SUCCINCT COMPRESSION: LOSSLESS COMPRESSION FOR FAST AND MEMORY-EFFICIENT DEEP NEURAL NETWORK INFERENCE

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper introduces "Succinct Compression", a method to provide lossless compression of Deep Neural Network (DNN) models for fast and memory-efficient inference. The key insight of our method leverages the concept of *Succinct Data Structures*, which supports fast queries without decompressing the compressed representations. Our method consists of three new insights. First, we introduce two basic building blocks to formulate DNN models, and how they can be extended to be synergistic with compressed models (e.g. pruned or quantized models). Then, we propose a scheme to enable mixed-formulation inference for different layers, to better extract its benefits. Finally, our method exploits a specialized execution pipeline to incorporate different model formulations for fast inference. We quantitatively demonstrate that: our method can (1) enable faster and more memory-efficient inference on uncompressed models; (2) be synergistic with a variety of structure-altered/unaltered compression schemes with better speedup and compression ratio, while preserving the accuracy; and (3) can outperform all other state-of-the-art Model Coding approaches.

1 INTRODUCTION

Recent efforts on Pareto improvements of compressed Deep Neural Network (DNN) models, on inference time, space consumption and the accuracy, have recently bloomed due to the great success of DNNs in practice. Prior works either aggressively simplify/optimize the structure of DNN models (e.g. Pruning and Neural Architecture Search) or retrench the representation of model parameters (e.g. Quantization and Model Coding), with a major focus on the compression ratio and the accuracy. Given a variety of methodologies for efficient compression, there still lacks a general method to further optimize the inference performance and compression ratio, without affecting the accuracy of both uncompressed and compressed models.

This paper introduces "Succinct Compression", a method to provide lossless compression of Deep Neural Network (DNN) models for fast and memory-efficient inference. The emphasis of our method is to enhance the inference performance and compression ratio without affecting the accuracy at the same time, for a variety classes of uncompressed and compressed models. The unique characteristic of our method is to exploit *Succinct Data Structures*, which enables fast queries without decompressing the compressed representations.

We consolidate three new insights to better incorporate *Succinct Data Structures*. ❶ we propose two semi-structured formulations to represent DNN models in element-wise or block-wise manners, and provide simple extensions to allow them for the combinations of other compression techniques. ❷ we enable mixed formulations of different layers in the model, to better extract the potential of *Succinct Data Structures*. ❸ we design a specialized execution pipeline to perform the inference on different formulations, by carefully engineering the inner operators of *Succinct Data Structures*.

Our evaluation shows that our method can be very effective for the inference efficiency, and generally applicable for uncompressed and compressed models (including for ResNet-50, ResNet-101, VGG-16, MobileNet-V2 and DeiT-B). For uncompressed models, our method can achieves most $1.07\times$ speedup and $1.17\times$ compression ratio at the same time, without affecting the accuracy. We then show that our method can bring significantly more benefits by combining other compression schemes, where all models are pre-processed via other compression methods. For instance, by combining structure-altered compression (such as pruning), our method enables the at most 8.8× acceleration of inference on ResNet-101, with 39.90× compression ratio meanwhile. Similarly, the speedup can be further enhanced to reach 9.3× by incorporating structure-unaltered method (such as quantization). We also compare our method with a variety of the state-of-the-art Model Coding schemes, and show that our method outperforms all of them.

2 Related Works

A large body of relevant works on compressing DNN models consists of two categories, based on the orientation of their methodology: structure-altered and -unaltered methodologies. We outline key directions in each category, briefly describe their features and justify the novelty of our method.

2.1 Structure-altered Methods for Compression

Structure-altered Methods refer to those compression methods by simplifying/optimizing the DNN model architectures, and representative methods in this direction include Pruning, Low-Rank Factorization, Neural Architecture Search (NAS) and Knowledge Distillation (KD). We describe each of them in brief as follow.

❶ **Pruning** removes the redundant connections within DNN models without incurring a considerable degradation of the accuracy. There are two categories of Pruning. One is Unstructured Pruning (Dong et al. (2017); Lee et al. (2019); XIAO et al. (2019); Park* et al. (2020)), which aggressively removes neurons with small relevance whenever it's possible. Though such an approach can deliver decent compression ratio with only a marginal degradation of the accuracy, the inference overheads suffers from the inefficient usage of the memory, due to the frequent operations on sparse matrices (Gale et al. (2019); Blalock et al. (2020)). The other is Structured Pruning (Huang & Wang (2018); Lin et al. (2018); Yu et al. (2018); He et al. (2019); Zhao et al. (2019); Yu et al. (2021)), which only removes irrelevant units of DNN models at a granularity of the elementary structures (e.g. weights, filters and layers). Though these methods can benefit the performance/compression ratio due to the reduction of the total computational costs, the accuracy is usually not as expected.

❷ **Low-Rank Factorization** (Mamalet & Garcia (2012); Sainath et al. (2013); Zhao et al. (2017); Li et al. (2018)) uncovers the latent compact structure of the network through low-rank matrix factorization of weight layers. Though these approaches may only incur a marginal degradation in terms of the accuracy, they requires extra computational costs and the benefits in memory efficiency may not be consistent in different models.

❸ **NAS** (Mellor et al. (2021); Zhao et al. (2021)) automatically output neural network architectures using specific search strategies applied to a large search space. Therefore, a huge amount of extra computational costs are required and such methods need to be performed before the deployments of the selected models.

❹ **KD** (Feng et al. (2021); Wang (2021); Zhu et al. (2021)) is to train a large model and then use it as a teacher to train a more compact model. Similarly, KD also demands a huge amount of extra computational costs for training different models, therefore they are usually performed off-line.

In this work, we consider Pruning as the representative method in this direction, to justify the compatibility of our method with Structure-altered methods (as described in Section 7.2).

2.2 Structure-unaltered Methods for Compression

Structure-unaltered Methodologies refer to those compression methods by compressing DNN models without altering the model architecture, and there are two representative methods in this direction, which are Quantization and Model Coding. We describe each of them in brief as follow.

❶ **Quantization** reduces the bitwidth of parameters within DNN models, and such an approach can be achieved via quantization-aware training (Bengio et al. (2013); Alizadeh et al. (2020)) or post-training quantization (Banner et al. (2019); Cai et al. (2020)). Note that it's also feasible to perform extreme quantization (e.g. binarization) for this purpose (Cai et al. (2017); Bulat et al. (2021)) but usually suffers from a significant degradation of the accuracy.

❷ **Model Coding** represents DNN models via an extra bit sequence. Several well-studied coding strategies like Huffman Coding (van Leeuwen (1976)), Tunstall Coding (Tunstall (1967)) and Arithmetic Coding (Witten et al. (1987)) have been attempted for compressing DNN models (Han et al. (2016a); Reagen et al. (2017); Zhe et al. (2021)). More recently, there are a growing interests in customized coding strategies in the context of DNN models (Louizos et al. (2017); Havasi et al. (2018); Oktay et al. (2020b)). However, these approaches suffers from costly pre-processing to convert model parameters as encoded values, and the overheads of decoding are significant during the inference runtime.

Our work considers Quantization as the representative method in this direction, to justify the synergy of our method with Structure-unaltered methods (as described in Section 7.3). Also, we compare our method against a variety of Model Coding methods to show that our method can outperform all other state-of-the-art methods (as described in Section 7.4).

