eDKM: An Efficient and Accurate Train-time Weight Clustering for Large Language Models

ABSTRACT

Since Large Language Models or LLMs have demonstrated high-quality performance on many complex language tasks, there is a great interest in bringing these LLMs to mobile devices for faster responses and better privacy protection. However, the size of LLMs (i.e., billions of parameters) requires highly effective compression to fit into storage-limited devices. Among many compression techniques, weight-clustering, a form of non-linear quantization, is one of the leading candidates for LLM compression, and supported by modern smartphones. Yet, its training overhead is prohibitively significant for LLM fine-tuning. Especially, Differentiable KMeans Clustering, or DKM, has shown the state-of-the-art trade-off between compression ratio and accuracy regression, but its large memory complexity makes it nearly impossible to apply to train-time LLM compression. In this paper, we propose a memory-efficient DKM implementation, eDKM powered by novel techniques to reduce the memory footprint of DKM by orders of magnitudes. For a given tensor to be saved on CPU for the backward pass of DKM, we compressed the tensor by applying uniquification and sharding after checking if there is no duplicated tensor previously copied to CPU. Our experimental results demonstrate that eDKM can fine-tune and compress a pretrained LLaMA 7B model from 12.6 GB to 2.5 GB (3bit/weight) with the Alpaca dataset by reducing the train-time memory footprint of a decoder layer by 130×, while delivering good accuracy on broader LLM benchmarks (i.e., 77.7% for PIQA, 66.1% for Winograde, and so on).

1. INTRODUCTION

Large language models or LLMs, and especially Generative Pre-trained Transformer (GPT) models have shown excellent performance on many complex language tasks [11, 23]. Such breakthrough leads to the desire to run these LLMs locally on mobile devices for user privacy [20, 21], but even small LLMs are too big for on-device execution. For example, the smallset LLaMA model has 7B parameters which is 14GB in FP16 [18], while high-end mobile devices have only up to 18GB DRAM. Therefore, aggressively compressing LLMs via train-time optimizations, such as sparsification, quantization, or weight clustering, is a crucial step for on-device LLM deployment [3, 5, 6, 8, 12, 13, 14, 15, 16, 16, 19, 20, 22, 24]

However, train-time optimization of LLM is highly expensive due to the model size and computational resource overheads. Especially, the computational resource demand from a train-time differentiable weight clustering in DKM [3], one of the state-of-the-art weight clustering algorithm is prohibitively high, as it needs to analyze the interactions between all the weights and all possible clustering options. Accordingly, many existing LLM compression techniques, such as GTPQ [7], and AWQ [9] rely on post-training optimization.

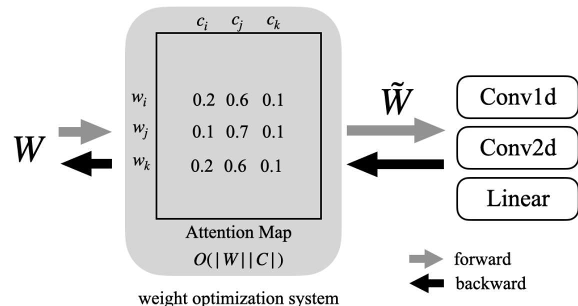

Figure 1: General overview of weight optimization systems. For DKM [3], an attention map for differentiable weight clustering is created inside the system.

In this work, we propose memory optimization techniques to enable train-time weight clustering and their applications to DKM [3], leading to eDKM. Our techniques include cross-device tensor marshaling and weight matrix uniquification/sharding. When we used eDKM to fine-tune and compress LLaMA 7B model into 3bit-per-weight, we achieved about 130× memory footprint reduction for a decoder stack, yet outperformed the existing 3bit compression techniques.

2. MEMORY-EFFICIENT DKM

Pruning, quantization, and normalization are all popular weight optimization techniques/systems that take in the original weights, W and output optimized weights $\tilde{W}$ for inference latency, test accuracy, or model size, as shown in Fig 1. Among techniques, we focus on weight clustering, notably the state-of-the-art train-time weight clustering algorithm, DKM [3]. Weight clustering is a non-linear weight discretization, and a weight matrix will be compressed into a lookup table and a list of low-precision indices to the lookup table, which can be consumed by modern inference accelerators [1].

