CoMERA: Computing- and Memory-Efficient Training via Rank-Adaptive Tensor Optimization

Zi Yang *
University at Albany, SUNY
zyang8@albany.edu

Ziyue Liu
University of California at Santa Barbara
ziyueliu@ucsb.edu

Samridhi Choudhary
Amazon Alexa AI
samridhc@amazon.com

Xinfeng Xie
Meta
xinfeng@meta.com

Cao Gao
Meta
caogao@meta.com

Siegfried Kunzmann
Amazon Alexa AI
kunzman@amazon.com

Zheng Zhang
University of California at Santa Barbara
zhengzhang@ece.ucsb.edu

Abstract

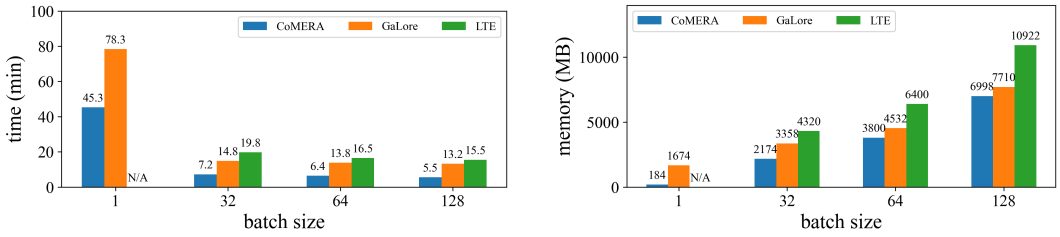
Training large AI models such as deep learning recommendation systems and large language models (LLMs) costs massive GPUs and computing time. The high training cost has become only affordable to big tech companies, meanwhile also causing increasing concerns about the environmental impact. This paper presents CoMERA, a **Co**mputing- and **M**emory-**E**fficient training method via **R**ank-**A**daptive tensor optimization. CoMERA achieves rank-adaptive tensor-compressed (pre)-training via a multi-objective optimization formulation and improves the training to provide both a high compression ratio and excellent accuracy in the training process. Our optimized numerical computation (e.g., optimized tensorized embedding and tensor-network contractions) and GPU implementation eliminate part of the run-time overhead in the tensorized training on GPU. This leads to, for the first time, $2-3\times$ speedup per training epoch compared with standard training. CoMERA also outperforms the recent GaLore in terms of both memory and computing efficiency. Specifically, CoMERA is $2\times$ faster per training epoch and $9\times$ more memory-efficient than GaLore on a tested six-encoder transformer with single-batch training. Our method also shows $\sim 2\times$ speedup than standard pre-training on a BERT-like code-generation LLM while achieving $4.23\times$ compression ratio in pre-training. With further HPC optimization, CoMERA may reduce the pre-training cost of many other LLMs.

1 Introduction

Deep neural networks have gained success in solving numerous engineering problems. These approaches usually use a huge number of variables to parametrize a network, and require massive hardware resources to train the model. For instance, the Deep Learning Recommendation Model (DLRM) released by Meta (which is smaller than the practical product) [32] has 4.2 billion parameters; GPT-3 [4] has 175 billion parameters. OpenAI shows that the computing power required for key AI tasks has doubled every 3.4 months [1] since 2012. Training a large language model like ChatGPT and LLaMA from scratch often takes several weeks or months on thousands of GPUs [3, 2].

*The majority of this work was done when the first author was a postdoc at UC Santa Barbara.

38th Conference on Neural Information Processing Systems (NeurIPS 2024).



(a) Training time per epoch.

(b) Peak memory consumption.

Figure 1: Training time and total memory cost of CoMERA, GaLore [41] and LTE [20] on a six-encoder transformer with varying batch sizes. The experiment is done on Nvidia RTX 3090 GPU.

Large AI models often have much redundancy. Therefore, numerous methods have been developed to reduce the cost of AI inference [5, 25, 7, 13, 16, 31, 30, 17]). However, training large AI models (especially from scratch) remains an extremely challenging task. Low-precision training [19, 27, 12, 36] has been popular in on-device setting, but its memory reduction is quite limited. Furthermore, it is hard to utilize ultra low-precision training on GPU since current GPUs only support limited precision formats for truly quantized training. Chen [6] employed the idea of robust matrix factorization to reduce the training cost. Similar low-rank matrix approximation techniques have been applied to train large AI models including large language models [41, 20]. Among them, GaLore [41] can train the 7B LLaMA model on an RTX 4090 GPU using a single-batch and layer-wise setting. However, this setting can lead to extremely long training time, which is infeasible in practical settings.

Compared with matrix compression, low-rank tensor compression has achieved much higher compression ratios on various neural networks [11, 22, 25, 29, 33, 37, 40, 43]. This idea has been studied in structure search for compact representations [43], post-training compression [11], fixed-rank training [5, 33, 24], zeroth-order training [42] and parameter-efficient fine tuning [39]. The recent work [14, 15] provides a rank-adaptive tensor-compressed training from a Bayesian perspective. However, to achieve a reduction in both memory and training time (especially on transformers), two open problems need to be addressed. Firstly, a more robust rank-adaptive tensor-compressed training model is desired, since the method in [14, 15] relies on a heuristic fixed-rank warm-up training. Secondly, while modern GPUs are well-optimized for large-size matrix computations, they are unfriendly for low-rank tensor-compressed training. Specifically, most operations in tensor-compressed training are small-size tensor contractions, which can cause significant runtime overhead on GPUs even though the theoretical computing FLOPS is very low. As a result, as far as we know no papers have reported real training speedup on GPU. This issue was also observed in [18]. SVDinsTN [43] controls tensor ranks to search for a compact tensor structure of a given tensor. HEAT [11] uses tensor decompositions for post-training model compression of trained models. Detailed comparisons with these works are shown in Appendix A.1.

Paper Contributions. In this work, we propose CoMERA, a tensor-compressed training method that can achieve, for the first time, *simultaneous reduction of both memory and runtime on GPU*. Our specific contributions are summarized as follows.

- **Multi-Objective Optimization for Rank-Adaptive Tensor-Compressed Training**. We propose a multi-objective optimization formulation to balance the compression ratio and model accuracy and to customize the model for a specific resource requirement. One by-product of this method is the partial capability of automatic architecture search: some layers are identified as unnecessary and can be completely removed by rank-adaptive training.
- **Performance Optimization of Tensor-Compressed Training.** While tensor-compressed training greatly reduces the memory cost and computing FLOPS, it often slows down the practical training on GPU. We propose three approaches to achieve real training speedup: ① optimizing the lookup process of tensorized embedding tables, ② optimizing the tensor-network contractions in both forward and backward propagation, ③ eliminating the GPU backend overhead via CUDA Graph.
- **Experimental Results.** We evaluate our method on the end-to-end training of a transformer with six encoders and the deep learning recommendation system model (DLRM). On these two benchmarks, our method achieves $80\times$ and $99\times$ compression ratios respectively, while maintaining the testing accuracy of standard uncompressed training. CoMERA also achieves $2-3\times$ speedup per training epoch compared with standard training methods on the transformer model. In a preliminary study of LLM pre-training, CoMERA shows $1.9\times$ to $2.3\times$ speedup in different pre-training stages on CodeBERT [21], while achieving $4.23\times$ overall model reduction in the pre-training process.

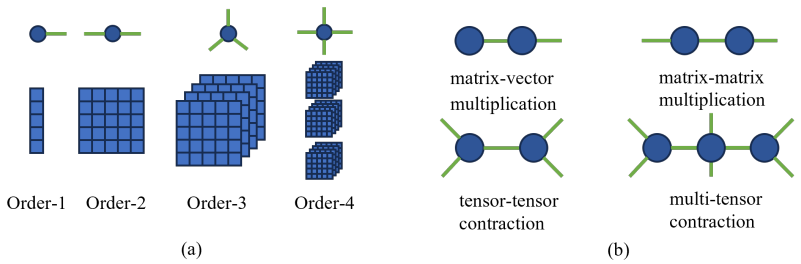


Figure 2: (a) Tensors. (b) Tensor contractions.

Figure 1 compares our CoMERA with GaLore [41] and the recent LoRA-based training method LTE [20] on the six-encoder transformer. When data and back-end memory cost are considered, CoMERA's memory consumption is $9\times$ less than GaLore in the single-batch training as adopted in [41], and it uses the least memory under all batch sizes. Our method is $2-3\times$ faster than GaLore and LTE in each training epoch, although CoMERA has not yet been fully optimized on GPU.

While this work focuses on reducing the memory and computing cost of training, it can also reduce the communication cost by orders of magnitude: only low-rank tensorized model parameters and gradients need to be communicated in a distributed setting. The CoMERA framework can also be implemented on resource-constraint edge devices to achieve energy-efficient on-device learning.

2 Background

The tensor [26, 28] $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is indexed as $\boldsymbol{\mathcal{A}} = (a_{i_1 \cdots i_d})_{1 \le i_j \le n_j}$ and is said to have order d and dimension $n_1, \ldots, n_d$. The Frobenius norm of tensor $\boldsymbol{\mathcal{A}}$ is defined as $\|\boldsymbol{\mathcal{A}}\| := \sqrt{\sum_{i_1,\ldots,i_d}^{n_1,\ldots,n_d} a_{i_1 \cdots i_d}^2}$. In tensor networks, the order-d tensor $\boldsymbol{\mathcal{A}}$ is represented as a node with d edges. Some tensor network representations are illustrated in Fig. 2 (a).