2.3 Novelty of Our Method

The novelty of our method is three-folded. ❶ We are the first to introduce *Succinct Data Structures* in the context of DNN models, which allow fast queries on compressed representations. ❷ we design specific formulations to make DNN models more compatible with *Succinct Data Structures*, which reduces the overheads for both pre-processing and decoding during the inference runtime. ❸ our method is generally applicable to available DNN models, and achieve Pareto improvements in both inference time and memory efficiency, without affecting the accuracy.

3 Formulating DNN Models

The first part of our method is to formulate DNN models appropriately, so that *Succinct Data Structures* can take advantage of. *Succinct Data Structures* exploits the delimiters within a long string, to perform fast queries directly on the compressed representations. To this end, we propose the model formulation called Runtime-Accessible Sequence (RAS), which refers to semi-structured format using a minimal amount of delimiters to construct hierarchical information (e.g. layers). Our proposed RAS consists of two basic building blocks, including (1) Element-wise RAS, which uses delimiters to separate different elementary operands within DNN models; and (2) Block-wise RAS, which applies delimiters to separate different sets of data operands within DNN models, based on the computation kernels. Based on the above Element-wise and Block-wise RAS, we provide simple extensions of RAS, to make them synergistic with other compression methods.

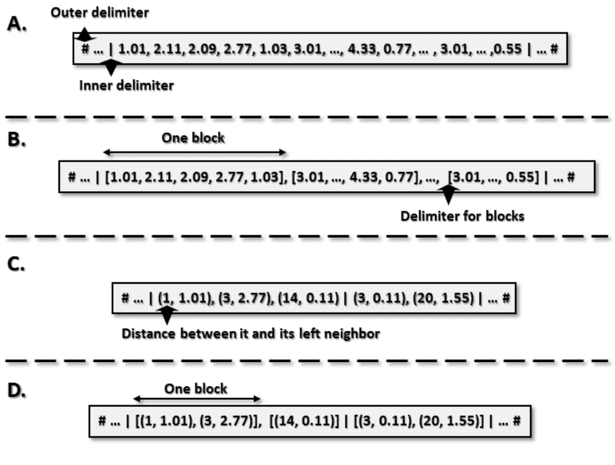

Figure 1: A comparison of different kinds of Runtime-Accessible Sequence (RAS).

3.1 ELEMENT-WISE RUNTIME-ACCESSIBLE SEQUENCE

One formulation in our method is Element-wise Runtime Accessible Sequence (denoted as Element-wise RAS). Element-wise RAS utilizes delimiters to separate elementary data operands. In the context of DNN models, the pre-defined delimiters (e.g. vertical bar and number sign) are used at the boundaries of different elementary data operands from DNN models, and these delimiters are used to query for elementary data operands accordingly.

Figure 1-(A) shows an example of Element-wise RAS; there are two vertical bars encompassing several elementary operands. This methodology forms the Element-wise RAS, and the number sign is used to represent the border of this union. To properly formulate the whole network into Element-wise RAS, we concatenate such unions by using a separate delimiter (e.g. '#').

3.2 BLOCK-WISE RUNTIME-ACCESSIBLE SEQUENCE

The limitation of Element-wise RAS is that frequent queries are required for every single data operand, before the computation for model inference. Therefore, to improve the efficiency of operand query, we suggest the other formulation of DNN models: Block-wise Runtime-Accessible Sequence (denoted as Block-wise RAS). Different from Element-wise RAS, Block-wise RAS forms basic building blocks for query and access based on the computation kernels, namely denoted as a block. Such a block stores a consecutive number of elementary data operands, which are used for a computation kernel. Between different blocks, Block-wise RAS exploits delimiters for separation, so that they can be efficiently queried.

Figure 1-(B) shows an example of Block-wise RAS: the Block-wise RAS aggregates five operands with two square brackets, as one individual block. This transformation of elementary operands, by synthesizing multiple operands and using a distinct delimiters, can provide faster queries by extracting them at one time over Element-wise RAS.

3.3 EXTENDING BOTH RAS FOR COMPRESSED MODELS

Since compressed models, supported by other compression schemes, usually maintain a high extent of the sparsity[1], current designs of Element/Block-wise RAS may not be capable to extract the maximum potentials of *Succinct Data Structures* on compressed models. To resolve this issue, we provide simple extensions to RAS so that they can be synergistic with other compression schemes. The key insight is to form elementary data operand in a similar manner as inverted indexes, by forming a tuple consisting of the exact values and the relative positions.

Figure 1-(C)/(D) shows examples of these optimized RAS formulated in Element- or Block-wise manner. The difference hereby is that, we refine the elementary operands as tuples. In such a tuple, the first element refers to the relative distance between this and its left neighbor (a number or a delimiter); and the second element stores the value of the corresponding data operand. This approach is synergistic with sparse models because for unstructured pruning, the relative distance, contained in the reshaped tuple, can effectively exploit the sparse model structures.

4 CONVERTING RAS TO *Succinct Data Structures*

The second part of our method is about how to covert RAS into *Succinct Data Structure* for efficient inference. We first provide a selection scheme to embrace the heterogeneity of different layers within DNN models, to maximize the benefits of our method. Then, we describe the rationale of our selected *Succinct Data Structure* - Wavelet Tree.

4.1 LAYER-WISE SELECTION OF THE RAS TYPE

In practice, soely using Element-wise or Block-wise RAS for one model cannot fully unleash the potentials, since the sparsity and access frequency vary across the whole model. Therefore, it's

[1]Note that Quantization and Model Coding can be viewed to exploit *bit-level* sparsity, so that the redundancy of data representation within DNN models can be reduced.

demanded to perform the selection of RAS at a certain granularity of model structure. We consider layer-level in this work, by connecting the size of elementary operands with the following criterion.

- Access Frequency of elementary data operands is used to determine which RAS to be used for the overall efficiency. More specifically, we define whether a particular layer is computationally intensive or not based on the ratio of FLOPs to #Parameters (namely FLOPs-Parameter-ratio, *FPr*). As long as above *FPr*, our method uses Block-wise RAS. Otherwise, Element-wise RAS is used.

How RAS Types are Distributed? We confirm that, from our experimental studies, all convolution layers and the weight matrices, inside the multi-head attention layers of Transformers, exploit Block-wise RAS, due to the intensively usage of stored data; And Element-wise RAS is applied to less-frequently-accessed layers (FC layers, in our experimental studies).

4.2 *Succinct Data Structures* AND WAVELET TREE

Succinct Data Structures were first pioneered by Jacobson (1988), which refers to a set of data structures using the near-information-theoretic bound space to store the compressed representation, and still provide fast query and access operations directly on these compressed representations. In general, *Succinct Data Structures* have the following representative inner operators.

Given a string S whose length and alphabet are L and σ, there are three operations directly on the compression (shown below).

• $Rank_q(x)$ returns the number of symbol q appearing in $S_{0:x}$ where $q \in \sigma$ and $x < L$.

• $Select_q(x)$ returns the position of x-th occurrence of symbol q in S.

• $Access(x)$ returns the symbol at the position x of S.

Though there are a number of *Succinct Data Structures* available for real-world applications, we choose Wavelet Tree (Grossi et al. (2003)) as the core of our method. We choose Wavelet Tree (WT) because there are already a number of evident successes in applying WT for large-scale, data-intensive applications, such as Data Store (Agarwal et al. (2015); Khandelwal et al. (2016)), Graph Processing (Khandelwal et al. (2017)), and it presents outstanding merits of runtime fast queries without losing memory efficiency: it allows $Rank$, $Select$ and $Access$ to only take $O(log\sigma)$ time while maintaining space consumption within $n\,log(\sigma) + O(n)$ bits (where input is of length n with σ distinct symbols). Therefore, our method deploys WT as the compression technique during the inference runtime.

