Applying the Residue Number System to Network Inference

Mohamed Abdelhamid MRHAMID@MIT.EDU
Skanda Koppula SKOPPULA@MIT.EDU

1. Introduction

Use of neural networks for computer vision, speech recognition, and other applications has exploded in recent years, in part due to their unprecedented performance on a variety of benchmarks. Nonetheless, high-throughput and energy-efficient evaluation of such neural networks, and in particular, convolutional neural networks (CNNs), remains an active field of research. Evaluation of networks is memory and compute intensive, with the bottleneck depending on the network topology and layer types (convolutional or fully-connected [FCN]).

Significant effort has been made to reduce the memory footprint of neural networks, motivated by the fact that many off-chip memory accesses can dominate energy consumption during evaluation (Zhou et al., 2016; Han et al., 2015; Courbariaux & Bengio). This work explores the lesser studied objective of optimizing the multiply-and-accumulates executed during evaluation of the network. In particular, we propose using the Residue Number System (RNS) as the internal number representation across all layer evaluations, allowing us to explore usage of the more power-efficient RNS multipliers and adders. We motivate our optimization with Table 1, which summarizes the large number of multiply-and-accumulates (MACs) required during evaluation of popular networks. Small improvements to the underlying efficiency of the core multiply and accumulate block can have large improvements to the overall network evaluation.

Table 1. Computation Accounting for Popular Networks

Net	MACs (10^6)	Params (10^6)
AlexNet	720	60
GoogLeNet	1550	6.8
SqueezeNet	1700	1.25
VGG-16	15300	138

Prior work applying RNS for efficient computation has largely focused on cryptographic applications and general purpose ALUs. In the machine learning domain, various optimizations such as Winograd and FFT transforms have been proposed to speed up network inference. As far as we are aware, this is the first attempt at applying RNS to neural network inference.

In section 2, we describe RNS and the requisite operations and methodology for RNS-based inference. Sections 3, 4, 5, and 6.1 details our implementation of core RNS hardware modules in Verilog, and simulation results with estimated power, area, and latency. In section 6.2, we describe accuracy estimates with our chosen RNS precision. In section 6.3, we tie together our results with an analysis of the break-even points at which it becomes beneficial to use RNS-based inference for low-power network evaluation. We conclude with a critique of our contributions.

All our Tensorflow, Bluespec/Verilog, and software model of digital hardware source code is available for re-use at https://github.mit.edu/mrhamid/6888_Project.

2. Preliminaries

2.1. Residue Number System

At its core, the Residue Number System relies on the Chinese Remainder Theorem (CRT) to represent a large integer as a tuple of smaller integers. This allows for more smaller and more parallelized arithmetic blocks. In RNS, an integer X mod M with moduli set $\{m_1, m_2, \ldots, m_k, \ldots m_n\}$ is represented as $(x_1, \ldots, x_n)$ where

$$x_k \equiv X \bmod m_k$$

The set of moduli is carefully chosen. In particular, the moduli are usually co-prime to reduce the number of distinct numbers that have the same RNS representation, and in our case, also chosen for hardware efficiency of operations. In this work, we fix our moduli to the structured 4-tuple $\{2^n \pm 1, 2^{n+1} \pm 1\}$. This set can represent numbers in range $[0, M = \frac{(2^{2n}-1)(2^{2n+2}-1)}{3}]$ (Sousa, 2007). We choose the $n = 7$ moduli set in this work. Every RNS number is stored using $7 + 8 + 8 + 9 = 32$ bits, and can fall in the range $[0, \frac{(2^{14}-1)(2^{16}-1)}{3} = 357886635]$. This is the representational range of a 28-bit unsigned integer.

Addition of two RNS numbers r^1 and r^2 occurs element-wise: $r^s = r^1_k + r^2_k \bmod m_k$. Multiplication is similarly elementwise: $r^p = r_k^1 \times r_k^2 \mod m_k$. Conversion in and out of RNS is based on CRT Theorems I, II, and III. Unfortunately, with the requisite divisions and iterative algorithms, conversion requires significant arithmetic overhead, as discussed by (Hiasat, 2003).

