# RNSnet: In-Memory Neural Network Acceleration Using Residue Number System

Sahand Salamat, Mohsen Imani, Sarangh Gupta, Tajana Rosing
Department of Computer Science and Engineering, UC San Diego, La Jolla, CA 92093, USA
{sasalama, moimani, sgupta, tajana}@ucsd.edu

***Abstract*—We live in a world where technological advances are continually creating more data than what we can deal with. Machine learning algorithms, in particular Deep Neural Networks (DNNs), are essential to process such large data. Computation of DNNs requires loading the trained network on the processing element and storing the result in memory. Therefore, running these applications need a high memory bandwidth. Traditional cores are memory limited in terms of the memory bandwidth. Hence, running DNNs on traditional cores results in high energy consumption and slows down processing speed due to a large amount of data movement between memory and processing units. Several prior works tried to address data movement issue by enabling Processing In-Memory (PIM) using crossbar analog multiplication. However, these designs suffer from the large overhead of data conversion between analog and digital domains. In this work, we propose RNSnet, which uses Residue Number System (RNS) to execute neural network completely in the digital domain in memory. RNSnet simplifies the fundamental neural network operations and maps them to in-memory addition and data access. We test the efficiency of the proposed design on several popular neural network applications. Our experimental result shows that RNSnet consumes 145.5× less energy and obtains 35.4× speedup as compared to NVIDIA GPU GTX 1080. In addition, our results show that RNSnet can achieve 8.5× higher energy-delay product as compared to the state-of-the-art neural network accelerators.**

## I. Introduction

As Internet of Things (IoT) becomes a reality, humans will be significantly outnumbered by the networked devices ready to respond to our every need. These IoT systems, in near future, will generate a large amount of data and put enormous pressure on computing nodes to analyze unreasonable quantities of data. IoT applications often process raw data by running machine learning algorithms. However, neural networks (NNs), the most common machine learning applications, are computationally expensive as it requires a large amount of resources to execute [1]. Most of the NNs are executed on mobile and embedded devices which have limited resources and energy budget. In addition, most of the NNs require a real-time response. Hence, they need to be executed with high performance while keeping the energy consumption to the lowest possible amount.

Several prior works tried to accelerate NN applications on general purpose processors such as CPUs and GPUs [2]. Additionally, works on approximate computing accelerate different applications, and in particular NNs. These works use bitwidth reduction [3], approximate synthesis [4, 5], to name but two. All these platforms, however, still suffer from data movement issue. Looking at the power consumption of NN applications, the main inefficiency in running these applications on traditional cores comes from the data movement between the processor and off-chip memory. Most of the NN applications need quite a large memory to store the trained weights, Moreover, NN computations need many memory accesses to load the stored data and store the generated results.

Processing In-Memory (PIM) is a promising solution to address the data movement issue by implementing data operations within the memory [6, 7, 8]. PIM architecture have been used to accelerate different applications including machine learning [9, 10], graph processing [11, 12], brain-inspired computing [13, 14], and approximate computing [15, 16]. Instead of sending a large amount of data to the processing cores for computations, PIM performs a part of computation tasks, e.g., bit-wise computations, inside the memory. Therefore, the application performance can be accelerated significantly by bypassing the memory access bottleneck. Several existing works have proposed PIM-based neural network accelerators which keep the input data and trained weights inside memory [6]. For example, the work in [7] showed that memristor devices can model multiplications and additions for each neuron. The authors of [7] store trained weights of each neuron as device resistance values, and pass currents corresponding to the digital input values in a similar way to spiking neuromorphic computing [17]. Although these approaches improve the efficiency of the design, these designs use analog execution blocks which are not scalable with technology.

Residue Number System (RNS) is an unorthodox number representation which is based on the Chinese Remainder Theorem (CRT) [18]. RNS is defined by a set of $k$ integer numbers which are relatively prime, called modulis. Each binary number is divided by each moduli and the remainders of the divisions are representative of the binary number in RNS. RNS decomposes each number to the remainders of the number when divided by a set of predefined modulis instead of representing the binary number. Each residue is always less than the corresponding moduli; thus, for representing each residue we need fewer bits. RNS simplifies the operations by breaking down any operation to the same operation on the residues which have less bit width than the binary representation. Reducing the operands bit width leads to simpler arithmetic units which are easier, and more efficient to implement in memory.

In this paper, we exploit the benefits provided by RNS to

978-1-5386-9170-0/18/$31.00 ©2018 IEEE

execute the NN in memory. We propose RNSnet, a novel neural network accelerator based on RNS. Instead of working with long 32-bit values, RNSnet maps all inputs and weights of NN to RNS. Therefore, it simplifies the operations to PIM friendly ones, then executes the NN in-memory. RNSnet models: (i) input-weight multiplication using additions and memory accesses; (ii) weighted input accumulation using local in-memory addition; (iii) activation function and pooling by using a novel comparison method. Our new RNS-based comparison method results in modeling both linear and non-linear activation functions. We evaluate the efficiency of the proposed RNSnet on several NN applications. Our evaluations show that the proposed RNSnet can achieve 145.5× energy efficiency and 35.4 speedup as compared to GPU implementation. Comparing the proposed design with the state-of-the-art accelerators shows that RNSnet can provide 2.2× faster and 3.9× higher energy efficiency while providing the same classification accuracy.

The rest of the paper is organized as the following. Section II reviews the related works. Section III proposes an RNS-based in-memory NN execution as well as the required background. Section IV depicts the architecture of proposed in-memory hardware. The experimental results are presented in Section V. Finally, Section VI concludes the paper.

## II. Related Work

### A. Neural Network Acceleration

Modern neural network algorithms are executed on diverse types of platforms such as CPUs, GPUs, and FPGAs [19, 20, 21]. However, in these designs the main computation still relies on CMOS-based cores, thus, it still suffers from the data movement issue. Additionally, Several recent works binarize the networks weights and inputs to reduce the complexity of the NN operations [22, 23, 24]. Binarizing the network simplifies the multiplication and addition operations to XOR and population count. However, Binarized Neural Networks (BNNs) cannot provide a comparable accuracy as the NNs using fixed/floating point weights. Moreover the training time of BNNs is higher than that of DNNs. To address data movement issue, prior works accelerate NNs by enabling in-memory computation [7, 25]. These designs use multi-level memristor devices which operate the multiplication and addition operations by converting digital values to analog signals. Therefore, They need to use Analog to Digital Converters (ADCs) and Digital to Analog Converters (DACs) which take more than 61% of the chip power consumption. Moreover, these approaches have potential design issues. For example, their designs require analog and digital-mixed circuits, e.g., ADC and DAC, which do not scale as the CMOS technology. In contrast, we design the RNSnet accelerator based on digital technology, which supports the neural network computations inside the memory.

### B. RNS

To convert a binary number to RNS the remainder of dividing the binary number to the modulis must calculated. Several works, as[26, 27, 28, 29, 30], exploit the effect of selecting the moduliset on complexity of the conversion. Several works such as [31, 32] investigate the effect of selecting the modulis on the complexity of implementing the operations on hardware. After converting each binary number into RNS the operations can perform in RNS domain; therefore, RNS operations should be introduced and modified for the given moduli set. RNS can reduce the bit width of the numbers at the cost of increasing the number of operations. After converting the binary numbers to RNS the operations need to be applied on each residue instead of the binary number.

Using RNS simplifies the hardware implementation and increases the ability of parallelism. RNS is used to implement compute-intensive applications, such as DSP [33, 34, 35], and NNs [36, 37]. These applications are implemented on different platforms such as ASIC, FPGA, GPU, and CPUs [31, 33, 38]. Several works including [39, 40, 41] take the advantage of RNS to detect and correct system errors. The work in [42] instead of using RNS representation to accelerate the hardware, improves the reliability of the hardware. By using Redundant RNS representation (RRNS) they enable detection and correction of errors caused due to both faulty storage and compute units.

Several recent efforts [43, 44, 45] have been focused on optimizing and simplifying the architecture of RNS operations, the work in [46] focuses on accelerating multipliers for neural network applications on FPGA. To accelerate the neural network applications, the authors in [47] propose a platform to interpolate the lost information by using convolutional neural networks (CNNs). They propose an FPGA implementation and a performance evaluation of CNN-based super-resolution system, which can process moving images in real time. Works in [36, 48] use nested RNS to reduce the bit width of numbers to enable lookup-table-based implementation. Although nested RNS improves the efficacy, it still has the drawback of data transfer between processing units and memory which is the main disadvantage of all CMOS-based designs.

