Ultra Memory-Efficient On-FPGA Training of Transformers via Tensor-Compressed Optimization

Jiayi Tian, Jinming Lu, Hai Li, Xiangwei Wang, Cong (Callie) Hao, Ian Young, *Fellow, IEEE*, Zheng Zhang

***Abstract*—Transformer models have achieved state-of-the-art performance across a wide range of machine learning tasks. There is growing interest in training transformers on resource-constrained edge devices due to considerations such as privacy, domain adaptation, and on-device scientific machine learning. However, the significant computational and memory demands required for transformer training often exceed the capabilities of an edge device. Leveraging low-rank tensor compression, this paper presents the first on-FPGA accelerator for transformer training. On the algorithm side, we present a bi-directional contraction flow for tensorized transformer training, significantly reducing the computational FLOPS and intra-layer memory costs compared to existing tensor operations. On the hardware side, we store all highly compressed model parameters and gradient information on chip, creating an on-chip-memory-only framework for each stage in training. This reduces off-chip communication and minimizes latency and energy costs. Additionally, we implement custom computing kernels for each training stage and employ intra-layer parallelism and pipe-lining to further enhance run-time and memory efficiency. Through experiments on transformer models within 36.7 to 93.5 MB using FP-32 data formats on the ATIS dataset, our tensorized FPGA accelerator could conduct single-batch end-to-end training on the AMD Alevo U50 FPGA, with a memory budget of less than 6-MB BRAM and 22.5-MB URAM. Compared to uncompressed training on the NVIDIA RTX 3090 GPU, our on-FPGA training achieves a memory reduction of $30\times$ to $51\times$. Our FPGA accelerator also achieves up to $4.0\times$ less energy cost per epoch compared with tensor transformer training on an NVIDIA RTX 3090 GPU. As an initial result, this work highlights the significant potential of large-scale tensor training on edge devices.**

***Index Terms*—Transformer Models, FPGA Accelerator, On-Device Training, Low-Rank Tensor Compression.**

I. Introduction

THERE has been growing interest in on-device training of machine learning models. In finance and healthcare [1], [2], privacy regulations and user expectations often prohibit raw data from being transmitted to external servers, making local training a practical and necessary solution. In robotic and autonomous systems, machine learning has shown great promise in high-dimensional safety verification and control [3]–[5], where the neural network needs to be trained again as the system dynamics or safety regions keep updated. Meanwhile, pre-trained AI model often suffers from performance degradation in practical deployment due to the uncertain environment and data distribution shift [6], necessitating on-device model adaptation. Due to the limited size, weight, and power budget, resource-constrained devices such as FPGA, micro-controllers, or embedded GPUs are often used for edge AI. In recent years, several edge training accelerators have been proposed for convolutional neural networks [7], [8], multilayer perceptron networks, or recurrent neural networks [9]–[11]. Most existing work focuses on on-device lightweight adaptation or partial fine-tuning, rather than the more challenging—but potentially more effective—full-model training, due to the limited computing and memory budget.

Transformer models [12] have achieved state-of-the-art performance in numerous application domains, including (but not limited to) natural language processing [12], [13], computer vision [14], [15] and scientific simulation [16]. Recently, on-device fine-tuning of transformers has been explored in practical settings such as multilingual voice assistants and document scanners, where adapting models to local data can enhance personalization while preserving user privacy [17], [18]. These scenarios underscore the growing demand for efficient transformer training directly on edge devices—particularly in applications where centralized retraining is impractical or undesirable. Compared with on-device fine-tuning, end-to-end on-device training is more important in physics intelligence (e.g., neural PDE solvers for safe robotic control) since the solution will change dramatically as sensor data and system dynamics change, leading to complete failure of fine-tuning. However, the substantial memory and computational burdens of transformers have posed new challenges for their training and deployment on resource-constrained devices. For instance, the model parameters of a medium-size transformer can easily exceed the memory capacity of an edge device (e.g. FPGA), making it nontrivial to perform inference and infeasible to perform training on it.

Recent works have investigated quantization [19]–[21], low-rank compression [22]–[24], pruning [25], [26] and knowledge distillation [27], [28] to reduce the cost of fine-tuning on GPU or inference on edge devices for transformer models. However, full-model transformer training on resource-constrained devices remains a great challenge, due to two main factors:

- Firstly, it is hard to meet the memory requirement of transformer training. A transformer model can easily consume $> 5\times$ more on-chip memory than the capacity of an FPGA. It is difficult to fill this memory gap even if sparse [29] or low-precision training [21], [30]–[32] methods are utilized.

J. Tian, J. Lu and Z. Zhang are with Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106; H. Li and I. Young are with Intel Corporation, Hillsboro, OR 97124; X. Wang is with Department of Computer Science, North Carolina State University. He participated in this work when he was an exchange undergraduate student at UCSB; C. Hao is with School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332. She participated in this work before Intel funded this project. (email: jiayi_tian@ucsb.edu; jinminglu@ucsb.edu; hai.li@intel.com; xwang258@ncsu.edu; callie.hao@gatech.edu; ian.young@intel.com; zhengzhang@ece.ucsb.edu).

For example, ideal memory reduction (which is hard to achieve in practice) is limited to $8\times$ even if ultra-low-precision 4-bit training [32] is used.

- Secondly, on-device training involves more computing tasks than on-device inference, such as backward propagation, gradient generation, and parameter update. This increases both the memory and computing burden on the edge devices. Furthermore, data dependency between forward and back propagations further complicates data communication, task scheduling, and memory management.

In this paper, we use FPGA as a demonstration platform to investigate the end-to-end training of transformer models on edge devices that leverage low-rank tensor optimization. As a high-dimensional generalization of matrix decomposition [33], tensor decomposition can achieve much higher compression ratios in neural networks. This may provide a huge reduction in memory and computing costs, enabling on-device training and inference of large AI models that were previously infeasible. Previous work has shown orders-of-magnitude model reduction in convolutional and recurrent networks by low-rank tensor decomposition during both post-training compression [34]–[36] and training-aware compression [37]–[39]. For transformer models, some studies have applied tensor decomposition to compress specific components such as embedding or attention layers [39], [40], but these approaches target only partial modules and achieve limited model compression ratio.

From a hardware execution standpoint, although tensor decomposition offers theoretical reductions in memory footprint and computation, practical speedup on GPUs is often limited by the tiny and sequential tensor-contracted operations. We profile the top-10 most time-consuming kernels during MM- and TT-based transformer training, where we found that TT kernels consistently exhibit $6.5\times$ lower occupancy and $3.0\times$ fewer blocks per streaming multiprocessor (SM) on average. This leads to inefficient scheduling, underutilized GPU resources, and increased latency. In contrast, FPGAs support custom dataflows and explicit memory control, making them more suitable for executing tensor-compressed transformer training efficiently on edge devices. While prior FPGA/ASIC [41]–[44], emerging computing platforms [45]–[47] designs have explored this potential, most focus solely on inference or small-model training acceleration, which do not address full-model on-device training for transformer models.

To fill the research gap described above, this paper presents, for the first time, an FPGA accelerator for low-rank tensor-compressed *full-model* transformer training. This research can be regarded as a proof-of-concept demonstration and a pathfinding effort toward future ASIC implementation. This article presents the following novel contributions:

- We design a tensor-compressed transformer training accelerator implemented on FPGA. Using tensor decomposition, transformer training becomes ultra memory-efficient, allowing full-model training of up to 93.5-MB model with an 28.4-MB on-chip memory budget.
- We present a novel bidirectional tensor contraction technique that significantly improves the computing and memory efficiency of tensor-compressed forward and backward propagation. This contraction flow surpasses the sequential computation flows employed in previous inference accelerators [41], [42]. In addition, we develop a comprehensive model for analyzing the computing and memory costs associated with various tensor decomposition formats and contraction orders.
- We optimize the parallel scheduling flow and develop an operation fusion strategy to enhance the trade-off between run-time and hardware cost. Additionally, we propose a tensor grouping technique to enhance the memory efficiency of storing tensor-compressed model weights.
- We implement the training algorithm on an AMD Alveo U50 FPGA, evaluate hardware performance using the ATIS dataset [48] on a transformer model with various number of encoding blocks. Experimental results demonstrate that our FPGA accelerator can achieve up to $4.0\times$ lower energy costs with $48.2\times$ reduction in computing memory compared to transformer training using Pytorch on a Nvidia RTX 3090 GPU, as highlighted in Fig. 1.

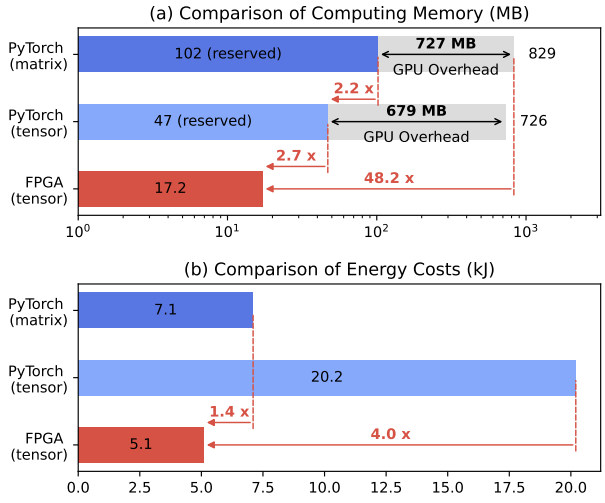


Fig. 1. Comparison of computing memory consumption (top) and energy costs (bottom) between PyTorch-based GPU training and our accelerator; blue bars represent reserved CUDA memory without additional overhead.

As a preliminary study, this work has not used any pruning or quantization techniques, and there hardware resource is still under utilized. Therefore, the performance gain can be further improved in the future. However, the current initial study suffices to demonstrate the great potential of low-rank tensor computation to scale up training on edge devices.

II. Background

In this section, we introduce some necessary background related to our work, including the transformer architecture, tensor decomposition and contraction, and tensor-compressed neural network.