DKM performs differentiable weight clustering by analyzing the interaction between the weights (denoted W) and centroids (denoted C), and has shown state-of-the-art trades-off between compression ratio and accuracy. Therefore, using DKM for LLM compression would yield high-quality result. However, DKM computes a large attention map with $O(|W||C|)$ memory complexity (i.e., the matrix in Fig. 1) for

line	code	GPU	CPU
0	x0 = torch.rand([1024,1024])	4	0
1	x1 = x0.view(-1,1)	4	0
2	y0 = x0.to('cpu')	4	4
3	y1 = x1.to('cpu')	4	8

Table 1: LLM fine-tuning may need to use CPU memory to offload large activations. Lacking cross-device tensor management can lead to redundant copies across devices (especially when the computation graph is complex), which can be particularly undesirable for LLM train-time optimization. For example, although $x0$ and $x1$ are the same tensor with just a different view, when copied to CPU, the resulting tensors y_0 and y_1 do not share the data storage while x_0 and x_1 do on GPU.

forward/backward passes (see the Appendix in [3]), which is particularly challenging for LLM compression. For example, a LLaMA 7B model needs at least 224GB just to compute an attention map for 4bit weight clustering.

Accordingly, we need to tap onto CPU memory to handle such large memory demand by overflowing to CPU memory and copying back to GPU when needed later. However, it will incur significant traffic between GPU and CPU (slowing down the training), and need immense CPU memory capacity. Hence, it is critical to reduce the number of transactions between CPU and GPU, and minimize the traffic of each transaction. To address such challenges, we introduce two novel memory optimization techniques in PyTorch.

- Cross-Device Tensor Marshaling: We track tensors being copied across devices and avoid redundant copying to reduce the memory footprint and expedite training.
- Weight Uniquification and Sharding: We use the fact that weights in 16 bits have only 2^{16} unique values to reduce the attention map (in Fig 1) representation and further shard it over multiple learners.

2.1 Cross-device Tensor Marshaling

PyTorch represents a tensor with data storage that links to the actual data layout and metadata that keeps the tensor shapes, types, and so on. Such tensor architecture lets PyTorch reuse the data storage whenever possible and efficiently reduces the memory footprint. However, when a tensor moves to another device (i.e., from GPU to CPU), the data storage cannot be reused and a new tensor needs to be created. Table 1 shows an example of the memory footprint overhead when a tensor moves between devices in PyTorch. The tensor, x_0 allocated in line 0, consumes 4MB on GPU. When its view is changed in line 1, no additional GPU memory is required as the underlying data storage can be reused (i.e., x_0 and x_1 are effectively identical). However, when x_0 and x_1 move to CPU as in lines 2 and 3, the CPU memory consumption becomes 8MB, although y_0 and y_1 could share the same data storage on CPU, which leads to the redundancy on CPU memory and increases GPU-CPU traffic.

To address such inefficiency, we place a marshaling layer as in Fig. 2 (b), where the black represents actual data storage and metadata, and the gray indicates only the metadata. Fig. 2 (a) illustrates the example in Table 1 (with the corresponding line numbers) where x_1 shares the data layout with x_0 but y_0 and y_1 have independent/duplicated data storage on CPU. By inserting a marshaling layer as in Fig. 2 (b), we avoid such redundancy and reduce the GPU-CPU traffic.