5 MODEL INFERENCE IN *Succinct Data Structures*

The third part of our method is to perform model inference via *Succinct Data Structures*. There are three steps: ❶ Identify the RAS flag to guide the subsequent steps; ❷ Execute different retrieval strategies for either Element- or Block-wise RAS; and ❸ Perform the inference.

❶ **Identify the RAS Flag.** Our method first identifies the RAS flag to obtain which type of RAS being used for the current layer. The flag is tagged at the beginning of all RAS for different layers. Therefore, identifying and parsing the flag is necessary for the subsequent steps.

❷ **Execute Different Retrieval Strategies based on the Flag.** Our method then retrieves all the operands within this layer. For a given type of RAS, the retrieval strategy is structurally organized similarly: for both Element-wise and Block-wise RAS: we use two $Select$ operators to locate the corresponding values for current layer within the compressed representation, while the access of these data differs. The differences between two strategies for different types of RAS are: since Block-wise RAS retrieves a series of operands at a time, additional operations for extracting them are necessary.

❸ **Inference over the extracted operands.** Our method finally performs the inference for the current layer. Note that, for inference on compressed models, the inference may require additional decoding, if the combined methods introduce more levels of indexes.

6 Experimental Methodology

Platform and Baseline: All experiments are done via the latest version of PyTorch (Paszke et al. (2019b)), on a platform equipped with Intel Core i9-12900KF and NVIDIA RTX 3090. We consider five mainstream models, including ResNet-50, ResNet-101 (He et al. (2015)), VGG-16 (Simonyan & Zisserman (2015)), MobileNet-V2 (Sandler et al. (2019)) and DeiT-B (Touvron et al. (2021)).

Methodology: For all uncompressed and compressed models, we measure (1) the end-to-end latency to examine the performance benefits; (2) the overall memory usage to examine the memory efficiency; and (3) the accuracy. To implement the execution pipeline of our method, the inference pipeline requires less-than-byte packing and unpacking. This can be done directly using Bit Manipulation Instructions extensions (e.g. NVIDIA (a)), due to the fact that the processing granularity of modern SIMD architectures (including CPU SIMD and GPU) is at byte-level. We use PyTorch Bindings with Fortran, to exploit these hardware intrinsic functions for the implementations (i.e. Alexeev).

Model Setups: All these models are trained on the ImageNet dataset (Deng et al. (2009)), which are used as uncompressed models (used in Section 7.1). To examine the benefits of our method over compressed models, we apply representative Pruning (described in Section 7.2) and Quantization methods (described in Section 7.3) on the trained models, and use them as the compressed models. Furthermore, we compare our method with other Model Coding methods to show the benefits.

7 Experimental Results

7.1 Impacts on Uncompressed Models

We first examine the impacts of our method on uncompressed models. Figure 2 reports the model size and speedup of our method over the uncompressed models. We draw two observations. First, our method provides considerable improvement for uncompressed models on both speedup and compression. For instance, MobileNet-V2 achieves 1.07× speedup and 1.17× compression ratio at the same time, and ResNet-101 achieves 1.04× speedup and 1.11× compression ratio at the same time. Second, we note that the improvements on VGG-16 are relatively worse than others. This is because the FC layers within VGG-16 incur significant overheads for the acceleration and compression.

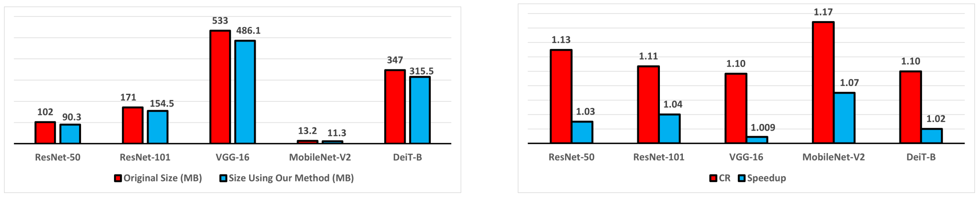

Figure 2: Results on uncompressed models for Model Size (left); CR and Speedup (right)

7.2 Impacts on Pruned Models

We then examine the impacts of our method on Pruned models. Table 1 reports the results on selected models after using three representative state-of-the-art pruning methods (i.e. layer-level pruning: DBP (Wang et al. (2019)); filter-level pruning: HRank (Lin et al. (2020)); and unstructured pruning: SNIP (Lee et al. (2020))). We first quantitatively demonstrate the outstanding impacts from our method on all models, then we elaborate the key observation.

Our method yields significantly better improvements when all models are pruned in advance, in terms of both speedup and compression ratio. The results show that our method achieves the speedup of 4.7×/8.8×/3.9×/3.8× and the compression ratio as 39.48×/39.90×/3.78×/3.81×, without affecting the accuracy, for ResNet-50/ResNet-101/VGG-16/DeiT-B. Our results suggest that our method is synergistic with the state-of-the-art pruning methods.

We make the key observation that: our method yields better benefits from filter-level/unstructured pruning, compared with layer-level pruning. As shown in Table 1, the speedup from our method can be up to 1.91×/8.07× for filter-level/unstructured pruned models, but the improvement for layer-level pruned models (DBP) is limited to just 1.28×. This is because the filter-level/unstructured pruning decreases the "width"/parameters of each layer, and there result in less *Access* operations for extracting data operands, since these layers are formulated in Block-wise RAS based on our methodology.

Table 1: Succinct Compression on Pruned Models: Target sparsity refers to the expected sparsity within the networks after conducting pruning, and we can tolerate the error within $\pm 4\%$. The entries "xx/yy" under the Size, CR (Compression Ratio) and Speedup columns show the comparison between only using pruning (left) and applying our method on the top of pruned models (right). UP-ViT (Yu & Wu (2021)) is the pruning method specialized for transformers.

