Residue Number Systems Quantization for Deep Learning Inference

SERGEY SIVKOV
Electrical Engineering Faculty
Perm National Research Polytechnic University
614013, Perm, 7 Professora Pozdeeva Street, Office 225
RUSSIA

Abstract: Quantization of learned CNN weights to Residue Number System can improve inference latency by taking advantage of fast and precise low bit integer arithmetic. In this paper we review the mathematical aspects of RNS operations for signed integer values and evaluate implementation choices for conversion of conventional float-point PyTorch weights of CNN models to RNS representation. We also present a workflow to convert weights of PyTorch neural network layers specific for computer vision domain to 4-bit RNS moduli-sets able to maintain classification accuracy within 5% of 8-bit quantization baseline.

Key-Words: FPGA, inference, RNS, Residue Number System, Verilog, PyTorch, Quantization

Received: July 28, 2023. Revised: October 23, 2023. Accepted: November 25, 2023. Published: December 31, 2023.

1 Introduction

At the present time, most neural networks are trained using weights represented by 32-bit single-precision floating-point numbers format. Pretrained weights can be converted (quantized)[1] to other floating-point number formats, such as 16-bit floating-point (IEEE fp16) without noticeable loss of object classification precision. It is also possible to convert weights to integer formats, with a greater loss of accuracy during inference.

The main advantage of quantization into integer formats is increased performance: for 8-bit integers this is, theoretically, a 16x acceleration in computation throughput and a 4x reduction of the transmitted data bandwidth [2]. Also, inference engine implemented on low-bit integers can be used on low-power or wearable devices that do not have a hardware based floating-point arithmetic.

The main contribution to the inference latency of neural network calculations is made by the process of matrix calculations, which comes down to a sequence of multiplication and addition operations. If we look at the implementation of an ripple carry adder and at the implementation of a multiplier in the form of an array of adders, it is obvious that the execution time of the addition operation linearly depends on the bit size of the operands O(N), and the execution time of the multiplication operation quadratically depends on the bitness of the operands $O(N^2)$. Also, reducing the bit content of the operands allows to implement a computer circuit with a shorter critical path, so we may to increase its clock frequency.

In addition to positional number systems, non-positional number systems are also known, for example, Residue Number Systems (RNS) [3], where the use of independent and parallel operations on low-bit coprime bases makes it possible to perform the same basic operations in time $O(\sqrt{N})$ for addition and $O(N)$ for multiplication. We propose to use RNS for the FPGA based implementation of base operations required to neural network accelerator.

2 Quantization Fundamentals

Increasing the speed of inference of neural networks is an urgent task that can be solved by quantizing weights into integers. Quantization is based on the affine transformation $f(x) = s \cdot x + z$, in which we transform the input value $x \in [\beta, \alpha]$ into the range $[-2^{b-1}, 2^{b-1} - 1]$.

Due to the peculiarities of the formation of the signed representation of numbers in RNS, we will consider only quantization into a symmetric interval, with a coefficient $z = 0$ (only scaling the interval, without moving the 0th point of the representation). This transformation is described for the interval $x \in$ [-α, α], and the coefficient s is found as

$$s = \frac{2^b - 1}{\alpha} \tag{1}$$

This scheme is implemented, for example, in the PyTorch torch.ao.quantization module, for quantization to 8-bit integers. In case the source code of the model is available, this approach allows to carry out layer-by-layer quantization of the model. Unfortunately, the module does not yet support quantization to smaller data types, for example, to 4-bit integers or quantization with the single coefficient s for the entire model. Also, this approach assumes the use

of either integers of higher bit capacity or float-point arithmetic to renorm weights after forward pass layer calculations, which will introduce additional delays into the inference latency for a low-bit CPU. The limitations inherent in calculations on numbers represented in the positional number system can be overcome by using calculations on numbers represented in the RNS.

2.1 Unsigned RNS Fundamentals

The limitations inherent in calculations on numbers represented in the positional number system can be overcome by using calculations on numbers represented in the RNS. Let us show the basic operations in RNS based arithmetic on a simple example with two coprime moduli-set:
$p1 = 3$ and $p2 = 7$
with such a choice of the moduli-set the range of representable numbers will be $[0..3 \cdot 7 - 1)$.

Let's take the number 17 and convert it to RNS (hereinafter, % is the operation of taking the remainder modulo):
$17 = (17\%3; 17\%7) = (2; 3)$

This method requires repeated use of the operation of obtaining the remainder of an integer division. There is another way to quickly convert from a positional number system, such as decimal, hexadecimal or binary, which is based on the basic property of RNS: performing operations on the bases in parallel and independently.

