TetriX: Flexible Architecture and Optimal Mapping for Tensorized Neural Network Processing

Jie-Fang Zhang, *Member, IEEE*, Cheng-Hsun Lu, *Graduate Student Member, IEEE*, and Zhengya Zhang, *Senior Member, IEEE*

***Abstract*—The continuous growth of deep neural network model size and complexity hinders the adoption of large models in resource-constrained platforms. Tensor decomposition has been shown effective in reducing the model size by large compression ratios, but the resulting tensorized neural networks (TNNs) require complex and versatile tensor shaping for tensor contraction, causing a low processing efficiency for existing hardware architectures. This work presents TetriX, a co-design of flexible architecture and optimal workload mapping for efficient and flexible TNN processing. TetriX adopts a unified processing architecture to support both inner and outer product. A hybrid mapping scheme is proposed to eliminate complex tensor shaping by alternating between inner and outer product in a sequence of tensor contractions. Finally, a mapping-aware contraction sequence search (MCSS) is proposed to identify the contraction sequence and workload mapping for achieving the optimal latency on TetriX. Remarkably, combining TetriX with MCSS outperforms the single-mode inner-product and outer-product baselines by up to 46.8× in performance across the collected TNN workloads. TetriX is the first work to support all existing tensor decomposition methods. Compared to a TNN accelerator designed for the hierarchical Tucker method, TetriX achieves improvements of 6.5× and 1.1× in inference throughput and efficiency, respectively.**

***Index Terms*—Tensor decomposition, tensorized neural network (TNN), neural network accelerator, tensor contraction sequence search.**

I. Introduction

THE recent advancement in artificial intelligence and machine learning is to a large degree attributed to the development of deep neural network (DNN), and its convolutional neural network (CNN) and recurrent neural network (RNN) variants. To keep improving the accuracy, DNN models grow larger every year, at a rate of 1.5 to 2× increase in model size and model complexity every year [1], [2]. The increasing model size and complexity create large storage and compute requirements for the underlying compute hardware that is becoming out of reach. State-of-the-art models can still be deployed on server-scale or desktop platforms using high-end GPUs and CPUs, but they are practically infeasible for resource-constrained platforms i.e., mobile, edge, and smart sensors, due to area, power, and cost budgets.

Manuscript received 2 May 2023; revised 26 December 2023; accepted 4 January 2024. Date of publication 14 February 2024; date of current version 9 April 2024. This work was supported by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by the Defense Advanced Research Projects Agency (DARPA). Recommended for acceptance by P. Stenstrom. *(Corresponding author: Jie-Fang Zhang.)*

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: jfzhang@umich.edu; lchsun@umich.edu; zhengya@umich.edu).

Digital Object Identifier 10.1109/TC.2024.3365936

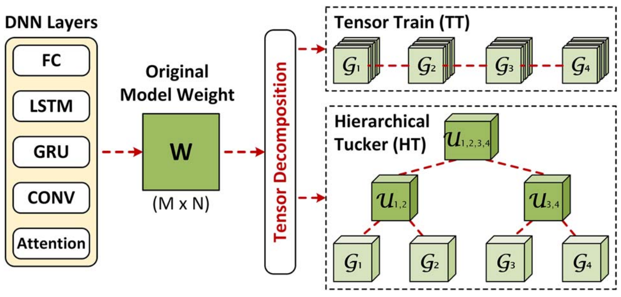


Fig. 1. Illustration of model weight and tensor decomposition for TNNs.

To meet these demanding requirements, many researches have been done to reduce the model size and complexity without degrading the accuracy. The popular compression methods through network pruning [3], e.g., unstructured pruning, offers good compression of a network model but the sparse data formats, e.g., compressed sparse row (CSR), often result in irregular computation and memory access, leading to lower hardware utilization. Alternatively, low-rank approximation methods, e.g., singular-value decomposition (SVD) [4] and matrix factorization [5], approximate the model's weight matrix using low-rank representations. These methods produce regular data structures, but they often struggle to reach a good balance between compression and accuracy.