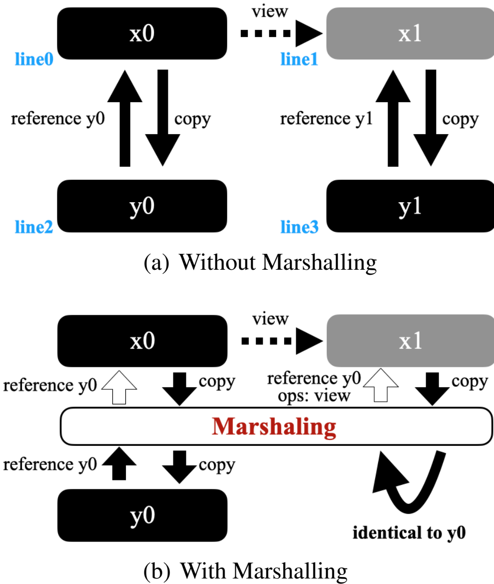

Figure 2: When the proposed cross-device tensor marshalling is applied to the case in Table 1, we can avoid duplication on the CPU side, which saves the memory/traffic. Before copying x_1 to CPU, our marshaling scheme checks if there exits tensor with the same data storage on the CPU (i.e., y_0). If there is, we reuse the reference for y_0 along with the required ops (*view* in this case) for future retrieval.

We use the *save-tensor-hook* in PyTorch (see [2] for reference) to implement such a marshaling scheme, where we examine if the same data storage has been already copied. However, checking whether the same tensor exists on the destination device is prohibitively expensive when using a convention scheme like hashing. Therefore, when a new tensor enters our marshaling system, we turn to the forward graph and check if there exists another tensor that is already on CPU and is reachable via only data-storage invariant operations (i.e., *view, transpose*, ...) from the new tensor within a few hops. If not found, the tensor is copied and a reference to the tensor is generated. If found, we return the reference of the existing tensor and the list of operations tracing back to the new tensor. For the example in Fig. 2 (b), instead of copying x_1 to CPU, we simply return the reference to y_0 and the *view* operation between x_1 and y_0.

Navigating the computation graph costs extra compute cycles, but saving on an unnecessary copy can compensate for such overhead. We found that searching within 4 hops is sufficient to detect all the qualified cases in the computation graph from the original DKM implementation.

2.2 Weights Uniquification and Sharding

In most LLM training, 16bit (e.g., BF16 or FP16) is widely used for weights, which means although there are multi-

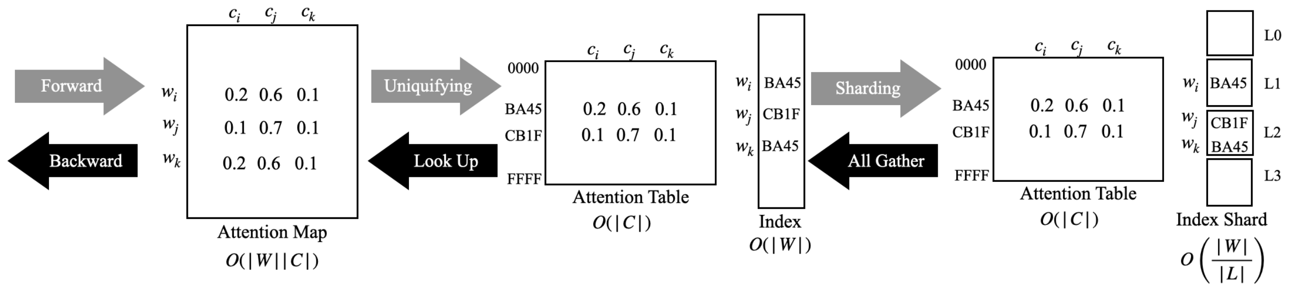

Figure 3: Weight Uniquification and Sharding: since w_i and w_k have the same bit value (BA45), both can share the same attention to centroids in the attention table, yet use the bit value as the offset to the table in the index list.

billion parameters in LLMs, there are only 2^{16} unique coefficients due to the bit-width. This allows an opportunity to significantly compress the attention map between weights and the centroids, as in Fig 3. By computing the attention to the centroids once for each unique weight value, the attention map can be converted into an attention table with $O(|C|)$ and the index list with $O(|W|)$. Note that the number of rows in the attention table is at most 65,536.

The index list (denoted L) can be further sharded over a set of learners (i.e., GPUs) in a fully synchronous training setup [4], as the weights are identical in each learner at any moment (thus, attention map and index list too). Such sharding will bring down the memory complexity to $O(\frac{|W|}{|L|})$. Uniquifying and sharding come with higher communication and computation costs, as the sharded weights need to be all-gathered and the attention table and index list need to be converted back to the attention map for backward propagation (see Table 2 for the runtime overhead).