For example, consider the conversion from the decimal system to RNS:
$17 = 1 * 10^1 + 7 * 10^0$

Knowing the representation of all powers of 10, in the range allowed for RNS:
$10^0 = (1; 1)$, because $1\%3 = 1$ and $1\%7 = 1$
$10^1 = (1; 3)$, because $10\%3 = 1$ and $10\%7 = 3$

Also, knowing the representation of each digit of the number in RNS:
$1 = (1\%3; 1\%7) = (1; 1)$
$7 = (7\%3; 7\%7) = (1; 0)$

We get the following representation of the translation operation:
$17 = 1*10^1+7*10^0 = (1; 1)*(1; 3)+(1; 0)*(1; 1) = (1; 3) + (1; 0) = (2; 3)$

2.2 Signed RNS Fundamentals

In the process of calculating the layers of a neural network, we need to be able to represent both positive and negative weights in the RNS. There are several approaches to solving this problem, let's look at representing numbers with a range shift (an socalled artificial number form), where both negative and positive numbers are mapped to positive numbers.

As an example, we can extend our example with the radix $p3 = 2$, and use a range-shifted representation of numbers, in which numbers in the range $[0..p1 \cdot p2)$ represent negative numbers, and numbers in the range $[p1 * p2..p1 * p2 * p3)$ are positive.

Then the form of representation of the number N with offset: $N' = p1 * p2 + N$
$N' = p1 * p2 - N$

The operations of addition and subtraction in this case can be derived as:
let $N1' = p1 * p2 + N1$ and $N2' = p1 * p2 + N2$
then $N1' + N2' = p1 * p2 + N1 + p1 * p2 + N2$
considering that $(N1 + N2)' = N1 + N2 + p1 * p2$
we get $(N1 + N2)' = N1' + N2' - p1 * p2$
Taking into account that $p1 * p2$ is $(0; 0; 1)$ in RNS with $p3 = 2$ added to moduli-sets,
we get $(N1 + N2)' = N1' + N2' - p1 * p2 = (N1 + N2)' = N1' + N2' + p1 * p2$

You can also notice that getting from the number $\left|N\right|$ numbers $-\left|N\right|$ perhaps by subtracting it from the number $(p1; p2; 1)$.

Consideration of the multiplication operation requires similar reasoning:
$N1' * N2' = N1 * N2 + p1 * p2 * N1 + p1 * p2 * N2 + p1 * p2 * p1 * p2$
those
$(N1 * N2)' = N1' * N2' + p1 * p2 - p1 * p2 * (N1 + N2 + p1 * p2)$
considering that $p1 * p2$ is $(0; 0; 1)$ in the RNS with $p3 = 2$ added, and that during the operation the numbers will be given in prime form, we get
$(N1 * N2)' = N1' * N2' + p1 * p2 * (1 + p1 * p2 + N1' + N2')$
since $p1 * p2$ is an odd number, we get
$(N1 * N2)' = N1' * N2' + p1 * p2 * (N1' + N2')$
Note that if $N1'$ and $N2'$ have the same parity, then
$p1 * p2 * (N1' + N2')$ is $(0; 0; 0)$
And if their parity is different, then
$p1 * p2 * (N1' + N2')$ is $(0; 0; 1)$, i.e. $p1 * p2$.

Also, for RNS it is quite simple, using precalculated auxiliary bit vectors, to implement the operations of checking the sign of a number and dividing by the bases of the system, which is sufficient both to implement the ReLU operation and to implement approximate dequantization after matrix operations.

More general comparison or division operations in RNS are also feasible, but require storage or calculation of additional parameters required for such operations. Because These operations are not used to implement the basic functions of the RNS inference

accelerator of the NN, then difficulties with the implementation of these operations do not affect our work.

2.3 Suggested workflow formulation

The objectives of this work are:

- selection of a family of NN models and their characteristic layers for implementation in the form of Verilog code of the RNS computer
- transfer of model weights to a quantized representation in RNS
- construction of a reference model for the possibility of checking the results of verification of the Verilog code of the calculator RNS
- implementation of basic blocks of RNS for FPGA
- comparison of the obtained result with the inference accuracy of float32 weights and int8 weights of the original model.

3 Problem Solution

As a target family of neural network models, we consider neural network architectures used in computer vision, due to the huge number of devices such as smart video cameras or wearable electronics that use them. Also, we consider only those types of layers that can be used in single-stage neural networks for classification, localization or detection tasks. This family includes convolutional neural networks with the following layers (in PyTorch terms):

- Conv2d layer providing convolution operation
- ReLU layer ensuring the introduction of nonlinearity into the neural network
- MaxPool2d layer providing selection of the maximum value in a given window of the input tensor
- AvgPool2d layer providing calculation of the average value in a given window of the input tensor
- Linear fully connected layer (MLP, perceptron).