2.2. Piecing Together RNS Inference

At its core, to construct complete end-to-end inference in RNS we need to support two operations: multiply-and-accumulate and ReLU. Our choice of moduli allow us to build on prior work for both: (Zimmermann, 1999) proposes digital architecture for multiplication and addition $\mod 2^n \pm 1$, and (Sousa, 2007) demonstrates comparison of RNS numbers $\mod 2^n \pm 1$ in VHDL. We use a comparison module to implement the ReLU nonlinearity.

We assume a discrete output for our network – e.g. 10-output image-recognition with CIFAR-10. This allows us to avoid the overhead of conversion out of RNS at the end of the network; instead, with our comparator, we compute a *max* over final layer softmax values, returning the index with the maximum value. All RNS operations occur in the realm of positive integers with fixed modulus. We explore this constraint in Section 6.2.

3. Comparison in RNS

Though RNS multiplication and addition are operationally intuitive, comparing two RNS numbers is non-trivial. We follow the procedure given by (Sousa, 2007). The crux of the algorithm is reducing comparison to parity (even/odd) checking. To compare two unsigned integers $A, B \mod M$, we compute the difference $C = A - B$ which becomes one of two values:

$$C = \begin{cases} A - B & \text{if } A \geq B \\ M + A - B & \text{if } A < B \end{cases}$$

Because with our chosen moduli M is odd, these two values have different parities. As such, we can compute a comparison given a formula for the $\mod 2$ parity X_P of an RNS number $X = (x_1, x_1^*, x_2, x_2^*)$. Parity is calculated with the following set of equations:

$$X_1 = x_1^* + (2^n + 1) \times (2^{n-1}(x_1 - x_1^*) \mod 2^n - 1)$$
$$X_2 = x_2^* + (2^{n+1} + 1) \times (2^n(x_2 - x_2^*) \mod 2^{n+1} - 1)$$
$$X_P = LSB(X_2) \oplus LSB((X_1 - X_2) \mod 2^{2n} - 1)$$

Proof is provided by (Sousa, 2007). This is the *full-comparator* we implement in combinational logic to execute the final layer maximum. In the ReLU, we are able to trim this combinational logic, because we compare with a fixed threshold $\frac{M}{2}$ (0 in our modulus world). We call this trimmed comparator a *half-comparator*. In particular, the parity of $B = \frac{M}{2}$ is fixed and pre-computed, as well as the value of the additive inverse $-B = -\frac{M}{2}$ we feed into the modulo adder.

The parity-checking combinational logic implemented in Verilog is given by Figure 1. This is used in both the full and half comparator design. We modify the circuit given in (Sousa, 2007), which we suspect, from our testing, does not evaluate correctly in an exhaustive sweep of all ∼3 billion inputs.

We include various optimizations. For example, our choice of modulus allows for use of an inverter to find the additive inverse. It also allows for calculating the remainder with a 16 bit number as a single addition (bottom-right). Additionally, we implement multiplication with $2^n \mod 2^{n+1} - 1$ as a right rotate.

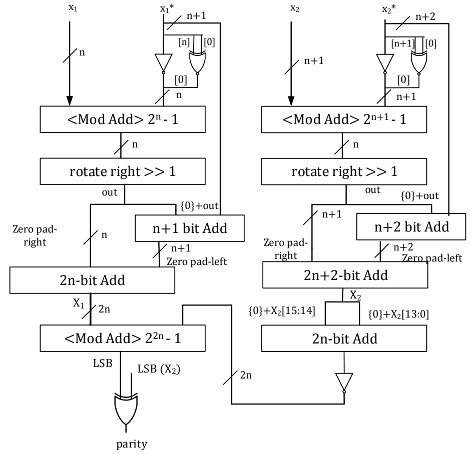

Figure 1. Combinational logic for calculating the parity of an RNS number (x_1, x_1^*, x_2, x_2^*)