Work in [37] implements an artificial neural network on ROM using RNS. In this work, a functionality of a neuron is modeled by several layers of ROMs. However, this technique suffers from the scalability. In this approach, increasing the number neuron inputs by one, increases the design memory requirement by a power of two. Therefore, this technique is not applicable to even the smallest neural networks.

## III. Proposed RNSnet

### A. RNS background

Consider $ModuliSet = \{M_1, M_2, ..., M_k\}$ where each $M_k$ in the ModuliSet has no common factor with the others. The remainder of dividing the binary number by each member of the ModuliSet represents that RNS value. By using RNS we can uniquely represent range of numbers called the Dynamic Range ($DR$) which is equal to $DR = \prod_{i=1}^{k} M_i$. It means that, $BinaryNumber \in [T, T+DR)$ can be represented uniquely as $\{R_1, R_2, ..., R_k\}$ where $R_k$ is $|X|_{M_k}$ ($X \bmod M_k$). Converting

the binary number to the RNS is called forward conversion which is equal to finding the $R_1, R_2, \ldots, R_k$. In order to have a hardware friendly forward conversion [32], in this work we select $ModuliSet = \{2^t - 1, 2^t, 2^t + 1\}$. In the following, we describe the forward conversion for the selected *ModuliSet*. Consider $X$ as an $n$-bit binary number and *ModuliSet* as a set of moduli with three relative prime numbers. The number is represented in RNS as $\{R_1, R_2, R_3\}$. Note that, if the bit width of the binary number is not a factor of 3, then we add zeros to the leftmost bits.

$$
\begin{aligned}
&X \equiv n \text{ bits binary number input}\\
&ModuliSet = \{2^{\lceil \frac{n}{3} \rceil} - 1, 2^{\lceil \frac{n}{3} \rceil}, 2^{\lceil \frac{n}{3} \rceil} + 1\},\ t \equiv \lceil \frac{n}{3} \rceil\\
&X = \{X_{n-1}, X_{n-2}, ..., X_1, X_0\}
\end{aligned}
$$

$$
\begin{cases}
a_1 \equiv X_{t-1}, X_{t-2}, ..., X_0 \\
a_2 \equiv X_{2t-1}, X_{2t-2}, ..., X_t \Rightarrow X = a_3 \times 2^{2t} + a_2 \times 2^{2t} + a_1 \\
a_3 \equiv X_{n-1}, X_{n-2}, ..., X_{2t}
\end{cases} \quad (1)
$$

$$
\begin{cases}
R_1 = |X|_{2^t-1} = |a_3 \times 2^{2t} + a_2 \times 2^t + a_1|_{2^t-1} \\
R_1 = |a_3 \times 2^{2t}|_{2^t-1} + |a_2 \times 2^t|_{2^t-1} + |a_1|_{2^t-1} \\
\xrightarrow[|2^t|_{2^t-1}=1]{|2^{2t}|_{2^t-1}=1} R_1 = |a_1 + a_2 + a_3|_{2^t-1}
\end{cases} \quad (2)
$$

$$
\begin{cases}
R_2 = |X|_{2^t} = |a_3 \times 2^{2t} + a_2 \times 2^t + a_1|_{2^t} \\
R_2 = |a_3 \times 2^{2t}|_{2^t} + |a_2 \times 2^t|_{2^t} + |a_1|_{2^t} \\
\xrightarrow[|2^t|_{2^t}=0]{|2^{2t}|_{2^t}=0} R_2 = a_1
\end{cases} \quad (3)
$$

$$
\begin{cases}
R_3 = |X|_{2^t+1} = |a_3 \times 2^{2t} + a_2 \times 2^t + a_1|_{2^t+1} \\
R_3 = |a_3 \times 2^{2t}|_{2^t+1} + |a_2 \times 2^t|_{2^t+1} + |a_1|_{2^t+1} \\
\xrightarrow[|2^t|_{2^t+1}=-1]{|2^{2t}|_{2^t+1}=1} R_3 = |a_1 - a_2 + a_3|_{2^t+1}
\end{cases} \quad (4)
$$

Binary numbers are converted to RNS using the above equations and $2^t - 1, 2^t, 2^t + 1$ residues are calculated using Equations (2), (3), (4) respectively. In order to calculate the residues, first, we split the binary number $X$ into three parts. The first $t$ bits of $X$ are called $a_1$ and the second $t$ bits are called $a_2$ and the remaining bits of $X$ are called $a_3$. To calculate $R_1 = |a_1 + a_2 + a_3|_{2^t-1}$, the $a_1$, $a_2$ and $a_3$ are added together and if the result is greater than the moduli$(2^t - 1)$, we subtract the moduli from the result of the addition, we call this addition as modulo-addition. Additionally, Modulo-adder is an adder whose output is the remainder of the addition divided by the moduli. The $2^t$ residue is equal to the $a_2$ and the $2^t + 1$residue is calculated in a similar way to calculating the $R_1$. Therefore, our design requires two additions, that each has three inputs, to perform forward conversion. For each number, two modulo-adders are used to calculate the residue of the binary number in the pre-defined ModuliSet. After converting binary numbers to RNS, operations, such as addition and multiplication, can be executed similarly to binary numbers. However, whenever the result exceeds the moduli,the corresponding residue is calculated. Moreover, after executing the application, the result should be converted back to binary using backward conversion methods such as [32]. In RNSnet instead of the complex backward, we map each RNS number to the corresponding binary number using lookup tables. Considering the fact that the RNSnet is calculated in memory and the input bit width as well as the modulis are fixed the look-up-based conversion is in-memory friendly and also it does not need a large memory.

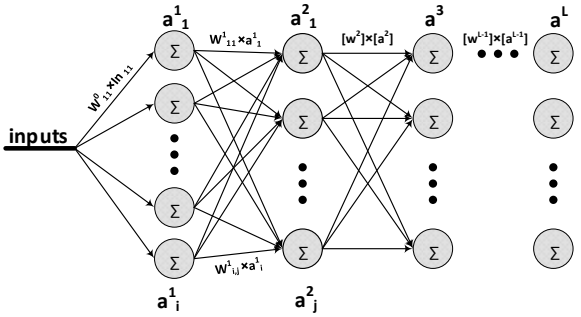

Fig. 1. Neural Network structure

### B. NN background

Fig 1 depicts the functionality of a Neural Network. Each NN contains multiple layers of neurons which are connected to the neurons of the previous layer (the neurons of the first layer are directly connected to the inputs) using a weighted link. The output of each neuron in the previous layer is multiplied by the weight of the corresponding link and the summation of these values is calculated in the neuron and thereafter, an activation function is applied to the result of the summation. As illustrated in Fig 1, $a^L$ denotes the layer l neurons and $W^L$ denotes a 2D weight matrix which links each neuron of layer l-1 to every neuron of layer l. Each node of layer l is linked to all neurons of the layer l-1 and each link requires a multiplication resulting in an immense amount of calculations. Additionally, the $W^i s$ are stored in memory and thus, each execution requires a memory access. Considering two above facts, in-memory computation would be a great solution for NN applications. However, only simple processes can be done in memory while most of the required calculations in NNs are multiplications and additions with usually 16 to 48 bits. Thanks to using RNS system all the bit width of numbers are reduced(16 to 48 bits numbers are mapped to 6 to 16 bit numbers).

### C. NN using RNS

Neural networks consist of multiple layers of neurons. Each neuron is connected to all available neurons in the next layer using pre-trained weights. In NNs, each neuron accumulates all the weighted-values received from neurons in the previous layer and then pass the output through an activation function as shown in Fig 2a. In this work, we propose RNS-based neural network (RNSnet) which can support all neural network operations in an efficient and scalable way in RNS domain.