Tensor Contraction. Let $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{l_1 \times l_2 \times \cdots \times l_m}$ be two tensors with $n_s = l_t$. The tensor contraction product $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} \times_{s,t} \boldsymbol{\mathcal{B}}$ has dimension $\Pi_{i \neq s} n_i \times \Pi_{j \neq t} l_j$ and the entries are

$$c_{(i_p)_{p \neq s},(j_p)_{p \neq t}} = \sum_{i_s = j_t = 1}^{n_s} a_{i_1 \cdots i_s \cdots i_m} b_{j_1 \cdots j_t \cdots j_k}. \tag{1}$$

This definition can be naturally generalized to multiple pairs. Figure 2(b) illustrates some tensor contractions. For general operations among multiple tensors, we use PyTorch einsum in the following

$$\boldsymbol{\mathcal{B}} = \textbf{einsum}(S_1, \ldots, S_m \Rightarrow T, [\boldsymbol{\mathcal{A}}_1, \ldots, \boldsymbol{\mathcal{A}}_m]), \tag{2}$$

where each S_i is a string of characters that specifies the dimension of $\boldsymbol{\mathcal{A}}_i$. The output tensor $\boldsymbol{\mathcal{B}}$ is obtained by summing over all other dimensions that are not in T. In the following, we show a few commonly used einsum operations. The Tensor-Train decomposition as in (3) is

$$\boldsymbol{\mathcal{A}} = \textbf{einsum}(n_1 r_1, \ldots, r_{d-1} n_d \Rightarrow n_1 n_2 \cdots n_d, [\boldsymbol{\mathcal{G}}_1, \boldsymbol{\mathcal{G}}_2 \ldots, \boldsymbol{\mathcal{G}}_d]).$$

For the batched matrices $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{b \times m \times k}, \boldsymbol{\mathcal{B}} \in \mathbb{R}^{b \times k \times n}$, the batched matrix multiplication is

$$\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\boldsymbol{\mathcal{B}} = \textbf{einsum}(bmk, bkn \Rightarrow bmn, [\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}]), \quad \text{where } \boldsymbol{\mathcal{C}}[i,:,:] = \boldsymbol{\mathcal{A}}[i,:,:]\boldsymbol{\mathcal{B}}[i,:,:].$$

Tensor Decomposition. In this paper, we will mainly use tensor-train (TT) [35] and tensor-train matrix (TTM) [34] decomposition for compressed neural network training. TT [35] represents the tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ as a set of small-size cores $\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_d$ such that $\boldsymbol{\mathcal{G}}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ and

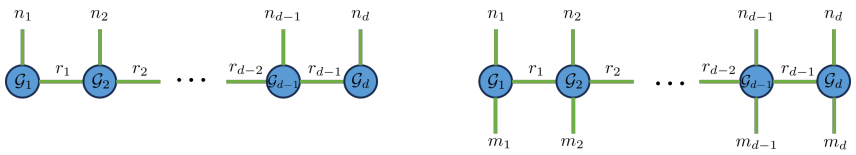
$$\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{G}}_1 \times_{3,1} \boldsymbol{\mathcal{G}}_2 \times_{3,1} \cdots \times_{3,1} \boldsymbol{\mathcal{G}}_d. \tag{3}$$

The tuple $(r_0, r_1, \ldots, r_d)$ is the TT rank of the TT decomposition (3) and must satisfy $r_0 = r_d = 1$. TTM considers an order-$2d$ tensor $\boldsymbol{\mathcal{B}}$ of dimension $m_1 \times n_2 \times \cdots \times m_d \times n_d$, and represents $\boldsymbol{\mathcal{B}}$ as

$$\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{F}}_1 \times_{4,1} \boldsymbol{\mathcal{F}}_2 \times_{4,1} \cdots \times_{4,1} \boldsymbol{\mathcal{F}}_d, \tag{4}$$

where $\boldsymbol{\mathcal{F}}_i \in \mathbb{R}^{r_{i-1} \times m_i \times n_i \times r_i}$ for $i = 1, \ldots, d$ and $r_0 = r_d = 1$. Figure 3 shows the tensor-network representations of TT and TTM decomposition.

In tensor-compressed neural networks, large weight matrices are reshaped to high-order tensors and compressed into small tensor cores in TT or TTM format. The weights of linear layers are often compressed into the TT format due to its efficiency in tensor-vector multiplications. The TTM format is more suitable for embedding tables whose dimension is highly unbalanced.



(a) (b)

Figure 3: Tensor networks for (a) tensor-train and (b) tensor-train-matrix decompositions.

3 The CoMERA Training Framework

The size of the tensor-compressed neural networks can be adjusted by modifying the tensor ranks. However, it also brings in an important problem: *how can we determine the tensor ranks automatically for a given resource limit?* We propose a multi-objective optimization to address this issue.

3.1 Multi-Objective Training Model

A Modified TT Representation. We consider the tensor-compressed training for a generic neural network. Suppose that the neural network is parameterized as $f(\mathbf{x}|\{\boldsymbol{\mathcal{G}}_1^i, \ldots, \boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P)$, where $\{\boldsymbol{\mathcal{G}}_j^i \in \mathbb{R}^{r_{j-1}^i \times n_j^i \times r_j^i}\}_{j=1}^{d_i}$ compress the original weight $\mathbf{W}_i$. Let $\{\mathbf{x}_k, y_k\}_{k=1}^N$ be training data and $\mathcal{L}$ be the loss function. The training is to minimize the following objective function

$$\min_{\{\boldsymbol{\mathcal{G}}_1^i, \ldots, \boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P} L := \sum_{k=1}^N \mathcal{L}(y_k, f(\mathbf{x}_k|\{\boldsymbol{\mathcal{G}}_1^i, \ldots, \boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P)). \tag{5}$$

We modify the TT compression and control the ranks of $\boldsymbol{\mathcal{G}}_1^i, \ldots, \boldsymbol{\mathcal{G}}_{d_i}^i$ by a set of diagonal matrices $\{\mathbf{D}_j^i \in \mathbb{R}^{r_j^i \times r_j^i}\}_{j=1}^{d-1}$. Specifically, let $\boldsymbol{\mathcal{W}}_i$ be the reshape of $\mathbf{W}_i$, and the compression of $\mathbf{W}_i$ is

$$\boldsymbol{\mathcal{W}}_i = \boldsymbol{\mathcal{G}}_1^i \times_{3,1} \mathbf{D}_1^i \times_{2,1} \boldsymbol{\mathcal{G}}_2^i \times_{3,1} \cdots \times_{3,1} \mathbf{D}_{d_i-1}^i \times_{2,1} \boldsymbol{\mathcal{G}}_{d_i}^i. \tag{6}$$

Now the tensor cores for $\boldsymbol{\mathcal{W}}_i$ have $S_i = n_1^i \|\mathbf{D}_1^i\|_0 + n_{d_i}^i \|\mathbf{D}_{d_i-1}^i\|_0 + \sum_{j=2}^{d_i-1} n_j^i \|\mathbf{D}_{j-1}^i\|_0 \|\mathbf{D}_j^i\|_0$ variables. For simplicity, we denote $\boldsymbol{\mathcal{G}} := \{\boldsymbol{\mathcal{G}}_1^i, \ldots, \boldsymbol{\mathcal{G}}_{d_i}^i\}_{i=1}^P$ and $\mathbf{D} := \{\mathbf{D}_1^i, \ldots, \mathbf{D}_{d_i-1}^i\}_{i=1}^P$.

Multi-Objective Optimization. We intend to minimize both the loss and compressed network size, which can be formulated as a multi-objective optimization $\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \{L(\boldsymbol{\mathcal{G}}, \mathbf{D}), S(\mathbf{D})\}$, where $S(\mathbf{D}) := \sum_{i=1}^P S_i(\mathbf{D})$. In most cases, we cannot find a point that minimizes the loss and model size simultaneously. Therefore, we look for a Pareto point $(\boldsymbol{\mathcal{G}}^*, \mathbf{D}^*)$, meaning that there exist no $\boldsymbol{\mathcal{G}}$ and $\mathbf{D}$ such that $L(\boldsymbol{\mathcal{G}}, \mathbf{D}) \leq L(\boldsymbol{\mathcal{G}}^*, \mathbf{D}^*)$, $S(\mathbf{D}) \leq S(\mathbf{D}^*)$, and at least one of inequalities is strict.

3.2 Training Methods

We convert a multi-objective optimization to a single-objective one via scalarization. We use different scalarization methods at the early and late stage of training. The late stage is optional, and it can further compress the model to enable efficient deployment on resource-constraint platforms.

Early Stage. At the early stage of CoMERA, aggressively pruning ranks dramatically hurts the convergence. Hence, we start the training with the following linear scalarization formulation [8]

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \; L(\boldsymbol{\mathcal{G}}, \mathbf{D}) + \gamma S(\mathbf{D}). \tag{7}$$

It is still hard to solve (7) since $S(\mathbf{D})$ uses $\|\cdot\|_0$ which is nonsmooth. Therefore, we replace $\|\cdot\|_0$ by the ℓ_1 norm $\|\cdot\|_1$ and get the convex relaxation

$$\hat{S}(\mathbf{D}) := \sum_{i=1}^P \left(\sum_{i=1}^P n_1^i \|\mathbf{D}_1^i\|_1 + n_{d_i}^i \|\mathbf{D}_{d_i-1}^i\|_1 + \sum_{j=2}^{d_i-1} n_j^i \|\mathbf{D}_{j-1}^i\|_1 \|\mathbf{D}_j^i\|_1 \right). \tag{8}$$

We note that $\hat{S}(\mathbf{D})$ can be arbitrarily close to 0 while keeping $L(\boldsymbol{\mathcal{G}}, \mathbf{D})$ unchangeable, since the corresponding slices of TT factors can be scaled accordingly. Therefore, a direct relaxation of the scalarization (11) does not have a minimizer. To address this issue, we add an ℓ_2 regularization $\|\boldsymbol{\mathcal{G}}\|^2 := \sum_{i=1}^P \sum_{j=1}^{d_i} \|\boldsymbol{\mathcal{G}}_j^i\|^2$ to the relaxation and get the formulation

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \; L(\boldsymbol{\mathcal{G}}, \mathbf{D}) + \gamma \hat{S}(\mathbf{D}) + \beta \|\boldsymbol{\mathcal{G}}\|^2. \tag{9}$$

The optimizer of Problem (9) is a Pareto point for a constrained problem, shown in the following.