Recently, tensor decomposition [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], a high-order generalization of the low-rank methods, has gained much progress in network model compression by demonstrating a larger compression than the 2D methods [4], [5] while maintaining the accuracy [26]. Fig. 1 illustrates the tensor decomposition of different DNN layer's weights. For example, a hierarchical Tucker (HT)-based long short term memory (LSTM) can achieve a 6,700× compression in parameter size while improving the accuracy by 17.8% in video action classification [19]; and tensor train (TT) was used to compress the fully connected (FC) layers in VGG-16 by 37,732× (7.4× overall network compression), losing only 1.1%

TABLE I
DECOMPOSITION OF NETWORK LAYERS

Layer Type	Decomposition Method	References
FC/MLP	Tucker/TT/TR/HT/BT	[6], [7], [8], [9], [10], [11], [12]
CONV	TT/HT/BT	[8], [9], [11]
GRU/LSTM	TT/TR/HT/BT	[9], [11], [13], [14], [15], [16], [17], [18], [19], [20]
Attention	TT/BT	[21], [22]
Embedding	TT	[23], [24]

in accuracy in image classification [6]. New tensor decomposition methods continue to emerge to improve the compression while minimizing any accuracy loss [25]. Note that the model compression ratio and the accuracy are solely attributed to the tensor decomposition method, the parameter selection, and the training or retraining approaches. They are unrelated to the compute hardware that performs the inference, except for the numerical quantization adopted by the hardware.

A network compressed by tensor decomposition is named a tensorized neural network (TNN). It is often the case that tensor decomposition is applied to certain layers of a model. As a result, a network can consist of a combination of uncompressed layers and "TNN layers" (or "tensorized layers"). Table I summarizes the layer types and the decomposition methods that have been used in existing TNN works.

The combination of high compression ratios and the ability to retain structured data formats have made TNNs particularly attractive for the mobile and edge inference applications. However, performing inference over TNNs requires high-order tensor contraction operations. Compared to a traditional vector or matrix multiplication, a tensor contraction requires additional **tensor shaping** steps, such as **arbitrary tensor reshape, permute, and transpose, in order to map a tensor into a 2D representation in memory for matrix multiplication**. These tensor shaping steps require additional memory operations, such as read-permute-write, read-transpose-write, which can reduce the computational efficiency on general-purpose processors. For custom accelerators, optimizing tensor shaping is possible by designing memory access patterns that align with the datapath and enable efficient processing through coalescing memory access and computation. However, existing DNN/CNN accelerators [27], [28] are optimized for fixed tensor orders (2 for DNN, 4 for CNN) and lack support for flexible TNN workloads with arbitrary tensor orders. On the other hand, existing TNN accelerators [29], [30], [31] are designed for specific tensor decomposition methods using fixed tensor shaping patterns, e.g., TT for TIE [29] and HT for FDHT [30]. This limitation restricts their ability to support a wider range of TNN workloads, including the newly proposed decomposition methods.

In this work, we present TetriX, a co-design that combines a flexible accelerator architecture with optimal workload mapping to enhance the efficiency of TNN inference. The TetriX architecture employs a configurable dataflow that supports both inner and outer product computations, along with index translation and output gathering mechanisms for efficient tensor shaping in arbitrary contraction sequences. TetriX's flexibility enables a hybrid mapping scheme that seamlessly switches between inner and outer product mappings, simplifying TNN processing by completely eliminating complex tensor shaping. To identify the optimal contraction sequence with minimum latency and memory usage on TetriX, we introduce mapping-aware contraction sequence search (MCSS), based on the hybrid mapping scheme. Unlike existing works that are limited to a single tensor decomposition method, TetriX is the first design to support all decomposition methods. In our evaluation, we observed that TetriX with MCSS consistently outperforms baseline accelerator designs that rely on fixed mapping and contraction schemes across diverse workloads for all decomposition methods. Compared to TIE and FDHT, TetriX demonstrates performance improvements of up to 5.2× for TT-TNNs and up to 31.5× for HT-TNNs, respectively.

II. BACKGROUND

In general, the DNN computation, i.e., FC, LSTM, gated-recurrent unit (GRU), and self-attention layers, can be formulated as a vector-matrix multiplication between an input vector $\boldsymbol{x} \in \mathbb{R}^N$ and a weight matrix $\boldsymbol{W} \in \mathbb{R}^{M \times N}$ to obtain an output vector $\boldsymbol{y} \in \mathbb{R}^M$ using $\boldsymbol{y}_{[j]} = \sum_{i=1}^{N} \boldsymbol{W}_{[j,i]} \cdot \boldsymbol{x}_{[i]}$. Here, we adopt a notation convention that represents vectors, matrices, and tensors (order ≥ 3) using boldface lowercase letters ($\boldsymbol{v}$), boldface capital letters ($\boldsymbol{M}$), and boldface script letters ($\boldsymbol{\mathcal{T}}$), respectively [32].