NN	M	Target Sparsity (%)	Accuracy (Top-1) (%)	Size (MB)	CR (×)	CR Improved (×)	Speedup (×)	Speedup Improved (×)
ResNet-50	-	0	75.5	102	1	1	1	1
	DBP	40	75.1	60.8/52.9	1.68/1.93	1.15	1.4/1.6	1.14
		50	74.8	50.2/43.6	2.03/2.34	1.15	1.7/1.8	1.06
		60	73.5	41.0/35.3	2.49/2.89	1.16	2.3/2.6	1.13
		70	72.3	33.2/28.6	3.07/3.56	1.16	3.0/3.4	1.13
	HRank	40	74.4	61.0/53.1	1.67/1.92	1.15	1.3/1.9	1.46
		50	72.5	49.9/43.2	2.04/2.36	1.15	1.4/2.6	1.86
		60	69.9	41.2/35.5	2.48/2.87	1.16	2.1/2.7	1.23
		70	68.1	31.0/26.7	3.29/**3.83**	1.16	2.7/**4.5**	1.67
	SNIP	90	74.8	10.3/8.7	9.90/11.69	1.18	1.12/2.8	2.50
		95	74.7	5.2/4.4	19.6/23.34	1.19	1.14/3.5	3.07
		97	74.7	3.1/2.6	32.9/**39.48**	**1.20**	1.15/**4.7**	**4.09**
ResNet-101	-	0	76.4	171	1	1	1	1
	DBP	40	76.2	103/91.2	1.66/1.88	1.13	1.2/1.5	1.25
		50	76.2	87/76.3	1.97/2.24	1.14	1.6/2.0	1.25
		60	75.3	70/61.2	2.44/2.79	1.14	1.9/2.4	1.26
		70	74.9	50.7/44.1	3.37/3.88	1.15	2.5/3.2	**1.28**
	HRank	40	76.3	102/90.3	1.68/1.89	1.13	1.1/2.1	**1.91**
		50	76.1	85/74.7	2.01/2.29	1.14	1.4/2.6	1.86
		60	75.0	71/62.3	2.41/2.75	1.14	1.8/2.9	1.61
		70	74.3	50/43.4	3.42/**3.94**	1.15	2.3/**4.4**	**1.91**
	SNIP	90	76.0	17.1/14.6	10.0/11.7	1.17	1.08/5.3	4.91
		95	75.9	8.6/7.3	19.9/23.56	**1.19**	1.09/6.4	5.87
		97	75.9	5.1/4.3	33.5/**39.90**	**1.19**	1.09/**8.8**	**8.07**
VGG-16	-	0	71.3	533	1	1	1	1
	DBP	40	71.3	320/290.4	1.67/1.84	1.10	1.7/1.9	1.12
		50	71.3	262/236.0	2.03/2.26	1.11	2.3/2.4	1.04
		60	71.1	210/189.2	2.54/2.82	1.11	2.7/2.9	1.07
		70	70.2	158/141.1	3.37/**3.78**	1.12	3.8/**3.9**	1.03
	HRank	40	71.3	323/292.8	1.65/1.82	1.10	1.5/1.7	1.13
		50	70.8	254/228.6	2.10/2.33	1.11	1.7/1.8	1.06
		60	70.5	214/192.1	2.49/2.77	1.11	1.8/2.1	1.17
		70	69.8	160/142.9	3.33/3.73	1.12	2.0/2.4	1.20
	SNIP	90	71.0	53/46.1	10.06/11.57	1.15	1.08/1.3	1.20
		95	69.9	26.7/22.9	19.96/23.26	**1.17**	1.08/1.4	1.30
		97	69.9	16/13.6	33.31/**39.08**	**1.17**	1.1/1.7	1.55
DeiT-B	-	0	81.8	347	1	1	1	1
	UP-ViT	40	81.7	210/189.2	1.65/1.83	1.11	1.5/1.9	1.27
		50	81.7	170/152.6	2.04/2.27	1.11	2.3/2.9	1.26
		60	81.6	141/125.8	2.46/2.76	1.12	2.6/3.3	1.27
		70	81.5	103/91.2	3.37/**3.81**	**1.13**	2.9/**3.8**	**1.31**

7.3 Impacts on Quantized Models

Next we examine the impacts of our method on quantized models. we use first quantize all models at a set of different levels with different bit precision (2/4/8 bits). To quantize our selected models, we use the state-of-the-art quantization methods, including LSQ (Esser et al. (2020)) for CNNs and PQT (Liu et al. (2021)) for transformer. After quantization, we apply our method upon these quantized models. We first report our quantitative results in terms of the speedup and compression ratio. Then, we elaborate the key observation.

Table 2: Succinct Compression on Quantized Models: The entries "xx/yy" under the Size, CR (Compression Ratio) and Speedup columns show the comparison between only using quantization (left) and applying our method on the top of quantized models (right). For the quantization of DeiT-B, we modify the framework PQT to output fixed-bits-length results and handcraft a DeiT-B model comprised of 2-bits parameters (marked with '*'), since PQT is incapable for the 2-bit precision.

NN	M	Precision (bits)	Accuracy (Top-1) (%)	Size (MB)	CR (×)	CR Improved (×)	Speedup (×)	Speedup Improved (×)
ResNet-50	-	32	75.5	102	1	1	1	1
	LSQ	8	75.4	25.5/21.6	4.00/4.72	1.18	3.5/3.8	1.09
		4	75.3	12.8/10.7	7.97/9.56	1.20	3.4/6.6	1.94
		2	72.5	6.4/5.1	15.94/**19.92**	**1.25**	2.9/**7.0**	**2.41**
ResNet-101	-	32	76.4	171	1	1	1	1
	LSQ	8	76.3	42.8/36.6	4.00/4.67	1.17	3.2/3.9	1.22
		4	76.3	21.4/18.0	7.99/9.51	1.19	3.1/7.2	2.32
		2	74.2	10.7/8.8	15.98/**19.50**	**1.22**	2.5/**9.3**	**3.72**
VGG-16	-	32	71.3	533	1	1	1	1
	LSQ	8	71.3	134/117.5	3.98/4.53	1.14	3.4/3.2	0.94
		4	71.5	66.6/56.9	8.00/9.36	1.17	3.1/3.4	**1.10**
		2	69.5	33.3/28.0	16.01/**19.05**	**1.19**	2.6/**3.8**	1.46
DeiT-B	-	32	81.8	347	1	1	1	1
	PQT	8	81.3	86.8/75.5	4.00/4.60	1.15	2.6/3.3	1.27
		4	75.9	43.6/36.9	7.96/9.39	1.18	2.4/5.2	2.17
	*	2	67.2	21.7/18.1	15.99/**19.19**	**1.20**	1.8/**6.7**	**3.72**

After quantization, our method can achieve the maximum level of speedup across all experiments, and still maintain a considerable amount of the reduction in terms of the memory footprint. On ResNet-101 quantized at 2-bit precision, the speedup can be achieved by 9.3×. For other quantized models at 2-bit granularity, ResNet-50/VGG-16/DeiT-B, the speedup can reach to 7.0×/3.8×/6.7×. As for the compress ratio, combining quantization and our method can further compress the models to a certain extent. For all four models selected for our experiments, our method can realize more than 19× compression ratio.

We make the key observations that: similar to our methods on Pruning, our method on quantized models also bring more benefits on CONV-dominated models, rather than FC-dominated models. For VGG-16 with large FC layers, the improvement of speedup is restricted to 1.10× at most. This is similar with the results when our method is applied on pruned models. We also report results by applying our method on DeiT-B. Though this model is distinctively different from CNNs, our method can still provide a considerable amount of space reduction and inference acceleration, by up to 19.19× compression and 6.7× speedup.

7.4 A Comparison against Other Model Coding Methods

Finally, we compare our method with the state-of-the-art Model Coding methods, including Rate-distortion Optimized Coding (ROC) (Zhe et al. (2021)), Entropy Penalized Reparameterization (EPR) (Oktay et al. (2020a)), DeepCABAC (Wiedemann et al. (2019)), Minimal Random Code

Table 3: A comparison between Succinct Compression and other Model Coding Methods: for a fair comparison, we only consider the best results from baseline methods. ('-' means the speedup is not capable to be validated, since they are not reported in the original paper.)