A. Transformer Models

A transformer model consists of a stack of encoder or decoder blocks. For discrimination tasks, one can use encoders

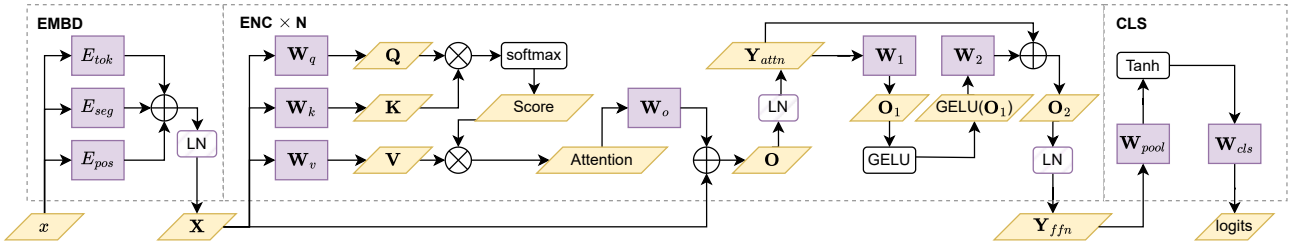


Fig. 2. Transformer structure for classification tasks. Inter-layer activation is represented using yellow blocks, embedding tables and linear layer weights are in purple blocks, and non-linear functions are in white blocks.

only to process the input sequence and generate output representations. For generative tasks, decoders should be used to generate the output sequence step by step, using both the encoded input and its own previous output. A encoder block and a decoder block have the same structure, a self-attention layer, and a feed-forward network (FFN). Both encoder and decoder blocks include residual connections and layer normalization to stabilize training. The computation in one encoder block could be written sequentially as

$$\mathbf{Q} = \mathbf{W}_q\mathbf{X} + \mathbf{B}_q, \mathbf{K} = \mathbf{W}_k\mathbf{X} + \mathbf{B}_k, \mathbf{V} = \mathbf{W}_v\mathbf{X} + \mathbf{B}_v,$$

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}, \tag{1}$$

$$\mathbf{Y}_{\text{attn}} = \text{LN}\left(\mathbf{W}_o\,\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) + \mathbf{B}_o + \mathbf{X}\right),$$

$$\mathbf{Y}_{\text{ffn}} = \text{LN}\left(\mathbf{W}_2\,\text{GELU}(\mathbf{W}_1\mathbf{Y}_{\text{attn}} + \mathbf{B}_1) + \mathbf{B}_2 + \mathbf{Y}_{\text{attn}}\right).$$

Let d_{hid} denote the hidden state dimension of the encoder, and let N_{seq} be the sequence length. Here, $\mathbf{X}$ is the input to the encoder, and $\mathbf{Y}_{\text{attn}}$ represents the output of the attention layer and feed-forward network, both with shape $d_{\text{hid}} \times N_{\text{seq}}$. The matrices $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}v$, and $\mathbf{W}o$ are the projection weights in the attention mechanism, while $\mathbf{W}1$ and $\mathbf{W}2$ are the weights in the feed-forward network, each with shape $d_{\text{hid}} \times d_{\text{hid}}$. The matrices $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$ denote the query, key, and value features, respectively, each of shape $d_{\text{hid}} \times N_{\text{seq}}$. LN denotes the layer normalization operation. The nonlinearities used are $\mathrm{Softmax}$ in the attention mechanism and GELU in the feed-forward network.

Before the encoder, an embedding layer transforms discrete tokens (e.g., words or subwords) into dense vectors for semantic processing by the attention mechanism. To encode positional information, positional embeddings $\mathrm{E}_{\text{pos}} \in \mathbb{R}^{d_{\text{hid}} \times N_{\text{seq}}}$ are added to the token representations. In natural language tasks, token embeddings $\mathrm{E}_{\text{tok}} \in \mathbb{R}^{d_{\text{hid}} \times |\mathcal{V}|}$ map tokens x_i from a vocabulary $\mathcal{V}$ into continuous vectors, while segment embeddings $\mathrm{E}_{\text{seg}} \in \mathbb{R}^{d_{\text{hid}} \times S}$ (where S is the number of segments, e.g., sentence A and B) distinguish tokens from different segments, enabling sentence-pair modeling and similar tasks. The embedding output $z_i \in \mathbb{R}^{d_{\text{hid}} \times N_{\text{seq}}}$ of a transformer can be expressed as

$$z_i = \mathrm{E}_{\text{tok}}(x_i) + \mathrm{E}_{\text{pos}}(i) + \mathrm{E}_{\text{seg}}(p_i), \tag{2}$$

where x_i is the i-th token in the input sequence and p_i indicates its segment ID.

For classification tasks, a classifier—typically consisting of one or more linear layers followed by a non-linear activation—is applied to the final hidden representation of the [CLS] token to perform prediction. Fig. 2 illustrates the model structure of a standard transformer used for classification.

B. Tensor Decomposition and Contraction

Tensors are a high-dimensional generalization of matrices and vectors. A generic d-way could be represented as $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$, where n_k is the size of the mode (or dimension) k. The $(i_1, i_2, \cdots, i_d)$-th element of $\boldsymbol{\mathcal{A}}$ is written as $a_{i_1,i_2,\cdots,i_d}$ or $\boldsymbol{\mathcal{A}}(i_1, i_2, \cdots, i_d)$. One can also use graphs to represent tensor operators. Specifically, a d-way tensor is denoted by a node with d edges. If two nodes are connected by an edge, then a tensor contraction happens between them, and the connected mode will diminish. Consider, for example, $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ and $\boldsymbol{\mathcal{B}} \in \mathbb{R}^{m_1 \times \ldots \times m_l}$. Assume that $n_s = m_t$, the tensor contraction of these two tensors could be formulated as

$$\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} \times_s^t \boldsymbol{\mathcal{B}}. \tag{3}$$

Tensor $\boldsymbol{\mathcal{C}} \in \mathbb{R}^{n_1 \times \cdots n_{s-1} \times n_{s+1} \cdots \times n_d \times m_1 \times \cdots m_{t-1} \times m_{t+1} \cdots \times m_l}$ has $d + l - 2$ modes, and each element of $\boldsymbol{\mathcal{C}}$ is

$$c_{i_1,\cdots,i_d,j_1,\cdots,j_l} = \sum_{i_s,j_t=1}^{n_s} a_{i_1,\cdots,i_s,\cdots,i_d} b_{j_1,\cdots,j_t,\cdots,j_l} \tag{4}$$

The contraction operations of multiple tensors can be represented as a tensor network with many nodes and edges. The number of free edges in a tensor network decides the final shape and size of the resulting tensor.

It is expensive to store and compute high-dimensional tensors directly. Fortunately, practical tensors can often be compressed by low-rank tensor decompositions [33]. We mainly use the tensor-train (TT) decomposition [49] and its variant tensor-train-matrix (TTM) decomposition for compressed training. TT decomposes a tensor $\boldsymbol{\mathcal{A}} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ as

$$\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{G}}_1 \times_3^1 \boldsymbol{\mathcal{G}}_2 \times_3^1 \ldots \boldsymbol{\mathcal{G}}_d, \tag{5}$$

where $\boldsymbol{\mathcal{G}}_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ is a TT factor, $(r_0, r_1, \cdots, r_d)$ is called TT rank, and $r_0 = r_d = 1$. TTM decomposes an order-$2d$ tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{n_1 \times \ldots \times n_d \times m_1 \times \ldots \times m_d}$ as

$$\boldsymbol{\mathcal{W}} = \boldsymbol{\mathcal{F}}_1 \times_4^1 \boldsymbol{\mathcal{F}}_2 \times_4^1 \ldots \boldsymbol{\mathcal{F}}_d, \tag{6}$$

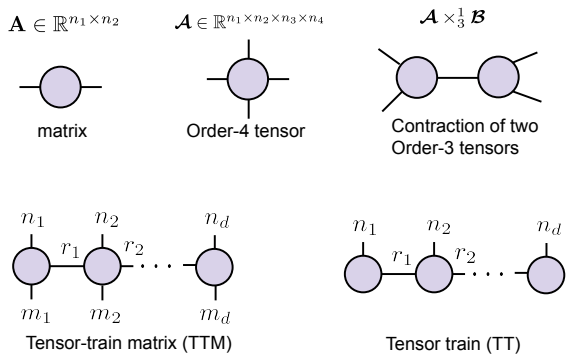


Fig. 3. Tensor graph representation for tensors, tensor contraction, and tensor networks for TTM and TT decomposition.

where $\boldsymbol{\mathcal{F}}_k \in \mathbb{R}^{r_{k-1} \times n_k \times m_k \times r_k}$, and $r_0 = r_d = 1$.

The graph representations for tensors, tensor contractions, and tensor networks for TTM and TT are shown in Fig. 3.

C. Tensor-Compressed Neural Network

Tensor decomposition has been widely adopted to train compressed neural networks from-scratch [37], [40], [50]–[53]. Among the various low-rank tensor formats, TT and TTM are the most widely used due to their exceptional accuracy, orders of magnitude compression ratios, and straightforward data structures [22], [37], [54]–[56]. In this work, we extend tensor-compressed training to transformer architectures, primarily using TT to compress the weight matrices and TTM to compress the embedding table.

We consider the weight matirx $\mathbf{W} \in \mathbb{R}^{M \times N}$ in a linear layer. Assume $M = \prod_i^d m_i$ and $N = \prod_i^d n_i$, and we reshape the matrix $\mathbf{W}$ into an order-$2d$ tensor $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 \times \ldots \times m_d \times n_1 \times \ldots \times n_d}$. With TT decomposition, we can represent $\boldsymbol{\mathcal{W}}$ with TT cores $\{\boldsymbol{\mathcal{G}}_k\}_{k=1}^{2d}$ as

$$\boldsymbol{\mathcal{W}} = \boldsymbol{\mathcal{G}}_1 \times_3^1 \ldots \boldsymbol{\mathcal{G}}_k \times_3^1 \ldots \boldsymbol{\mathcal{G}}_d \times_3^1 \ldots \boldsymbol{\mathcal{G}}_{d+k} \times_3^1 \boldsymbol{\mathcal{G}}_{2d}, \tag{7}$$

Here $\boldsymbol{\mathcal{G}}_k \in \mathbb{R}^{r_{k-1} \times m_k \times r_k}$ when $1 \le k \le d$ and $\boldsymbol{\mathcal{G}}_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ when $d+1 \le k \le 2d$. The total number of variables in $\boldsymbol{\mathcal{W}}$ is reduced to $\sum_{k=1}^{d}(r_{k-1} m_k r_k + r_{k-1+d} n_k r_{k+d})$ from $\prod_{k=1}^{d} m_k n_k$, which approximately achieves $O(m^d n^d) \to O(dr^2(m+n))$ memory reduction.

For the embedding table, we follow [39], [56] and use TTM to compress the vocab dictionary. We first reshape the token embedding layer $\mathbf{E}_{\text{tok}} \in \mathbb{R}^{M \times N}$ to $\boldsymbol{\mathcal{E}}_{\text{tok}} \in \mathbb{R}^{m_1 \times n_1 \ldots \times m_d \times n_d}$, where $M = \prod_i^d m_i$ and $N = \prod_i^d n_i$. We define the TTM representation for the token embedding layer as

$$\boldsymbol{\mathcal{E}}_{\text{tok}} = \boldsymbol{\mathcal{F}}_1 \times_4^1 \ldots \boldsymbol{\mathcal{F}}_k \times_4^1 \ldots \boldsymbol{\mathcal{F}}_d, \tag{8}$$

where $\boldsymbol{\mathcal{F}}_k \in \mathbb{R}^{r_{k-1} \times m_k \times n_k \times r_k}$. TTM reduces the number of parameters from $\prod_{k=1}^{d} m_k n_k$ to $\sum_{k=1}^{d}(r_{k-1} m_k n_k r_k)$, achieving $O(m^{d-1} n^{d-1} / dr^2)$ memory reduction.

III. Tensorized Transformer Training

A. Overall Training Framework

We consider the transformer in Fig. 2, which has one embedding layer, N encoder blocks and a task-specific classifier. The three embedding tables for token, position, and segment are compressed to TTM as done in [56]. All weight matrices (including those in the attention layers and feed-forward layers) of the encoder blocks are compressed to TT. The fully connected weight matrices in the classifier are also compressed to TT. The last task-specific linear layer for classification is kept uncompressed. In the training process, we use stochastic gradient descent (SGD) to solve the optimization problem

$$\min_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}, \mathcal{D}), \tag{9}$$

where $\mathcal{D}$ is the training dataset. Note that the training variables $\boldsymbol{\theta}$ include compression factors once a matrix is parameterized in a TT or TTM format. This differs from standard training where only large-scale 2-D matrices are trained.