Assume $\{w_i, w_j, w_k\} \in W$ and $\{c_i, c_j, c_k\} \in C$, which denote the weights and centroids respectively in Fig. 3. Further consider the case where $\{w_i, w_k\}$ have the same 16bit representation $BA45$ and w_j has $CB1F$. Then, when an attention map is computed during forward pass, w_i and w_k shall have the same attention to C. After uniquification, the attention map is decomposed into an attention table with $O(|C|)$ memory complexity and an index list with $O(|W|)$ complexity. For example, the 16bit value, $BA45$ of w_i and w_k can serve as an offset to the attention table in the index list. The index list can be further sharded over $|L|$ learners to reduce the complexity in each learner into $O(\frac{|W|}{|L|})$. The original attention map needs to be reconstructed for backward pass to stay compatible with the existing autograd implementation. Therefore, we take the reverse steps to restore the attention map by performing all-gather and look-up.

3. EXPERIMENTAL RESULTS

We used the PyTorch 2.0.01 and applied Fully Sharded Data Parallel (FSDP) to fine-tune the pretrained LLaMA 7B model in brainfloat16 with the Alpaca dataset [17]. We fine-tuned for 2 epochs while compressing the model on a single node with 8× A100-80GB GPUs using eDKM. The maximum sequence length during fine-tuning was 256. We used AdamW optimizer with learning rate as 5e-5, weight decay as 0, and betas as $(0.9, 0.95)$. The global batch size is 64, and the gradient norm clipping with 1.0 is used.

3.1 LLM Accuracy

We compared eDKM against other quantization-based compression schemes: round-to-nearest (RTN), SmoothQuant, GPTQ [7], AWQ [9] and LLM-QAT [10]. For eDKM, we also compressed the embedding layers with 8 bits.

Table 3 reports the accuracy with Common Sense Reasoning, and Few-Shot benchmarks with the compressed LLaMA 7B models from each technique.

- eDKM allows the 3bit compressed LLaMA 7B model to outperform all other schemes in the 3bit configuration.
- eDKM even delivers the best accuracy for ARC-e benchmarks across 3 and 4bit configurations.
- eDKM yields the competitive performance for PIQA and MMLU benchmarks with 4bit compressed models.

3.2 Ablation Study

For the ablation study, we made an example with one attention layer from the LLaMA 7B decoder stack and measured the trade-off between the memory footprint vs. the forward-backward speed with 3bit compression, as shown in Table 2.

Cross-device tensor marshaling alone reduces the memory footprint by 2.9× with little runtime overhead, and the additional savings of 23.5× and 16.4× are achieved with sharding and uniquification, respectively. When all techniques combined, as in Fig. 3, eDKM offered about 130x reduction. Although these steps require extra computation/communications (i.e., all-gather), the runtime overhead is insignificant, as the traffic between GPU and CPU has decreased substantially.

M[a]	S[b]	U[c]	Memory (MB)	Memmory Reduction (×)	Runtime (sec)
			1600	1	8.67
✓			544	2.9	8.97
✓	✓		68	23.5	9.5
✓		✓	97	16.4	15.9
✓	✓	✓	12	129.9	14.9

[a] M: using marshaling layer
[b] S: using sharding
[c] U: using uniquification

Table 2: Ablation study to understand the effects of each techniques: With the proposed techniques, the memory footprint can be reduced by 130x with 1.7x slow down.

Method	bits	Model Size(GB)	Common Sense Reasoning					Few-shot	
			PIQA	HellaSwag	Winogrande	ARC-e	ARC-c	TriviaQA	MMLU
LLaMA-7B	16	12.6	79.3	76.1	70.0	73.0	48.0	57.0	35.2
RTN	4	3.5	77.3	72.7	66.9	68.8	46.4	44.9	28.9
GPTQ g128[c]	4	3.7	77.2	54.0	65.7	61.6	–[a]	–	–
AWQ g128	4	3.7	78.1	55.8	65.8	66.8	–	–	–
LLM-QAT	4	3.5	78.3	74.0	69.0	70.0	45.0	50.8	30.8
GPTQ g128	3	3.0	70.9	46.8	60.9	66.1	–	–	–
AWQ g128	3	3.0	76.7	53.6	66.1	65.7	–	–	–
eDKM	3	2.5	77.7	54.6	66.1	72.3	40.3	35.2[b]	30.3

[a] The result is not reported for the corresponding scheme; [b] One-shot is applied; [c] Group size is 128.