Neural Network Models	Method	Size (MB)	CR (×)	Speedup (×)	Error (Top-1) (%)
ResNet-50	Uncompressed	102	1	1	24.5
	OPQ	2.68	38.0	1.3	**24.5**
	Deep Compression	6.41	15.91	0.8	25.7
	DeepCABAC	6.06	16.8	-	25.9
	EPR(DFT)	5.49	18.6	-	26.0
	ROC (Tunstall)	5.10	20.0	4.3	24.9
	Our Method	2.10	**48.56**	**9.2**	**24.5**
MobileNet-V2	Uncompressed	13.2	1	1	29.0
	OPQ	0.57	23.2	1.2	**29.3**
	Deep Compression	0.97	13.62	1.1	30.5
	ROC (Tunstall)	1.2	11.0	1.7	29.8
	Our Method	0.44	**30.11**	**5.6**	**29.3**
Small VGG-16	Uncompressed	60	1	1	6.6
	OPQ	0.13	461.5	1.5	**7.0**
	Deep Compression	1.64	36.57	1.2	7.2
	DeepCABAC	0.95	63.2	-	9.0
	MIRACLE	0.16	375.0	-	10.0
	EPR (DFT)	0.10	600.0	-	10.0
	Our Method	0.09	**641.5**	**4.7**	**7.0**

Learning (MIRACLE) (Havasi et al. (2018)), Deep Compression (Han et al. (2016b)). Our experiments aim to examine the tradeoffs of our method on compression ratio, inference time and accuracy loss. In order to have a fair comparison, the input models for our method are pre-processed using OPQ method for Pruning and Quantization (Hu et al. (2021)), since other approaches also apply Pruning and Quantization methods (or other methods) before applying their methods.

Table 3 reports the results of this experiment. Our results show that our method can significantly outperform the state-of-the-art Model Coding methods on both compression ratio and speedup by a large magnitude, with the lowest loss of the accuracy. On ResNet-50, MobileNet-V2 and Small VGG-16, our method demonstrates the compression ratio up to 48.56×/30.11×/641.5× with conspicuous 9.2×/5.6×/4.7× speedup, with the minimum accuracy loss.

We make the key observation that: incorporating Pruning and Quantization with our method amplifies the benefits significantly to achieve the best outcome throughput the whole paper, compared with solely utilizing either Pruning or Quantization. On ResNet-50, our method outperforms the best results reported in Section 7.2 (i.e. 39.48× compression ratio, 4.7× speedup, 25.3% top-1 error), with 48.56× compression ratio, 9.2× speedup and only 24.5% top-1 error. Second, our method can enable lossless compression and acceleration at the same time. As shown in Table 3, there is no accuracy degradation for all models by using our method.

8 CONCLUSIONS

We present "Succinct Compression", a method to provide lossless compression of Deep Neural Network (DNN) models for fast and memory-efficient inference. Our method exploits *Succinct Data Structures* with three novel insights, which can be generally applicable to a variety of models (i.e. CNNs and Transformers). Our method is also synergistic with both structure-altered/-unaltered compress schemes (such as Pruning and Quantization). Our results justify the above benefits. On uncompressed models, our method provides at most 1.07× speedup with 1.17× compression ratio. The benefits from our method can be further amplified. By incorporating Pruning (and/or Quantization), we can achieve the maximum speedup/compression ratio at 9.3×/641.5×. We extensively examine and show that our method can significantly outperform six other Model Coding methods.

REFERENCES

Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pp. 337–350. USENIX Association, 2015. URL `https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal`.

Dmitry Alexeev. Fortran Bindings in PyTorch. URL `https://github.com/alexeedm/pytorch-fortran`.

Milad Alizadeh, Arash Behboodi, Mart van Baalen, Christos Louizos, Tijmen Blankevoort, and Max Welling. Gradient ℓ_1 regularization for quantization robustness, 2020.

Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. Post-training 4-bit quantization of convolution networks for rapid-deployment, 2019.

Gilad Baruch, Shmuel T. Klein, and Dana Shapira. Accelerated partial decoding in wavelet trees. *Discrete Applied Mathematics*, 274:2–10, 2020. ISSN 0166-218X. doi:https://doi.org/10.1016/j.dam.2018.07.016. URL `https://www.sciencedirect.com/science/article/pii/S0166218X18303974`. Stringology Algorithms.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation, 2013.

Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning?, 2020.

Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. High-capacity expert binary networks, 2021.

Yaohui Cai, Zhewei Yao, Zhen Dong, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. Zeroq: A novel zero shot quantization framework, 2020.

Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization, 2017.

Tien-Fu Chen. *Data Prefetching for High-Performance Processors*. PhD thesis, USA, 1993. UMI Order No. GAX94-09294.

Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995. doi:10.1109/12.381947.

Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015. ISSN 0306-4379. doi:https://doi.org/10.1016/j.is.2014.06.002. URL `https://www.sciencedirect.com/science/article/pii/S0306437914000945`.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255, 2009. doi:10.1109/CVPR.2009.5206848.

Xin Dong, Shangyu Chen, and Sinno Jialin Pan. Learning to prune deep neural networks via layer-wise optimal brain surgeon, 2017.

Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization, 2020.

Haozhe Feng, Zhaoyang You, Minghao Chen, Tianye Zhang, Minfeng Zhu, Fei Wu, Chao Wu, and Wei Chen. KD3A: unsupervised multi-source decentralized domain adaptation via knowledge distillation. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 3274–3283. PMLR, 2021. URL `http://proceedings.mlr.press/v139/feng21f.html`.

Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019.

Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pp. 841–850, USA, 2003. Society for Industrial and Applied Mathematics. ISBN 0898715385.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016a. URL `http://arxiv.org/abs/1510.00149`.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun (eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016b. URL `http://arxiv.org/abs/1510.00149`.

Marton Havasi, Robert Peharz, and José Miguel Hernández-Lobato. Minimal random code learning: Getting bits back from compressed model parameters, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices, 2019.

Peng Hu, Xi Peng, Hongyuan Zhu, Mohamed M. Sabry Aly, and Jie Lin. OPQ: compressing deep neural networks with one-shot pruning-quantization. In *Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021*, pp. 7780–7788. AAAI Press, 2021. URL `https://ojs.aaai.org/index.php/AAAI/article/view/16950`.

Zehao Huang and Naiyan Wang. Data-driven sparse structure selection for deep neural networks, 2018.

J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. ISSN 0304-3975. doi:https://doi.org/10.1016/j.tcs.2015.11.011. URL `https://www.sciencedirect.com/science/article/pii/S0304397515010087`. Pattern Matching, Text Data Structures and Compression.

Guy Joseph Jacobson. *Succinct Static Data Structures*. PhD thesis, USA, 1988. AAI8918056.

Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In Katerina J. Argyraki and Rebecca Isaacs (eds.), *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016*, pp. 485–500. USENIX Association, 2016. URL `https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/khandelwal`.

Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 1149–1164. ACM, 2017. doi:10.1145/3035918.3064012. URL `https://doi.org/10.1145/3035918.3064012`.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Snip: Single-shot network pruning based on connection sensitivity, 2019.

Namhoon Lee, Thalaiyasingam Ajanthan, Stephen Gould, and Philip H. S. Torr. A signal propagation perspective for pruning neural networks at initialization, 2020.

Chao Li, Zhun Sun, Jinshi Yu, Ming Hou, and Qibin Zhao. Low-rank embedding of kernels in convolutional neural networks under random shuffling, 2018.

Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map, 2020.

Shaohui Lin, Rongrong Ji, Yuchao Li, Yongjian Wu, Feiyue Huang, and Baochang Zhang. Accelerating convolutional networks via global dynamic filter pruning. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pp. 2425–2432. International Joint Conferences on Artificial Intelligence Organization, 7 2018. doi:10.24963/ijcai.2018/336. URL `https://doi.org/10.24963/ijcai.2018/336`.

Zhenhua Liu, Yunhe Wang, Kai Han, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer, 2021.

Christos Louizos, Karen Ullrich, and Max Welling. Bayesian compression for deep learning, 2017.