Tensorized transformer training includes three stages:

1) **Forward propagation (FP)**. In a forward pass, the input tokens are fed into the embedding layer to generate hidden features, then the N encoders sequentially process the input features, and finally the classifier computes predictions and a loss function computes the loss using the predictions and labels. The main computations, matrix-vector products, are replaced with low-rank tensor-network contractions, which leads to a significant reduction in memory and FLOPs.
2) **Backward propagation (BP)**. The BP process first computes the gradients of the loss with respect to feature values according to a chain rule. After that, the gradient with respect to each TT factor is also computed via a tensor network contraction.
3) **Model parameter update (PU)**. The model parameters are updated as $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{\theta}'$, where α is a learning rate and $\boldsymbol{\theta}' = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ is the gradient. Since we use tensor-compressed training, updates are also performed on TT or TTM factors for compressed layers. For example, a TT core $\boldsymbol{\mathcal{G}}_k$ is updated as $\boldsymbol{\mathcal{G}}_k \leftarrow \boldsymbol{\mathcal{G}}_k - \alpha \boldsymbol{\mathcal{G}}'_k$ in each iteration.

B. Tensorized Linear Layer

We use tensorized linear layers in most computing blocks of a transformer, including the feed-forward networks and the linear transforms in the attention layers.

Forward propagation. Most computations in the forward propagation of a transformer involve the matrix-vector product $\mathbf{y} = \mathbf{W}\mathbf{x}$, with $\mathbf{W} \in \mathbb{R}^{M \times N}$ and $\mathbf{x} \in \mathbb{R}^{N}$. Assume that $M = \prod_{i=1}^{d} m_i$ and $N = \prod_{i=1}^{d} n_i$, and that $\mathbf{W}$ is reshaped to a tensor $\boldsymbol{\mathcal{W}}$ of order $2d$. With the TT representation in (7), the matrix-vector product is efficiently computed by low-rank tensor network contraction in our framework. Given the $2d$ TT cores $\{\boldsymbol{\mathcal{G}}_k\}_{k=1}^{2d}$ that represent $\mathbf{W}$, we compute $\mathbf{y} = \mathbf{W}\mathbf{x}$ in a tensor-compressed form as

$$y_{i_1,\ldots i_d} = \boldsymbol{\mathcal{G}}_1[i_1] \ldots \boldsymbol{\mathcal{G}}_d[i_d] \sum_{j_1,\ldots,j_d} \boldsymbol{\mathcal{G}}_{d+1}[j_1] \ldots \boldsymbol{\mathcal{G}}_{2d}[j_d] x_{j_1,\ldots,j_d} \tag{13}$$

where $x_{j_1,\ldots,j_d}$ and $y_{i_1,\ldots i_d}$ are elements in the tensor-format input $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times n_2 \times \cdots n_d}$ and tensor-format output $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{m_1 \times m_2 \times \cdots m_d}$, separately. Here $\boldsymbol{\mathcal{G}}_k[i_k]$ denotes one slice of

$$\mathcal{G}'_{d+k}[i_{d+k}] = \sum_{i_1,\ldots,i_d,j_1,\ldots,j_{k-1},j_{k+1},\ldots j_d} \left(\mathcal{G}^T_{d+k-1}[j_{k-1}]\ldots\mathcal{G}^T_1[i_1]y'_{i_1,\ldots,i_d}\right)\left(x_{j_1,\ldots j_d}\mathcal{G}^T_{2d}[j_d]\ldots\mathcal{G}^T_{d+k+1}[j_{k+1}]\right), \tag{10}$$

$$\mathcal{G}'_{k}[i_{k}] = \sum_{i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_d,j_1,\ldots,j_d} \left(\mathcal{G}^T_{k-1}[i_{k-1}]\ldots\mathcal{G}^T_1[i_1]y'_{i_1,\ldots,i_d}\right)\left(x_{j_1,\ldots j_d}\mathcal{G}^T_{2d}[j_d]\ldots\mathcal{G}^T_{k+1}[i_{k+1}]\right), \quad \text{for } k \in [1,d] \tag{11}$$

$$\mathcal{F}'_{k}[i_k,j_k] = \sum_{i_1,\ldots,i_{k-1},i_{k+1},\ldots,i_d} \left(\mathcal{F}^T_{k-1}[i_{k-1},j_{k-1}]\ldots\mathcal{F}^T_1[i_1,j_1]y'_{i_1,\ldots,i_d}\right)\left(\mathcal{F}^T_{d}[i_d,j_d]\ldots\mathcal{F}^T_{k+1}[i_{k+1},j_{k+1}]\right) \tag{12}$$

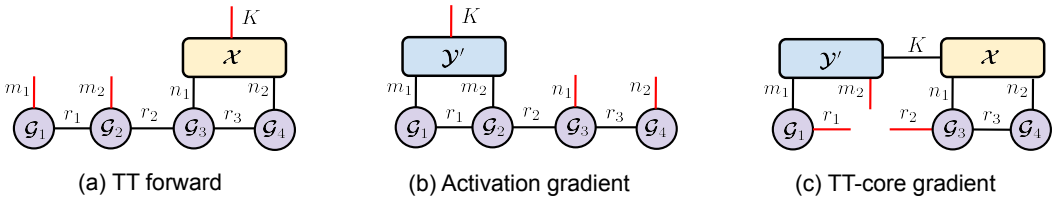


Fig. 4. Tensor graph representations for the TT-format (a) forward propagation, (b) gradient computation w.r.t. activations and (c) gradient computation w.r.t. a TT core (with $\mathcal{G}_2$ as an example) of a linear layer. For simplicity, here we assume $d = 2$ and that the matrix is fold to a tensor of order $2d = 4$. The red lines denote free edges of a tensor network, which indicate the dimension of the result tensor.

the order-3 TT core $\mathcal{G}_k$ by fixing its 2nd index as i_k. The above equation shows that contraction happens among $\mathcal{X}$ and the last d TT cores. Fig. 4(a) shows the graph representation of the above equation when d is 3. Here, we have an additional free edge of size K, because in practice we often perform training with an input dimension K which is the multiplication of a batch size multiplied with the sequence length. Therefore, $\mathbf{x}$ should be replaced with a matrix $\mathbf{X} \in \mathbb{R}^{N\times K}$.

Backward propagation. With low-rank TT parameterization, the BP process for gradient computation can also be done via tensor network contractions. Let us first define the following two tensors:

$$\mathcal{X}' = \frac{\partial \mathcal{L}}{\partial \mathcal{X}}, \quad \mathcal{Y}' = \frac{\partial \mathcal{L}}{\partial \mathcal{Y}}, \tag{14}$$

which are the order-d tensor representations for $\mathbf{x}' = \frac{\partial \mathcal{L}}{\partial \mathbf{x}}$ and $\mathbf{y}' = \frac{\partial \mathcal{L}}{\partial \mathbf{y}}$, respectively. In standard uncompressed training with $\mathbf{y} = \mathbf{W}\mathbf{x}$, it is known that

$$\mathbf{W}' = \frac{\partial \mathcal{L}}{\partial \mathbf{W}} = \mathbf{y}'\mathbf{x}^T. \tag{15}$$

With $\mathcal{Y}'$ being the high-order tensor representation for $\mathbf{Y}'$, the gradient with respect to the TT factors can be computed in two steps:

- We first compute $\mathcal{X}'$ via a tensor-network contraction

$$x'_{j_1,\ldots j_d} = \mathcal{G}^T_{2d}[j_d]\ldots\mathcal{G}^T_{d+1}[j_1] \sum_{i_1,\ldots,i_d} \mathcal{G}^T_d[i_d]\ldots\mathcal{G}^T_1[i_1]y'_{i_1,\ldots,i_d}. \tag{16}$$

- Then the gradients with respect to the $2d$ TT cores can be computed according to Eq. (10) and Eq. (11) for $k \in [1, d]$. Note that $\mathcal{G}'_k[i_k]$ denotes the i_k-th slice of the derivative tensor $\mathcal{G}'_k = \frac{\partial \mathcal{L}}{\partial \mathcal{G}_k}$ by fixing its 2nd index as i_k.

Fig. 4 (b) and Fig. 4 (c) show the tensor networks of activation gradient and TT core gradients in one linear layer. Note that Fig. 4 (c) illustrates how to compute $\mathcal{G}'_4$ by eliminating $\mathcal{G}_4$ from the computing graph. In general, we can remove $\mathcal{G}_k$ from the tensor network while keeping all other nodes if we want to compute the gradient $\frac{\partial \mathcal{L}}{\partial \mathcal{G}_k}$.

C. Tensorized Embedding Table

The embedding layer in a transformer has huge look-up tables, which consumes a large amount of memory. The TTM representation in (6) can greatly reduce the memory cost of embedding layers, but its forward and backward propagation uses different tensor-network contractions.

Forward propagation. Given the d TTM factors $\{\mathcal{F}_k\}_{k=1}^d$ that represent $\mathrm{E}_{\mathrm{tok}} \in \mathbb{R}^{\mathrm{M}\times\mathrm{N}}$, where $M = \prod_{i=1}^{d} m_i$ and $N = \prod_{i=1}^{d} n_i$, and thus $\mathrm{E}_{\mathrm{tok}}$ is reshaped to a tensor $\mathcal{E}$ of order $2d$. In the look-up process, a column of $\mathrm{E}_{\mathrm{tok}}$ is selected for each input token. We use an index vector $\mathbf{j} = [j_1, j_2, \cdots j_d]$ to select the j_k -th subtensor of each TTM core $\mathcal{F}_k$. The output feature $\mathbf{y}$ could be computed as

$$y_{i_1,\ldots i_d} = \mathcal{F}_1[i_1,j_1]\mathcal{F}_2[i_2,j_2]\ldots\mathcal{F}_d[i_d,j_d], \tag{17}$$

where $\mathcal{F}_k[i_k,j_k]$ is a 2-D slice of $\mathcal{F}$ by fixing its 2nd and 3rd indices as i_k and j_k, respectively. According to the above equation, using the TTM embedding table, we first look up each TTM core using the d-element index $\mathbf{j}$, and use the selected slices to form the output feature.

Backward propagation. Since the input index does not require gradients, we only compute the gradients for the tensor cores in the TTM embedding table according to Eq. (12). Here, $\mathcal{F}'_k[i_k,j_k]$ denotes a slice of the derivative tensor $\mathcal{F}'_k = \frac{\partial \mathcal{L}}{\partial \mathcal{F}_k}$ by fixing its 2nd and 3rd indices as i_k and j_k, respectively.