Our design uses a forward conversion logic to convert all input data and all trained weights of the neural network to RNS. All neurons in the input layer of the NN will take the

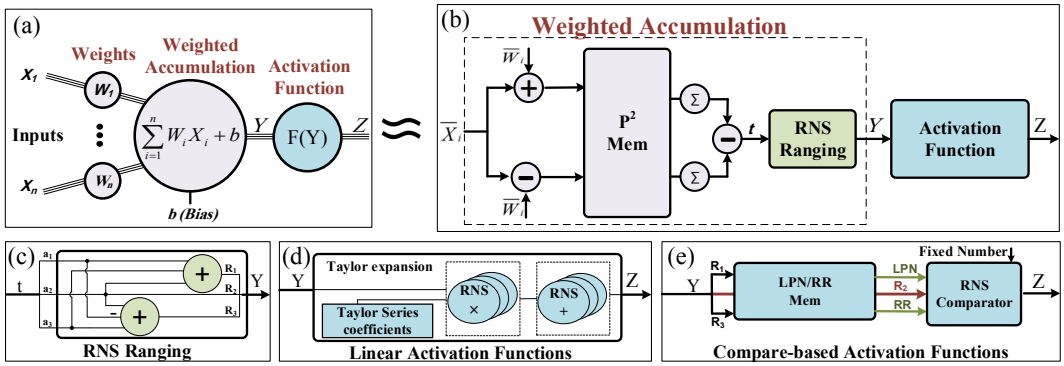


Fig. 2. (a) the functionality of each neuron in neural network. (b) A representation of a neuron computations in the proposed RNSnet. (c) RNSnet supporting the activation function using Taylor expansion. (d) the compare operation required for activation function in RNSnet.

input values in RNS form. Then, the entire neuron functionality is implemented in RNS without any backward conversion. Figure 2 shows the overview of the proposed RNSnet. This figure shows how RNSnet models each neuron using additions and memory accesses. Our design models multiplication between neuron inputs and weights, by adding, subtracting and memory lookup. Since each neuron multiplies several inputs and weights, the results of all multiplications accumulate in the memory block. As the result of RNS accumulation can be out of the RNS range, our Ranging block (shown in Figure 2c), converts the value to a valid RNS format. Finally, this value is passed through the activation function (shown in Fig 2d, e), which can be a linear function, e.g. *Sigmoid*, or a compare-based function, e.g. rectified linear unit (*ReLU*). We approximate the linear/non-linear activation functions by their equivalent Taylor expansion terms, which are implemented using the proposed RNS addition and multiplication. Additionally, the compare-based activation functions and pooling layers are implemented using the proposed compare method.

Now, we explain how RNSnet models the above-mentioned fundamental neural network operations in RNS, and in the following section we propose the in-memory implementation of the operations.

**Addition**

As illustrated in equation (5), in order to add the binary numbers we can add their RNS representation. To add RNS numbers, it is enough to add the residues of each moduli. In case of using ModuliSet with the size three, RNS requires three additions, modulo-$2^t-1$, modulo-$2^t$, and modulo-$2^t+1$ addition.

$$A+B \xrightarrow{RNS}$$

$$\begin{cases} R_1^{A+B} = |A+B|_{2^t-1} = |A|_{2^t-1} + |B|_{2^t-1} = |R_1^A + R_1^B|_{2^t-1} \\ R_2^{A+B} = |A+B|_{2^t} = |A|_{2^t} + |B|_{2^t} = |R_2^A + R_2^B|_{2^t} \\ R_3^{A+B} = |A+B|_{2^t+1} = |A|_{2^t+1} + |B|_{2^t+1} = |R_3^A + R_3^B|_{2^t+1} \end{cases} \quad (5)$$

As shown in Figure 3, when the result of the addition is greater than the moduli we subtract the moduli from the addition result. To compare the result of the addition with the moduli, we subtract the moduli from the addition result and the sign bit of the subtraction represents the greater number. When the sign bit of the subtraction result is '0' the addition result is greater than the moduli and therefore the result of the modulo-$2^t-1$ addition is $A_{2^t-1}+B_{2^t-1}-2^t-1$. The structure of modulo-$2^t+1$ adder is completely similar to that of the modulo-$2^t-1$ adder. The remainder of dividing each number on $2^t$ is the $t$ least significant bits of the number; thus, The modulo-$2^t$ adder uses the $t$ rightmost bits. In fully connected

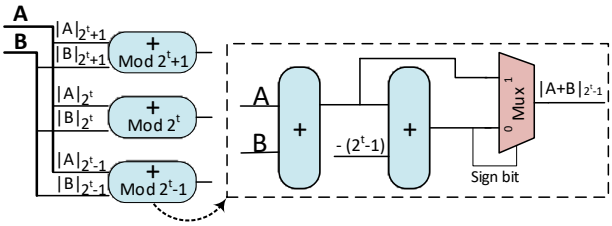


Fig. 3. RNS addition implementation and Modulo-$2^t-1$ adder structure.

NN structures, each neuron is connected to all neurons in the previous layer. Therefore, each neuron adds several numbers together in RNS form. To calculate the summation of all weighted-values, we use parallel in-memory adder. Our design assigns a single crossbar memory to each neuron and supports in-memory addition in a tree-based structure. We explain the details of the hardware in Section IV. To achieve fast summation, instead of using modulo adders which are costlier than normal adders, our design uses normal adders and avoids ranging the addition result to RNS form in the intermediate stages. Our design adds all values together in a tree-based structure and converts the addition result to RNS form in the end. Generally, adding $m$ $t$-bit numbers results in a value with $t+\lceil \log_2 m \rceil$ bits. In order to return this number to RNS range, it is necessary to find the remainder of dividing the result by the moduli. However, the division is a costly function which is very difficult to be implemented in-memory. In this work, we simplify finding the remainder of division to a function similar to the forward conversion. Our design works based on the fact that the remainder of $2^t$ divided by $2^t-1$ and $2^t+1$ is +1 and -1 respectively. In general, the addition result can be written as $S_2 \times 2^{2t} + S_1 \times 2^t + S_0$. Therefore, we can use the same conversion as the forward conversion to return the addition result to the RNS range. When the weights and the

inputs are 18 bits, the ModuliSet is $\{2^6-1, 2^6, 2^6+1\}$, thus the summation result of 2048 numbers has $6+\log_2 2048 = 17$ bits for $2^6$ and $2^6-1$ modulis. Similarly, for $2^6+1$ moduli the summation result has 18 bits. Based on the value of $t$, we can simplify the summation result to $S_{17:12} \times 2^{12} + S_{11:6} \times 2^6 + S_{5:0}$. Therefore, $S \ mod \ 2^6-1$ is equal to $S_{17:12} + S_{11:6} + S_{5:0}$. Using this technique, the residue of $2^6$ and $2^6+1$ is calculated to be $S_{5:0}$ and $S_{17:12} - S_{11:6} + S_{5:0}$ respectively.

**Multiplication**

A Neural network involves several multiplications. The output of each neural network layer multiplies by a weighted network to generate inputs for the next layer. Although converting the binary numbers to RNS reduces the bit width, the multiplication is still costly. additionally, similar to addition, after the multiplication, a division is required to back the multiplication result to the RNS range (calculating the remainder).

In this work, we replace the multiplication between two RNS values (e,g, $\forall_{i=1}^{k} {R_{M_i}^a}^2 \times {R_{M_i}^b}^2$ which we call it $a \times b$ briefly ), by an equivalent term $\frac{(a+b)^2}{4} - \frac{(a-b)^2}{4}$. Considering the square of binomial, $(a+b)^2 = a^2+b^2+2ab$, and $(a-b)^2 = a^2+b-^2 ab$. We can calculate the terms $a+b$ and $a-b$ using in-memory addition. The result of either $a+b$ or $a-b$ is a $t+1$-bit number (a and b are t-bit numbers). Then, our design pre-stores all possibilities for $x^2$ where x is a t+1-bit number in a crossbar memory. Therefore, the result of $\frac{(a+b)^2}{4}$ and $\frac{(a-b)^2}{4}$ is pre-stored in the memory. Thus, to calculate the result of a multiplication, for each residue, first, we calculate $a+b$ and $a-b$. The result of these additions are directly an address to a memory which stores the result of the square binomial. Two memory accesses gives us the values of $\frac{(a+b)^2}{4}$ and $\frac{(a-b)^2}{4}$. Considering the architecture of neural networks, the multiplied values are accumulated. Therefore, instead of $\Sigma(\frac{(X_i+W_i)^2}{4} - \frac{(X_i-W_i)^2}{4})$ we calculate the equivalent term $\Sigma\frac{(X_i+W_i)^2}{4} - \Sigma\frac{(X_i-W_i)^2}{4}$. In this way, as it is illustrated in Fig 2b, we eliminate a subtraction in each multiplication, thereby, increasing the efficacy of the proposed RNSnet.