Proposition 3.1. *For all $\gamma > 0, \beta > 0$, there exists some constant $C > 0$ such that the solution to the problem* (9) *is a Pareto point of the following multi-objective optimization problem*

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \quad (L(\boldsymbol{\mathcal{G}},\mathbf{D}), \hat{S}(\mathbf{D})) \quad \textit{subject to } \|\boldsymbol{\mathcal{G}}\|^2 \leq C. \tag{10}$$

Proof. See Appendix A.2 for the complete proof. □

Late Stage (Optional). The early-stage training can provide us with a Pareto point, but we cannot control where the Pareto point is. In the late stage of CoMERA, we may continue training the model towards a preferred loss L_0 and a preferred model size S_0 for deployment requirements. This can be achieved by the achievement scalarization [8] that leads to a Pareto point close to (L_0, S_0):

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \ \max\{w_1(L(\boldsymbol{\mathcal{G}},\mathbf{D}) - L_0), w_2(S(\mathbf{D}) - S_0)\} + \rho(L(\boldsymbol{\mathcal{G}},\mathbf{D}) + S(\mathbf{D})). \tag{11}$$

Here $w_1, w_2 > 0$ scale the objectives into proper ranges, and $\rho > 0$ is a small constant. After relaxing $S(\mathbf{D})$ to $\hat{S}(\mathbf{D})$ and adding the regularization term, we get the following problem

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \ \max\{w_1(L(\boldsymbol{\mathcal{G}},\mathbf{D}) - L_0), w_2(S(\mathbf{D}) - S_0)\} + \rho(L(\boldsymbol{\mathcal{G}},\mathbf{D}) + \hat{S}(\mathbf{D})) + \beta\|\boldsymbol{\mathcal{G}}\|^2, \tag{12}$$

where $\beta > 0$ is a positive constant. Note that the $S(\mathbf{D})$ inside $\max$ is not relaxed now for accurate comparisons. When $w_1(L(\boldsymbol{\mathcal{G}},\mathbf{D}) - L_0) \geq w_2(S(\mathbf{D}) - S_0)$, we consider the following problem

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \ w_1(L(\boldsymbol{\mathcal{G}},\mathbf{D}) - L_0) + \rho(L(\boldsymbol{\mathcal{G}},\mathbf{D}) + \hat{S}(\mathbf{D})) + \beta\|\boldsymbol{\mathcal{G}}\|^2. \tag{13}$$

We run a step of a gradient-based algorithm on this problem. When $w_1(L(\boldsymbol{\mathcal{G}},\mathbf{D}) - L_0) < w_2(S(\mathbf{D}) - S_0)$, we relax the $S(\mathbf{D})$ again and get the following problem

$$\min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \ w_2(\hat{S}(\mathbf{D}) - S_0) + \rho(L(\boldsymbol{\mathcal{G}},\mathbf{D}) + \hat{S}(\mathbf{D})) + \beta\|\boldsymbol{\mathcal{G}}\|^2, \tag{14}$$

and run a step of a gradient-based algorithm on this problem. The Algorithm 1 is summarized in Appendix A.3. The late stage optimization can be independently applied to a trained tensor-compressed model for further model size reductions.

4 Performance Optimization of CoMERA

While CoMERA can greatly reduce training variables and memory cost, the low-rank and small-size tensor operations in CoMERA are not efficiently supported by GPU. This often slows the training process. This section presents three methods to achieve real training speedup on GPU.

4.1 Performance Optimization of TTM Embedding Tables.

Embedding tables are widely used to transfer discrete features into continuous hidden space. The row size of embedding tables is usually much larger than the column size, making TTM compression more suitable than the TT format. In the following, we use an order-4 TTM embedding table to illustrate how to accelerate the lookup process.

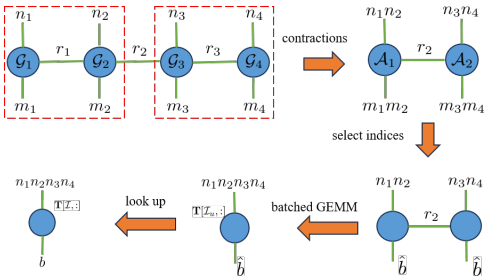
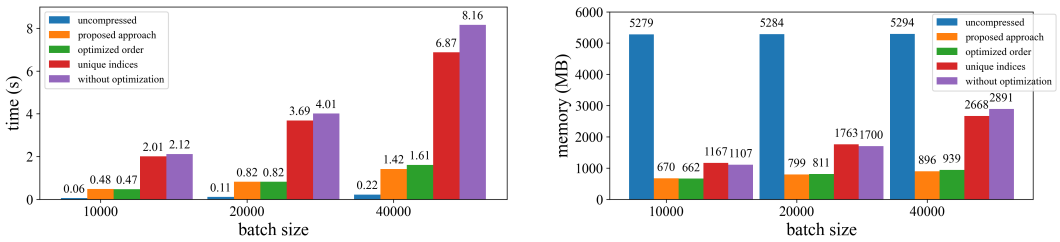


Figure 4: Optimized TTM embedding lookup.

We consider an embedding table $\mathbf{T} \in \mathbb{R}^{m\times n}$. A look-up operation selects the submatrix $\mathbf{T}[\mathcal{I},:] \in \mathbb{R}^{b\times n}$ for the index set $\mathcal{I} = \{i_k\}_{k=1}^b$. This operation is fast and inexpensive. However, the full embedding table itself is extremely memory-consuming. Suppose that $m = m_1m_2m_3m_4, n = n_1n_2n_3n_4$, then we reshape $\mathbf{T}$ into tensor $\boldsymbol{\mathcal{T}} \in \mathbb{R}^{m_1\times n_1\times\cdots\times m_4\times n_4}$ and represent it in TTM format

$$\boldsymbol{\mathcal{T}} = \boldsymbol{\mathcal{G}}_1 \times_{4,1} \boldsymbol{\mathcal{G}}_2 \times_{4,1} \boldsymbol{\mathcal{G}}_3 \times_{4,1} \boldsymbol{\mathcal{G}}_4. \tag{15}$$

The compressed embedding table does not have the matrix $\mathbf{T}$ explicitly. We convert each row index $i_k \in \mathcal{I}$ to a tensor index vector $(z_1^k, z_2^k, z_3^k, z_4^k)$ and denote $\mathcal{Z}_t = \{z_t^k\}_{k=1}^b$, then $\mathbf{T}[\mathcal{I},:]$ can be computed by contracting the tensors $\{\boldsymbol{\mathcal{G}}_t[:,\mathcal{Z}_t,:,:]\}_{t=1}^4$ where each has size $r_{t-1}\times b\times n_i\times n_t$. The $\boldsymbol{\mathcal{G}}_t[:,\mathcal{Z}_t,:,:]$ stores many duplicated values especially when the set $\mathcal{I}$ is large. Therefore, directly computing the tensor contractions can cause much computing and memory overhead.



(a) Speed-up (b) Memory reduction

Figure 5: Performance of optimized TTM embedding table lookup. The labels *uncompressed, proposed approach, optimized order, unique indices, without optimization* represent standard embedding with sparse gradients, the new method in 4.1, the method that only uses the unique order, the method that only uses the unique indices, and the method without optimization, respectively.

We optimize the tensor contraction by eliminating the redundant computation at two levels. **Row-index level.** We construct the index set $\mathcal{I}_u = \{i_k\}_{k=1}^{\hat{b}}$ containing all unique indices in $\mathcal{I}$. We can easily obtain $\mathbf{T}[\mathcal{I}, :]$ from $\mathbf{T}[\mathcal{I}_u, :]$. **Tensor-index level.** The reduced index set $\mathcal{I}_u$ leads to $\hat{b}$ associated tensor index vectors $(z_1^k, z_2^k, z_3^k, z_4^k)$, but at most $m_1 m_2$ pairs of (z_1^k, z_2^k) and $m_3 m_4$ pairs of (z_3^k, z_4^k) are unique. For instance, (2,3,1,3) and (2,3,2,4) are common in (2,3), so we only compute (2,3) entry once. Therefore, we can consider all unique pairs (z_1^k, z_2^k) and (z_3^k, z_4^k) and compute

$$\boldsymbol{\mathcal{A}}_1 = \textbf{einsum}(r_0 m_1 n_1 r_1, r_1 m_2 n_2 r_2 \Rightarrow (m_1 m_2)(n_1 n_2) r_2, [\boldsymbol{\mathcal{G}}_1, \boldsymbol{\mathcal{G}}_2]), \tag{16}$$

$$\boldsymbol{\mathcal{A}}_2 = \textbf{einsum}(r_2 m_3 n_3 r_3, r_3 m_4 n_4 r_4 \Rightarrow r_2 (m_3 m_4)(n_3 n_4), [\boldsymbol{\mathcal{G}}_3, \boldsymbol{\mathcal{G}}_4]). \tag{17}$$

For each $i_k \in \mathcal{I}_u$, let (j_1^k, j_2^k) be the coordinate of i_k for size $(m_1 m_2, m_3 m_4)$. We denote $\mathcal{J}_1 = \{j_1^k\}_{k=1}^{\hat{b}}$ and $\mathcal{J}_2 = \{j_2^k\}_{k=1}^{\hat{b}}$, then compute the unique rows of $\mathbf{T}$ as

$$\mathbf{T}[\mathcal{I}_u, :] = \textbf{einsum}(\hat{b}(n_1 n_2) r_2, r_2 \hat{b}(n_3 n_4) \Rightarrow \hat{b}(n_1 n_2 n_3 n_4), [\boldsymbol{\mathcal{A}}_1[\mathcal{J}_1, :, :], \boldsymbol{\mathcal{A}}_1[:, \mathcal{J}_2, :]]). \tag{18}$$

Figure 4 summarizes the whole process of TTM embedding table look-up. This approach can be easily applied to higher-order embedding tables by first grouping some small tensor cores to obtain intermediate tensors and then utilizing them to compute unique row vectors.

Performance. We demonstrate the optimized TTM embedding tables on a single RTX 3090 GPU. We consider an embedding table of TTM shape $[[80, 50, 54, 50], [4, 4, 4, 2]]$ and rank 32, extracted from a practical DLRM model. As shown in Figure 5, our proposed method achieves about $4 - 5\times$ speed-up and $2-3\times$ memory saving than the standard TTM embedding without any optimization. The uncompressed embedding with sparse gradients is faster than our approach since our TTM embedding table requires extra computation, but it uses much more memory than the TTM embedding table.

4.2 Contraction Path Optimization for TT-Vector Multiplications

Next, we optimize the forward- and back- propagation of linear layers in the TT format. We consider the linear layer $\mathbf{Y} = \mathbf{X}\mathbf{W}$, where $\mathbf{Y} \in \mathbb{R}^{b \times N_2}$, $\mathbf{W} \in \mathbb{R}^{N_1 \times N_2}$, $\mathbf{X} \in \mathbb{R}^{b \times N_1}$. The $\mathbf{W}$ is compressed into the Tensor-Train format: $\boldsymbol{\mathcal{W}} = [\![\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_{2d}]\!] \in \mathbb{R}^{n_1 \times \cdots \times n_{2d}}$,where $\boldsymbol{\mathcal{G}}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ and $N_1 = n_1 \cdots n_d, N_2 = n_{d+1} \cdots n_{2d}$. The forward-propagation in the einsum form is

$$\mathbf{Y} = \mathbf{X}\mathbf{W} = \textbf{einsum}(b n_1 \ldots n_d, S_1, \ldots, S_{2d} \Rightarrow b n_{d+1} \ldots n_{2d}, [\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_{2d}]) \tag{19}$$

where $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{b \times n_1 \times \cdots \times n_d}$ is the reshaping of $\mathbf{X}$ and S_i denotes $r_{i-1} n_i r_i$. Suppose that the gradient to $\mathbf{Y}$ is $\mathbf{g}_{\mathbf{Y}}$, then the gradients to $\boldsymbol{\mathcal{G}}_i$ and $\boldsymbol{\mathcal{X}}$ can be computed as follows:

$$\mathbf{g}_{\boldsymbol{\mathcal{G}}_i} = \textbf{einsum}\big(b n_1 \ldots n_d, b n_{d+1} \ldots n_{2d}, S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_{2d} \Rightarrow S_i, [\boldsymbol{\mathcal{X}}, \mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_{i-1}, \boldsymbol{\mathcal{G}}_{i+1}, \ldots, \boldsymbol{\mathcal{G}}_{2d}]\big), \tag{20}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{X}}} = \textbf{einsum}\big(b n_{d+1} \ldots n_{2d}, S_1, \ldots, S_{2d} \Rightarrow b n_1 \ldots n_d, [\mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_{2d}]\big). \tag{21}$$

In total, $2d + 2$ contraction sequences are needed for the TT-format forward- and back- propagation. To reduce the computational costs, it is critical to find an optimal or near-optimal contraction path.