IV. Algorithm Implementation

As discussed earlier, training of tensor-compressed models is based on tensor contractions. While the contraction order does not affect the training curve, the associated computational and memory complexities are highly sensitive to it. Existing

$$\mathrm{MUL}_{\mathrm{TT}} = K \sum_{k=0}^{d-1} \left(r_{2d-k-1} r_{2d-k} \prod_{i=1}^{d-k} n_i + r_{d-k-1} r_{d-k} \prod_{i=d-k}^{d} m_i \right) \tag{18}$$

$$\mathrm{Memory}_{\mathrm{TT}} = K r_d + K \sum_{k=0}^{d-2} \left(r_{2d-k-1} \prod_{i=1}^{d-k-1} n_i + r_{d-k-1} \prod_{i=d-k}^{d} m_i \right) \tag{19}$$

$$\mathrm{MUL}_{\mathrm{BTT}} = \sum_{k=0}^{d-2} \left(r_{2d-k-1} r_{2d-k-2} \left(\prod_{i=d-k-1}^{d} n_i \right) + r_{k+1} r_{k+2} \left(\prod_{i=1}^{k+2} m_i \right) \right) + K r_d \left(\prod_{i=1}^{d} m_i + \prod_{i=1}^{d} n_i \right) \tag{20}$$

$$\mathrm{Memory}_{\mathrm{BTT}} = \sum_{k=0}^{d-2} \left(r_{2d-k-2} \left(\prod_{i=d-k-1}^{d} n_i \right) + r_{k+1} \left(\prod_{i=1}^{k+2} m_i \right) \right) + K r_d. \tag{21}$$

tensor-compressed inference accelerators typically adopt a right-to-left contraction sequence for both tensor-train matrix (TTM) [37], [41] and TT formats [42]. In this section, we propose an optimized contraction order for the tensor-train (TT)-format linear layer to enhance the computational and memory efficiency without changing the model performance.

A. Complexity of Right-to-Left Contractions

We first evaluate the computational and memory complexity of the right-to-left TT contraction in a linear layer. We only analyze the forward propagation, since computing the gradients with respect to activations and TT cores uses a similar contraction process. The total computational cost in training is roughly $3\times$ of the inference cost.

Computational Cost. The forward of a TT-format linear layer has $2d$ contraction steps. The total number of multiplications in a forward pass is estimated in Eq. (18), where the first and second terms accounts for the number of multiplications in the initial and final d contraction steps, separately.

Memory Cost. Training with the TT-format incurs additional intra-layer memory overhead due to the multiple contraction steps involved, which is estimated in Eq. (19). Specifically, there are $2d-1$ intermediate results generated during $2d$ contraction steps, , excluding the final step which produces the output and is therefore not counted. The first term in the equation captures the memory cost at the d-th contraction step, while the second and third terms correspond to the costs incurred during the initial and final $d-1$ steps, respectively. All of these intermediate results need to be stored for reuse in back propagation.

Fig. 5 (top) shows the contraction sequence for a linear layer in TT format when $d=2$. In the right-to-left contraction flow, the computation and memory complexity of every step depend on the input dimension K which is the batch size multiplied by the sequence length. In NLP and computer vision tasks, K can be large, leading to low hardware efficiency. To address this limitation, we propose an efficient bidirectional contraction scheme that reduces computational and memory costs by eliminating the dependence on K in most contraction steps.

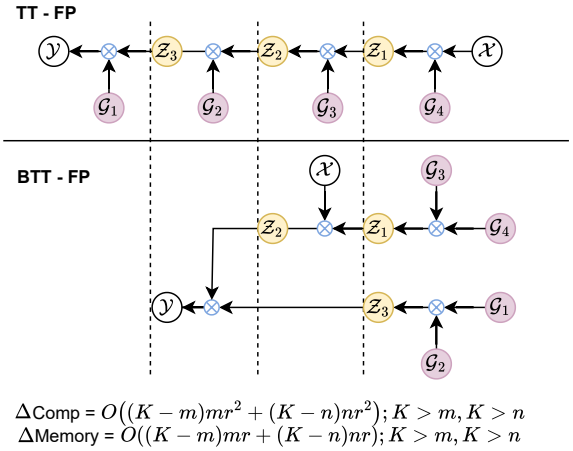


Fig. 5. Comparison of the computing flow of the TT-format and our modified BTT forward propagation when $d=2$. Contraction operations are represented in blue multipliers. Here white nodes represents input tensor $\boldsymbol{\mathcal{X}}$ and output tensor $\boldsymbol{\mathcal{Y}}$; purple nodes are TT cores $\boldsymbol{\mathcal{G}}_1$ to $\boldsymbol{\mathcal{G}}_4$; nodes marked in yellow denotes the intermediate contraction results of the involved tensors.

B. Bi-directional Tensor-Train Contraction

Our bidirectional tensor-train (BTT) contraction is shown in Fig. 5 (bottom). Since the left and right d TT cores have no data dependency, we can perform contraction from the left and right towards the middle in parallel. This can reduce the total number of computation stages from $2d$ to $d+1$. To clearly demonstrate the efficiency brought by our BTT contraction, we formulate the computing and memory costs and conduct a comprehensive comparison with the right-to-left TT contraction.

The core idea of our BTT is to modify the computing path of the tensor-compressed linear layer to explore the parallelism and reduce involving large K in most contraction steps to reduce the computing and memory efficiency. The resulting total computational cost and memory consumption are estimated as Eq. (20) and Eq. (21). In both equations, the first term—independent of the factor K—captures the costs

TABLE I
COMPUTATIONAL AND MEMORY TRAINING COMPLEXITIES OF EACH LINEAR LAYER FOR TTM, RIGHT-TO-LEFT TT CONTRACTION, AND THE PROPOSED BTT CONTRACTION METHOD.

Method	**FLOPs**	**Memory**	
		Weight	**Activation**
Matrix-Matrix multiplication (MM)	$O(3Kn^{2d})$	$O(n^{2d})$	0
Tensor-Train-Matrix contraction (TTM)	$O(3Kn^{d+1}((d-2)r^2+2r))$	$O(n^2((d-2)r^2+2r))$	$O(Kn^d(d-1)r)$
Right-to-left Tensor-Train contraction (TT)	$O(6K(\sum_{k=1}^{d-1} n^k r^2 + n^d r))$	$O(2n((d-2)r^2+2r))$	$O(2K\sum_{k=1}^{d-1} n^k r + Kr)$
Bidirectional Tensor-Train contraction (BTT, ours)	$O(6\sum_{k=2}^{d} n^k r^2 + 6Kn^d r)$		$O(2\sum_{k=2}^{d} n^k r + Kr)$

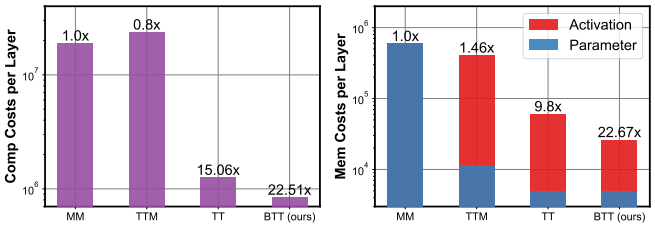


Fig. 6. Comparison of the computation and memory costs of MM, TTM, TT and our BTT.

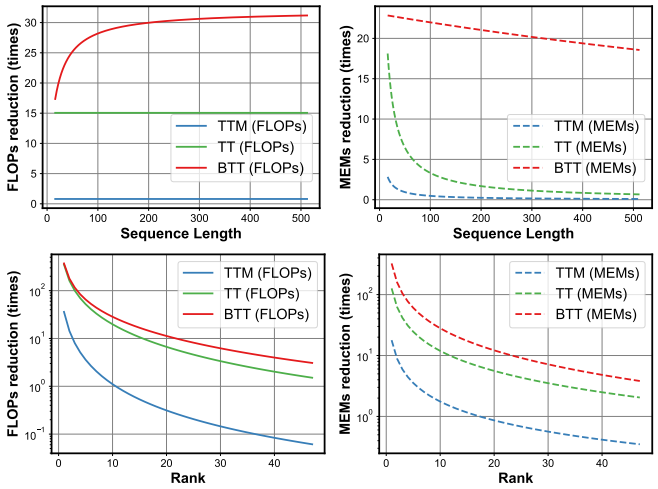


Fig. 7. Computational and memory costs of TTM-based contraction, TT-based contraction and our BTT-based contraction corresponding to different sequence length (top) and rank (bottom). The reduction ratio is calculated based on matrix-matrix multiplication.

associated with contracting the left and right TT cores during the initial d steps. In contrast, the final term, which is scaled by K, reflects the costs incurred in the final two contraction steps. Based on the above equations, BTT always have lower computational complexity when m_i and n_i are smaller than K, which is common in modern NLP and CV tasks.

To provide a comprehensive comparison between our BTT method and existing contraction schemes, we summarize the theoretical computing and memory costs of various methods in Table I. For simplicity, we assume $m = n$ in all equations.

Example. We consider a hidden dimension of 768 for the query matrix, and set $d = 3$ with $\{n_1, n_2, n_3\}$ and $\{m_1, m_2, m_3\}$ being $\{12, 8, 8\}$ and $\{8, 8, 12\}$, respectively. We also set the TT rank to 12 and the sequence length to 32. Fig. 6 compares the computation and memory costs of a linear layer implemented with different contraction methods. Clearly, our BTT contraction method consumes the least computation and memory cost. Compared with standard matrix-matrix multiplication, our BTT method is $22.51\times$ more computing efficient and $22.67\times$ more memory efficient. Compared with the right-to-left TT contraction, our BTT implementation further reduces the computing cost and the memory cost by $1.49\times$ and $2.31\times$, respectively.

To evaluate the influences of the hyper-parameters on our BTT contraction scheme, we show the computing and memory costs associated with different sequence length and TT ranks in Fig. 7. The standard matrix-matrix multiplication is used a baseline to estimate the reduction ratios regarding FLOPs and memory cost. In the upper figure, we fix the TT rank as 12 and increase the sequence length from 8 to 512. In the bottom figure, we fix the sequence length as 32 and increase the TT rank from 1 to 48. As the sequence length increases, our BTT contraction shows greater advantages over the right-to-left contraction of TT and TTM methods in terms of both FLOPS and memory cost. As the TT rank increases, the computing and memory advantage of all tensor-compressed methods degrades, but our BTT scheme always has the highest reduction ratios in both FLOPs and memory costs.

V. FPGA ACCELERATOR DESIGN

Based on the new proposed computing scheme, we develop an FPGA accelerator for tensor-compressed training. While the BTT contraction strategy effectively mitigates the sequential bottleneck inherent in conventional tensor contraction and significantly reduces both FLOPs and memory usage, it introduces several trade-offs—such as increased resource consumption due to duplicated compute and memory buffers, and elevated control complexity within the FPGA implementation. To address these challenges, we propose two key hardware-aware design strategies. First, we apply task rescheduling and tensor fusion to improve compute and memory efficiency during tensor-compressed transformer training. Second, we propose an on-chip memory management scheme based on tensor grouping, which enables efficient storage and access of tensor-compressed weights. Our training accelerator is implemented in C++ and transferred to RTL via high-level synthesis (HLS).