Yifan Lu, Lu Yang, Virendrakumar C. Bhavsar, and Neetesh Kumar. Tree structured data processing on gpus. In *2017 7th International Conference on Cloud Computing, Data Science Engineering - Confluence*, pp. 498–505, 2017. doi:10.1109/CONFLUENCE.2017.7943203.

Franck Mamalet and Christophe Garcia. Simplifying convnets for fast learning. 09 2012. ISBN 978-3-642-33265-4. doi:10.1007/978-3-642-33266-1_8.

Joe Mellor, Jack Turner, Amos J. Storkey, and Elliot J. Crowley. Neural architecture search without training. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 7588–7598. PMLR, 2021. URL `http://proceedings.mlr.press/v139/mellor21a.html`.

Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014. ISSN 1570-8667. doi:https://doi.org/10.1016/j.jda.2013.07.004. URL `https://www.sciencedirect.com/science/article/pii/S1570866713000610`. 23rd Annual Symposium on Combinatorial Pattern Matching.

NVIDIA. Bit Manipulation Functions (Table 6) in CUDA FORTRAN Programming Guide, a. URL `https://docs.nvidia.com/hpc-sdk/compilers/cuda-fortran-prog-guide/`.

NVIDIA. NVIDIA: How to access global memory efficiently in CUDA C/C++ kernels. `https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/`, b. Accessed: 2022-08-26.

NVIDIA. NVIDIA: Hardware implementation. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#hardware-implementation`, c. Accessed: 2022-08-26.

Deniz Oktay, Johannes Ballé, Saurabh Singh, and Abhinav Shrivastava. Scalable model compression by entropy penalized reparameterization, 2020a.

Deniz Oktay, Johannes Ballé, Saurabh Singh, and Abhinav Shrivastava. Scalable model compression by entropy penalized reparameterization, 2020b.

Sejun Park*, Jaeho Lee*, Sangwoo Mo, and Jinwoo Shin. Lookahead: A far-sighted alternative of magnitude-based pruning. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=ryl3ygHYDB`.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina

Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019a. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019b. URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Brandon Reagen, Udit Gupta, Robert Adolf, Michael M. Mitzenmacher, Alexander M. Rush, Gu-Yeon Wei, and David Brooks. Weightless: Lossy weight encoding for deep neural network compression, 2017.

Tara N. Sainath, Brian Kingsbury, Vikas Sindhwani, Ebru Arisoy, and Bhuvana Ramabhadran. Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6655–6659, 2013. doi:10.1109/ICASSP.2013.6638949.

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Herve Jegou. Training data-efficient image transformers amp; distillation through attention. In *International Conference on Machine Learning*, volume 139, pp. 10347–10357, July 2021.

B. P. Tunstall. Synthesis of noiseless compression codes. 1967.

Jan van Leeuwen. On the construction of huffman trees. In S. Michaelson and Robin Milner (eds.), *Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, UK, July 20-23, 1976*, pp. 382–410. Edinburgh University Press, 1976.

Wenxiao Wang, Shuai Zhao, Minghao Chen, Jinming Hu, Deng Cai, and Haifeng Liu. Dbp: Discrimination based block-level pruning for deep model acceleration, 2019.

Zi Wang. Zero-shot knowledge distillation from a decision-based black-box model. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 10675–10685. PMLR, 2021. URL http://proceedings.mlr.press/v139/wang21a.html.

Simon Wiedemann, Heiner Kirchhoffer, Stefan Matlage, Paul Haase, Arturo Marban, Talmaj Marinc, David Neumann, Ahmed Osman, Detlev Marpe, Heiko Schwarz, Thomas Wiegand, and Wojciech Samek. Deepcabac: Context-adaptive binary arithmetic coding for deep neural network compression, 2019.

Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, jun 1987. ISSN 0001-0782. doi:10.1145/214762.214771. URL https://doi.org/10.1145/214762.214771.

XIA XIAO, Zigeng Wang, and Sanguthevar Rajasekaran. Autoprune: Automatic network pruning by regularizing auxiliary parameters. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/4efc9e02abdab6b6166251918570a307-Paper.pdf.

Hao Yu and Jianxin Wu. A unified pruning framework for vision transformers, 2021.

Ruichi Yu, Ang Li, Chun-Fu Chen, Jui-Hsin Lai, Vlad I. Morariu, Xintong Han, Mingfei Gao, Ching-Yung Lin, and Larry S. Davis. Nisp: Pruning networks using neuron importance score propagation, 2018.

Shixing Yu, Zhewei Yao, Amir Gholami, Zhen Dong, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. Hessian-aware pruning and optimal neural implant, 2021.

Chenglong Zhao, Bingbing Ni, Jian Zhang, Qiwei Zhao, Wenjun Zhang, and Qi Tian. Variational convolutional neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2780–2789, 2019.

Qibin Zhao, Masashi Sugiyama, and Andrzej Cichocki. Learning efficient tensor representations with ring structure networks, 2017.

Yiyang Zhao, Linnan Wang, Yuandong Tian, Rodrigo Fonseca, and Tian Guo. Few-shot neural architecture search. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12707–12718. PMLR, 2021. URL http://proceedings.mlr.press/v139/zhao21d.html.

Wang Zhe, Jie Lin, Mohamed Sabry Aly, Sean Young, Vijay Chandrasekhar, and Bernd Girod. Rate-distortion optimized coding for efficient cnn compression. In *2021 Data Compression Conference (DCC)*, pp. 253–262, 2021. doi:10.1109/DCC50243.2021.00033.

Zhuangdi Zhu, Junyuan Hong, and Jiayu Zhou. Data-free knowledge distillation for heterogeneous federated learning. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pp. 12878–12889. PMLR, 2021. URL http://proceedings.mlr.press/v139/zhu21b.html.

A Additional Details of the Experiments

We disclose additional experimental details to make our work reproducible by others. More specifically, we focus on the detailed configurations of RAS types for different layers in the Deep Neural Network models, which is the most influential factor on the inference performance.

To this end, we provide additional details on how we decide the usage of different RAS and the quantitative supports, based on the requirement described in Section 3.3. Since our selection criteria is based on *FPr* (FLOPs-Parameter-ratio), we also provide relevant details for each layer.

Table A demonstrates the detailed results of this method on ResNet-50. As explained in our paper, the CONV layers demand Block-wise RAS for coarse-grained accesses, since accesses to these layers are relatively frequent; and the FC layers can be used via Element-wise RAS, since accesses to these layers are less frequent.

Table 4: **ResNet-50 MACs and RAS Type**

Layers	#Parameters	MACs	MACs/#Parameters	RAS Type
conv1	9.41k	118.01M	12541	Block-wise
layer1.conv	215.81k	680.39M	3153	Block-wise
layer2.conv	1.22M	1.04G	852	Block-wise
layer3.conv	7.1M	1.47G	207	Block-wise
layer4.conv	14.96M	811.02M	54213	Block-wise
fc	2.05M	2.05M	1	Element-wise
Total	**25.6M**	**4121.47M**	**161**	

The same trend of the RAS distribution can also be found on VGG-16, where FC layers accounts for around 90% of all model parameters. Such an overwhelming proportion directly results in the fact that: the inference performance of our method on VGG-16 is significantly weakened, compared to the results on other models (where FC layers occupy less space).

