version 11.1.105 and NVIDIA Driver version 455.32.00. The source code was compiled with the -O3 option.

Table 1. Overview of the GPU hardware used in the experiments. The column labeled "CC" indicates the compute capability version of the GPU.

Hardware	# Cores	Clock Speed	Memory Size/Type	Bandwidth	CC
NVIDIA RTX 2080	2944	1515 MHz	8 GB/GDDR6	448.0 GB/s	7.5

We carried out the performance tests using 7 different sets of RNS moduli that provide dynamic ranges from 64 to 4096 bits. An overview of the moduli sets is given in Table 2, and the tool used to generate these sets is available on the GitHub repository: https://github.com/kisupov/rns-moduli-generator.

Table 2. Overview of the moduli sets used in the experiments.

Size of Set, n	Dynamic Range, M (Approx.)	Bit-Length	$m_1 \ldots m_n$
4	$1.891730206351222500900 \times 10^{19}$	64	65,947 … 65,953
8	$3.486474761596273374449 \times 10^{38}$	128	65,725 … 65,749
16	$1.182869237276559892956 \times 10^{77}$	256	65,599 … 65,657
32	$1.381750867498453484869 \times 10^{154}$	512	65,533 … 65,683
64	$1.834972082650114435387 \times 10^{308}$	1024	65,379 … 65,771
128	$3.267493893788783073405 \times 10^{616}$	2048	65,139 … 66,071
256	$1.113716837551166769174 \times 10^{1233}$	4096	64,491 … 66,889

The measurements were performed with arrays of fixed length $N = 5{,}000{,}000$, and random residues were used for the data generation, i.e., in the case of an n-moduli RNS, the input dataset (an array of N RNS numbers) was obtained from a random integer array of size $N \times n$. The GNU MP library was used to validate the results.

We report the performance of the tested CUDA implementations in Table 3. We clearly see that the proposed approach significantly outperforms the naive and mixed-radix variants for all test cases. The MRC method has a quadratic behavior, while the proposed and naive approaches based on floating-point interval evaluations have a linear behavior. In fact, the Szabo and Tanaka algorithm requires $n(n-1)$ operations modulo m_i to compute the mixed-radix representation in an n-moduli RNS, while sequential (single-threaded) computation of the interval evaluation using Algorithm 1 requires only n operations modulo m_i and $4n$ standard floating-point operations, assuming no ambiguity resolution or refinement iterations are required. The mixed-radix implementation has not been evaluated for the 256-moduli set due to excessive memory consumption.

Table 3. Running times (in milliseconds) of the different approaches for calculating the maximum element of an array of RNS numbers on an NVIDIA RTX 2080.

Size of Set, n	Proposed	Naive	Mixed-Radix
4	1.964	2.851	1.517
8	2.395	4.654	9.737
16	4.126	8.377	63.033
32	7.579	16.004	262.495
64	11.262	24.162	1017.980
128	107.726	268.151	4223.760
256	227.690	749.596	N/A

Although in some cases Algorithm 1 calls the MRC procedure, these cases are very rare when the numbers are randomly distributed. Moreover, if it is known in advance that the RNS number is small enough, then a quick-and-dirty computation of the interval evaluation is possible, which is obtained from Algorithm 1 by eliminating steps 9 to 11.

In Figure 5, we show the performance gains of the proposed approach over the other approaches tested. The superiority of the proposed approach over the naive one increases with an increase in the size of the moduli set. In turn, for the case of the 128-moduli set, the superiority of the proposed approach over the mixed-radix method is less than that for the case of the 64-moduli set. One possible reason for this is a decrease in the GPU memory bandwidth due to strided accesses to the input array of RNS numbers. The reader can refer to [28] for further details. An interval addressing scheme, in which the residues of the RNS numbers are interleaved, will provide more efficient access to the global GPU memory. We plan to explore this in the future.

Table 4 shows the memory consumption (in MB) of the tested implementations. Memory consumption is calculated as the size of the auxiliary buffer that needs to be allocated in the global GPU memory for a particular implementation to work properly.