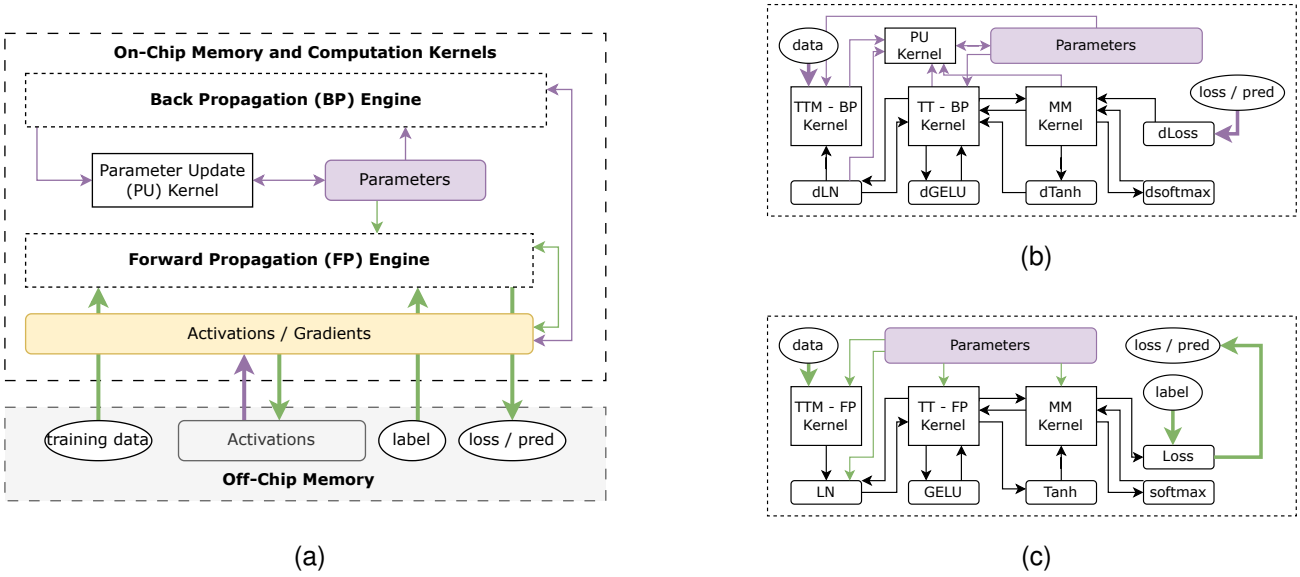


Fig. 8. (a): Overall architecture of the tensorized transformer training accelerator. (b) and (c): The detailed interactions between bottom-level computing kernels during backward and forward propagation. Note that in (b) and (c), the on-chip memory storing activation / gradient interacts with all computing kernels and thus is not shown in the figures.

A. Overall Architecture

Fig. 8 (a) shows the overall architecture of our tensor-compressed transformer training accelerator, including tensor-compressed forward propagation (FP) and backward propagation (BP) engines, on-chip and off-chip storage, and data transfer between difference training stages.

Due to the small number of model parameters caused by tensor compression, all training parameters and most activation values are stored in on-chip memory. This reduces the latency and energy overhead caused by off-chip communications. Since the volume of model parameters and activation values increase linearly with the number of layers, the total memory cost still exceeds the on-chip memory budget, thus off-chip memory access is inevitable. We observe that the activation values consume much more memory than the tensor-compressed model parameters. Some interlayer activations produced in FP are only needed in BP. To support more complex tasks with more layers, we store these activations in off-chip memory and fetch them to on-chip memory when computing the gradients with respect to tensor factors.

The detailed computing kernels and dataflows inside the tensor-compressed BP and FP engines are shown in Figs. 8 (b) and (c), respectively. To support the whole training progress, multiple computing kernels are developed for various computing schemes, including tensor contraction in the TTM-format embedding table and TT-format linear layers, matrix-multiplication (MM) in the attention parts and classifier, and various non-linear functions (softmax, GELU, Tanh, LayerNorm) used in transformer models.

B. Efficient Parallelism for Tensorized Linear Layers

The tensor contraction inside a TT-linear layer is highly sequential. Fortunately, our BTT-linear layer eliminates the data dependency in the first $d-1$ contraction steps, which brings parallelism inside the tensor linear layers and reduces the total length of contraction from $2d$ to $d+1$. However, directly applying intra-layer parallelism without considering inter-layer dataflow will cause inefficient resource usage. We propose task rescheduling and tensor fusion to tackle this problem.

1) Task Rescheduling for Attention Layers: Fig. 9 compares the rescheduled parallel BTT and the default scheme for forward propagation of an attention block. Here, we define the contraction kernels as follows:

- **Kernel MUL0**: This kernel denotes the contraction between $\mathcal{G}_4$ and $\mathcal{G}_3$ that produces $\mathcal{Z}_1$, or the contraction between $\mathcal{G}_1$ and $\mathcal{G}_2$ which produces $\mathcal{Z}_3$.
- **Kernel MUL1**: this kernel denotes the contraction between the input tensor $\mathcal{X}$ and the intermediate tensor $\mathcal{Z}_1$ that produces $\mathcal{Z}_2$.
- **Kernel MUL2**: this kernel denotes the contraction between intermediate tensors $\mathcal{Z}_2$ and $\mathcal{Z}_3$, producing the output of a linear layer $\mathcal{Y}$. In an attention block, $\mathcal{Y}$ can be the high-order tensorization of $\mathbf{Q}$, $\mathbf{K}$ or $\mathbf{V}$ features.

A naive parallelization (Fig. 9 top right) of the $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ computation can lead to resource overhead on FPGA. It runs all MUL0 kernels in the linear layers to compute $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ simultaneously since they are independent. As a result, six MUL0 kernels are needed in total. We use task rescheduling to reduce the hardware cost while achieving the same speedup on FPGA. The optimized tasks scheduling moves non-urgent MUL0 to later time steps and run them with other multipliers in parallel without increasing the total latency. This requires only 2 (rather than 6) reusable computing kernels for MUL0, leading to better resource utilization.

2) Fused Parallel BTT: The intermediate results inside a tensor linear layer consume much more memory than the compressed model parameters. Therefore, we propose a fused parallel BTT dataflow to eliminate the memory overhead

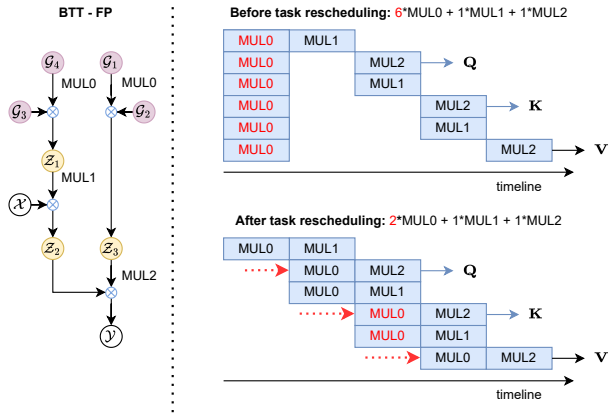


Fig. 9. Schedule visualization of the **Q**, **K**, and **V** linear layers in BTT-format forward propagation before and after task rescheduling. Left: Computational graph for a single linear layer in BTT forward propagation. Right: Timeline-based schedule indicating the number of required compute kernels, with key resource usage (e.g., MUL0 units) highlighted in red.

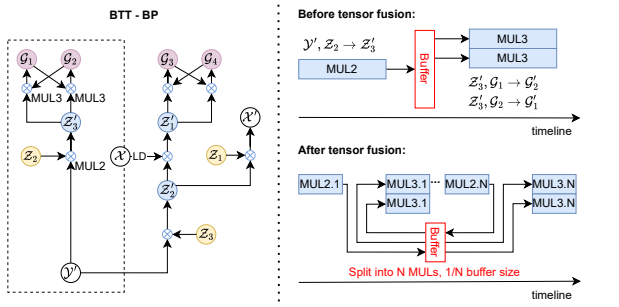


Fig. 10. Schedule viewer of BTT-format back propagation before and after tensor fusion. Left: Computational graph of the BTT-linear layer during back-propagation. Right: Timeline-based execution schedule illustrating how tensor fusion enables buffer reuse and reduces buffer size, with key improvements highlighted in red.

caused by the intermediate results, as shown in Fig. 10. Here we define the contraction kernels as follows:

- **Kernel MUL2:** this kernel contracts between the output gradient $\mathcal{Y}'$ and the intermediate result $\mathcal{Z}_2$, resulting in $\mathcal{Z}_3'$.
- **Kernel MUL3:** this kernel contracts the intermediate result $\mathcal{Z}_3'$ and the tensor factor ($\mathcal{G}_1$ or $\mathcal{G}_2$), produces $\mathcal{G}_2'$ or $\mathcal{G}_1'$, respectively, and then performs a parameter update.

We mainly apply operation fusion in these two contraction steps. As denoted in the original scheme (Fig. 10. top), all multiplications for a tensor contraction are completed before the next contraction step. In this way, all intermediate results between contraction steps are required to be stored, which creates a large buffer with high memory consumption. To relieve memory usage, we divide each contraction operation into multiple fine-grained contractions as follows:

- **Fine-Grained Contraction for MUL2:** we perform contraction between sub-tensor $\mathcal{Y}'[:, i_1, i_2]$ with $\mathcal{Z}_2$ and produce subtensor $\mathcal{Z}_3'[i_1, i_2, :]$.
- **Fine-Grained Contraction for MUL3:** we perform the contraction between a subtensor $\mathcal{Z}_3'[i_1, i_2, :]$ with another sub-tensor $\mathcal{G}_1[i_1, :]$ (or $\mathcal{G}_2[:, i_2, :]$), and produce a result sub-tensor $\mathcal{G}_2'[:, i_2, :]$ (or $\mathcal{G}_1'[i_1, :]$).

Here $i_1 \in \{1, ..., n_1\}, i_2 \in \{1, ..., n_2\}$, and we need to repeat the fine-grained multiplication for $n_1 n_2$ times. When one fine-grained contraction (Fig. 10 bottom) creates a small intermediate result, the next fine-grained contraction step uses it immediately, and thus only a small buffer with size $O(r)$ is required for all $n_1 n_2$ fine-grained contraction steps. The memory cost of intermediate contraction results is completely eliminated in the parameter gradient computation process.

C. Memory Management on FPGA

Although tensor-compressed training allows storing all model parameters in the BRAM, creating separate memory blocks for multiple small-size TT cores can result in dramatic

BRAM 36K	
Width	Depth
1	36k
2	18k
4	9k
9	4k
18	2k
36	1k
72	512

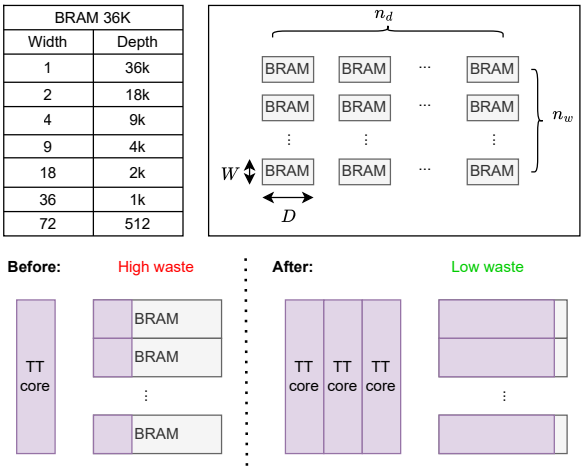


Fig. 11. Configurations of BRAM 36K. Number of BRAM of one array. BRAM usage efficiency before and after tensor grouping.

memory waste because of the fixed BRAM block size on FPGA. We propose a tensor core grouping method to improve BRAM utilization efficiency.