Large batch case. We denote $\boldsymbol{\mathcal{A}}_i := \boldsymbol{\mathcal{G}}_1 \times \cdots \times \boldsymbol{\mathcal{G}}_i, \boldsymbol{\mathcal{A}}_{-i} = \boldsymbol{\mathcal{G}}_{d-i+1} \times \cdots \times \boldsymbol{\mathcal{G}}_d, \boldsymbol{\mathcal{B}}_i := \boldsymbol{\mathcal{G}}_{d+1} \times \cdots \times \boldsymbol{\mathcal{G}}_{d+i}, \boldsymbol{\mathcal{B}}_{-i} = \boldsymbol{\mathcal{G}}_{2d-i+1} \times \cdots \times \boldsymbol{\mathcal{G}}_{2d}$, which are all computed sequentially. In practice, we only need to compute $\boldsymbol{\mathcal{A}}_d, \boldsymbol{\mathcal{A}}_{-d}, \boldsymbol{\mathcal{B}}_d, \boldsymbol{\mathcal{B}}_{-d}$ and store the intermediate results. The forward-propagation (19) is then computed in the following way

$$\boldsymbol{\mathcal{T}}_1 = \textbf{einsum}(bn_1 \dots n_d, n_1 \dots n_d r_d \Rightarrow br_d, [\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{A}}_d]) \tag{22}$$

$$\boldsymbol{\mathcal{Y}} = \textbf{einsum}(br_d, r_d n_{d+1} \dots n_{2d} \Rightarrow bn_1 \dots n_d, [\boldsymbol{\mathcal{T}}_1, \boldsymbol{\mathcal{B}}_d]). \tag{23}$$

In backward propagation, the gradients are computed in the following way:

- The gradient $\mathbf{g}_{\boldsymbol{\mathcal{X}}}$ is computed as

$$\boldsymbol{\mathcal{U}}_1 = \textbf{einsum}(bn_{d+1} \dots n_{2d}, r_d n_{d+1} \dots n_{2d} \Rightarrow br_d, [\mathbf{g}_{\boldsymbol{\mathcal{Y}}}, \boldsymbol{\mathcal{B}}_d]) \tag{24}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{X}}} = \textbf{einsum}(br_d, n_1 \dots n_d r_d \Rightarrow bn_1 \dots n_d, [\boldsymbol{\mathcal{U}}_1, \boldsymbol{\mathcal{A}}_d]). \tag{25}$$

- The gradients $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$ for $i \geq d+1$ can be computed as

$$\boldsymbol{\mathcal{T}}_2 = \textbf{einsum}(br_d, bn_{d+1} \dots n_{2d} \Rightarrow r_d n_{d+1} \dots n_{2d}, [\boldsymbol{\mathcal{T}}_1, \mathbf{g}_{\boldsymbol{\mathcal{Y}}}]) \tag{26}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{G}}_i} = \textbf{einsum}(r_d n_{d+1} \dots n_{2d}, r_d n_{d+1} \dots n_{i-1} r_{i-1}, r_i n_{i+1} \dots n_{2d} \Rightarrow r_{i-1} n_i r_i,\ [\boldsymbol{\mathcal{T}}_2, \boldsymbol{\mathcal{B}}_{i-1-d}, \boldsymbol{\mathcal{B}}_{-(2d-i)}]). \tag{27}$$

- Similarly, the gradients $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$ for $i \leq d$ can be computed as

$$\boldsymbol{\mathcal{U}}_2 = \textbf{einsum}(br_d, bn_1 \dots n_d \Rightarrow r_d n_1 \dots n_d, [\boldsymbol{\mathcal{U}}_1, \boldsymbol{\mathcal{X}}]) \tag{28}$$

$$\mathbf{g}_{\boldsymbol{\mathcal{G}}_i} = \textbf{einsum}(r_d n_1 \dots n_d, n_1 \dots n_{i-1} r_{i-1}, r_i n_{i+1} \dots n_d r_d \Rightarrow r_{i-1} n_i r_i,\ [\boldsymbol{\mathcal{U}}_2, \boldsymbol{\mathcal{A}}_i, \boldsymbol{\mathcal{A}}_{-(d-i)}]). \tag{29}$$

The contraction paths of forward- and back- propagation are summarized in Appendix A.5.

Analysis. The proposed empirical path is near-optimal for large batch sizes. The following result analyzes the contraction path for forward-propagation.

Proposition 4.1. *Suppose that the TT ranks satisfy $1 = r_0 < r_1 \leq \cdots \leq r_d \geq r_{d-1} > \cdots \geq r_{2d} = 1$ and the batch size b is large enough. There exist groups $\{S_i\}_{i=1}^k$ where $S_i = \{\boldsymbol{\mathcal{G}}_{j_i+1}, \dots, \boldsymbol{\mathcal{G}}_{j_{i+1}}\}$ containing consecutive tensor cores for $0 = j_1 < \cdots < j_k < j_{k+1} = 2d$. Then, the contraction path with the least number of flops for the forward-propagation* (19) *first contracts the tensor cores in each S_i to obtain $\boldsymbol{\mathcal{V}}_i$ with dimension $r_{j_i} \times n_{j_i+1} \times \cdots \times n_{j_{i+1}} \times r_{j_{i+1}}$ and then contract the input tensor $\boldsymbol{\mathcal{X}}$ with tensors $\{\boldsymbol{\mathcal{V}}_i\}_{i=1}^k$ in the sequential order.*

Proof. See Appendix A.4 for the complete proof. □

Proposition 4.1 implies that the optimal path first contracts some consecutive tensor cores and then contracts obtained tensors with the input tensor sequentially. The groups $\{S_i\}_{i=1}^k$ depend on the dimensions, ranks, and batch size. The proposed contraction path satisfies the property shown in Proposition 4.1 and has flops roughly $b(n_1 \cdots n_d + n_{d+1} \cdots n_{2d})r_d$. The optimal contraction path has flops about $bn_1 \cdots n_d c_1 + bn_{d+1} \cdots n_{2d} c_2$, where c_1, c_2 are some constants. Hence, the proposed is near-optimal and has a comparable complexity to the optimal path. Suppose the optimal path is different from the proposed empirical path. Then the optimal path will likely involve a few more large intermediate tensors, which pose more memory costs during training and inference especially for static computational graphs. The empirical path is a good choice to balance time and memory consumption. Similar arguments can be applied to the contractions for back-propagation.

When the batch size is small, the optimal path may have much fewer flops. However, the execution time is almost the same as the proposed path since all the operations are too small. Hence, we can use the proposed path for most batch sizes. See Appendix A.6 for more analysis.

4.3 GPU Performance Optimization via CUDA Graph

While CoMERA consumes much less computing FLOPS than standard uncompressed training, it can be slower on GPU if not implemented carefully. Therefore, it is crucial to optimize the GPU performance to achieve real speedup. Modern GPUs are highly optimized for large-size matrix multiplications. However, the small-size tensor contractions in CoMERA are not yet optimized on GPU and require many small-size GPU kernels, causing significant runtime overhead. During the training, Cuda Graph launches and executes the whole computing graph rather than launching a large number of kernels sequentially. This can eliminate lots of back-end overhead and lead to significant training speedup. It is more suitable for CoMERA since tensor-compressed training has much more small kernels than uncompressed training. This is just an initial step of GPU optimization. We expect that a more dedicated GPU optimization can achieve a more significant training speedup.

Table 1: Result of Transformer on MNLI of batch size 128.

	validation	total size (MB)	compressed size (MB)
uncompressed training	62.2%	256 (1×)	253 (1×)
CoMERA (early stage)	63.3%	5.9 (43×)	3.4 (74×)
CoMERA (late stage), target ratio: 0.8	62.2%	4.9 (52×)	2.4 (105×)
CoMERA (late stage), target ratio: 0.5	62.1%	3.9 (65×)	1.4 (181×)
CoMERA (late stage), target ratio: 0.2	61.5%	3.2 (80×)	0.7 (361×)

Table 2: The change of ranks of layers in the fifth encoder block.

	before training	early-stage rank	late-stage rank
Q-layer in attention	(12, 30, 30, 30, 12)	(12, 30, 30, 30, 12)	(0, 0, 0, 0, 0)
K-layer in attention	(12, 30, 30, 30, 12)	(12, 30, 30, 30, 12)	(0, 0, 0, 0, 0)
V-layer in attention	(12, 30, 30, 30, 12)	(12, 30, 30, 30, 12)	(9, 11, 11, 7, 9)
FC-layer in attention	(12, 30, 30, 30, 12)	(12, 30, 29, 30, 12)	(9, 8, 10, 8, 8)
#1 linear-layer in Feed-Forward	(12, 30, 30, 30, 16)	(0, 0, 0, 0, 0)	(0, 0, 0, 0, 0)
#2 linear-layer in Feed-Forward	(16, 30, 30, 30, 12)	(0, 0, 0, 0, 0)	(0, 0, 0, 0, 0)

5 End-to-End Training Results

In this section, we test the performance of CoMERA on a few benchmarks. Our experiments are run on a Nvidia RTX 3090 GPU with 24GB RAM.

5.1 A Medium-Size Transformer with Six Encoders

We first consider a six-encoder transformer. The embedding tables and all linear layers are represented as tensor cores in the training process as detailed in Appendix A.7. We train this model on the MNLI dataset [38] with the maximum sequence length 128 and compare the accuracy, resulting model size, and training time of CoMERA with the standard uncompressed training.