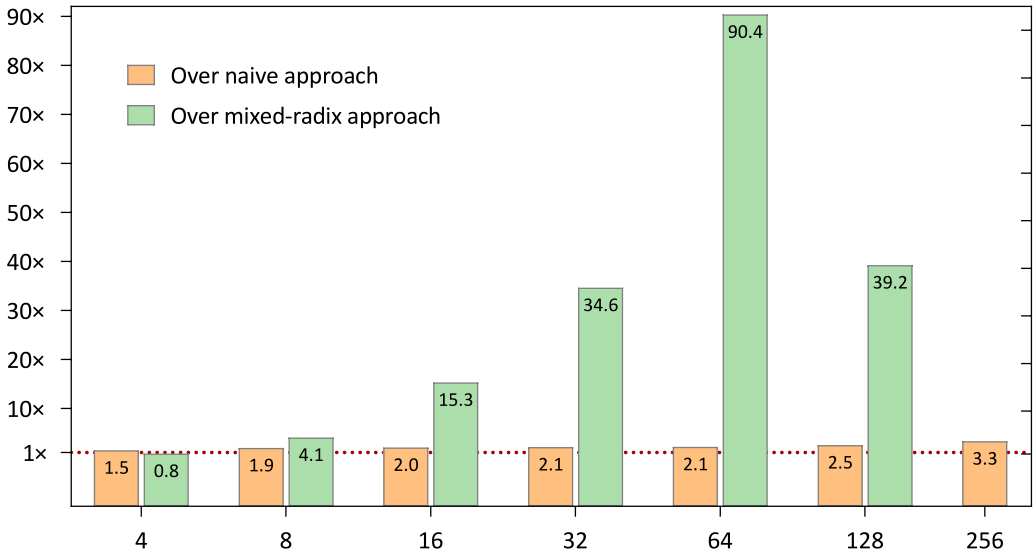


Figure 5. Performance gains of the proposed approach over the other two approaches.

The memory requirements of the mixed-radix implementation increase in proportion to the number of moduli, since the size of each mixed-radix representation is equal to the size of the corresponding RNS representation. In contrast, the memory requirements of the naive and proposed implementations are constant, since the size of each floating-point interval evaluation is fixed (40 bytes in our setting, including padding) and does not depend on the size of the moduli set.

Table 4. Memory consumption (in MB) of the different approaches.

Size of Set, n	Proposed	Naive	Mixed-Radix
4	190.735	0.000 977	95.367
8	190.735	0.000 977	171.661
16	190.735	0.000 977	324.249
32	190.735	0.000 977	629.425
64	190.735	0.000 977	1239.777
128	190.735	0.000 977	2460.480
256	190.735	0.000 977	4901.886

The naive implementation requires less memory, since it stores only partial results generated by the first reduction kernel, while the proposed implementation requires storing the computed interval evaluations for all inputs. However, the memory consumption of our proposed implementation do not seem critical, since the NVIDIA RTX 2080 graphics card has 8 GB of GDDR6 RAM.

7. Conclusions

An efficient method has been considered for performing difficult operations in a residue number system with arbitrary moduli sets. It relies only on very fast standard floating-point operations and integer modulo m_i arithmetic, which allows it to be implemented on many general-purpose computing architectures. The method can be used to improve the efficiency of various parallel algorithms operating in the RNS domain. In particular, it has now been successfully employed in CUDA-accelerated multiple-precision linear algebra kernels [10]. It is also implemented in GRNS, a new software library for high-performance RNS computations on CPU and GPU architectures (available at https://github.com/kisupov/grns). Similar to the MAX operation presented in this paper, in the future we plan to use our method to build other important data-parallel primitives over RNS arrays, such as SUM and SCAN.

Funding: This research was funded by the Russian Science Foundation grant number 20-71-00046.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data sharing is not applicable to this article.

Conflicts of Interest: The author declares no conflict of interest.

Sample Availability: The source code is available at: https://github.com/kisupov/grns.

Abbreviations

The following abbreviations are used in this manuscript:

RNS	Residue number system
GPUs	Graphics processing units
CRT	Chinese Remainder Theorem
MRC	Mixed-radix conversion
CUDA	Compute Unified Device Architecture