Each BRAM block in the FPGA has a fixed storage size C, and can be configured with various widths (W) and depths (D) with $C = W \times D$. For example, $C = 36,864$ bits for most state-of-the-art AMD FPGA. A BRAM block can be configured as $W = 1 \sim 72$ as shown in Fig. 11 (top left). In order to store a large data array, $n_w \times n_d$ BRAM blocks are concatenated in the dimension of "depth" or "width", as shown in Fig. 11 (top right). Here, n_w is calculated by the width of the data array divided by W, n_d is calculated by the depth of the data array divided by D. Both should be rounded up to their nearest integers, respectively. The data array width and depth are decided by the dimensions of the TT core (or grouped TT cores) after it is reshaped into a 2D structure.

To efficiently store the TT cores, we first consider the influence of parallel computing inside each tensor contraction operation. In our design, we implement parallelism over the rank index across all tensor contractions involving TT cores,

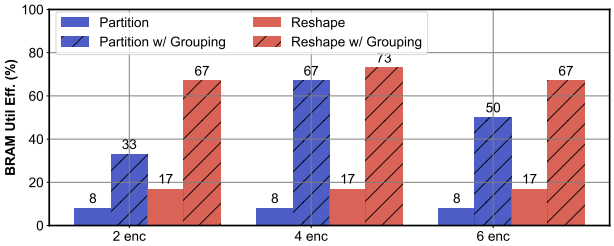


Fig. 12. Comparison of the BRAM utilization efficiency using different memory management strategies in different model sizes.

which requires r parallel read operations from each TT core. However, since each BRAM block has only two read/write ports, storing an entire TT core in a single BRAM block cannot support parallel access. High-Level Synthesis (HLS) offers two solutions to address this issue. The first approach, known as array partitioning, could partition a large data array into multiple small arrays and map them to separate BRAM blocks. Therefore, we use r BRAM blocks per TT core to enable parallel data loading. The second approach, known as array reshaping, concatenates multiple elements of the array by increasing the bit-width in order to support parallel data processing. This method is more memory efficient because it only consumes one BRAM block when $B_w r < \max(W)$, or $\lceil \frac{B_w r}{\max(W)} \rceil$ blocks otherwise. In this way, since we use floating-point 32, where $B_w = 32$ ($< \max(W) = 72$), the required number of BRAM blocks is always smaller than r.

Assuming that the tensor-compressed transformer involves N TT cores having the same size for simplicity, the total number of BRAM blocks can be roughly estimated as $N n_w \times n_d$. Let $\lceil x \rceil$ denote rounding up x to its nearest integer, then n_w and n_d can be decided as

$$\text{Array partitioning: } n_w = r\lceil \frac{B_w}{W} \rceil, n_d = \lceil \frac{nr}{D} \rceil, \tag{22}$$

$$\text{Array reshaping: } n_w = \lceil \frac{B_w r}{W} \rceil, n_d = \lceil \frac{nr}{D} \rceil. \tag{23}$$

The above analysis indicates that n_d will always be 1 since generally $nr << D$, leading to a huge waste of BRAM usage.

To improve BRAM efficiency, we increase the depth of the array by grouping some TT cores. Specifically, we concatenate K TT-cores without data dependency into a single array along the depth size, then assign one BRAM block to the whole group. In our implementation, TT cores are concatenated across encoder layers and contraction directions. Given a total of L encoder layers and $2d$ tensor factors, we set $K = (d-1)L$. In this case, the total number of BRAM blocks required is $\frac{N}{K} n_w \times n_d$, with n_w and n_d decided below:

$$\text{Array partitioning: } n_w = r\lceil \frac{B_w}{W} \rceil, n_d = \lceil \frac{Knr}{D} \rceil, \tag{24}$$

$$\text{Array reshaping: } n_w = \lceil \frac{B_w r}{W} \rceil, n_d = \lceil \frac{Knr}{D} \rceil. \tag{25}$$

Here, scaling up small $\frac{nr}{D}$ using K brings $\frac{Knr}{D}$ closer to its rounded-up integer, and thus we can greatly improve the BRAM utilization efficiency, as shown in Fig. 11 (bottom).

TABLE II
CONFIGURATIONS OF EACH LAYER IN UNCOMPRESSED AND TENSORIZED TRANSFORMER.

	Format	Matrix shape	Tensor shape	Rank
Embedding	TTM	$(1000, 768)$	$((10, 10, 10), (12, 8, 8))$	30
Attention	TT	$(768, 768)$	$(12, 8, 8, 8, 8, 12)$	12
Feed-forward	TT	$(768, 768)$	$(12, 8, 8, 8, 8, 12)$	12
Classification	TT	$(768, 768)$	$(12, 8, 8, 8, 8, 12)$	12

TABLE III
PERFORMANCE OF TENSOR-COMPRESSED VERSUS STANDARD MATRIX-FORMAT TRANSFORMER TRAINING ON ATIS DATASET.

Model	Intent Test Acc	Slot Test Acc	Size (MB)
2-ENC matrix	96.2%	**97.3%**	36.7
2-ENC tensor	**97.0%**	97.2%	**1.2 (30.5×)**
4-ENC matrix	95.7%	**97.4%**	65.1
4-ENC tensor	**96.6%**	**97.4%**	**1.5 (43.4×)**
6-ENC matrix	96.4%	**97.5%**	93.5
6-ENC tensor	**97.0%**	97.2%	**1.8 (52.0×)**

The BRAM management problem is formulated as

$$\min_{\theta} F(\theta, \beta) = N n_w n_d, \theta = \{W, D\}, \beta = \{N, K, n, r, B_w\}.$$

With a tensorization setting β, we can find W and D to maximize the BRAM utilization according to Eq. (22) – (25).

To show the effectiveness of our method, we define the efficiency of BRAM utilization as $\eta = \frac{N_{\min}}{N_{\text{total}}}$, where $N_{\min}$ is the ideal total memory usage for all tensor factors without considering the capacity of each BRAM block, and N_{total} is obtained using our BRAM management scheme. Fig. 12 shows that our BRAM management method can achieve a $3.9\times$ to $8.4\times$ higher utilization efficiency than the default under various partitioning strategies and model volume.

VI. EXPERIMENTS

A. Experimental Setup

We implement our tensor-compressed transformer training accelerator with high-level synthesis (HLS) using C++, and perform synthesis, placement, and routing using Vitis HLS 2023.2 on an AMD Alveo U50 FPGA board. The programmable logic fabric has 872k LUTs, 5952 DSPs, 5.9-MB BRAMs, and 22.5-MB URAMs. All model parameters, activations, and gradients during training are implemented in the floating-point 32-bit format with 100-MHz frequency.

B. Algorithmic Evaluation

We evaluated the FPGA accelerator by training a transformer on the ATIS dataset [48]. This is a data set widely used in natural language understanding (NLU), including audio recordings of customers booking flight tickets. The input sequence length is set to 32, but the accelerator is also reconfigurable to support larger sequence length and for other datasets. For functionality evaluation, we use the stochastic gradient descent (SGD) optimizer with a learning rate 4×10^{-3} and batch size of 1. The hyper-parameters of uncompressed training and our tensor-compressed training are listed in Table II. Table III shows that the tensor-compressed method could achieve the same accuracy with $30.5\times$ to $52.0\times$ compression ratios for transformers with 2 to 6 encoding blocks.

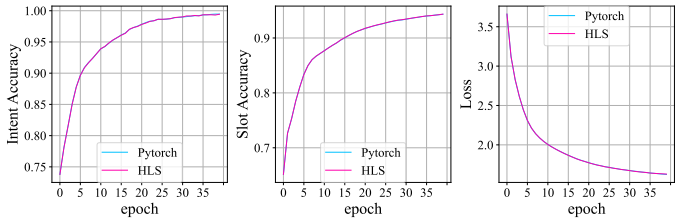


Fig. 13. Training accuracy and loss of the accelerator (2-ENC) compared with PyTorch in ATIS dataset through the whole training process (40 epochs).

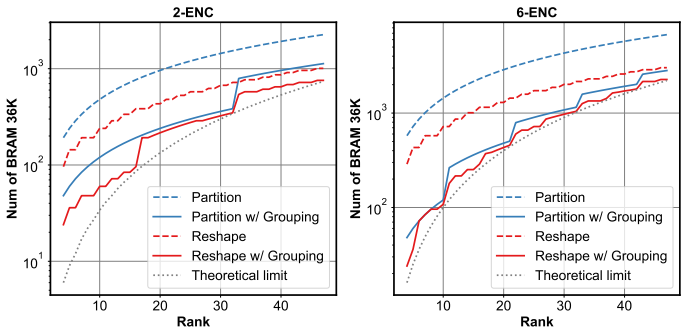


Fig. 14. BRAM usage for all tensor factors using different memory management strategy versus rank.

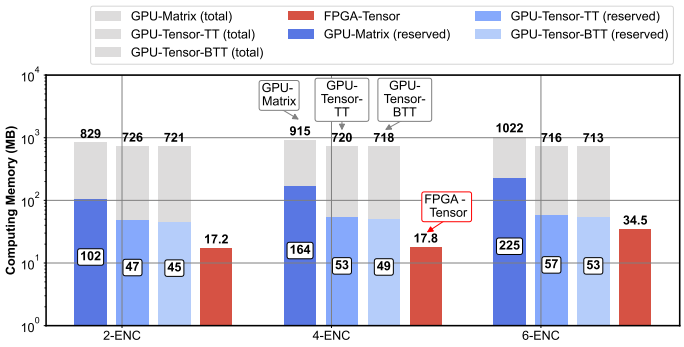


Fig. 15. Comparison of the computing memory costs between GPU (with or without additional overhead) and FPGA in different model size.

C. FPGA Accelerator Evaluation

1) Functionality Evaluation: To validate the functionality of our transformer training accelerator with parallel BTT implementation, we compare the training loss and accuracy curves of our high-level synthesis (HLS) FPGA implementation with PyTorch training on a GPU, as illustrated in Fig. 13. The ATIS dataset is a multi-task dataset for both intent classification and slot filling. Intent classification aims to determine the overarching goal or purpose of the user's query and is treated as a text classification problem. Slot filling identifies key entities or attributes (slots) within a user's utterance and classifies them into predefined categories. The accuracy and loss curves for both tasks demonstrate that the results from our HLS FPGA implementation closely match those from PyTorch training, confirming the consistency and reliability of our accelerator design.

2) Effects of BRAM Optimization: Fig. 14 illustrates the number of BRAMs required to store all TT cores using different allocation methods in Section V-C. It is shown that our proposed BRAM management strategy always consumes the least BRAM resources and even reaches the theoretical limit in some specific rank settings. Our method could bring the actual cost of BRAM closer to the ideal usage, because the TT-core grouping could reduce the waste of depth per BRAM block. Combining this BRAM optimization with tensor compression, the memory burden is further relieved, allowing storing all compressed model parameters in on-chip memory.