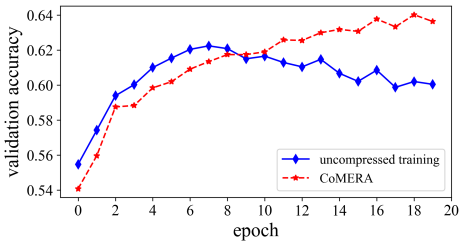


Figure 6: Behavior of early-stage CoMERA training on the MNLI dataset.

CoMERA Accuracy and Compression Performance. Table 1 summarizes the training results. The **early-stage training** of CoMERA achieves 74× compression ratio on all tensorized layers, and the validation accuracy is even higher than the uncompressed training. Figure 6 shows the validation accuracy of CoMERA. In the **late stage** of CoMERA, we set different target compression ratios for more aggressive rank pruning. The target compression ratios are for the tensorized layers rather than for the whole model. The late-stage training can reach the desired compression ratio with very little accuracy drop. The smallest model has a compression ratio of 80× for the whole model due to a 361× compression on the tensorized layers with slightly worse accuracy.

Architecture Search Capability of CoMERA. A major challenge in training is architecture search: shall we keep certain layers of a model? Interestingly, CoMERA has some capability of automatic architecture search. Specifically, the ranks of some layers become zero in the training, and thus the whole layer can be removed. For the target compression ratio 0.2, the whole second last encoder and some linear layers in other encoders are completely removed after late-stage rank-adaptive training. The change of ranks of layers in the 5th encoder is shown in Table 2.

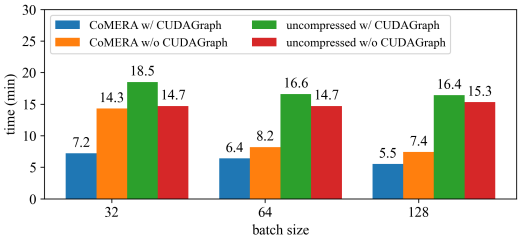
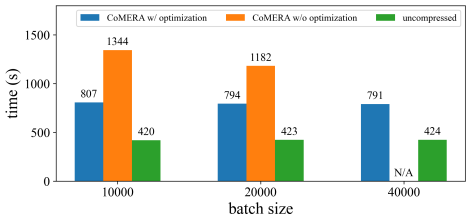
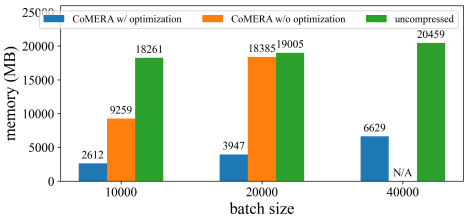


Figure 7: Training time per epoch for the six-encoder transformer model on the MNLI dataset.

Training Time. As shown in Figure 7, CoMERA with CUDAGraph achieves around 3× speed-up than uncompressed training. CoMERA without CUDAGraph can take much longer time in small batch-size setting due to the launching overhead of too many small kernels. The uncompressed training with CUDAGraph takes longer time than the one without CUDAGraph. This is because CUDAGraph requires all batches to have the same sequence length, and the consequent computing overhead is more than the time reduction of CUDAGraph. In contrary, CoMERA has much fewer computing FLOPS and the computing part accounts for a much smaller portion of the overall



(a) Training time per epoch.



(b) Memory cost.

Figure 9: Performance of optimized CoMERA on training DLRM.

runtime. Empirically CoMERA is $2-3\times$ faster in the whole training than uncompressed training for transformers on a single GPU, but we do not have theoretical guarantees about the number of epochs although they are similar in our experiments. Appendix A.8 provides more details about the run-time comparison on this benchmark, showing that CoMERA is still faster than standard training even if the compression ratio is close to 1.

5.2 A DLRM Model with 4-GB Model Size

We further test CoMERA on DLRM [32] released by Meta on Criteo Ad Kaggle dataset [23]. We compress the ten largest embedding tables into the TTM format as in Section 4.1. All fully connected layers with sizes > 128 are compressed into TT format. The model is trained for two epochs.

Effect of Optimized TTM Embedding. The training time per epoch and peak memory cost are shown in Figure 9. Our optimized TTM lookup speeds up the training process by around $2\times$ and remarkably reduces the memory cost by $4-5\times$.

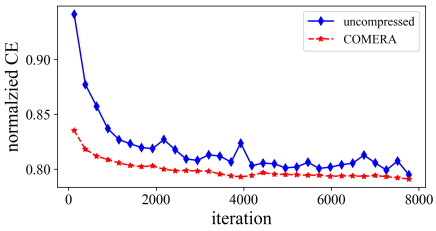


Figure 8: NCE loss curve of DLRM on the validation dataset.

Overall Performance of CoMERA. Table 3 shows the testing accuracy, testing loss (measured as normalized CE), memory costs, and model sizes of CoMERA and uncompressed training. CoMERA achieves similar accuracy as the uncompressed training, while CoMERA compresses the whole model by $99\times$ and saves $7\times$ peak memory cost (with consideration of the data and backend overhead) in the training process. The reduction of model size and memory cost mainly comes from the compact TTM tensor representation of large embedding tables. Standard uncompressed training is faster than CoMERA since DLRM is an embedding-intensive model, and the computation in the embedding table is look-up rather than matrix multiplications. However, CoMERA uses much less memory, saving 6.9X, 4.8X, and 3.1X memory for batch sizes 10000, 2000, and 4000, respectively. Furthermore, CoMERA has a similar convergence curve and needs fewer iterations than standard training for DLRM, as shown in Figure 8.

5.3 Comparison with GaLore and LTE

We compare our method with two recent low-rank compressed training frameworks: GaLore [41] and LTE [20]. GaLore [41] reduces the memory cost by performing SVD compression on the gradient, and LTE represents the weights as the sum of parallel low-rank matrix factorizations. We evaluate their memory costs and training times per epoch on the six-encoder transformer model under different batch sizes. We do not compare the total training time because the training epochs of various methods are highly case-dependent. The CoMERA and GaLore achieve almost the same validation accuracy, 64%, on the MNLI dataset. However, the LTE approach does not converge on the task using its default setting.

Table 3: Training results on the DLRM model with a batch size $10,000$.

	uncompressed	CoMERA
accuracy	78.68%	78.76%
normalized CE	0.793	0.792
model size (GB)	4.081	0.041 (**99**×)
peak memory (GB)	18.275	2.612 (**7**×)

Training Time Per Epoch. We use rank 128 for the low-rank gradients in GoLore, and rank 32 and head number 16 for the low-rank adapters in LTE. For a fair comparison, all methods are executed with CUDA graph to reduce the overhead of launching CUDA kernels. The runtimes per training epochs are reported in Figure 1(a). For the LTE, we only report the results for batch sizes $32, 64, 128$ since it requires the batch size to be a multiple of the head number. Overall, our CoMERA is about $2\times$ faster than GaLore and $3\times$ faster than LTE for all batch sizes, because the forward and backward propagation using low-rank tensor-network contractions dramatically reduce the computing FLOPS.

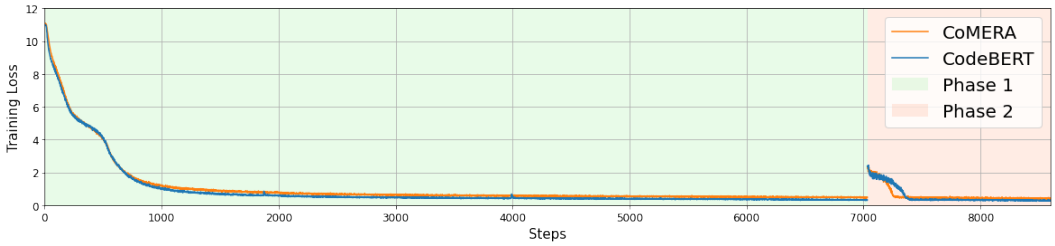


Figure 10: Pre-training loss curves of CodeBERT and CoMERA.

Memory Cost. Figure 1 (b) shows the memory cost of all three training methods. In the single-batch setting as used in [41], our CoMERA method is $9\times$ more memory-efficient than Galore on the tested case (with consideration of data and back-end cost). As the batch size increases, the memory overhead caused by data and activation functions becomes more significant, leading to less memory reduction ratios. However, our proposed CoMERA still uses the least memory.

We run the experiments on the RTX 3090 GPU. The work GaLore[41] uses the RTX 4090 GPU for experiments, so we also compare them on the RTX 3090 GPU. The results are in Appendix A.9.

5.4 Preliminary LLM Pre-Training Results: Case Study on CodeBERT

To show the benefit of CoMERA in pre-training (domain-specific) LLMs, we follow the setup from CodeBERT [10] to pre-train a BERT-like model for code generation. The pre-training dataset is the CodeSearchNet [21], a collection of 2M (comment, code) pairs and 6M pure code sequences from open-source libraries with 6 types of programming languages. We pre-train CodeBERT$_{\text{LARGE}}$ (357M) and its CoMERA (84M) variant using the masked language modeling (MLM) objective and compare their training loss in Figure 10. We achieve up to $12.72\times$ compression on tensorized layers and $4.23\times$ overall compression with final loss of 0.40 vs 0.28. There is a small gap between the final losses. However, this does not necessarily imply performance degradation on downstream tasks, based on our observation on BERT$_{\text{LARGE}}$[9] shown in Appendix A.10. Furthermore, CoMERA is $2.3\times$ and $1.9\times$ faster than standard pre-training in Phase 1 and Phase 2 respectively, when evaluated on the Nvidia RTX 3090 GPU. Our current CoMERA implementation is still slower than standard pre-training on massive GPUs, since no performance optimization has been done on HPC.

6 Conclusions and Future work

This work has presented CoMERA framework to reduce the memory and computing time of training AI models. We have investigated rank-adaptive training via multi-objective optimization to meet specific model sizes while maintaining model performance. We have achieved real training speedup on GPU via three optimizations: optimizing the tensorized embedding tables, optimizing the contraction path in tensorized forward and backward propagation, and optimizing the GPU latency via CUDAGraph. The experiments on a transformer model demonstrated that CoMERA can achieve $2-3\times$ speedup per training epoch. The model sizes of the transformer and a DLRM model have been reduced by $43\times$ to $99\times$ in the training process, leading to significant peak memory reduction (e.g., $7\times$ total reduction in large-batch training of DLRM on a single GPU). Our method has also outperformed the latest GaLore and LTE frameworks in both memory and runtime efficiency. More importantly, our method has demonstrated significant speedup and model compression in pre-training CodeBERT, a domain-specific LLM for automatic code generation. We have also observed further speedup by combining CoMERA with mixed-precision computation. The discussions and some preliminary results are in Appendix A.11.

