Model Compression by Exponent Sharing

Prachi Kashikar
School of Computer Science and Mathematics
Indian Institute of Technology Goa, India

I. Introduction

AI on the edge has become a significant focus of research in the past decade. This area involves deploying various applications in computer vision and natural language processing on small devices with limited on-chip memory and battery power. The neural networks require large storage space, It is crucial to compress the models to make them fit into these tiny devices.

The current model compression techniques, such as quantization [7], weight sharing, and pruning [8], reduce the size of the model by sacrificing some accuracy. We propose a new lossless model compression approach that makes use of the exponent distribution in weights for a trained model. We demonstrate how exponent sharing in weights can help reduce memory requirements. It is important to note that our proposed "exponent sharing in weights" method is different from the existing "sharing of weights" approach. Additionally, one can apply exponent sharing on top of other existing compression methods as well. We also discuss how exponent sharing affects the memory and execution time of processors and FPGAs [4].

II. Method

The size of a machine learning model depends on the number and type of parameters used to represent it. These parameters are typically stored in floating-point formats, which offer greater precision and can represent a wider range of real numbers. According to the IEEE 754 standard for floating-point representation [2], every floating-point value is a combination of sign, exponent, and mantissa.

A. Proposed Floating Point Storage Format

In the IEEE single-precision format, there are 256 possible exponent values. However, in reality, there are thousands of weights in a given model, and many of these weights have the same exponent value. For instance, in a trained LeNet model as shown in Figure 1, only 6.25% of possible exponents are present in the weights. To exploit this fact, we propose a new floating-point storage format that leverages the presence of multiple exponents with the same values. This format can be applied to any real value representation format that utilizes the mantissa-exponent method for number representation.

We have devised a new method to store only the unique exponents of floating-point weights in a separate exponent table. This method replaces the exponents from IEEE floating-point formats with respective indices that refer to the exponent table. As a result, every floating-point weight becomes a combination of three components: sign, index, and mantissa. If a layer's weights have k distinct exponents, the number of index bits required is determined by, $i = ceil(\log_2(k))$.

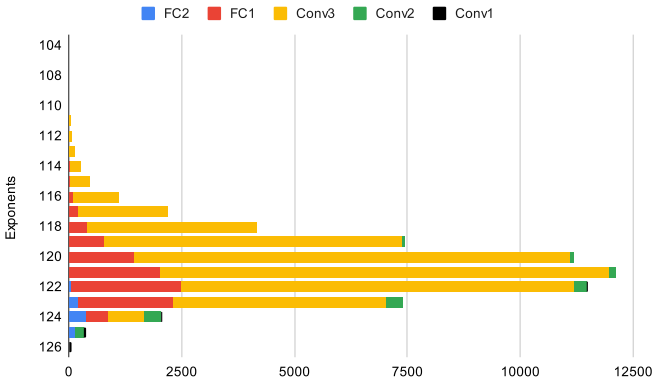

Fig. 1. Exponent frequency distribution in LeNet

If s, e and m denote the number of bits for sign, exponent and mantissa, respectively then the memory needed after exponent sharing (M_{comp}) is,

$$M_{comp} = N \times (s + i + m) + e \times k, \tag{1}$$

where N is the total number of weights in a layer and i is the number of index bits. Our method [3] exhibits better performance when layer-wise exponent sharing is applied. Figure 2 provides an illustration of how the exponent sharing works on a small weight matrix.

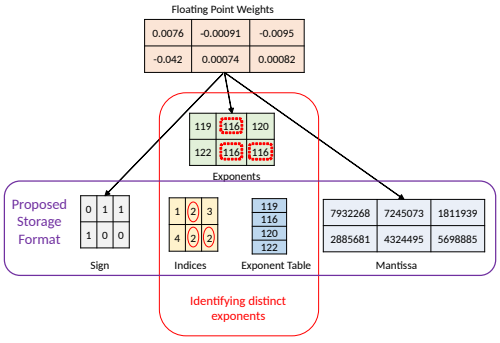

Fig. 2. Illustration of storing weights in the proposed format

B. Execution Time Overhead

The time it takes for a model to execute depends largely on how many generalized matrix multiplications (GEMM) are present in each layer. Our proposed indexing-based method requires three reads, as opposed to just one read when such

indexing isn't used. This could potentially increase execution time when using single-threaded implementations on microprocessors. However, this overhead in execution can be minimized by performing parallel reads on accelerators like FPGAs [5].

1) Sequential Architecture: Our proposed method requires three additional reads to access the same weight that is fetched in a single read in sequential architecture. For a GEMM of any layer, if the weight tensor is $M \times N$ and the input tensor is $N \times O$, then the cycle count can be given by

$$C_{expSharing} = C_{orig} + M \times N \times O, \tag{2}$$

where C_{orig} is the number of clock cycles before exponent sharing and $C_{expSharing}$ is the number of clock cycles after exponent sharing.