3) Overall Accelerator Performance: Table IV presents the hardware resource utilization of our FPGA training accelerator in various model sizes from RTL implementation reports. We evaluated the accelerator with a transformer with 2 to 6 encoder layers. The utilization of DSPs, LUTs, and FFs remains consistent across different model configurations, as the same computational kernels are employed regardless of the model settings. However, BRAM usage decreases while URAM usage increases with a greater number of layers. This behavior can be attributed to the application of our grouping technique to intermediate arrays, which demands additional memory as the number of layers grows. Consequently, HLS automatically allocates these memory requirements to URAM to enhance the utilization efficiency of on-chip memory.

D. Performance Comparison with GPU

We perform end-to-end performance evaluation of our FPGA accelerator (operating at 100 MHz) compared to the NVIDIA RTX 3090 GPU (acting at 1395 MHz). The evaluation includes metrics such as memory usage, latency, power consumption, and overall energy efficiency across both platforms. Specifically, we benchmark matrix-based, sequential TT-based, and bidirectional TT (BTT)-based training on the GPU, and BTT-based training on our FPGA accelerator. For each setup, we report the total system-level power consumption, measured during full training execution.

1) Memory Cost: Fig. 15 shows the memory cost of the RTX 3090 GPU and our FPGA accelerator on the ATIS dataset with different model configurations. For GPU memory, we report two metrics: total memory (extracted from the nvidia-smi command) and reserved memory (extracted with the CUDA memory_reserved functions). The former measures the memory cost in the entire training process, encompassing model allocations as well as framework-level overheads. In contrast, the reserved memory reflects only the portion occupied by model parameters, activations, and optimizer states. The computing memory of FPGA is the on-chip memory consumption collected from the RTL implementation report. Unlike GPUs, FPGAs allow for explicit allocation of BRAM and URAM, offering deterministic and fragmentation-free memory usage.

Benefiting from the hardware and algorithm co-design for tensor-compressed training, our FPGA accelerator can perform on-device training with $20.7\times$ to $42.2\times$ lower memory cost compared to tensor-compressed training on GPU. When excluding the framework-level overhead on the GPU, our BTT algorithm achieves a $2.3\times$ to $4.2\times$ memory reduction on

TABLE IV
RESOURCE UTILIZATION AND POWER CONSUMPTION WITH VARIOUS MODEL STRUCTURE SETTINGS AND INPUT WORKLOAD.

Model Type	DSP	LUT	FF	BRAM	URAM	Power (W)		
						Dynamic	Static	Total
2-ENC	2396 (40%)	565k (65%)	475k (27%)	1216 (90%)	114 (18%)	20.68	6.00	26.68
4-ENC	2396 (40%)	572k (66%)	485k (28%)	1163 (86%)	128 (20%)	20.85	5.97	26.82
6-ENC	2396 (40%)	579k (67%)	499k (29%)	1089 (81%)	374 (58%)	20.97	6.09	27.06
Available	5952	872k	1,743k	1344	640	75		

TABLE V
PERFORMANCE COMPARISON BETWEEN GPU AND OUR FPGA ACCELERATOR IN VARIOUS MODEL SETTINGS AND CONTRACTION TYPES.

Model Settings	Platform-Type	Latency per epoch (sec)	Power (W)	Computing Memory		Energy per epoch	
				(MB)	**Ratio** (×)	(kJ)	**Ratio** (×)
L2-S32-FP32	GPU-Matrix	47	150	829	**48.2**	7.1	**1.38**
	GPU-TT	144	140	726	**42.2**	20.2	**3.96**
	GPU-BTT	129	138	721	**41.9**	17.8	**3.49**
	FPGA-BTT (ours)	191	26.68	17.2	**1.0**	5.1	**1.0**
L4-S32-FP32	GPU-Matrix	77	150	915	**51.4**	11.6	**1.29**
	GPU-TT	243	138	720	**40.4**	33.5	**3.73**
	GPU-BTT	222	138	718	**40.3**	30.6	**3.41**
	FPGA-BTT (ours)	335	26.82	17.8	**1.0**	9.0	**1.0**
L6-S32-FP32	GPU-Matrix	108	152	1022	**29.6**	16.4	**1.26**
	GPU-TT	347	138	716	**20.8**	47.9	**3.67**
	GPU-BTT	324	138	713	**20.7**	44.7	**3.43**
	FPGA-BTT (ours)	482	27.06	34.5	**1.0**	13.0	**1.0**

the GPU compared to a standard matrix-based Transformer. Furthermore, our fine-grained on-chip memory management offers an additional $1.5\times$ to $2.7\times$ memory reduction relative to the GPU's reserved memory footprint.

2) Energy Cost: Table V further summarizes the latency per epoch, power consumption, and energy per training epoch of our FPGA accelerator collected from HLS synthesis and RTL implementation reports, compared to both uncompressed training and tensor-compressed training on the GPU. On GPU, BTT achieves lower energy consumption and reduced compute memory usage compared to sequential TT, but the improvements remain relatively modest. Due to the huge difference in clock frequency, our FPGA accelerator has a higher training latency. Nevertheless, it achieves over $3.6\times$ and $3.4\times$ lower energy consumption than TT and BTT training on GPU, respectively. We also compare against standard matrix-based GPU training, which benefits from optimized hardware support for matrix operations. While this matrix-format GPU training is more energy-efficient than tensor-compressed GPU training, our FPGA accelerator still achieves approximately $1.3\times$ lower energy consumption.

3) Comparison with Previous Work: Some previous works [57]–[59] achieved higher energy efficiency improvement compared with GPU. However, these works are all CNN training accelerators supporting small models, and their GPU baseline is less powerful than ours. For instance, Ref. [57] using Xilinx ZU19EG FPGA achieved $8.5\times$ improvement in energy efficiency in LeNet-10 compared to the GTX 1080 Ti GPU. However, LeNet-10 has $129\times$ to $328\times$ fewer model parameters and requires $22\times$ to $61\times$ lower total computing costs than our transformer training. Similarly, the work in [59] using Intel Stratix-10 MX FPGA achieved $4.5\times$ higher energy efficiency than Tesla V100 in training the ResNet-20 model. Our transformer model is $41\times$ to $106\times$ larger than ResNet-20, and our computing cost is $6.9\times$ to $18.7\times$ higher. Compared with these results, our accelerator can achieve higher energy efficiency than the more powerful RTX 3090 GPU, and can train much larger models with limited memory.

VII. CONCLUSION

In this work, we have developed the first proof-of-concept end-to-end tensor-compressed transformer training accelerator on FPGA. We have identified a key limitation of the prior tensor network accelerators as the low-parallel computing flow and proposed a bidirectional flow with both lower computational costs and memory costs. We have further optimized dataflow and proposed a BRAM memory management method to achieve better latency, memory, and energy efficiency. The proposed tensorized transformer accelerator has achieved up to $4.0\times$ lower energy costs and $51\times$ lower computing memory costs than the Nvidia RTX 3090 GPU. The model can scale up to 6 encoder layers and has the potential to solve more complex training tasks on FPGA.

ACKNOWLEDGMENTS

This work is co-funded by Intel Strategic Research Sectors (SRS) - Systems Integration SRS & Devices SRS.

REFERENCES

[1] B. Liu, M. Ding, S. Shaham, W. Rahayu, F. Farokhi, and Z. Lin, "When machine learning meets privacy: A survey and outlook," *ACM Computing Surveys (CSUR)*, vol. 54, no. 2, pp. 1–36, 2021.

[2] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "Sok: Security and privacy in machine learning," in *European symposium on security and privacy*, 2018, pp. 399–414.

[3] S. Bansal and C. J. Tomlin, "DeepReach: a deep learning approach to high-dimensional reachability," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2021, pp. 1817–1824.

[4] D. Onken, L. Nurbekyan, X. Li, S. Wu Fung, S. Osher, and L. Ruthotto, "A neural network approach applied to multi-agent optimal control," in *European Control Conference*, 2021.

[5] D. Sun, S. Jha, and C. Fan, "Learning certified control using contraction metric," in *Conference on Robot Learning*. PMLR, 2021, pp. 1519–1539.

[6] S. Lee and S. Nirjon, "Learning in the wild: When, how, and what to learn for on-device dataset adaptation," in *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, 2020, pp. 34–40.

[7] Y. Tang, X. Zhang, P. Zhou, and J. Hu, "EF-train: Enable efficient on-device CNN training on FPGA through data reshaping for online adaptation or personalization," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 5, pp. 1–36, 2022.

[8] C. Guo, B. Lou, X. Liu, D. Boland, P. H. Leong, and C. Zhuo, "BOOST: block minifloat-based on-device CNN training accelerator with transfer learning," in *Intl. Conf. Computer Aided Design*, 2023, pp. 1–9.

[9] J. Lu, C. Ni, and Z. Wang, "ETA: an efficient training accelerator for DNNs based on hardware-algorithm co-optimization," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.

[10] S. Fox, J. Faraone, D. Boland, K. Vissers, and P. H. Leong, "Training deep neural networks in low-precision with high accuracy using FPGAs," in *Intl. Conf. Field-Programmable Technology*, 2019, pp. 1–9.

[11] S. Q. Zhang, B. McDanel, and H. Kung, "FAST: DNN training under variable precision block floating point with stochastic rounding," in *Int. Symp. High-Performance Computer Architecture*, 2022, pp. 846–860.

[12] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[13] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, 2019, p. 2.

[14] A. Dosovitskiy, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[15] A. Kirillov, E. Mintun, N. Ravi, H. Mao, C. Rolland, L. Gustafson, T. Xiao, S. Whitehead, A. C. Berg, W.-Y. Lo *et al.*, "Segment anything," in *Proc. CVPR*, 2023, pp. 4015–4026.

[16] Z. Zhao, X. Ding, and B. A. Prakash, "PINNsFormer: A transformer-based framework for physics-informed neural networks," in *Int. Conf. Learning Representations*, 2024.

[17] W. Yu, J. Freiwald, S. Tewes, F. Huennemeyer, and D. Kolossa, "Federated learning in asr: Not as easy as you think," in *Speech Communication; 14th ITG Conference*. VDE, 2021, pp. 1–5.

[18] J. Jia, J. Mahadeokar, W. Zheng, Y. Shangguan, O. Kalinli, and F. Seide, "Federated domain adaptation for asr with full self-supervision," *arXiv preprint arXiv:2203.15966*, 2022.

[19] J. Tian, C. Fang, H. Wang, and Z. Wang, "BEBERT: efficient and robust binary ensemble BERT," in *Intl. Conf. Acoustics, Speech and Signal Processing*, 2023, pp. 1–5.

[20] Y. Ji, C. Fang, and Z. Wang, "BETA: binarized energy-efficient transformer accelerator at the edge," *arXiv preprint arXiv:2401.11851*, 2024.

[21] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *NIPS*, 2018, pp. 7675–7684.

[22] Z. Yang, S. Choudhary, S. Kunzmann, and Z. Zhang, "Quantization-aware and tensor-compressed training of transformers for natural language understanding," *arXiv preprint arXiv:2306.01076*, 2023.

[23] Y. Yang, J. Zhou, N. Wong, and Z. Zhang, "LoRETTA: low-rank economic tensor-train adaptation for ultra-low-parameter fine-tuning of large language models," in *Proc. North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2024, pp. 3161–3176.