4. Converting Input into RNS

For the chosen moduli set, the residue generation relies on the periodicity of the binary weights (2^i) in the modulus domain. As pointed out by (Piestrak, 1994), the residue of a binary number is given by:

$$X \mod (2^{n_1} - 1) = \sum_{i=0}^{n-1} 2^i x_i \mod (2^{n_1} - 1)$$

taking the modulus causes the binary weights to repeat themselves when $i > n_1$. Therefore, the residue is calculated by periodically folding back the higher weights and adding them to the lower weights using a tree of modulo adders.

5. RNS Arithmetic

CNNs perform a large number of computations such as multiplication and accumulation throughout the convolutional as well as the fully connected layers. To fully exploit the advantages of transforming the network to the RNS system, efficient modulo arithmetic circuits have to be designed. This section presents efficient modulo multiplication and addition implementation using the same moduli set as the comparison, namely $\{2^{n_1} \pm 1,\ 2^{n_2} \pm 1\}$.

5.1. Modulo Addition

The main advantage of RNS arithmetic is that each residue operates separately in parallel without any carry propagation from one residue to the other. Our conjugate moduli set requires modulo $(2^n - 1)$ addition/multiplication as well as $(2^n + 1)$.

First, the modulo $(2^n - 1)$ addition can be expressed as conventional n-bit addition if the sum is less than the modulus, while a correction is added if the sum overflows the modulus as follows:

$$(A + B) \bmod (2^n - 1) = \begin{cases} A + B - (2^n - 1) & \\ \quad = A + B + 1 & \text{if A+B} \geq 2^n - 1, \\ A + B & \text{otherwise} \end{cases}$$

Since, the output carry (c_{out}) of an n-bit adder can be used to detect the overflow condition which determines whether to increment the sum or not, then such carry can be fed back into the adder as proposed by (Zimmermann, 1999) for $(2^n - 1)$ addition.

$$(A + B) \quad \bmod (2^n - 1) = (A + B + c_{out}) \quad \bmod 2^n$$

Second, a similar analysis can be done for the modulo $(2^n + 1)$ addition to show that

$$(A + B + 1) \quad \bmod (2^n + 1) = (A + B + \overline{c_{out}}) \quad \bmod 2^n$$

where diminished-1 numbers can be used for the inputs, or a correction circuit is added to the output to account for the extra '1'.

Since the addition in both moduli depends on the output carry, then fast parallel prefix adders are the most suitable implementation for the modulo adders. Figure 2 shows the a modulo parallel prefix adder where the inputs are preprocessed into carry generate and propagate signals then a tree of fixed operation propagates the carry in only 4 levels. Each solid circle represents a dot operation which combines the group carry generate and propagate bits. Finally, a modulo end-around carry correction is required to feedback back the output carry (c_{out}) or its complement $(\overline{c_{out}})$ according to the designated modulus.

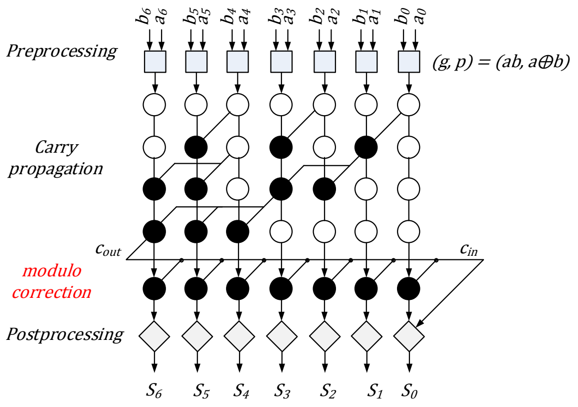

Figure 2. Parallel prefix adder for mod $2^7 - 1$ addition

5.2. Modulo multiplication

N-bit binary multiplication relies on generating N partial products and accumulating them all to produce the final product. Modulo multiplication relies on the same concept as well as the periodicity of the binary weights which causes the higher order partial products to rotate folding back into n-bit weights. Therefore, the partial products can be generated, similar to in (Zimmermann, 1999), as