Figure 4 shows the multiplication in RNS format. As we are using three modulis in this work, we require six additions, six memory reads, and three subtractions to perform the multiplication. However, as we discussed, we do not do the subtraction for each multiplication and we subtract the results of the summations at the end. The main advantage of using square-based calculation is its low memory requirement. For instance, for multiplication of RNS values using square-based multiplication, we require $3 \times 2^t + 1$ rows of memory (each memory has 2 read ports) to pre-store the result of the RNS multiplication, where $t$ is the bit width of the numbers in RNS. While, without square-based multiplication technique, if we want to lookup the multiplication result using $a$ and $b$ combination as a memory address, the required memory is $3 \times 2^{2\times t}$ rows, which is $2^{t-1}$ times more than square-based technique. Moreover, without using RNS the memory-based implementation of the multiplication needs $2^{2\times 3\times t}$ memory rows, which is significantly higher than the RNS-based multiplication.

**Activation Function and Pooling**

In a neural network, each neuron accumulates the weighted signals from the neurons in the previous layer. Then, it passes the result of the summation through an activation function $f(\Sigma W_{i,j}^l \times a_j^{l-1})$. There are several types of linear or non-linear activation functions. Linear functions such as $ax+b$ are totally supported using RNS addition and multiplication explained in previous parts.

RNSnet supports also nonlinear activation functions. we can approximate the nonlinear activation functions using a few first terms of the Taylor expansion (Fig 2d). To implement compared-based activation functions, such as Rectified Linear Unit (ReLU), defining a comparison operator is necessary. For convolutional layers, our design needs to support Pooling layers most of which need comparison operations. Min and Max pooling are two common types of pooling layers. However, the comparison operation in RNS is not as straight forward as the multiplication and addition operations[49]. In RNSnet we implement the compared-based activation function and pooling layers as shown in (Fig 2e) which is described hereunder. Most of the implemented compare operators use backward conversion to bring the data back to binary and then compare the numbers in binary format [50]. These methods impose a significant cost to the system, especially in NNs where the comparison operation is required for each neuron. In this work, we propose a technique to compare numbers in RNS. In our design each RNS number is represented by three residues $\{R_1, R_2, R_3\}$ and as illustrated in Equation below. In our RNS, adding $(2^t+1)\times(2^t-1)$ (which is a repetition period of $R_1$ and $R_3$) to each RNS number, results in a number with the same value for the $2^t+1$ and $2^t-1$ residues, while the $2^t$ residue is decremented by one. This pattern is shown below:

$$X \xrightarrow[RNS]{ModuliSet=\{2^t+1, 2^t, 2^t-1\}} = \{R_1, R_2, R_3\}$$

$$|X|_{2^t-1} = R_1, |X|_{2^t} = R_2, |X|_{2^t+1} = R_3$$

$$\begin{cases} R'_1 \equiv |X + ((2^t-1)\times(2^t+1))|_{2^t-1} \\ |(2^t-1)\times(2^t+1)|_{2^t-1} = 0 \rightarrow R'_1 = |X|_{2^t-1} = R_1 \end{cases}$$

$$\begin{cases} R'_3 \equiv |X + ((2^t-1)\times(2^t+1))|_{2^t+1} \\ |(2^t-1)\times(2^t+1)|_{2^t+1} = 0 \rightarrow R'_3 = |X|_{2^t+1} = R_3 \end{cases}$$

$$\begin{cases} R'_2 \equiv |X + ((2^t-1)\times(2^t+1))|_{2^t} \\ R'_2 = |X|_{2^t+1} + |(2^t-1)\times(2^t+1)|_{2^t} \\ R'_2 = R_2 + |(2^t-1)|_{2^t} \times |(2^t+1)|_{2^t} \\ R'_2 = R_2 + (-1 \times +1) = R_2 - 1 \end{cases}$$

Table I shows the trend of the values in RNS using $\{7, 8, 9\}$ as the ModuliSet. The left half of the Table I shows how RNS values change when the binary value increases from 1 to 66. As we can see, the value of $R_1$ and $R_3$ change periodically in $7 \times 9 = 63$ step while in each period the value of the $R_2$ decrements by one. The right half of the Table I shows how the RNS values change over periodic binary numbers. In order to compare RNS numbers, first, we need to define two

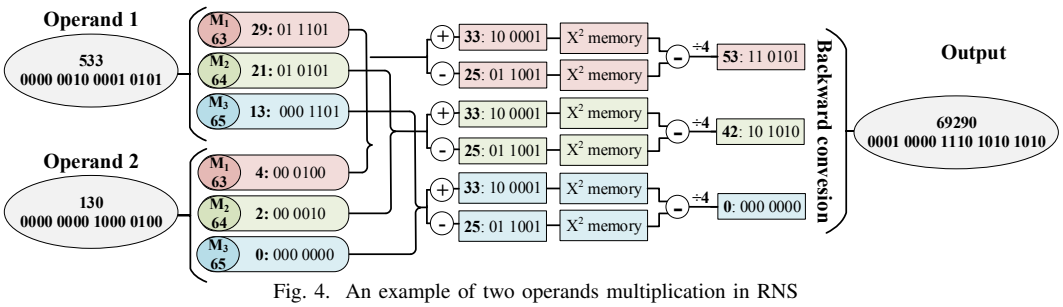


Fig. 4. An example of two operands multiplication in RNS

parameters: for a given $R_1$, $R_2$, and $R_3$ (i) The Least Possible Number (LPN) is the least number that has the same $R_1$ and $R_3$ as its $2^t-1$ and $2^t+1$ residues respectively, and (ii) The Reference Residue is defined as $R_2$ of the corresponding LPN.

For instance, considering ModuliSet=$\{7,8,9\}$, the LPN for every RNS number with $R_1=2$ and $R_3=3$ is 30. Additionally, the RR for every RNS number that has $R_1=2$ and $R_3=3$ is 6, since $|30|_8=6 \rightarrow R_2=6$. Therefore, We can define each binary number as:

$$X \xrightarrow[RNS]{ModuliSet=\{2^t+1,2^t,2^t-1\}} = \{R_1,R_2,R_3\}$$
$$X' \equiv (2^t+1)\times(2^t-1)\times(RR_X-R_2)+LPN_X$$

$$X' \xrightarrow[RNS]{ModuliSet=\{2^t+1,2^t,2^t-1\}} = \{R'_1,R'_2,R'_3\}$$
$$\begin{cases} R'_1 = |X'|_{2^t-1} = |LPN_X|_{2^t-1} = RR_X = R_1 \\ R'_2 = |X'|_{2^t} = |-1\times(RR-R_2)|_{2^t} + |LPN_X|_{2^t} \rightarrow \\ R'_2 = R_2 - RR_X + RR_X = R_2 \\ R'_3 = |X'|_{2^t+1} = |LPN_X|_{2^t+1} = RR_X = R_3 \end{cases}$$
$$X' = X \rightarrow X = (2^t+1)\times(2^t-1)\times(RR_X-R_2)+LPN_X$$

As far as the comparison is concerned, due to the fact that $2^t+1\times 2^t-1$ is greater than the maximum possible LPN, each number that has a greater value of $(R_2-RR)$ is greater. Therefore, to compare two numbers, first, we subtract their $R_2$ from the $RR$, then, we compare these results for both numbers and the number with the greater difference is the greater one. If the difference is equal for two numbers, the number that has a larger LPN is the greater one. In order to compare the numbers in RNS, first, we generate a memory that stores the LPN, and RR for each possible values of $R_1$ and $R_3$. To compare RNS numbers, we use both $R_1$ and $R_3$ as the memory address. Then, we read the LPN and RR from the memory and we compare the numbers using the above-mentioned technique.

TABLE I
TREND OF RNS REPRESENTATION OVER ORDERED AND PERIODIC BINARY NUMBERS

| ordered numbers | | | | Periodic Numbers | | | |
|---|---|---|---|---|---|---|---|
| Binary | $R_1$ | $R_2$ | $R_3$ | Binary | $R_1$ | $R_2$ | $R_3$ |
| **1** | 1 | 1 | 1 | 30 | 2 | 6 | 3 |
| **2** | 2 | 2 | 2 | 93 | 2 | 5 | 3 |
| **3** | 3 | 3 | 3 | 156 | 2 | 4 | 3 |
| ... | ... | ... | ... | 219 | 2 | 3 | 3 |
| **63** | 0 | 7 | 0 | 282 | 2 | 2 | 3 |
| **64** | 1 | 0 | 1 | 345 | 2 | 1 | 3 |
| **65** | 2 | 1 | 2 | 408 | 2 | 0 | 3 |
| **66** | 3 | 2 | 3 | 471 | 2 | 7 | 3 |

## IV. HARDWARE SUPPORT

### A. RNS Addition

As shown in Section III-C, RNS addition requires an integer addition, a comparison and final selection. We first add the two inputs. We then perform the comparison by adding $(-)$moduli to the previous output. We then select the appropriate output based on the sign bit of the comparison output. If the sign bit is '0,' the output of comparison is selected otherwise the output before comparison is selected as the result of RNS addition.