2) Parallel Architecture: We observed about a 9-10 % impact of exponent sharing on the execution cycles when non-pipelined hardware is used. To reduce this impact we designed parallel architecture where reads are done in parallel. Here, multiple reads are possible in the same clock cycle due to multi-port on-chip memories like Block RAM. We used pragmas like *Pipeline* in the high-level synthesis tool Vivado HLS to parallelize operations, including memory reads. In this scenario, the cycle count is given by

$$C_{expSharing} = C_{orig} + M \times O \tag{3}$$

Using equations 2 and 3, we can find out the execution cycle overhead that would result because of exponent sharing in a GEMM of any layer.

III. Experiments and Observations

We discuss results using BFloat16 [1] floating point format as it has more benefit of proposed exponent sharing. It is a 16 bit representation of a float having half space (8 bits) allocated for exponents. The results reported are using Vivado HLS set for part xc7z020clg484-1 on Zynq Zedboard with a default clock period of 10 ns.

A. Compression Results

We have demonstrated our approach on the layers of the Tiny-tiny-tiny YOLO (TTT-YOLO) model from the Darknet framework with pre-trained weights [6]. This model is a tinier version of YOLO having 8 convolutional layers. Here, we discuss the results on three smaller layers. Table I shows the dimensions of GEMM in those three layers of this model.

TABLE I
GEMM sizes of Layers in Tiny-Tiny-Tiny Yolo

Layer	Input	Filters	GEMM Size
Conv_2	28x28x32	(3x3x32)x128	[128x288][288x560]
Conv_5	7x5x512	(1x1x512)x256	[256x512][512x35]
Conv_7	7x5x512	(1x1x512)x125	[125x512][512x35]

Table II reports the total weight memory in bits *Before* exponent sharing and *After* exponent sharing as well as the compression ratio achieved in percentage.

TABLE II
Compression on the Layers of TTT-YOLO

Layer	Before	After	% Memory Saved
Conv_2	9437184	7667920	18.74
Conv_5	2097152	1704100	18.742
Conv_7	1024000	832160	18.734

B. Execution Speed Overhead

To find out the applicability of proposed eq 2 and eq3 reporting the execution time overhead, we measured the clock cycles required by the GEMM of a layer with and without exponent sharing on various architectures.

As sequential reading after exponent sharing needs three more reads for every weight, the performance is impacted as per eq 2. The impact on clock cycles is less than 10% percent in all the layers. We have demonstrated exponent sharing on GEMM with *Pipeline* pragma from Vivado HLS resulting in parallel reads. The performance impact abides by the relationship shown in eq3. In all of the cases, the impact is less than 1%. The complete results along with few other variations are discussed in [4].

TABLE III
Clock Cycles Impact after Exponent Sharing on the layers of TTT-YOLO on FPGA

Layer	% Increase in Clock Cycles
Conv_2	0.069
Conv_5	0.039
Conv_7	0.039

IV. Conclusion

The proposed method is capable of compressing models without affecting their accuracy. It does not require any additional fine-tuning after model compression, thus avoiding any potential overhead. In FPGA implementations, parallel reads can be used to fill the gap in execution time before and after exponent sharing. The code for the GEMM with our implementations is available on GitHub[1]. In our future work, we aim to enhance memory savings by using exponent approximations during the training phase itself.

Acknowledgments

This work is guided by Dr. Sharad Sinha, IIT Goa. We thank Prof. Olivier Sentieys, Inria, Rennes, France for his valuable inputs for this work. This work is supported by DST-INRIA-CNRS Project IFC/4131/DST- Inria/2018-2019/1 through CEFIPRA, India.

References

[1] G. Henry, P. Tang, and A. Heinecke. "Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019, pp. 69–76.

[2] W. Kahan. "IEEE standard 754 for binary floating-point arithmetic". In: *Lecture Notes on the Status of IEEE* 754.94720-1776 (1996), p. 11.

[1] https://github.com/prachikashikar/Expo-Share-In-GEMM

[3] P. Kashikar, S. Sinha, and A. K. Verma. “Exploiting Weight Statistics for Compressed Neural Network Implementation on Hardware”. In: *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2021, pp. 1–4.

[4] P. Kashikar, O. Sentieys, and S. Sinha. “Lossless Neural Network Model Compression Through Exponent Sharing”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31.11 (2023), pp. 1816–1825. DOI: 10.1109/TVLSI.2023.3307607.

[5] W. MacLean. “An Evaluation of the Suitability of FPGAs for Embedded Vision Systems”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*. 2005, pp. 131–131.

[6] *Tiny-Tiny-Tiny YOLO*. https://github.com/k5iogura/darknet_ttt. Last Accessed: 28 Feb 2023.

[7] X. Wei, H. Chen, W. Liu, and Y. Xie. “Mixed-Precision Quantization for CNN-Based Remote Sensing Scene Classification”. In: *IEEE Geoscience and Remote Sensing Letters* (2020), pp. 1–5.

[8] R. Yu, A. Li, C. Chen, J. Lai, V. Morariu, X. Han, et al. “NISP: Pruning Networks Using Neuron Importance Score Propagation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.