Unlike large-size matrix operations, the small low-rank tensor operations used in CoMERA are not yet well-supported by existing GPU kernels. The performance of CoMERA can be further boosted significantly after a comprehensive GPU and HPC optimization. The existing optimizers, e.g. Adam, are well-studied for uncompressed training. However, CoMERA has a very different optimization landscape due to the tensorized structure. Therefore, it is also worth studying the optimization algorithms specifically for CoMERA in the future.

7 Acknowledgement

The pre-training task used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0030039.

References

[1] AI and compute. https://openai.com/blog/ai-and-compute/. OpenAI in 2019.

[2] ChatGPT and generative AI are booming, but the costs can be extraordinary. https://www.cnbc.com/2023/03/13/chatgpt-and-generative-ai-are-booming-but-at-a-very-expensive-price.html. Accessed: 2023-03-15.

[3] Update: ChatGPT runs 10K Nvidia training GPUs with potential for thousands more. https://www.fierceelectronics.com/sensors/chatgpt-runs-10k-nvidia-training-gpus-potential-thousands-more. Accessed: 2023-03-15.

[4] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[5] Giuseppe G Calvi, Ahmad Moniri, Mahmoud Mahfouz, Qibin Zhao, and Danilo P Mandic. Compression and interpretability of deep neural networks via tucker tensor layer. *arXiv:1903.06133*, 2019.

[6] Beidi Chen, Tri Dao, Kaizhao Liang, Jiaming Yang, Zhao Song, Atri Rudra, and Christopher Re. Pixelated Butterfly: Simple and efficient sparse training for neural network models. *arXiv preprint arXiv:2112.00029*, 2021.

[7] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[8] Kalyanmoy Deb, Karthik Sindhya, and Jussi Hakanen. Multi-objective optimization. In *Decision sciences*, pages 161–200. CRC Press, 2016.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. Association for Computational Linguistics: Human Language Technologies*, pages 4171–4186, 2019.

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[11] Jiaqi Gu, Ben Keller, Jean Kossaifi, Anima Anandkumar, Brucek Khailany, and David Z Pan. Heat: Hardware-efficient automatic tensor decomposition for transformer compression. *arXiv preprint arXiv:2211.16749*, 2022.

[12] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

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

[14] Cole Hawkins, Xing Liu, and Zheng Zhang. Towards compact neural networks via end-to-end training: A bayesian tensor approach with automatic rank determination. *SIAM Journal on Mathematics of Data Science*, 4(1):46–71, 2022.

[15] Cole Hawkins and Zheng Zhang. Bayesian tensorized neural networks with automatic rank selection. *Neurocomputing*, 453:172–180, 2021.

[16] Qinyao He, He Wen, Shuchang Zhou, Yuxin Wu, Cong Yao, Xinyu Zhou, and Yuheng Zou. Effective quantization methods for recurrent neural networks. *arXiv preprint arXiv:1611.10176*, 2016.

[17] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[18] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

[19] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[20] Minyoung Huh, Brian Cheung, Jeremy Bernstein, Phillip Isola, and Pulkit Agrawal. Training neural networks from scratch with parallel low-rank adapters. *arXiv preprint arXiv:2402.16828*, 2024.

[21] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[22] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions. *arXiv preprint arXiv:1405.3866*, 2014.

[23] Olivier Chapelle Jean-Baptiste Tien, joycenv. Display advertising challenge, 2014.

[24] Valentin Khrulkov, Oleksii Hrinchuk, Leyla Mirvakhabova, and Ivan Oseledets. Tensorized embedding layers for efficient model compression. *arXiv preprint arXiv:1901.10787*, 2019.

[25] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv preprint arXiv:1511.06530*, 2015.

[26] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, Aug. 2009.

[27] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. In *NIPS*, pages 1742–1752, 2017.

[28] Joseph M Landsberg. Tensors: geometry and applications. *Representation theory*, 381(402):3, 2012.

[29] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. *arXiv preprint arXiv:1412.6553*, 2014.

[30] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[31] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[32] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[33] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in neural information processing systems*, pages 442–450, 2015.

[34] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. In *Advances in Neural Information Processing Systems 28*, pages 442–450, 2015.

[35] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[36] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Viji Srinivasan, and Kailash Gopalakrishnan. Ultra-low precision 4-bit training of deep neural networks. *NIPS*, 33, 2020.

[37] Andros Tjandra, Sakriani Sakti, and Satoshi Nakamura. Compressing recurrent neural network with tensor train. *CoRR*, abs/1705.08052, 2017.

[38] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1112–1122. Association for Computational Linguistics, 2018.

[39] Yifan Yang, Jiajun Zhou, Ngai Wong, and Zheng Zhang. LoRETTA: Low-rank economic tensor-train adaptation for ultra-low-parameter fine-tuning of large language models. *arXiv preprint arXiv:2402.11417*, 2024.

[40] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *Proc. the 34th International Conference on Machine Learning*, volume 70, pages 3891–3900, Sydney, Australia, 06–11 Aug 2017.

[41] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient LLM training by gradient low-rank projection, 2024.

[42] Yequan Zhao, Xinling Yu, Zhixiong Chen, Ziyue Liu, Sijia Liu, and Zheng Zhang. Tensor-compressed back-propagation-free training for (physics-informed) neural networks. *arXiv preprint arXiv:2308.09858*, 2023.

[43] Yu-Bang Zheng, Xi-Le Zhao, Junhua Zeng, Chao Li, Qibin Zhao, Heng-Chao Li, and Ting-Zhu Huang. Svdinstn: A tensor network paradigm for efficient structure search from regularized modeling perspective. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 26254–26263, 2024.

A Supplementary Material

A.1 Comparison with existing works

SVDinsTN [43] uses sparse diagonal matrices and the ℓ_1 regularization to control tensor ranks for a compact tensor structure of a given tensor. In contrast, our work compresses weights during end-to-end training without any prior information on the tensor (i.e., model parameters). Both works use sparse diagonal matrices and ℓ_1 terms to control tensor ranks. Using diagonal matrices to control the ranks of matrices is very common, like SVD. The ℓ_1 norm is also widely used to induce sparsity in various models, like compressed sensing and Lasso regression. It is natural to combine these two techniques to control tensor ranks, regardless of tensor formats. Moreover, our work formulates the problem as a more generic multi-objective problem and uses a two-stage algorithm to solve it. The formulation in SVDinsTN is similar to linear scalarization approach in our early stage. Our work further uses the achievement scalarization in the late stage to find a model close to our preferred model performance and size.

HEAT [11] also considers contraction optimization for post-training model compression of trained models. In contrast, CoMERA considers end-to-end tensor-compressed training, where no model parameters are known prior to training. In addition, HEAT only discusses the single path optimization for forward propagation in CP format. We have optimized $d + 2$ contraction paths **jointly** in both forward- and back- propagation in TT format. Since these contractions can be coupled, we have also minimized the overall computation costs by reusing intermediate results.

A.2 Proof of Proposition 3.1

Proof. The objective function in (9) is bounded below by 0. Hence, the problem (9) has a finite infimum value f^*. Let $\{\boldsymbol{\mathcal{G}}^k, \mathbf{D}^k\}_{k=1}^{\infty}$ be a sequence such that $\lim_{k\to\infty} L(\boldsymbol{\mathcal{G}}^k, \mathbf{D}^k) + \gamma\hat{S}(\mathbf{D}^k) + \beta\|\boldsymbol{\mathcal{G}}^k\|^2 = f^*$. The sequence must be bounded because of the l_1 regularization of $\mathbf{D}$ and the ℓ_2 regularization of $\boldsymbol{\mathcal{G}}$. As a result, the sequence has a cluster point $(\boldsymbol{\mathcal{G}}^*, \mathbf{D}^*)$ which is a minimizer of the (9). Let $C := \|\boldsymbol{\mathcal{G}}^*\|^2$. The relaxation (9) is equivalent to the constrained optimization problem.

$$\begin{aligned} \min_{\boldsymbol{\mathcal{G}},\mathbf{D}} \quad & L(\boldsymbol{\mathcal{G}}, \mathbf{D}) + \gamma\hat{S}(\mathbf{D}) \\ \text{s.t.} \quad & \|\boldsymbol{\mathcal{G}}\|^2 \le C. \end{aligned} \tag{30}$$

It implies that the solution to the training problem (9) is a Pareto point of the multi-objective optimization problem $\min_{\boldsymbol{\mathcal{G}},\mathbf{D}}(L(\boldsymbol{\mathcal{G}}, \mathbf{D}), \hat{S}(\mathbf{D}))$. □

A.3 Algorithm for Late Stage Optimization in Section 3.2

The algorithm for the late stage optimization in Section 3.2 is summarized in Algorithm 1.

Algorithm 1 Solve relaxed scalarization problem (12)

Input: Initializations $\boldsymbol{\mathcal{G}}_0, \mathbf{D}_0$, constants $L_0, S_0, w_1, w_2, \rho, \beta$, and an optimization algorithm $\mathcal{O}$.
Output: Tensor cores $\boldsymbol{\mathcal{G}}_T$ and rank-control parameters $\mathbf{D}_T$.

for $t = 0, \ldots, T-1$ **do**
  if $w_1(L(\boldsymbol{\mathcal{G}}_t, \mathbf{D}_t) - L_0) \ge w_2(S(\mathbf{D}_t) - S_0)$ **then**
    The optimization algorithm $\mathcal{O}$ runs one step on the problem (13).
  else
    The optimization algorithm $\mathcal{O}$ runs one step on the problem (14).
  end if
end for

A.4 Proof of Proposition 4.1

Proof. For convenience, let V_i be a string of characters to specify the dimension of $\boldsymbol{\mathcal{V}}_i$, C_i be the set of tensor cores used to obtain $\boldsymbol{\mathcal{V}}_i$, and $\boldsymbol{\mathcal{X}}_i$ be the tensor by contracting $\boldsymbol{\mathcal{X}}$ with $\boldsymbol{\mathcal{V}}_1, \ldots \boldsymbol{\mathcal{V}}_i$, denoted by the string X_i.