Table 3: When compared our techniques against the state-of-the-art compression scheme, eDKM offered the smallest model size, yet similar or better accuracy for the broader set of benchmarks with the 3bit compressed LLaMA 7B model.

4. CONCLUSION

In this work, we propose a memory-efficient differentiable weight clustering scheme, eDKM, to provide train-time compression for LLMs. With the proposed techniques, the memory consumption was reduced by almost 130x, and the resulting 3bit compressed LLaMA model yields state-of-the-art accuracy on various LLM-harness benchmarks.

REFERENCES

[1] https://coremltools.readme.io/docs/training-time-palettization.

[2] https://pytorch.org/docs/stable/autograd.html#torch.autograd.graph.saved_tensors_hooks.

[3] M. Cho, K. Alizadeh-Vahid, S. Adya, and M. Rastegari, "DKM: Differentiable K-Means Clustering Layer for Neural Network Compression," in *International Conference on Learning Representations*, 2022.

[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. a. Ranzato, A. Senior, P. Tucker, K. Yang, Q. Le, and A. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012.

[5] Z. Dong, Z. Yao, D. Arfeen, A. Gholami, M. W. Mahoney, and K. Keutzer, "HAWQ-V2: Hessian Aware trace-Weighted Quantization of Neural Networks," in *Advances in Neural Information Processing Systems*, 2020.

[6] A. Fan, P. Stock, B. Graham, E. Grave, R. Gribonval, H. Jégou, and A. Joulin, "Training with quantization noise for extreme model compression," in *International Conference on Learning Representations*, 2021.

[7] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers," in *arXiv*, 2023.

[8] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *International Conference on Learning Representations*, 2016.

[9] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration," *arXiv*, 2023.

[10] Z. Liu, B. Oguz, C. Zhao, E. Chang, P. Stock, Y. Mehdad, Y. Shi, R. Krishnamoorthi, and V. Chandra, "LLM-QAT: Data-Free Quantization Aware Training for Large Language Models," *arXiv*, 2023.

[11] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," in *Advances in Neural Information Processing Systems*, 2022.

[12] E. Park, S. Yoo, and P. Vajda, "Value-aware quantization for training and inference of neural networks," in *European Conference on Computer Vision*, 2018.

[13] S. Park, J. Lee, S. Mo, and J. Shin, "Lookahead: A far-sighted alternative of magnitude-based pruning," in *International Conference on Learning Representations*, 2019.

[14] A. Polino, R. Pascanu, and D.-A. Alistarh, "Model compression via distillation and quantization," in *International Conference on Learning Representations*, 2018.

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

[16] P. Stock, A. Joulin, R. Gribonval, B. Graham, and H. Jégou, "And the bit goes down: Revisiting the quantization of neural networks," in *International Conference on Learning Representations*, 2020.

[17] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Stanford Alpaca: An Instruction-following LLaMA model," https://github.com/tatsu-lab/stanford_alpaca, 2023.

[18] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," in *arXiv*, 2023.

[19] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," in *International Conference on Learning Representations*, 2017.

[20] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[21] J. Wu, Y. Wang, Z. Wu, Z. Wang, A. Veeraraghavan, and Y. Lin, "Deep k-means: Re-training and parameter sharing with harder cluster assignments for compressing deep convolutions," in *International Conference on Machine Learning*, 2018.

[22] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "Nisp: Pruning networks using neuron importance score propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9194–9203.

[23] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," in *arXiv*, 2022.

[24] D. Zhou, X. Jin, Q. Hou, K. Wang, J. Yang, and J. Feng, "Neural epitome search for architecture-agnostic network compression," in *International Conference on Learning Representations*, 2019.