$$X \cdot Y \quad \bmod (2^n - 1) = \sum_{i=0}^{n-1} x_i \cdot (Y << i) \quad \bmod (2^n - 1)$$

where the $<<$ operator represents a circular shift. Similarly, an expression for the partial products of the $(2^n + 1)$ modulus can be derived to be

$$PP_i = x_i \cdot y_{n-i-1} \cdots y_0 y_n \cdots y_{n-i} + \overline{x_i} \cdot 0 \cdots 01 \cdots 1$$

Such multipliers can be designed in a modular way where a block generates the required partial products according to the selected modulus. Then, a modulo carry save adder tree as shown in Figure 3 generates a redundant sum output (P_C, P_S) which is then added using a modulo adder to produce the final product.

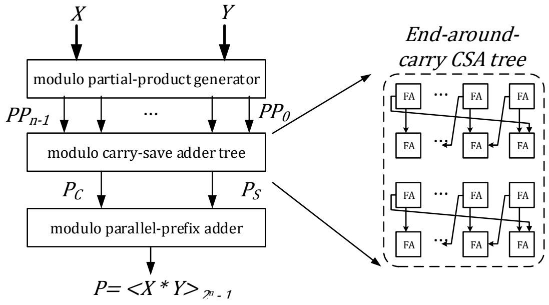
X
Y
modulo partial-product generator
PP_{n-1}
...
PP_0
modulo carry-save adder tree
P_C
P_S
modulo parallel-prefix adder
P= <X * Y>_{2^n - 1}
End-around-carry CSA tree
FA

Figure 3. Mod $2^n - 1$ multiplier architecture showing the modulo carry save adder

6. Results

6.1. RNS Power Consumption

We built several building blocks for RNS-based network inference using Bluespec SystemVerilog and synthesized them using a commercial LP65nm CMOS process. Table 2 shows the power and frequency of operation of the RNS blocks as well as their 32-bit counterparts. It is worth noting that the multiplier consumes almost half the power of the 32-bit block with a positive slack allowing for higher frequency of operation.

Table 2. Synthesis results

Block	P (mW)	f (MHz)	Slack(ps)
Adder32	1.05	625	15.9
AdderRNS	1.18	625	17.6
Multiplier32	3.04	250	7.1
MultiplierRNS	1.56	250	95.4
ConvertToRNS	2.6	250	1.1
ReLU-RNS	0.88	156	109.5
CompareRNS	1.67	156	93.1

6.2. Maintaining a Modulus Integer Network

A limitation using RNS is the necessity to maintain positive integer weights and activations within a given modulus M. To demonstrate the feasibility of this, and obtain a rough estimate of accuracy degradation when imposing such constraints, we train different flavors of a 8-layer (7 CNN/1 FC) network on the Street-View House Numbers (SVHN) dataset (Netzer et al., 2011). We denote a (W, A)-FP/INT network as a network with W-bit weights and A-bit activations in either floating point or integer, respectively. Note that negative integers are interpreted as their respective positive value in a wrap-around modulus.

We first trained (32, 32)-FP. We used a set of shadow floating point weights, initialized to (32, 32)-FP, , and truncate these shadow weights in the forward pass to generate our (6, 6)-FP network (gradients get passed to the shadow weights). In our (32, 32)-INT and (6,6)-INT networks, we modify this truncation operation to be a suitable affine transformation to fit our bit width and desired range. Note that our activation function in the integer network changes to compare with $\frac{M}{2}$. Networks were trained for 15 epochs, with data augmentation, 50% last layer dropout, and selecting the best model-checkpoint with highest validation accuracy. Note that a 6-bit integer is able to fit within each modulus of our RNS representation.