Table 5: **VGG-16 MACs and RAS Type**

Layers	#Parameters	MACs	MACs/#Parameters	RAS Type
conv1	1.79k	89.92M	50179	Block-wise
conv2	36.93k	1.85G	50095	Block-wise
conv3	73.86k	926.45M	12543	Block-wise
conv4	147.58k	1.85G	12536	Block-wise
conv5	295.17k	925.65M	3136	Block-wise
conv6	590.08k	1.85G	3135	Block-wise
conv7	590.08k	1.85G	3135	Block-wise
conv8	1.18M	925.25M	784	Block-wise
conv9	2.36M	1.85G	783898	Block-wise
conv10	2.36M	1.85G	783898	Block-wise
conv11	2.36M	462.52M	196	Block-wise
conv12	2.36M	462.52M	196	Block-wise
conv13	2.36M	462.52M	196	Block-wise
fc1	102.76M	102.76M	1	Element-wise
fc2	16.78M	16.78M	1	Element-wise
fc3	4.1M	4.1M	1	Element-wise
Total	**138.355M**	**15478M**	**112**	

B INFERENCE ENGINE

As for our hand-crafted inference engine, some key implementation details (like tensor interface, Wavelet Tree querying module and querying schedule module) are to be illustrated comprehensively for the easy reproduction of our experimental results.

Our inference engine is comprised of three parts, which collaboratively work together to maximize the inference performance on Wavelet Tree while keeping compatible with current prevalent deep learning framework (e.g., PyTorch Paszke et al. (2019a)). As shown in Figure 3, the top section of this framework is a well-designed tensor interface merging the crevasse between tensor calculation and Wavelet Tree access. In the following discussion, we will verify that by leveraging this interface, our inference engine can directly benefit from the already ripe modules in PyTorch with fewest engineering efforts. What's under this interface is a GPU-accelerated Wavelet Tree querying module. It's the mainstay of our inference framework which supports the fast inference directly on compressed data. Our Wavelet Tree inference scheduler, next to querying module, plays the role as a controller of Wavelet Tree querying module with the goal of prefetching necessary data to mitigate access latency.

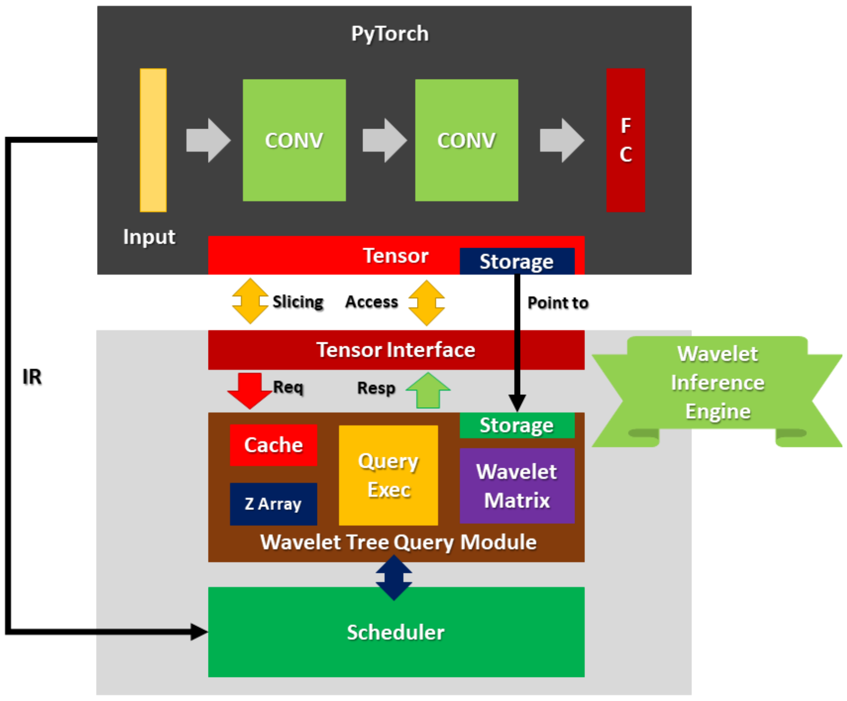

Figure 3: Wavelet Inference Engine

B.1 TENSOR INTERFACE

We place this interface under the implementation of PyTorch tensor as an override of its storage, slice and data access mechanisms, which will not alter tensor's outer API and, hence, achieve the maximal compatibility with PyTorch's other modules. In the following section, we will demonstrate the reason and rationale behind our override.

The data storage format of PyTorch's tensor becomes inefficient in the case of managing a complicated succinct data structure, and, this urges us to reorganize its data arrangement to lay a foundation for our fast Wavelet Tree inference. In PyTorch's design, it separates the calculation logic and data storage for its most important component tensor. Data storage is managed by a special module with the assumption that all the data can be stored as a whole block of continuous memory, and the interpretation of these data (e.g., slice and view) is easy to be offloaded to tensor's implementation. However, this assumption doesn't hold true when we decide to use Wavelet Tree as our underlying data structure, since, in succinct data structure, query mechanism and data arrangement format are highly associated, which necessitates the co-design of both if we want to realize high throughput and low latency. Therefore, to boost our runtime performance, tensor's data storage is displaced with our fine-tuned data structure for Wavelet Tree. We encourage interested readers to have a closer look at it in Section B.2.

Slicing is an indispensable function of tensor manipulation, and we strive to support this function by replacing original offset-based slicing with Wavelet Tree's select primitive. On our RAS formulation, select operation is a native equivalent of slicing. By leveraging the delimiters we inject to RAS, select can locate any sub parts within constant time. For instance, there is a slicing request $T[4][2][1]$ on a 4-dimension tensor T whose size is $(128, 3, 224, 224)$. To response this request, we can execute 5 select operations in order to get the required range, which are $select_{d1}(5)$, $select_{d2}(3)$, $select_{d3}(2)$ and $select_{d3}(3)$ (where $d1$, $d2$ and $d3$ are the delimiters for dimension-1/2/3, respectively). Considering this equivalence relationship between select and slicing, we, hence, are able to substitute classic slicing with Wavelet-Tree-friendly select operations.

Address-based access, another rudimentary operation supported by tensor, is also incompatible with Wavelet Tree, and this enforces us to override original access function with Wavelet-Tree-specific access query. Unlike tensor whose data are arranged continuously, data in Wavelet Tree are stored in a tree-like structure with positions specifically decided by Wavelet Tree construction process (e.g., Ian Munro et al. (2016)), which disables the direct address-based access on it. To query data in Wavelet Tree, traditional practice is to conduct recursive $rank_0$ queries on each level's bitmap Navarro (2014). However, as the size of bitmap goes up (e.g., to 1MB), the query process becomes incredible slow (for instance, 1ms per access), and will eventually slow down the whole inference. In an attempt to address this issue, we propose a combination of Wavelet Matrix Claude et al. (2015) and Accelerated Decoding Baruch et al. (2020) to accelerate access query on GPU. We refer interested readers to Section B.2 for an understanding of its detailed pipeline.

In this section, we demonstrate how to connect underlying data stored in Wavelet Tree seamlessly with upper modules through a well defined tensor interface overriding the three basic functions (i.e., storage, slicing and data access). From the perspective of upper modules (like conv2d), the behavior of tensor doesn't change after our replacement, and, therefore, can be reused by our inference engine without any additional engineering effort.

B.2 Wavelet Tree Querying Module

As mentioned in Section B.1, co-architecturing data storage and corresponding query on it (for example, access and select) is the key to boosting the performance of succinct data structure like Wavelet Tree to meet the requirement of high throughput and low latency when it comes to neural network inference. Hereby, we will articulate the architecture design details in our Wavelet Tree Querying Module from the aspects of tailor-made data storage format and highly-optimized query algorithms.