Neural networks (f.ex. LeNet, AlexNet, VGG) built only using the listed layers have repeatedly achieved SOTA results on standard datasets (MNIST, CIFAR, ImageNet) for computer vision tasks.

We chose MNIST as a dataset to evaluate the results obtained.

An example code of a PyTorch NN model that achieves an accuracy of 92.67% on float32 weights on the MNIST dataset presented at Fig.1.

The estimate of the number of model parameters and its computational complexity for the

```
#part of class constructor
#resposible for neuronet structure:
self.conv1 = nn.Conv2d(1,4,3,1,1)
self.relu1 = nn.ReLU()
self.mp1   = nn.MaxPool2d(2)
self.conv2 = nn.Conv2d(4,8,3,1,1)
self.relu2 = nn.ReLU()
self.mp2   = nn.MaxPool2d(2)
self.conv3 = nn.Conv2d(8,16,3,1,1)
self.relu3 = nn.ReLU()
self.ap3   = nn.AvgPool2d(7)
self.out   = nn.Linear(16, 10)

#method resposible for model
#inference:
def forward(self, x):
  x = self.conv1(x)
  x = self.relu1(x)
  x = self.mp1(x)
  x = self.conv2(x)
  x = self.relu2(x)
  x = self.mp2(x)
  x = self.conv3(x)
  x = self.relu3(x)
  x = self.ap3(x)
  x = x.view(x.size(0), -1)
  output = self.out(x)
  return output
```

Fig.1: CNN Model structure and forward pass method

module	#params or shape	#flops
model	1.674K	0.141M
conv1	40	28.224K
conv1.weight	(4, 1, 3, 3)	
conv1.bias	(4,)	
conv2	0.296K	56.448K
conv2.weight	(8, 4, 3, 3)	
conv2.bias	(8,)	
conv3	1.168K	56.448K
conv3.weight	(16, 8, 3, 3)	
conv3.bias	(16,)	
out	0.17K	0.16K
out.weight	(10, 16)	
out.bias	(10,)	

Fig.2: CNN Model parameters and FLOPS

above model by FaceBook's PyTorch package `fvcore.nn.FlopCountAnalysis` presented at Fig.2.

The reference model was implemented in C++ in 5 versions:

1. with weights represented by real 32-bit numbers
2. with weights represented by 16-bit integers
3. with weights represented by 8-bit integers
4. with weights represented by 4-bit integers
5. with weights represented by class objects that implement RNS operations, each base of which does not go beyond a 4-bit integer.

The real weights for the reference model were converted directly from the weights of the corresponding PyTorch tensors of the NN model.

The values of the output vectors shown by the reference model with real weights coincided with the weights shown by the PyTorch NN model.

The Verilog implementation of the RNS calculator is based on a code generator for the given RNS bases and the required NS functionality.

The capabilities of our own code generator are clear from its prompt presented at Fig.3. An example of generated code for the multiplication operation in RNS based on $P = 3$ presented at Fig.4. Example of generated code for the multiplication operation for RNS with radix $(P1 = 2; P2 = 3; P3 = 7)$ presented at Fig.5.

4 Conclusion

The PyTorch model of the convolutional neural network presented at Fig.1 was trained, and 92.67% accuracy was achieved after 10 epochs of training on the MNIST dataset with batch 50 as showed at Fig.6.

The reference model implemented in C++ showed the same accuracy and the same output vectors.

Running the reference model with a single scaling factor on int16 scales shows 80.12% accuracy.

Running the reference model with a single scaling factor on int8 scales shows 74.30% accuracy.

Running the reference model with a single scaling factor on int4 scales shows 14.58% accuracy.

Launch of a reference model with a single scaling coefficient on RNS scales with bases $(P1 = 2, P2 = 3, P3 = 5, P4 = 7)$, none of which go beyond int4 shows 71.13% accuracy.

Below is a Table 1 comparing the number of LEs involved when implementing the multiplication operation both in the positional number system and in the RNS: We used RNS with about the same bit arity as their 2^n*2^n scheme as shown at Table 2.

```
gen.py --help
usage: g.py [-h] [--signed SIGNED]
  [--op OP]
  [--base BASE] [--rnsop RNSOP]
  [--vecop VECOP]
  [--bases BASES [BASES ...] ]
  [--tests TESTS]

optional arguments:
   -h, --help show this help message
     and exit
   --signed SIGNED Use artificial number
     representation
   --op OP Type of operation to generate:
     mul, add, div2, neg (by default all
     modules)
   --base BASE Value of base to generate
   --rnsop RNSOP Type of RNS operation to
     generate: mul, add, div2, neg, lte
     (by default all appropriate modules)
   --vecop VECOP Type of operation to
     generate: mul, add, div2, neg, lte,
     k3x3, max_pool, gac_pool, relu
     (by default all appropriate modules)
   --bases BASES [BASES ...]
     List of bases to use for vector
     operations
   --tests TESTS Generate coverage and
     randomized tests for required
     operations
     (may be valued as [1..100])
```