We use Memristor Aided loGIC (MAGIC) `NOR` [51, 52, 53] to execute logic functions in memory due to its simplicity and independence of execution from the data stored in memory. Applying voltages to wordlines to execute logic within a particular column triggers same operation in other columns as well. This allows execution of logic in parallel, thereby improving the speed of the process. Figure 5 shows a simple `NOR` operation in memory. A row driver applies an execution voltage, $V_0$, to the output row in which the `NOR` result is to be written while the rows that `NOR` operates on are grounded. Using this in-memory `NOR` operation, the design proposed in [52] evaluates sum ($S$) and carry ($C_{out}$) bits of 1-bit full addition (inputs being $A,B,C$) as following:

$$C_{out} = ((A+B)'+(B+C)'+(C+A)')'$$
$$S = (((A'+B'+C')'+((A+B+C)'+C_{out})')')'$$

Here, $C_{out}$ is realized as a series of 4 `NOR` operations while $S$ is obtained by 3 `NOT` operations (evaluation of $A',B'$, and $C'$) followed by 5 `NOR` operations. A `NOT` operation is implemented as a `NOR` operation with 1 input. From this point onward, a `NOR` operation by default implies a `MAGIC NOR` operation. This design takes $12N+1$ cycles to add two $N$-bit numbers.

Figure 6 shows the in-memory execution of RNS addition. The result of in-memory addition of the two inputs is stored in Result Row 1 (RR1). This addition requires $12t+1$ cycles for the addition of inputs corresponding to modulis $2^t-1$ and $2^t$, and $12*(t+1)+1$ cycles for the $2^t+1$ moduli. Then $-$(moduli) is added to RR1 and the result is stored in RR2. This addition take the same number of cycles as in the previous step, resulting in effective latency of $2*(12t+1)$ and $2*(12t+13)$ respectively. Then the sign bit of RR2 is read out. If the read bit is '1,' RR2 is selected otherwise RR1 is selected. The selected row is copied to RR3, which acts as the output of RNS addition. The memory also contains 11 additional rows, which

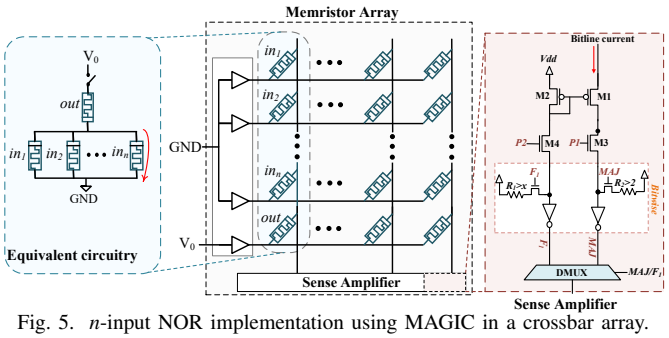


Fig. 5. $n$-input NOR implementation using MAGIC in a crossbar array.

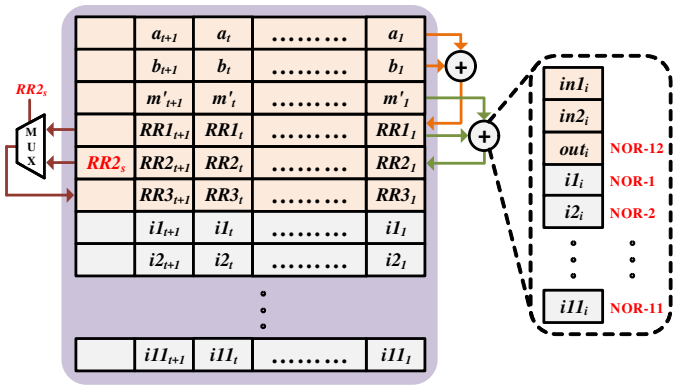


Fig. 6. In-memory execution of RNS addition

are required to store the intermediate output of MAGIC-based addition.

### B. RNS Multiplication

The RNS multiplication discussed in Section III-C simplifies RNS multiplication $a \times b$ to an equivalent term $\frac{(a+b)^2}{4} - \frac{(a-b)^2}{4}$. This can be implemented using additions and memory accesses. First, we perform in-memory RNS addition $(a+b)$ and subtraction $(a-b)$ for each residue. We have two data memories where each address location, $k$, stores the value $k^2/4$. The values $a+b$ and $a-b$ are used to address the memories, which generates $\frac{(a+b)^2}{4}$ and $\frac{(a-b)^2}{4}$. Now, as explained in Section III-C, we accumulate $\frac{(a_i+b_i)^2}{4}$ and $\frac{(a_i-b_i)^2}{4}$ for all the residues. Then we subtract the two accumulated results to obtain $\Sigma(\frac{(a+b)^2}{4} - \frac{(a-b)^2}{4})$. This is followed by RNS ranging, which converts the accumulated values back to RNS domain.

*1) Parallelism in Multiplication:* Since, the operations corresponding to all the residues are same, we discuss the implementation for the $2^t+1$ residue. The same implementation is used for other residues as well. As explained previously, each RNS addition/subtraction takes $t_{add} = 2*(12t+13)$ cycles. Many such additions/subtractions, say $p$, can be parallelized, generating $p$ outputs at the end of $t_{add}$ cycles. This can be done with no extra latency overhead since MAGIC inherently supports in-memory parallelism [52].

Then, each output is used to read a value from the data memory, which is then stored in the accumulator. This requires three cycles in total, one cycle to access the data memory and two cycles to write the read value in the accumulator. For $p$ outputs, it takes $t_{ma\_p} = 3*p$ cycles.

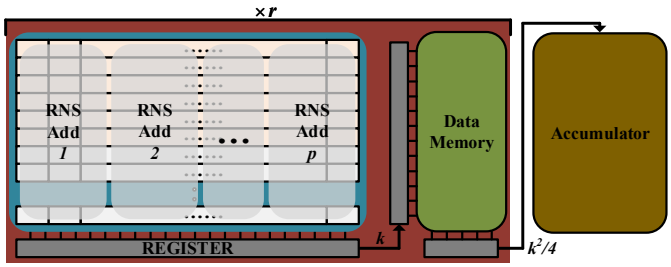


Fig. 7. In-memory execution of RNS multiplication

**Accumulator** The accumulator adds the $p$ outputs of data memory. PIM addition is slow when adding several numbers together. This PIM inefficiency comes from slow carry propagation among all bits when adding all three inputs together. Our design exploits the idea of carry save adder to add inputs together in a tree structure [53, 54]. In carry save adder, the inputs are added together in groups of three without propagating the carry bit, generating *two outputs for three inputs*.

For example, when we add nine numbers together (e.g. 32-bit each), we require three adder groups to add such numbers together in the first stage. In the second stage, the results of adder blocks are again added together in a tree structure. This continues until the numbers to be added reduces to 2. The last stage adds the final two numbers while propagating the carry bit. All these stages except the last stage can perform in-memory addition in 13 cycles each. The adder in last stage of carry save adder needs to propagate the carry bit, and thus it requires $12N+1$ cycles, where $N$ is the size of the final two numbers. The work in [53] details the hardware execution of this adder.

The number of carry save addition (3:2 reduction) stages required are given by $\lceil log_{3/2}p \rceil$, each requiring 13 cycles. Addition of $p$ numbers of size $t+1$ bits each results in two outputs of size $y = (t+1)+\lceil log_2 p \rceil$ bits. The last stage hence requires $12*(y)+1$ cycles for generating the final output. In total, the accumulator requires $t_{acc\_p} = 13*\lceil log_{3/2}p \rceil + 12*(y)+1$.

*2) Pipelining Multiplication:* The above discussion implies that the latencies of the three major parts of RNS multiplication behave with $p$ in different ways. The latency of RNS addition remains constant, while the latency of memory access increases linearly, and that of the accumulator increases logarithmically. In this subsection, we present a multi-level pipelined design, as shown in Figure 7, which increases the throughput of the design, while balancing the different parts of the multiplication.