References

1. Ananda Mohan, P.V. *Residue Number Systems: Theory and Applications*; Birkhäuser: Cham, Switzerland, 2016.
2. Omondi, A.; Premkumar, B. *Residue Number Systems: Theory and Implementation*; Imperial College Press: London, UK, 2007.
3. Szabo, N.S.; Tanaka, R.I. *Residue Arithmetic and Its Application to Computer Technology*; McGraw-Hill: New York, NY, USA, 1967.
4. Qaisar Ahmad Al Badawi, A.; Polyakov, Y.; Aung, K.M.M.; Veeravalli, B.; Rohloff, K. Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme. *IEEE Trans. Emerg. Top. Comput.* **2019**. [CrossRef]
5. Bajard, J.C.; Eynard, J.; Hasan, M.A.; Zucca, V. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *Selected Areas in Cryptography—SAC 2016*; Avanzi, R., Heys, H., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 423–442.
6. Celesti, A.; Fazio, M.; Villari, M.; Puliafito, A. Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. *J. Netw. Comput. Appl.* **2016**, *59*, 208–218. [CrossRef]
7. Givaki, K.; Hojabr, R.; Najafi, M.H.; Khonsari, A.; Gholamrezayi, M.H.; Gorgin, S.; Rahmati, D. Using Residue Number Systems to Accelerate Deterministic Bit-stream Multiplication. In Proceedings of the 2019 IEEE 30th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), New York, NY, USA, 15–17 July 2019; p. 40. [CrossRef]
8. Vayalil, N.C.; Paul, M.; Kong, Y. A Residue Number System Hardware Design of Fast-Search Variable-Motion-Estimation Accelerator for HEVC/H.265. *IEEE Trans. Circuits Syst. Video Technol.* **2019**, *29*, 572–581. [CrossRef]
9. Chen, J.; Hu, J. Energy-Efficient Digital Signal Processing via Voltage-Overscaling-Based Residue Number System. *IEEE Trans. Very Large Scale Integr. Syst.* **2013**, *21*, 1322–1332. [CrossRef]
10. Isupov, K.; Knyazkov, V.; Kuvaev, A. Design and implementation of multiple-precision BLAS Level 1 functions for graphics processing units. *J. Parallel Distrib. Comput.* **2020**, *140*, 25–36. [CrossRef]
11. Guo, Z.; Gao, Z.; Mei, H.; Zhao, M.; Yang, J. Design and Optimization for Storage Mechanism of the Public Blockchain Based on Redundant Residual Number System. *IEEE Access* **2019**, *7*, 98546–98554. [CrossRef]
12. Gayoso, C.A.; Arnone, L.; González, C.; Moreira, J.C. A general construction method for Pseudo-Random Number Generators based on the Residue Number System. In Proceedings of the 2019 XVIII Workshop on Information Processing and Control (RPIC), Bahía Blanca, Argentina, 18–20 September 2019; pp. 25–30. [CrossRef]
13. Salamat, S.; Imani, M.; Gupta, S.; Rosing, T. RNSnet: In-Memory Neural Network Acceleration Using Residue Number System. In Proceedings of the 2018 IEEE International Conference on Rebooting Computing (ICRC), Tysons, VA, USA, 7–9 November 2018; pp. 1–12. [CrossRef]
14. Torabi, Z.; Jaberipur, G.; Belghadr, A. Fast division in the residue number system $\{2^n+1, 2^n, 2^n-1\}$ based on shortcut mixed radix conversion. *Comput. Electr. Eng.* **2020**, *83*, 106571. [CrossRef]
15. Kumar, S.; Chang, C.; Tay, T.F. New Algorithm for Signed Integer Comparison in $\{2^{n+k}, 2^n-1, 2^n+1, 2^{n\pm1}-1\}$ and Its Efficient Hardware Implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2017**, *64*, 1481–1493. [CrossRef]
16. Hiasat, A. General Frameworks for Designing Arithmetic Components for Residue Number Systems. In *Intelligent Methods in Computing, Communications and Control*; Dzitac, I., Dzitac, S., Filip, F.G., Kacprzyk, J., Manolescu, M.J., Oros, H., Eds.; Springer International Publishing: Cham, Switzerland, 2021; pp. 82–92. [CrossRef]
17. Brönnimann, H.; Emiris, I.Z.; Pan, V.Y.; Pion, S. Sign determination in residue number systems. *Theor. Comput. Sci.* **1999**, *210*, 173–197. [CrossRef]
18. Dimauro, G.; Impedovo, S.; Pirlo, G. A new technique for fast number comparison in the residue number system. *IEEE Trans. Comput.* **1993**, *42*, 608–612. [CrossRef]

19. Wang, Y.; Song, X.; Aboulhamid, M. A new algorithm for RNS magnitude comparison based on New Chinese Remainder Theorem II. In Proceedings of the Ninth Great Lakes Symposium on VLSI, Ypsilanti, Michigan, 4–6 March 1999; pp. 362–365. [CrossRef]
20. Gbolagade, K.A.; Cotofana, S.D. An $O(n)$ Residue Number System to Mixed Radix Conversion technique. In Proceedings of the IEEE International Symposium on Circuits and Systems, Taipei, Taiwan, 24–27 May 2009; pp. 521–524. [CrossRef]
21. Isupov, K. Using Floating-Point Intervals for Non-Modular Computations in Residue Number System. *IEEE Access* **2020**, *8*, 58603–58619. [CrossRef]
22. Lu, M. *Arithmetic and Logic in Computer Systems*; John Wiley & Sons, Ltd.: Hoboken, NJ, USA, 2004. [CrossRef]
23. Taylor, F.J. Residue Arithmetic A Tutorial with Examples. *Computer* **1984**, *17*, 50–62. [CrossRef]
24. Vu, T.V. Efficient Implementations of the Chinese Remainder Theorem for Sign Detection and Residue Decoding. *IEEE Trans. Comput.* **1985**, *100*, 646–651. [CrossRef]
25. Soderstrand, M.; Vernia, C.; Chang, J.H. An improved residue number system digital-to-analog converter. *IEEE Trans. Circuits Syst.* **1983**, *30*, 903–907. [CrossRef]
26. Dean, J.; Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
27. Farber, R. *CUDA Application Design and Development*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2011.
28. Harris, M. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. Available online: https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ (accessed on 11 November 2020).