As expected, reducing bitwidth increases error. Moving to integer networks appears to slightly decrease accuracy. The precise reason for this is unclear; perhaps, something wonky is occuring with the gradient updates and gradient magnitude. Networks were implemented in Tensorflow/Tensorpack.

Table 3. SVHN Test Error Rate

Network	SVHN Test Error
(32, 32)-FP	3.95%
(6, 6)-FP	6.69%
(32, 32)-Int	4.54%
(6, 6)-Int	7.07%

6.3. Estimation of the RNS Break-Even Point

Use of RNS incurs overhead proportional to the output size, because of the comparatively expensive ReLU-RNS modules. If we have a $Y \times X$ fully-connected layer, we can compare the relative energy costs associated with performing the corresponding MACs and ReLU in RNS or non-RNS:

$$\begin{aligned} &Y \times E_{RNSReLU} + XY \times (E_{RNSMult} + E_{RNSAdd}) \\ &\qquad > Y \times E_{ReLU} + XY \times (E_{Mult} + E_{Add}) \end{aligned}$$

E_X is the energy per operation for hardware block X. Given our simulation results, this simplifies to:

$$\begin{aligned} Y &> \frac{E_{ReLU} - E_{RNSReLU}}{(E_{Mult} + E_{Add}) - (E_{RNSMult} + E_{RNSAdd})} \\ &\approx 0.98 \end{aligned}$$

This hints that it could be possible to achieve energy savings through RNS on FC layers of any size, because of our ReLU overhead/MAC savings ratio. It is demonstratable that the same result applies for a CNN layer, in which we would replace Y with $C_{in}K_XK_Y$, the size of channel-output filter. Note that this estimation is ignoring costs of memory accesses. Though, because of our choice of moduli, the amount of data being shuffled is similar in both systems.

7. Conclusion

In this work, we outlined use of the Residue Number System to perform inference on neural networks. Using our single-block implementation power estimates, we showed theoretical analysis of the advantages of RNS for an end-to-end system.

7.1. Critique and Future Improvements

In our next steps, we aim to demonstrate end-to-end inference stringing together our RNS blocks, comparing this system with a non-RNS system. It would be useful to explore other methods of translating networks to the integer domain, and testing accuracy drops in more networks and datasets. By fiddling with choice of adder design, we suspect that we can improve our RNS multipliers and comparator power.

References

Courbariaux, M and Bengio, Y. Binarynet: Training deep neural networks with weights and activations constrained to+ 1 or-1. CoRR abs/1602.02830 (2016).

Han, Song, Mao, Huizi, and Dally, William J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hiasat, Ahmad. Residue number system to binary converter for the moduli set (2^{n-1}, $2^n - 1$, $2^n + 1$). *Journal of systems architecture*, 2003.

Netzer, Yuval, Wang, Tao, Coates, Adam, Bissacco, Alessandro, Wu, Bo, and Ng, Andrew Y. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, pp. 5, 2011.

Piestrak, S. J. Design of residue generators and multioperand modular adders using carry-save adders. *IEEE Transactions on Computers*, 43(1):68–77, Jan 1994. ISSN 0018-9340. doi: 10.1109/12.250610.

Sousa, Leonel. Efficient method for magnitude comparison in RNS based on two pairs of conjugate moduli. In *Computer Arithmetic, 2007. ARITH'07. 18th IEEE Symposium on*, pp. 240–250. IEEE, 2007.

Zhou, Shuchang, Wu, Yuxin, Ni, Zekun, Zhou, Xinyu, Wen, He, and Zou, Yuheng. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.

Zimmermann, Reto. Efficient VLSI implementation of modulo ($2^{n_1} \pm 1$) addition and multiplication. In *Computer Arithmetic, 1999. Proceedings. 14th IEEE Symposium on*, pp. 158–167. IEEE, 1999.