We first show that $\boldsymbol{\mathcal{V}}_1$ must be in the proposed format. Suppose otherwise for contradiction. Let $\boldsymbol{\mathcal{G}}_i$ be the first tensor core used to obtain $\boldsymbol{\mathcal{V}}_1$. If $i = 1$, then we write $V_1 = V_1^1 V_1^2$ where the tensor $\boldsymbol{\mathcal{V}}_1^1$ corresponding to V_1^1 is obtained by contractions of longest consecutive tensor cores containing $\boldsymbol{\mathcal{G}}_i$ in the set C_1. Let $Z_1 = V_i^1 \cap \tilde{X}$, $Z_2 = V_i^2 \cap \tilde{X}$. The number of flops for the contraction between $\widetilde{\boldsymbol{\mathcal{X}}}$ and $\boldsymbol{\mathcal{V}}_1$ is $\frac{\pi(X)\pi(V_1^1)\pi(V_1^2)}{\pi(Z_1)\pi(Z_2)}$. If we first contract $\widetilde{\boldsymbol{\mathcal{X}}}$ with $\boldsymbol{\mathcal{V}}_1^1$ and then contract the obtained tensor with $\boldsymbol{\mathcal{V}}_1^2$, the number of flops is $\frac{\pi(X)\pi(V_1^1)}{\pi(Z_1)} + \frac{\pi(X)\pi(V_1^1)\pi(V_1^2)}{\pi(Z_1)^2\pi(Z_2)}$ which is less than $\frac{\pi(X)\pi(V_1^1)\pi(V_1^2)}{\pi(Z_1)\pi(Z_2)}$ since $Z_2 \neq V_1^2$. It contradicts our assumption that this is the optimal path. If $d \geq i > 1$, let $\boldsymbol{\mathcal{S}}$ be the tensor generated by the longest consecutive tensor cores containing $\boldsymbol{\mathcal{G}}_{i-1}$ and used in the optimal path. A better path is to first contract $\boldsymbol{\mathcal{V}}_1$ with $\boldsymbol{\mathcal{S}}$ to obtain $\boldsymbol{\mathcal{W}}$, then contract $\boldsymbol{\mathcal{W}}$ with $\boldsymbol{\mathcal{X}}$ and all other unused parts in the optimal path. It is better because the number of flops for contracting $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{X}}$ is no greater than that for contracting $\boldsymbol{\mathcal{V}}_1$ and $\boldsymbol{\mathcal{X}}$ and the new path reduces the number of flops in the remaining contractions. The reduction in the remaining contractions is more than the potential flop increase in obtaining $\boldsymbol{\mathcal{W}}$ when the batch size b is big enough. Finally, we consider the case that $i > d$. Let $\boldsymbol{\mathcal{G}}_j$ be the first tensor core used to obtain $\boldsymbol{\mathcal{V}}_2$. If $j > d$, then first contracting $\boldsymbol{\mathcal{V}}_1$ and $\boldsymbol{\mathcal{V}}_2$ and then all other parts is a better choice. Otherwise, if $j \leq d$, let $\boldsymbol{\mathcal{V}}_2^1$ be the tensor contracted by the consecutive tensors containing $\boldsymbol{\mathcal{G}}_j$ in C_2. The tensor $\boldsymbol{\mathcal{V}}$ can be represented by the contraction of $\boldsymbol{\mathcal{V}}_2^1$ and another tensor $\boldsymbol{\mathcal{V}}_2^2$ generated by the remaining tensor cores in C_2. In this scenario, we can contract $\boldsymbol{\mathcal{V}}_2^1$ with $\boldsymbol{\mathcal{V}}_1$ to get $\boldsymbol{\mathcal{S}}$, then $\boldsymbol{\mathcal{S}}$ with $\boldsymbol{\mathcal{X}}$ to get $\boldsymbol{\mathcal{W}}$, then $\boldsymbol{\mathcal{V}}_2^2$ with $\boldsymbol{\mathcal{W}}$, and finally the obtained tensor with $\boldsymbol{\mathcal{V}}_3, \ldots, \boldsymbol{\mathcal{V}}_k$. It is not hard to verify contracting $\boldsymbol{\mathcal{V}}_2^1$ and $\boldsymbol{\mathcal{V}}_1$, $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{S}}$ and $\boldsymbol{\mathcal{X}}$, and $\boldsymbol{\mathcal{W}}$ and $\boldsymbol{\mathcal{V}}_2^2$ uses less flops than directly contracting $\boldsymbol{\mathcal{X}}$ with $\boldsymbol{\mathcal{V}}_1$ and $\boldsymbol{\mathcal{V}}_2$ directly when the batch size b is large. Summarizing everything above, we can conclude that $\boldsymbol{\mathcal{V}}_1$ must be in the proposed format.

The contraction of $\boldsymbol{\mathcal{X}}_i$ and $\boldsymbol{\mathcal{V}}_{i+1}, \ldots, \boldsymbol{\mathcal{V}}_k$ has the similar structure to the contraction of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{V}}_1, \ldots, \boldsymbol{\mathcal{V}}_k$. By applying the same proof, we conclude that the tensors $\boldsymbol{\mathcal{V}}_i$'s must be in the format stated in the proposition and we will contract the input tensor $\boldsymbol{\mathcal{X}}$ with the tensors $tenV_1, \ldots, \boldsymbol{\mathcal{V}}_k$ in the sequential order. □

A.5 Algorithm for Contraction Path in Section 4.2

The empirical near-optimal contraction path for tensor-compressed training is shown in Algorithm 2.

Figure 11 presents the tensor diagrams for contraction paths of TT forward- and back- propagation as discussed in Section 4.2.

Algorithm 2 Empirical path for tensor-compressed forward- and back- propagation

Forward Input: Tensor cores $\boldsymbol{\mathcal{G}}_1, \ldots, \boldsymbol{\mathcal{G}}_{2d}$ and input matrix $\mathbf{X}$.
Forward Output: Output matrix $\mathbf{Y}$, and intermediate results.

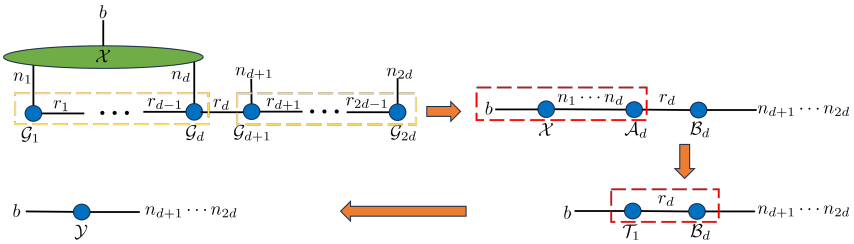
1: Reshape the matrix $\mathbf{X}$ to the tensor $\boldsymbol{\mathcal{X}}$.
2: Compute $\boldsymbol{\mathcal{A}}_d, \boldsymbol{\mathcal{A}}_{-d}, \boldsymbol{\mathcal{B}}_d, \boldsymbol{\mathcal{B}}_{-d}$ in the sequential order and store intermediate results of $\{\boldsymbol{\mathcal{A}}_i, \boldsymbol{\mathcal{A}}_{-i}, \boldsymbol{\mathcal{B}}_i, \boldsymbol{\mathcal{B}}_{-i}\}_{i=1}^d$ for back-propagation.
3: Compute $\boldsymbol{\mathcal{T}}_1$ as in (22) to store it for back-propagation.
4: Compute $\boldsymbol{\mathcal{Y}}$ as in (23) and reshape it to the appropriate matrix $\mathbf{Y}$.

Backward Input: Inputs of **Forward**, stored results from **Forward**, and output gradient $\mathbf{g}_{\mathbf{Y}}$.
Backward Output: Gradients $\mathbf{g}_{\mathbf{X}}, \mathbf{g}_{\boldsymbol{\mathcal{G}}_1}, \ldots, \mathbf{g}_{\boldsymbol{\mathcal{G}}_{2d}}$.

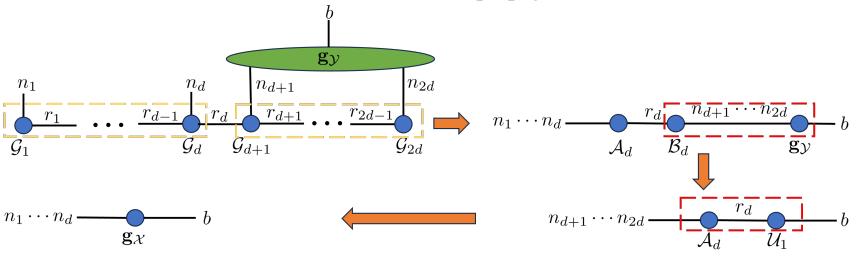
1: Reshape the gradient $\mathbf{g}_{\mathbf{Y}}$ to the tensor $\mathbf{g}_{\boldsymbol{\mathcal{Y}}}$.
2: Compute $\boldsymbol{\mathcal{U}}_1$ and $\mathbf{g}_{\boldsymbol{\mathcal{X}}}$ as in (24), (25) and store $\boldsymbol{\mathcal{U}}_1$ for future use.
3: Compute $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$ for $i \geq d+1$ as in (26), (27) using stored tensors.
4: Compute $\mathbf{g}_{\boldsymbol{\mathcal{G}}_i}$ for $i \leq d$ as in (22), (29) using stored tensors.

A.6 Small Batch Case for Contraction Path in Section 4.2

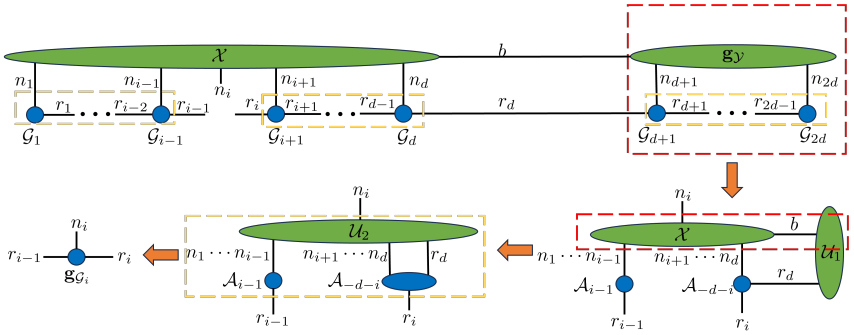
Small batch case. The empirical contraction path in Algorithm 2 eliminates the batch size dimension b early, so it is nearly optimal when the batch size is large. We may search for a better path using a greedy search algorithm to minimize the total operations. In each iteration, we prioritize the pairs that output the smallest tensors. Such a choice can quickly eliminate large intermediate dimensions to reduce the total number of operations. When the batch size is large, the searched path is almost identical to the empirical path in Algorithm 2 which eliminates the batch size dimension b