First, we balance the time taken by RNS addition and memory access stages. We select $p$ such that $t_{ma\_p} \approx t_{add}$, giving $p = \lfloor t_{add}/3 \rfloor$. This ensures that the time taken by the memory access stage is almost equal to the RNS addition stage. A register of size $p \times (t+1)$ is added after RNS addition block. After every RNS addition, the $p$ outputs are stored in the register. Then, the RNS addition block operates on the next batch of inputs. In parallel, the memory access stage operates on the values stored in the register. Hence, except the first iteration, after every $t_{add}$ cycles, the accumulator contains

TABLE II
NEURAL NETWORK MODELS AND BASELINE ERROR RATES FOR 4 APPLICATIONS

| Dataset | Network Topology | Error |
|---|---|---|
| **MNIST** | 784, 512, 512, 10 | 1.5% |
| **ISOLET** | 617, 512, 512, 26 | 3.6% |
| **INDOOR** | 520, 512, 512, 13 | 4.2% |
| **CIFAR-10** | $32\times32\times3$, *Conv* : $32\times3\times3$, *Pooling* : $2\times2$, *Conv* : $64\times3\times3$, *ConvV* : $64\times3\times3$, 512, 10 | 14.4% |

a new set of $p$ values. Now, $t_{acc\_p}$ is significantly greater than $t_{ma\_p}$. Instead, we accumulate $q$ values together such that $t_{acc\_q} \approx t_{ma\_p} * \lceil q/p \rceil$. This requires $r = \lceil q/p \rceil$ iterations of the $p$-input RNS addition and memory access stages.

This is followed by the subtraction of the two accumulated values as discussed in Section III-C. The result is then converted to RNS domain. Since, this happens only once for a neuron, we implement it in periphery using CMOS logic. Similarly, the activation function is also implemented using simple peripheral circuits.

## V. EXPERIMENTAL RESULTS

### A. Experimental Setup

We exploit HSPICE design tool for circuit-level simulations and calculate energy consumption and performance of all the RNSnet memory blocks. The energy consumption and performance are also cross-validated using NVSIM [55]. We adopt memristor device with a large OFF/ON resistance [56] for the memory devices. The robustness of all proposed circuits has been verified by considering 10% process variations on the size and threshold voltage of transistors using 5000 Monte Carlo simulations. The RNSnet controller has been designed using System Verilog and synthesized using *Synopsys Design compiler* in 45nm TSMC technology. We compare the proposed RNSnet accelerator with GPU-based DNN implementations, running on NVIDIA GPU GTX 1080. The performance and energy of GPU are measured by the `nvidia-smi` tool.

Tensorflow [57] is used to realize the NN in the customization unit. We evaluate RNSnet on four applications described below.

**Handwritten Digits (MNIST):** MNIST is a popular machine learning data set including images of handwritten digits [58]. The objective is to classify an input picture to one of the ten digits $\{0 \ldots 9\}$.

**Voice Recognition (ISOLET):** Many mobile applications require online processing of vocal data. We evaluate RNSnet with the Isolet dataset [59] which consists of speech collected from 150 speakers. The goal of this task is to classify the vocal signal to one of the 26 English letters.

**Indoor Localization (INDOOR)** [60]: We designed a DNN model for the indoor localization dataset. This DNN localizes into one of 13 places where there is a high loss in GPS signals.

**Object Recognition (CIFAR)** [61]: CIFAR-10 dataset includes 50000 training and 10000 testing images belonging to 10 classes. The goal is to classify an input image to the correct category, e.g., animals, airplane, automobile, ship, truck, etc.

We use stochastic gradient descent with momentum [62] to train each network. The momentum is set to 0.1, the learning rate is set to 0.001, and a batch size of 10 is used. Table II presents the baseline NN Topologies and their error rates running four applications. The activation functions are set to "Rectified Linear Unit" clamped at 6. A "Softmax" function is applied to the output layer.

### B. RNSnet Exploration

Although machine learning algorithms such as neural networks require the precision of floating point unit in training phase, in inference, these algorithms are working with fixed point values. Even in integer values, reducing the number bit width, from 32-bit, usually does not affect the final classification accuracy. Figure 8 shows the classification accuracy of different neural network applications to the bit width. As result shows, applications have different sensitivity to reducing the bit width. For instance, MNIST application works with maximum accuracy when the bit width of numbers is reduced to 12. In this configuration, the MNIST using 12-bit numbers can provide 5.7× and 4.9× higher energy efficiency improvement and speedup as compared to the network with full 32-bit values. Figure 8 depicts the effects of reducing the bit width on energy efficacy and the performance of applications. The bottom x-axis shows the bit width of numbers which sweeps from 24-bit to 4-bit and the top x-axis shows the Quality Loss of applications in the corresponding bit width. The y-axis shows the energy efficiency improvement and performance speedup of each application using different bit width as compared to GPU core. Our evaluation shows that using 16-bit numbers, leads to 145.5×, and 35.4 energy efficacy improvement and speed up respectively, while providing the maximum classification accuracy. Accepting less than 1% (2%) quality loss, improves the energy efficiency and speedup to 188.7× and 42.1× (202.3× and 59.7×) respectively.

Reducing the bit width not only improves the performance of the RNSnet, but also reduces the memory requirement. Table III shows the average memory requirement of each neuron in RNSnet when the bit width changes. In general, decreasing the bit width of the binary numbers reduces the required dynamic range; therefore, we can represent the binary numbers in RNS with fewer bits. For instance, for implementing 16-bit binary numbers we select $\{2^6-1, 2^6, 2^6+1\}$ $(t = \lceil \frac{16}{3} \rceil = 6)$ as our ModuliSet. While, for 12-bit binary number we use $ModuliSet = \{2^4-1, 2^4, 2^4+1\}(t=4)$, since we need less dynamic range to represent a 12-bit binary number than a 16-bit number. The memory required by RNSnet consists of two parts: (i) memory required to implement the multiplication. We pre-store the square of RNS numbers to implement the multiplication and therefore the size of this memory is doubled when the bit width of RNS increments by one. (ii) the memory required to implement the activation function. The number of rows in this memory is equal to $(2^t-1)\times(2^t+1)$.

Our result shows that using 24 bits numbers, RNSnet models each neuron in average using 43.4KB memory size. Reducing the bit width of numbers to 16-bit improves the neuron memory efficiency to 10.1KB while all applications

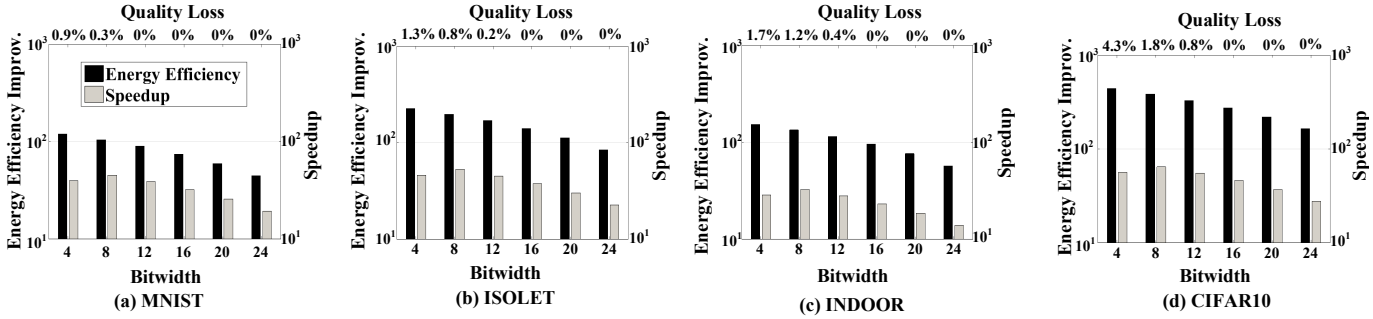


Fig. 8. Energy efficiency improvement, speedup and quality loss of the RNSnet in different bit width.