For data storage, instead of storing a tree-like structure which is the standard formulation of Wavelet Tree, we rearrange the Wavelet Tree to be formed in a GPU-friendly format, 2D array. Compared to common-used linear data structure like 2D array, tree structure faces the difficulty in data transfer and runtime access. Since GPU prefers to transfer linear data structure to maximize its throughput, the existence of tree pointers makes the data copy complicated and might cripple the overall performance Lu et al. (2017). Besides this, coalesced data access is also hard to be conducted on tree structure for the reason that long distance between two tree data accesses makes it impossible to fetch them at once, considering the limited bandwidth (for example, 128 bytes) NVIDIA (b). Therefore, traditional Wavelet Tree might can't be accelerated on a GPU platform. To address this issue, we leverage Wavelet Matrix Claude et al. (2015), an equivalent of Wavelet tree but expressed as 2D array, to realize our Wavelet Tree on GPU. As shown in Figure 4, our implementation of a Wavelet Tree now becomes a matrix with bitmaps, which, under the hood, can be stored in a 1D byte array. This linear data structure enables us to fully utilize GPU's features and capability.

To mitigate the high access latency in traditional Wavelet Tree query algorithms, we contrive a new one with outstanding runtime performance under the assistance of cache. Classic Wavelet Tree query algorithms (like the methodologies described in Navarro (2014)) usually have a poor performance on workloads with massive assess operations. As reported in Baruch et al. (2020), one access query

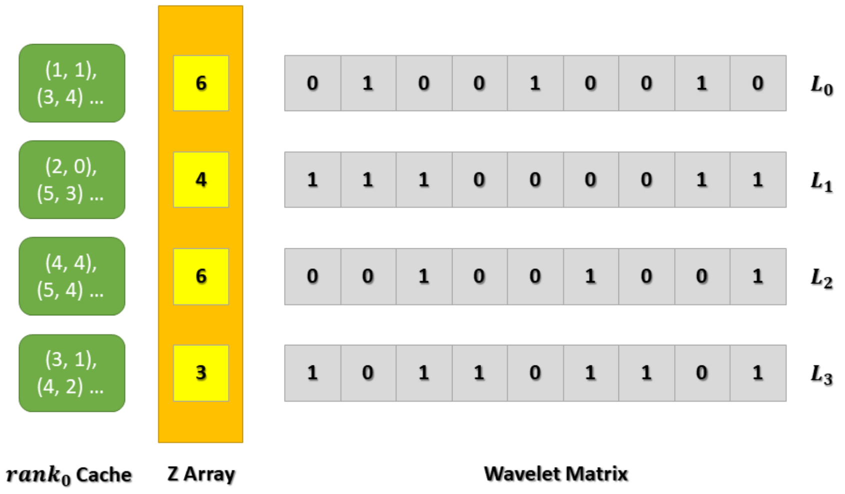

Figure 4: Wavelet Tree Query Module

on a text file of size 200MB suffers from millisecond-level latency, a prohibitive cost especially in a access-latency sensitive scenario like neural network inference. In order to resolve this issue, we first point out the root cause of high latency by doing ablation study on access operation and then come up with a cache-based acceleration algorithm. After studying the time consumption of access operation step-by-step, we find out that $rank_0$ function [2] accounts for more than 90% computation time and is usually overlapped with other $rank_0$ calculations if we conduct access on adjacent elements. For instance, we need to call twice $rank_0$ functions per layer, even though we are accessing two consecutive elements in a string, in which case the calculation result of former $rank_0$ should have been able to be reused by latter one. Baruch et al. (2020) has already proposed a solution for this by keeping the previous $rank_0$ result in a specific register. We further extend their work by specially allocating a block of cache for each layer to facilitate the reuse of previous results. Details of our method can be seen in Figure 4. The previous calculation results are stored as tuples in per layer cache, and when we invoke a new $rank_0$ function, it can directly start from the nearest recording. For a formal description of our algorithm[3], we display its pseudo-codes in Algorithm 1.

Please note that, in practice, too many branches in program's execution path will limit its performance on GPU, considering the restricted control ability of GPU's SIMT architecture NVIDIA (c). Therefore, we invoke thread blocks on vertically partitioned Wavelet Matrix segments with each one containing at most three levels, and bridge block-wise data transference through global memory.

In this section, we cover the two co-designed components of our Wavelet Tree inference module, which are Wavelet-Matrix-based data storage format and fast Wavelet Tree querying algorithm. Built on these two parts, our inference module can achieve negligible runtime latency.

B.3 Querying Schedule Module

We conduct software-based prefectching to further hide our data access latency. We take intermediate representation (IR) of the program as input and generate prefetching plan according to the data access operations in IR. Unlike traditional software-based prefetching on CPU whose plans are usually hard-coded into additional instructions in program's executable files Chen (1993), our prefetching plan is actually comprised of several match-action entries and will be executed in a lazy way. For instance, one entry in our plan can be a tuple

[2] $rank_0$ is performed on each layer's bitmaps to count the number of zeros in a certain range.

[3] Strictly speaking, our algorithm also contains the rank and select primitives for Wavelet Tree. However, their implementations are very similar to Claude et al. (2015). You can find detailed description in that paper.

Algorithm 1 Access(x)

```
Require: $0 \leq x < the\ length\ of\ bitmap$
Ensure: $symbol,\ where\ symbol\ is\ the\ x_{th}\ element\ of\ original\ string$
  $l \leftarrow 0$
  $r \leftarrow len(\sigma)$                                  ▷ $\sigma$ is the alphabet
  $level \leftarrow 0$
  $B \leftarrow WM[level]$                    ▷ $WM$ is the Wavelet Matrix comprised of bitmaps
  while $r - l \neq 1$ do
      $(y, preResult) \leftarrow GetNearestRecording(x)$
      $q \leftarrow rank_0(B, y, x) + preResult$          ▷ $rank_0$ counts the zeros between y and x
      $StoreRecording(x, q)$
      if $B[x] = 0$ then
          $x \leftarrow q$
          $r \leftarrow (r - l)/2$
      else
          $x \leftarrow Z[level] + (x - q)$      ▷ $Z[level]$ is the number of zeros in current level
          $l \leftarrow (r - l)/2$
      end if
      $level \leftarrow level + 1$
      $B \leftarrow WM[level]$
  end while
  $symbol \leftarrow \sigma[l]$
```

$(access(x), [access(y_0),\ access(y_1),\ ...,\ access(y_n)])$ whose first element is the key and the second place holds the actions. When we access data x, prefetching on positions $y_0,\ y_1,\ ...,\ y_n$ will be performed in a subprocess simultaneously to mitigate the data loading latency. By doing so, a considerable speedup (approximately 10%) can be observed on our experiment platform equipped with NVIDIA RTX3090. However, other possible schedulings are not tested in this work due to implementation difficulties. Hardware-based prefetching usually outperforms software-based ones because there are no overheads of additional prefetching instructions and the optimization can be characterized according to hardware's internal status at runtime Chen & Baer (1995). Besides this, a more fine-grained software-based prefetching might also be attractive. Instead of conducting prefetching at the granularity of Wavelet Tree's access operation, we might try to plan our prefetching by taking GPU's intrinsics into consideration. We encourage interested researchers to test these possibilities, and we are also going to do a more comprehensive performance analysis in a separate work in the near future.