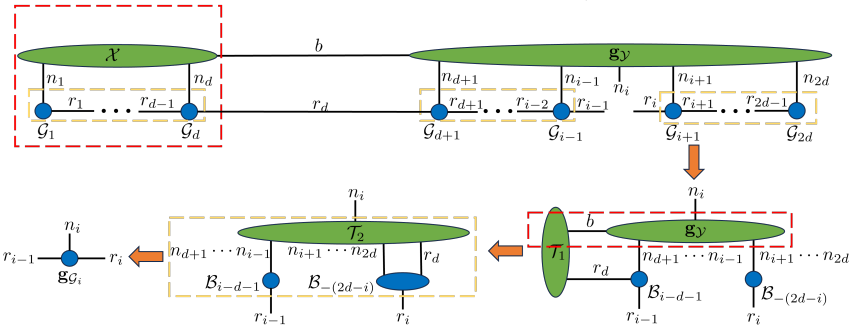
(a) TT-vector forward-propagation



(b) TT-vector back-propagation: compute $\mathbf{g}_{\mathcal{X}}$



(c) TT-vector back-propagation: compute $\mathbf{g}_{\mathcal{G}_i}, 1 \leq i \leq d$



(d) TT-vector back-propagation: compute $\mathbf{g}_{\mathcal{G}_i}, d+1 \leq i \leq 2d$

Figure 11: Tensor diagrams for contraction paths of TT forward- and back- propagation.

early. The searched path may differ from Algorithm 2 for small batch sizes, but their execution times on GPU are almost the same. This is because the tensor contractions for smaller batch sizes have a minor impact on the GPU running times. Consequently, despite certain tensor contractions in the empirical path being larger than those in the optimal path, the actual GPU execution times between them exhibit only negligible differences. Therefore, the empirical contraction path in Algorithm 2 is adopted for all batch sizes in CoMERA.

A.7 Compression Settings for the Experiment in Section 5.1

The compression settings for the experiment in Section 5.1 are shown in Table 4.

Table 4: Tensorized setting for the Transformer model in CoMERA.

	format	linear shape	tensor shape	rank
embedding	TTM	(30527,768)	(64,80,80,60)	30
attention	TT	(768,768)	(12,8,8,8,8,12)	30
feed-forward	TT	(768,3072)	(12,8,8,12,16,16)	30

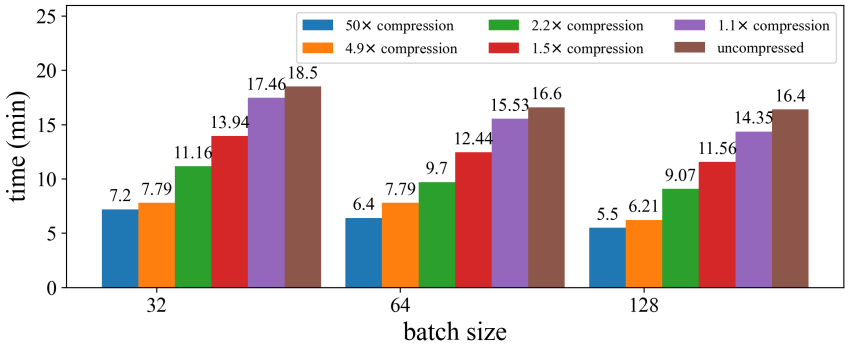


Figure 12: Per epoch training time of CoMERA on MNLI for various compression ratios.

A.8 Per Epoch Training Time of CoMERA on MNLI for Various Compression Ratios

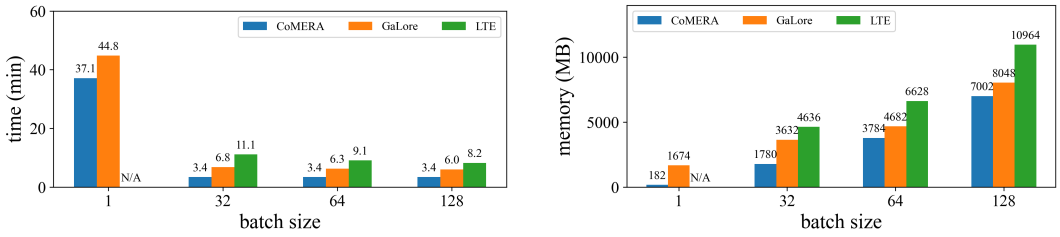
Table 12 shows the per-epoch training time of CoMERA on MNLI dataset for different compression ratios. The acceleration is more obvious for larger compression ratios. When the compression ratio is greater than 1, CoMERA always has speedup. When the compression ratio approaches 1, the time of CoMERA approaches that of uncompressed training.

A.9 Comparison with GaLore and LTE on a single RTX 4090 GPU

Since GaLore[41] uses a single RTX 4090 GPU for experiments in the original paper, we also run the experiments on the RTX 3090 GPU and compare the results. Figure 13 presents the training time and peak memory consumption. Compared to RTX 3090, the training on RTX 4090 uses similar memory and takes less training time, and CoMERA is still the fastest method and consumes the least memory among all three techniques. The memory savings are almost the same as the results reported in Figure 1 in our paper. The speed-up factors are almost identical for batch sizes 32, 64, and 128. For batch size 1, our method is $1.2\times$ faster and $1.7\times$ faster than GaLore on RTX 3090, respectively. The difference is that RTX 4090 GPU significantly accelerates matrix multiplications of batch size 1, while it does not accelerate that much for smaller tensor contractions. We find that $r = 30$ matrix multiplication on RTX 3090 has a similar speedup for both batch sizes, whereas the same multiplication on RTX 4090 only has speedup for batch 32 and does not have any speedup for batch 1. We would like to note that it might be caused by that different GPU platforms have different backend overhead, which can become more dominant as computation decreases to batch=1. We will continue optimizing GPU-level kernels to accelerate small tensor contractions and expect to see a similar speedup.

A.10 CoMERA Pretraining Result on $\text{BERT}_{\text{LARGE}}$

CoMERA on Original $\text{BERT}_{\text{LARGE}}$. Our results on CodeBERT is rather preliminary as only pre-training loss is available. For the original $\text{BERT}_{\text{LARGE}}$[9] (336M) and its CoMERA (125M) variant that we trained by using Wikipedia (2500M words), we achieve up to $6.36\times$ compression on tensorized layers and $2.69\times$ overall compression, with final loss of 1.45 vs 1.26. On downstream tasks, CoMERA ourperforms $\text{BERT}_{\text{LARGE}}$ on SST-2 (accuracy: 92.10% vs 91.74%) and MRPC (accuracy: 86.82% vs 86.00%), underperforms $\text{BERT}_{\text{LARGE}}$ on SQuAD (f1: 88.76% vs 90.68%).



(a) Training time per epoch.

(b) Peak memory consumption.

Figure 13: Training time and total memory cost of CoMERA, GaLore [41] and LTE [20] on a six-encoder transformer with varying batch sizes. The experiment is done on Nvidia RTX 4090 GPU.

Table 5: Speed-up of mixed-precision computation on tensor-compressed linear layers.

shape (b,m,n)	tensor-vector		matrix-vector	
	FP8-mix	FP32	FP8	FP32
(10000,1024,1024)	1.95	1.63	1.02	2.82
(20000,1024,1024)	2.02	3.37	1.93	5.41
(40000,1024,1024)	2.55	6.82	4.27	10.93
(10000,1024,4096)	1.97	3.96	3.69	10.28
(20000,1024,4096)	2.96	8.32	7.26	20.93
(40000,1024,4096)	5.47	17.13	15.27	45.60

A.11 Discussion: Mixed-Precision CoMERA

Modern GPUs offer low-precision computation to speed up the training and inference. It is natural to combine low-rank tensor compression and quantization to achieve the best training efficiency. However, CoMERA involves many small-size low-rank tensor contractions, and a naive low-precision implementation may even slow down the training due to the overhead caused by precision conversions.

To resolve the above issue, we implement mixed-precision computation in CoMERA based on one simple observation: large-size contractions enjoy much more benefits of low-precision computation than small-size ones. This is because the overhead caused by precision conversions can dominate the runtime in small-size contractions. In large-batch tensor-compressed training, small- and large-size tensor contractions can be distinguished by whether the batch size dimension b is involved. In general, a contraction with the batch b is regarded as large and is computed in a low precision. Otherwise, it is regarded small and is computed in full-precision. The actual mixed-precision algorithm depends on the contraction path used in the forward- and back- propagations of CoMERA.

Runtime. We evaluate the mixed-precision forward and backward propagations of CoMERA in a FP8 precision on the NVIDIA L4 GPU. We consider a single linear layer. The shapes $(1024, 1024)$ and $(1024, 4096)$ are converted to the TT shapes $(16, 8, 8, 8, 8, 16)$ and $(16, 8, 8, 16, 16, 16)$ respectively, and the ranks are both 32. The total execution time for 1000 forward and backward propagations are shown in Table 5. The FP8 tensor-compressed linear layer has about $3\times$ speed-up compared to the FP8 vanilla linear layer when the batch size and layer size are large. When the batch size is small, the FP8 vanilla linear layer is even faster. This is because the tensor-compressed linear layer consists of a few sequential computations that are not well supported by current GPU kernels. We expect to see a more significant acceleration after optimizing the GPU kernels.

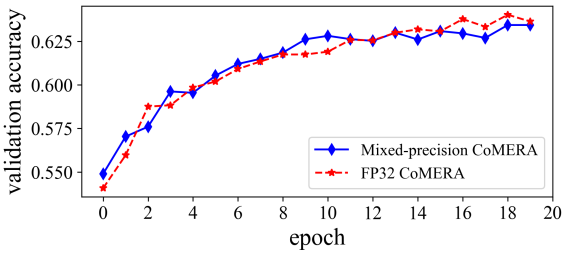


Figure 14: Convergence of mixed-precision CoMERA on the six-encoder transformer.

Convergence. We use the mixed-precision CoMERA to train the DLRM model and the six-encoder transformer. The result on DLRM is shown in Table 6. The convergence curve of the six-encoder transformer is shown in Figure 14. The experiments demonstrate that the accuracy of FP8 training is

Table 6: Training results of mixed-precision CoMERA on DLRM (batch size=10,000).

	accuracy	normalized CE
FP32 CoMERA	78.76%	0.792
FP8/FP32 mixed-precision CoMERA	78.88%	0.793

similar to FP32 training. However, we did not see much acceleration of using FP8 in the experiments. This is mainly because of ① the computation overhead of slow data type casting between FP32 and FP8; ② the sequential execution of small tensor contractions that are not well supported by current GPUs; ③ the relatively small sizes of linear layers in the tested models. We will investigate these problems in the future, and are optimistic to see significant acceleration on larger models.