TABLE III
AVERAGE MEMORY REQUIREMENT OF EACH NEURON IN DIFFERENT APPLICATIONS USING DIFFERENT BIT WIDTH

| | 4-bit | 8-bit | 12-bit | 16-bit | 20-bit | 24-bit |
|---|---|---|---|---|---|---|
| **MNIST** | 0.24KB | 0.76KB | 1.9KB | 11.2KB | 26.8KB | 49.7KB |
| **ISOLET** | 0.22KB | 0.71KB | 1.8KB | 10.5KB | 25.0KB | 46.1KB |
| **INDOOR** | 0.21KB | 0.67KB | 1.7KB | 10.1KB | 23.9KB | 44.7KB |
| **CIFAR-10** | 0.18KB | 0.59KB | 1.3KB | 8.6KB | 17.1KB | 33.2KB |

can provide the maximum classification accuracy. Accepting less than 1% quality loss on applications, RNSnet can further reduce memory requirement to 1KB. Note that the RNSnet memory requirement is negligible as compared to prior RNS works. For instance, work in [37] used 5,681,237 to model a neuron with 12 inputs. In addition, their memory requirement increases exponentially with the number of inputs, while RNSnet memory linearly depends on the number of inputs.

### C. RNS Efficiency

In this section we compare the energy consumption and execution time of RNSnet with DaDianNao [63] and ISAAC [7]. All designs have been tested over four different applications. For NN accelerators, we select the best configuration reported in the papers [7, 63]. For instance, ISAAC design works at 1.2GHz and uses 8-bit ADC, 1-bit DAC, 128×128 array size where each memristor cell stores 2 bits. DaDianNao works at 600MHz, with 36MB eDRAM size (4 per tile), 16 neural functional units, and 128-bit global bus. We see that of the previously proposed designs, ISAAC performs better over all datasets.

Table IV lists the energy improvement and speed up of DaDianNao, ISAAC and proposed design RNSnet as compared to GPU architecture. The result shows that DaDianNao can speeds up the GPU computation by 10.6× in average over four tested applications. Our evaluation shows that RNSnet execute 3.7× faster and 8.4× more energy efficient as compared to DaDianNao. This efficiency comes because DaDianNao does not totally address data movement issue. Similarly, as compare to ISAAC, the proposed RNSnet can provide 2.5× and 1.6× higher energy efficiency and speedup respectively. The advantage of RNSnet to ISAAC comes from RNSnet digital based operation which removes the necessity of using ADC and DAC blocks.

Accepting less than 1% quality loss, RNSnet energy efficiency and speedup improve by 5×, 2.6× thanks to using lower bit width values. Additionally, our RNSnet @1% ex-

TABLE IV
ENERGY EFFICIENCY IMPROVEMENT AND SPEEDUP OF THE PROPOSED DESIGN AND OTHER NEURAL NETWORK ACCELERATORS (GPU=1).

| Applications | | MNIST | ISOLET | INDOOR | CIFAR10 |
|---|---|---|---|---|---|
| **DaDiano** | *Energy Improv.* | 13.3× | 17.2× | 14.4× | 24.6× |
| | *Speedup* | 11.0× | 12.8× | 5.9× | 8.4× |
| **ISAAC** | *Energy Improv.* | 33.3× | 51.5× | 44.7× | 98.6× |
| | *Speedup* | 26.5× | 30.6× | 14.0× | 20.1× |
| **RNSnet** | *Energy Improv.* | 71.5× | 140.9× | 95.6× | 273.9× |
| | *Speedup* | 34.5× | 37.8× | 23.2× | 46.1× |
| **RNSnet @1%** | *Energy Improv.* | 114.5× | 197.2× | 114.6× | 328.7× |
| | *Speedup* | 35.2× | 45.3× | 32.6× | 55.3× |

TABLE V
COMPARISON OF DEEP AND BINARIZED NEURAL NETWORK IN TERMS OF ACCURACY AND EFFICIENCY.

| | DNN | | | BNN | | |
|---|---|---|---|---|---|---|
| | **Accuracy** | **Training (s)** | **Testing (s)** | **Accuracy** | **Training (s)** | **Testing (s)** |
| **MNIST** | 98.5% | 13.2 | 0.1712 | 98.0% | 157.0 | 0.0176 |
| **ISOLET** | 97.3% | 3.9 | 0.1379 | 94.8% | 44.3 | 0.0210 |
| **INDOOR** | 96.4% | 3.7 | 0.2324 | 95.6% | 53.8 | 0.0215 |
| **CIFAR-10** | 85.6% | 125.2 | 0.6921 | 83.4% | 2035.1 | 0.1841 |

ecutes the application 1.3× more energy efficient and 1.2× faster than our RNSnet. Increasing the network size, significantly, increases the computation cost of the neural network on conventional GPU core, however the RNSnet energy and performance scales linearly with network size. Therefore, we expect to see much higher efficiency of RNSnet on applications using large network.

Table V compares the classification accuracy and execution time of deep neural network and Binarized Neural Networks (BNNs). The DNNs and BNNs are trained for 10 and 100 epochs respectively. The timing results are reported for NVIDIA GTX 1080 GPU. Our evaluation shows that although BNNs are efficient in testing, they are very slow in training. As results in Table V shows, BNNs trained for 10 times more number of epochs still cannot provide the similar accuracy that DNN provides. In contract, in this paper we proposed RNSnet, which can provide the same accuracy as DNN using fixed point values.

### D. Energy-Execution Breakdown

Figure 9 shows the breakdown of the energy and execution time of RNSnet over different applications. Our result shows that in network using fully connected layers the weighted accumulations (multiplication and accumulations) takes 73% and 84% of DNN energy and execution time. After that, activation function is taking 21% and 11% of energy and execution of the DNN. Looking at the overhead of RNS conversion, we

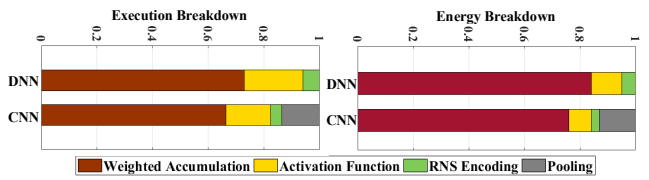

Fig. 9. Breakdown of RNSnet energy consumption and execution time on differed applications.

observe that this conversion takes less than 6% of total energy and execution of the neural network. In CNN network with convolutional layers, the Pooling takes comparable energy and execution time as compared to activation function. However, the cost of weighted-accumulation still dominate the total DNN/CNN energy consumption and execution time.

## VI. Conclusion

Our proposed RNSnet takes the advantage of using Residue Number System (RNS) to accelerate deep neural network applications. Our approach addresses the data movement issue by locally processing the data inside the memory blocks. Thanks to the value representation in RNS, RNSnet simplifies all DNN operations and compute them using memory-friendly operations such as addition and memory access. We evaluate the efficiency of the proposed RNSnet on wide range of DNN applications. Our evaluations show that RNSnet can perform 35.4× faster and 145.5× more energy efficient as compared to GPU implementation.

## Acknowledgements

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

## References

[1] S. Lin *et al.*, "Fft-based deep learning deployment in embedded systems," in *DATE*, pp. 1045–1050, IEEE, 2018.

[2] M. A. Bhuiyan *et al.*, "Acceleration of spiking neural networks in emerging multi-core and gpu architectures," in *IPDPSW*, pp. 1–8, IEEE, 2010.

[3] S. Hashemi, N. Anthony, H. Tann, R. Bahar, and S. Reda, "Understanding the impact of precision quantization on the accuracy and energy of neural networks," in *Proceedings of the Conference on Design, Automation & Test in Europe*, pp. 1478–1483, European Design and Automation Association, 2017.

[4] X. Jiao, V. Akhlaghi, Y. Jiang, and R. K. Gupta, "Energy-efficient neural networks using approximate computation reuse," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 1223–1228, IEEE, 2018.

[5] S. Salamat, M. Ahmadi, B. Alizadeh, and M. Fujita, "Systematic approximate logic optimization using don't care conditions," in *Quality Electronic Design (ISQED), 2017 18th International Symposium on*, pp. 419–425, IEEE, 2017.

[6] P. Chi *et al.*, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 27–39, IEEE Press, 2016.

[7] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, pp. 14–26, IEEE Press, 2016.

[8] X. Yin *et al.*, "Exploiting ferroelectric fets for low-power non-volatile logic-in-memory circuits," in *ICCAD*, p. 121, ACM, 2016.

[9] M. Imani *et al.*, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.

[10] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *DATE*, pp. 1779–1784, IEEE, 2017.

[11] M. Zhou *et al.*, "Gas: A heterogeneous memory architecture for graph processing," in *ACM ISLPED*, p. 27, ACM, 2018.