[24] Y. Yang, K. Zhen, E. Banijamal, A. Mouchtaris, and Z. Zhang, "AdaZeta: adaptive zeroth-order tensor-train adaption for memory-efficient large language models fine-tuning," in *Proceedings of the Empirical Methods in Natural Language Processing*, 2024.

[25] C. Fang, A. Zhou, and Z. Wang, "An algorithm–hardware co-optimized framework for accelerating N: M sparse transformers," *IEEE Trans. Very Large Scale Integration Systems*, vol. 30, no. 11, pp. 1573–1586, 2022.

[26] C. Fang, W. Sun, A. Zhou, and Z. Wang, "Efficient N: M sparse DNN training using algorithm, architecture, and dataflow co-design," *IEEE Trans. CAD of Integrated Circuits and Systems*, 2023.

[27] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "MINILM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," *Advances in Neural Information Processing Systems*, vol. 33, pp. 5776–5788, 2020.

[28] Y. Yang, K. Zhen, B. Ganesh, A. Galstyan, G. Huybrechts, M. Müller, J. M. Kübler, R. V. Swaminathan, A. Mouchtaris, S. B. Bodapati *et al.*, "Wanda++: Pruning large language models via regional gradients," in *Sparsity in LLMs (SLLM): Deep Dive into Mixture of Experts, Quantization, Hardware, and Inference*.

[29] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, "Compressing dma engine: Leveraging activation sparsity for training deep neural networks," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 78–91.

[30] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[31] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[32] X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan, and K. Gopalakrishnan, "Ultra-low precision 4-bit training of deep neural networks," *NIPS*, vol. 33, 2020.

[33] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[34] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," in *3rd International Conference on Learning Representations, ICLR 2015-Conference Track Proceedings*, 2015.

[35] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[36] P. Zhen, B. Liu, Y. Cheng, H.-B. Chen, and H. Yu, "Fast video facial expression recognition by deeply tensor-compressed LSTM neural network on mobile device," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 298–300.

[37] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," *Advances in neural information processing systems*, vol. 28, 2015.

[38] G. G. Calvi, A. Moniri, M. Mahfouz, Q. Zhao, and D. P. Mandic, "Compression and interpretability of deep neural networks via Tucker tensor layer," *arXiv:1903.06133*, 2019.

[39] O. Hrinchuk, V. Khrulkov, L. Mirvakhabova, E. Orlova, and I. Oseledets, "Tensorized embedding layers," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 4847–4860.

[40] X. Ma, P. Zhang, S. Zhang, N. Duan, Y. Hou, M. Zhou, and D. Song, "A tensorized transformer for language modeling," *Advances in neural information processing systems*, vol. 32, 2019.

[41] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, "TIE: energy-efficient tensor train-based inference engine for deep neural network," in *Proc. Int. Symp. Computer Architecture*, 2019, pp. 264–278.

[42] Y. Gong, M. Yin, L. Huang, J. Xiao, Y. Sui, C. Deng, and B. Yuan, "ETTE: Efficient tensor-train-based computing engine for deep neural networks," in *Proc. Intl. Symp. Computer Architecture*, 2023, pp. 1–13.

[43] F.-H. Meng, Y. Wu, Z. Zhang, and W. D. Lu, "TT-CIM: Tensor train decomposition for neural network in rram-based compute-in-memory systems," *IEEE Trans. Circuits and Systems I: Regular Papers*, 2023.

[44] R. Guo, Z. Yue, X. Si, T. Hu, H. Li, L. Tang, Y. Wang, L. Liu, M.-F. Chang, Q. Li *et al.*, "15.4 a 5.99-to-691.1 TOPS/W tensor-train in-memory-computing processor using bit-level-sparsity-based optimization and variable-precision quantization," in *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 64, 2021, pp. 242–244.

[45] Y. Zhao, X. Yu, Z. Chen, Z. Liu, S. Liu, and Z. Zhang, "Tensor-compressed back-propagation-free training for (physics-informed) neural networks," *arXiv preprint arXiv:2308.09858*, 2023.

[46] Y. Zhao, X. Xiao, G. Kurczveil, R. G. Beausoleil, and Z. Zhang, "Tensorized optical multimodal fusion network," in *2023 Conference on Lasers and Electro-Optics (CLEO)*. IEEE, 2023, pp. 1–2.

[47] Y. Zhao, X. Xiao, X. Yu, Z. Liu, Z. Chen, G. Kurczveil, R. G. Beausoleil, and Z. Zhang, "Real-time fj/mac pde solvers via tensorized, back-propagation-free optical pinn training," in *Machine Learning with New Compute Paradigms*.

[48] C. T. Hemphill, J. J. Godfrey, and G. R. Doddington, "The ATIS spoken language systems pilot corpus," in *Workship on Speech and Natural Language*, 1990.

[49] I. V. Oseledets, "Tensor-train decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, 2011.

[50] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *International Conference on Machine Learning*. PMLR, 2017, pp. 3891–3900.

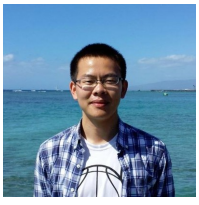
[51] J. Ye, L. Wang, G. Li, D. Chen, S. Zhe, X. Chu, and Z. Xu, "Learning compact recurrent neural networks with block-term tensor decomposition," in *Proc. Computer Vision and Pattern Recognition*, 2018, pp. 9378–9387.

[52] C. Hawkins, X. Liu, and Z. 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*, vol. 4, no. 1, pp. 46–71, 2022.
[53] C. Hawkins and Z. Zhang, "Bayesian tensorized neural networks with automatic rank selection," *Neurocomputing*, vol. 453, pp. 172–180, 2021.
[54] D. Wang, G. Zhao, G. Li, L. Deng, and Y. Wu, "Compressing 3D CNNs based on tensor train decomposition," *Neural Networks*, vol. 131, pp. 215–230, 2020.
[55] D. Wang, G. Zhao, H. Chen, Z. Liu, L. Deng, and G. Li, "Nonlinear tensor train format for deep neural network compression," *Neural Networks*, vol. 144, pp. 320–333, 2021.
[56] Z. Yang, Z. Liu, S. Choudhary, X. Xie, C. Gao, S. Kunzmann, and Z. Zhang, "CoMERA: Computing-and memory-efficient training via rank-adaptive tensor optimization," in *Annual Conference on Neural Information Processing Systems*, 2024.
[57] Z. Liu, Y. Dou, J. Jiang, Q. Wang, and P. Chow, "An FPGA-based processor for training convolutional neural networks," in *Intl. Conf. on Field Programmable Technology*. IEEE, 2017, pp. 207–210.
[58] S. K. Venkataramanaiah, Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao, and J.-S. Seo, "Automatic compiler based FPGA accelerator for CNN training," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 166–172.
[59] S. K. Venkataramanaiah, H.-S. Suh, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, and J.-s. Seo, "FPGA-based low-batch training accelerator for modern CNNs featuring high bandwidth memory," in *Proc. International Conference on Computer-Aided Design*, 2020, pp. 1–8.

VIII. Biography Section

Jiayi Tian Jiayi Tian received the B.Eng. degree in VLSI Design & System Integration from Nanjing University, Nanjing, China, in 2023, and is currently a Ph.D. student in computer engineering in University of California, Santa Barbara, advised by Prof. Zheng Zhang. Her current research interests are algorithm & hardware co-design for efficient large language model training and inference.

Jinming Lu Jinming Lu received the B.S. degree in microelectronics from Nankai University, Tianjin, China, in 2018, and the Ph.D. degree in information and communication engineering from Nanjing University, Nanjing, China, in 2023. Currently, he is a post-doc in University of California, Santa Barbara. His current research interests include automatic speech recognition and deep learning, especially its hardware acceleration and compression algorithms.

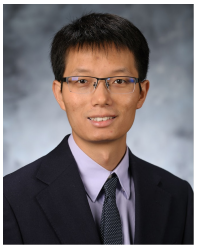


Hai Li Hai Li received the B.S. degree in applied physics from the Huazhong University of Science and Technology, Wuhan, China, in 2011, and the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2015 and 2016, respectively, where he developed foundational theorem of next generation storage system. He joined Intel Corporation in 2016, starting with specialized technologies program. He is currently a Senior Research Scientist with the Exploratory Integrated Circuits, Components Research, Intel Corporation, Hillsboro, OR, USA. He is working on emerging technologies in the joint area of novel device, circuit, and computing paradigm, while managing university collaboration. He serves as Co-Chair of Intel Emerging Technology Strategic Research Sector (SRS).

Xiangwei Wang Xiangwei Wang received the B.Eng. degree in cyberspace security from Huazhong University of Science and Technology, Wuhan, China, in 2023, and currently is a Ph.D. student in the Department of Computer Science at North Carolina State University. He was an exchange student at the University of California, Santa Barbara (2022–2023), supervised by Prof. Zheng Zhang. His current research interests are programming languages, compiler and program optimization, and distributed systems for LLMs.

Cong (Callie) Hao Cong (Callie) Hao (Associate Member, IEEE) received the Ph.D. degree from Waseda University, Tokyo, Japan, in 2017. She was a Postdoctoral Fellow with the Georgia Institute of Technology (Georgia Tech), Atlanta, GA, USA, from 2020 to 2021 and also worked as a Postdoctoral Researcher of ECE with the University of Illinois at Urbana-Champaign, Champaign, IL, USA, from 2018 to 2020. She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, Georgia Tech. Her current research interests lie in the joint area of efficient hardware design and machine learning algorithms, reconfigurable and high-efficiency computing, and electronic design automation tools. Dr. Hao was a recipient of the NSF CAREER Award.

Ian Young Ian. A. Young [M'788, SM96, F'99] is a senior fellow and director of Advanced Circuits and Technology Integration in the Technology and Manufacturing Group at Intel Corporation. He received his Bachelor's and Master's degrees in electrical engineering from the University of Melbourne, Australia, in 1972 and 1975. He received his Ph.D. in electrical engineering from the University of California, Berkeley in 1978. He joined Intel in 1983. Currently, he is responsible for defining future circuit directions with emerging novel devices and identifying leading options to manufacture energy efficient integrated circuits for computing in the beyond-CMOS era. He has authored or coauthored more than 300 technical articles and holds over 250 U.S. patents. Dr. Young has received three Intel Achievement Awards. He was a recipient of the 2009 International Solid-State Circuits Conference's Jack Raper Award for outstanding technology directions paper and the 2018 IEEE Frederik Philips Award "for leadership in research and development on circuits and processes for the evolution of microprocessors."



Zheng Zhang Zheng Zhang [M'15] has been an Associate Professor of Electrical and Computer Engineering with the University of California at Santa Barbara (UCSB), since September 2023. He received his Ph.D in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology (MIT), Cambridge, MA, in 2015, M.Phil from the University of Hong Kong in 2010, and B. Eng from Huazhong University of Science and Technology in 2008. His research interests include uncertainty-aware design automation for electronics, photonics, and quantum circuits; small-data and data-free scientific machine learning for multi-physics design of 3D IC and chiplet, and tensor-compressed methods for sustainable training of large AI models and for resource-constraint on-device learning.