Fig.3: Help screen of Verilog code generator

```
module mulP3
  (input logic[1:0] a, b,
   output logic[1:0] y);
    logic[1:0] m;
    always_comb
      case ({a,b})
        'b01_01: m <= 1;
        'b01_10: m <= 2;
        'b10_01: m <= 2;
        'b10_10: m <= 1;
        default: m <= 0;
      endcase
    assign y = m;
endmodule
```

Fig.4: Generated Verilog code for one RNS moduli base operation

```
module mul2P2P3P7
  (input logic[2:0] a1, b1,
   input logic[2:0] a2, b2,
   input logic[3:0] a3, b3
   output logic[2:0] y1,
   output logic[2:0] y2,
   output logic[3:0] y3);
    mulP2(a1, b1, y1);
    mulP3(a2, b2, y2);
    mulP7(a3, b3, y3);
endmodule
```

Fig.5: Generated Verilog code for RNS moduli-set operation

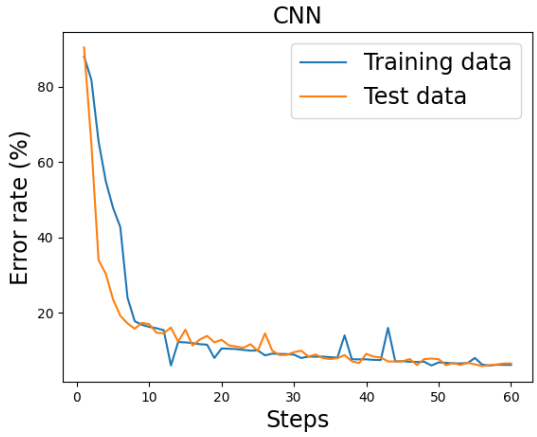

Fig.6: Traing/test accuracy

Table 1: Circuit sizes and latency

Bit-arity	multiplier		RNS moduli-set	
	LEs	delay	LEs	delay
8* 8	46	17ns	20	11ns
16*16	183	26ns	245	15ns
32*32	625	32ns	3577	23ns
64*64	2273	47ns	13009	31ns

Table 2: List of used RNS moduli-set bases

Bit-arity	RNS moduli-set bases
2^8	2; 3; 5; 7
2^{16}	11; 17; 19; 23
2^{32}	11; 17; 19; 23; 29; 31; 61
2^{64}	11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59

An we can see RNS scheme with 4 bases, two of which are 2-bit and another two are 4-bit, shows an accuracy that differs by 4.46% from the accuracy of a scheme on 8-bit integers and exceeds the accuracy of a scheme on 4-bit integers by 387.86%.

At the same time, the speed of the multiplication operation in the RNS, which is basic for matrix calculations, is 64.7% faster than the 8-bit multiplier and the required number of LEs for implementation is 43.47% less.

The paper [4] showed a nice example of specific moduli-set $(P1 = 2^n - 1, P2 = 2^n, P3 = 2^n + 1)$. Such moduli-set allows very concise, in terms of LEs, design. Also, the paper shows an important idea to measure power consumption of our scheme. This question will be the subject of our future research.

From a practical point of view, it is important to be able to accept as input not quantized data converted from the positional number system to RNS, but the original stream from the CMOS sensor, for example, in the $R_5G_6B_5$ format, to reduce the load on the CPU and/or directly connect the CMOS sensor to the FPGA and quantize them In parallel with the calculation of the convolutions of the first layer, ideas for deriving a recurrence formula for each base are outlined in Omondi's monograph [3]. Authors of [5] paper use another method to convert numbers to RNS but also show a good performance boost for their design.

We plan to implement a more compact version of this accelerator design based on the following idea: because we know all weights for all neural network layers for inference tasks, no one multiplicator or summator needs in two argument realization. I.e. one argument already known, so the Verilog module for vector operation should be more compact too.

The second idea for future research is to use polyadic summators with a few internal carry propagation chains.

References:

[1] Benoit Jacob, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*, CVPR, 2018

[2] Hao Wu, *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*, arXiv:2004.09602, 2020

[3] Omondi A., Premkumar B. *Residue number systems: theory and implementation*, Imperial College Press, 2007.

[4] Salamat S. *RNSnet: In-Memory Neural Network Acceleration Using Residue Number System*, IEEE International Conference on Rebooting Computing, 2018.

[5] Nagornov N. *RNS-Based FPGA Accelerators for High-Quality 3D Medical Image Wavelet Processing Using Scaled Filter Coefficients*, IEEE Access, 2022.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

All work done solely by a single author of this article.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflicts of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.