[12] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*, pp. 25–32, IEEE, 2017.

[13] S. Saransh *et al.*, "Felix: Fast and energy-efficient logic in memory," *IEEE/ACM ICCAD*, 2018.

[14] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *HPCA*, pp. 445–456, IEEE, 2017.

[15] A. Aziz *et al.*, "Computing with ferroelectric fets: Devices, models, systems, and applications," in *DATE*, pp. 1289–1298, IEEE, 2018.

[16] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *DATE*, pp. 373–378, IEEE/ACM, 2016.

[17] T. Serrano-Gotarredona *et al.*, "Stdp and stdp variations with memristors for spiking neuromorphic learning systems," *Frontiers in neuroscience*, vol. 7, p. 2, 2013.

[18] L. Y. Lam and T. S. Ang, "Fleeting footsteps: Tracing the conception of arithmetic and algebra in ancient china," *World scientific*, 2004.

[19] S. Gupta *et al.*, "Deep learning with limited numerical precision," in *ICML*, pp. 1737–1746, 2015.

[20] M. Nazemi *et al.*, "A hardware-friendly algorithm for scalable training and deployment of dimensionality reduction models on fpga," *arXiv preprint arXiv:1801.04014*, 2018.

[21] V. Akhlaghi *et al.*, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," ISCA, 2018.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, pp. 525–542, Springer, 2016.

[23] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International*

*Symposium on Field-Programmable Gate Arrays*, pp. 65–74, ACM, 2017.

[24] K. Ando, K. Ueyoshi, K. Orimo, H. Yonekawa, S. Sato, H. Nakahara, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, T. Kuroda, *et al.*, "Brein memory: A single-chip binary/ternary reconfigurable in-memory deep neural network accelerator achieving 1.4 tops at 0.6 w," *IEEE Journal of Solid-State Circuits*, 2017.

[25] X. Chen *et al.*, "Design and optimization of fefet-based crossbars for binary convolution neural networks," in *DATE*, pp. 1205–1210, IEEE, 2018.

[26] P. A. Mohan, "Binary to residue conversion," in *Residue Number Systems*, pp. 27–38, Springer, 2016.

[27] K. Anitha, T. Arulananth, R. Karthik, and P. B. Reddy, "Design and implementation of modified sequential parallel rns forward converters," *International Journal of Applied Engineering Research*, vol. 12, no. 16, pp. 6159–6163, 2017.

[28] P. M. Matutino, R. Chaves, and L. Sousa, "Arithmetic-based binary-to-rns converter modulo $2^n + k$ for jn-bit dynamic range," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 3, pp. 603–607, 2015.

[29] E. Gholami, R. Farshidi, M. Hosseinzadeh, and K. Navi, "High speed residue number system comparison for the moduli set {2 (superscript n)-1, 2 (superscript n), 2 (superscript n)+ 1}," *Journal of communication and computer*, vol. 6, no. 3, pp. 40–46, 2009.

[30] A. A. Hiasat and S. Abdel-Aty-Zohdy, "Residue-to-binary arithmetic converter for the moduli set (2/sup k/, 2/sup k/-1, 2/sup k-1/-1)," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 45, no. 2, pp. 204–209, 1998.

[31] J. Liu *et al.*, "Optimizing residue number system on fpga," in *iThings and GreenCom and CPSCom and SmartData*, IEEE, 2016.

[32] A. R. Omondi and B. Premkumar, *Residue number systems: theory and implementation*. World Scientific, 2007.

[33] R. Chaves and L. Sousa., "A risc dsp based on residue number system," in *Digital System Design, 2003. Proceedings. Euromicro Symposium on*, IEEE, 2003.

[34] D. Younes and P. Steffan, "Efficient image processing application using residue number system," in *MIXDES*, IEEE, 2013.

[35] G. C. Cardarilli *et al.*, "Residue number system for low-power dsp applications," in *ACSSC*, IEEE, 2017.

[36] H. Nakahara and T. Sasao, "A deep convolutional neural network based on nested residue number system," in *FPL*, IEEE, 2015.

[37] G. Alia and E. Martinelli, "Neurom: a rom based rns digital neuron," *Neural networks 18.2*, pp. 179–189, 2005 organization=Elsevier.

[38] S. Duquesne and N. Guillermin, "A fpga pairing implementation using the residue number system," *IACR Cryptology ePrint Archive 2011*, 2011.

[39] J. James and A. Pe, "A novel method for error correction using redundant residue number system in digital communication systems," in *Advances in Computing, Communications and Informatics (ICACCI), 2015 International Conference on*, pp. 1793–1798, IEEE, 2015.

[40] T. F. Tay and C.-H. Chang, "A new algorithm for single residue digit error correction in redundant residue number system," in *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pp. 1748–1751, IEEE, 2014.

[41] T. F. Tay and C.-H. Chang, "Fault-tolerant computing in redundant residue number system," in *Embedded Systems Design with Special Arithmetic and Number Systems*, pp. 65–88, Springer, 2017.

[42] B. Deng, S. Srikanth, E. R. Hein, P. G. Rabbat, T. M. Conte, E. DeBenedictis, and J. Cook, "Computationally-redundant energy-efficient processing for y'all (creepy)," in *Rebooting Computing (ICRC), IEEE International Conference on*, pp. 1–8, IEEE, 2016.

[43] R. de Matos *et al.*, "Efficient implementation of modular multiplication by constants applied to rns reverse converters," in *ISCAS*, IEEE, 2017.

[44] R. o. Dhanabal, "Implementation of floating point mac using residue number system," in *ICROIT*, IEEE, 2014.

[45] Chervyakov *et al.*, "Fast modular multiplication execution in residue number system," in *ITMQIS*, IEEE, 2016.

[46] E. B. Olsen, "Rns hardware matrix multiplier for high precision neural network acceleration:" rns tpu"," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.

[47] T. Manabe, Y. Shibata, and K. Oguri, "Fpga implementation of a real-time super-resolution system with a cnn based on a residue number system," in *Field Programmable Technology (ICFPT), 2017 International Conference on*, pp. 299–300, IEEE, 2017.

[48] H. Nakahara and T. Sasao, "A high-speed low-power deep neural network on an fpga based on the nested rns: Applied to an object detector," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.

[49] V. Krasnobayev, A. Yanko, and S. Koshman, "A method for arithmetic comparison of data represented in a residue number system," *Cybernetics and Systems Analysis*, vol. 52, no. 1, pp. 145–150, 2016.

[50] L. Sousa and P. Martins, "Sign detection and number comparison on rns 3-moduli sets \{2^ n-1, 2^{n+ x}, 2^ n+ 1\} {2 n-1, 2 n+ x, 2 n+ 1}," *Circuits, Systems, and Signal Processing*, vol. 36, no. 3, pp. 1224–1246, 2017.

[51] S. a. o. Kvatinsky, "Magicmemristor-aided logic," *TCAS II*, vol. 61, no. 11, pp. 895–899, 2014.

[52] N. Talati *et al.*, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.

[53] M. Imani *et al.*, "Ultra-efficient processing in-memory for data intensive applications," in *DAC*, p. 6, ACM,

2017.
[54] M. Imani *et al.*, "Efficient query processing in crossbar memory," in *Low Power Electronics and Design (ISLPED, 2017 IEEE/ACM International Symposium on*, pp. 1–6, IEEE, 2017.
[55] X. Dong *et al.*, "Nvsim: A circuit-level performance, energy, and area model for emerging non-volatile memory," in *Emerging Memory Technologies*, pp. 15–50, Springer, 2014.
[56] S. Kvatinsky *et al.*, "Vteam: A general model for voltage-controlled memristors," *TCAS II*, vol. 62, no. 8, pp. 786–790, 2015.
[57] M. Abadi *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.
[58] Y. LeCun *et al.*, "The mnist database of handwritten digits," 1998.
[59] "Uci:." http://archive.ics.uci.edu/ml/datasets/ISOLET.
[60] "Uci:." https://archive.ics.uci.edu/ml/datasets/UJIIndoorLoc.
[61] "The cifar dataset." https://www.cs.toronto.edu/~kriz/cifar.html.
[62] I. Sutskever *et al.*, "On the importance of initialization and momentum in deep learning.," *ICML (3)*, vol. 28, pp. 1139–1147, 2013.
[63] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *Micro*, pp. 609–622, IEEE, 2014.