In a TNN, a large weight matrix is decomposed into a series of small and high-order tensors using tensor decomposition methods. For tensor decomposition, the 2D weight matrix $\boldsymbol{W}$ is first tensorized into either $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{(m_1 \times n_1) \times \cdots \times (m_d \times n_d)}$ or $\boldsymbol{\mathcal{W}} \in \mathbb{R}^{m_1 \times \cdots \times m_d \times n_1 \times \cdots \times n_d}$ (where $M = \prod_{k=1}^{d} m_k$, $N = \prod_{k=1}^{d} n_k$) depending on the tensor decomposition method. Similarly, input vector $\boldsymbol{x}$ and output vector $\boldsymbol{y}$ must be tensorized into input tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and output tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$, respectively, and d is the order of input and output tensors. The computation of a TNN layer can be represented by a tensor contraction as in Eq. (1).

$$\boldsymbol{\mathcal{Y}}_{[j_1,\ldots,j_d]} = \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} \boldsymbol{\mathcal{W}}_{[j_1,i_1,\ldots,j_d,i_d]} \cdot \boldsymbol{\mathcal{X}}_{[i_1,\ldots,i_d]} \quad (1)$$

A. Tensor Decomposition Methods

Research works have demonstrated the effectiveness of several tensor decomposition methods in compressing DNNs, CNNs, and RNNs for diverse applications such as video classification, natural language understanding, image classification, and large-scale recommendation model [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], [26]. These tensor decomposition methods vary in terms of their type, number, order, and topology of the decomposed tensors, resulting in different compression ratios and model accuracies. Here, we provide a brief overview of the commonly used tensor decomposition methods. To ensure simplicity and consistency, we modify the naming and notation conventions from existing literature.

TABLE II
COMPARISON OF TENSOR DECOMPOSITION METHODS

Method	Network Topology	Number of Nodes	Number of Parameters
Matrix	-	1	MN
TT	Chain	d	$O(dmnr^2)$
HT	Binary Tree	$2d-1$	$O(dr(mn+r^2))$
TR	Ring	d	$O(dr^2(n+m))$
BT	Tree	$S(d+1)$	$O(S(dmnr+r^d))$

Tensor Train (TT): The TT decomposition method decomposes a weight tensor $\boldsymbol{\mathcal{W}}$ into d4th-order tensors $\boldsymbol{\mathcal{G}}^{(k)} \in \mathbb{R}^{r_{k-1}\times m_k\times n_k\times r_k}$, where $k\in[1,d]$, forming a chain structure [33]. Here, $\boldsymbol{\mathcal{G}}^{(k)}$ and r_k represent the k-th core tensor and its rank, respectively, and $r_0=r_d=1$ by definition.

Hierarchical Tucker (HT): The HT decomposition method hierarchically decomposes a weight tensor $\boldsymbol{\mathcal{W}}$ into d 3rd-order tensors $\boldsymbol{\mathcal{G}}^{(k)}\in\mathbb{R}^{r_k\times m_k\times n_k}$, for $k\in[1,d]$, and $d-1$ 3rd-order transfer tensors $\boldsymbol{\mathcal{U}}^{(s)}\in\mathbb{R}^{r_s\times r_{s_1}\times r_{s_2}}$. This decomposition results in a binary tree structure [34]. Here, $\boldsymbol{\mathcal{G}}^{(k)}$ and $\boldsymbol{\mathcal{U}}^{(s)}$ represent the leaf and non-leaf nodes of the binary tree, respectively.

Tensor Ring (TR): The TR decomposition method is a variation of the TT decomposition method. It creates a ring structure by connecting the first and last tensors using $r_0=r_d=R$, where $R\geq 1$, to enhance its expressiveness [35]. In the TR decomposition, a d-th-order weight tensor $\boldsymbol{\mathcal{W}}$ is decomposed into d_n 3rd-order tensors $\boldsymbol{\mathcal{G}}^{(k_n)}\in\mathbb{R}^{r_{k_n-1}\times n_k\times r_{k_n}}$, for $k_n\in[1,d_n]$, as well as d_m 3rd-order tensors $\boldsymbol{\mathcal{G}}^{(k_m)}\in\mathbb{R}^{r_{k_m-1}\times m_k\times r_{k_m}}$, for $k_m\in[d_n+1,d]$. Here, $d=d_n+d_m$.

Block Term (BT): The BT decomposition method combines the key features of the canonical polyadic (CP) decomposition and the Tucker decomposition [32], [36]. In a BT-TNN with a CP-rank S, the d-th-order weight tensor $\boldsymbol{\mathcal{W}}$ is decomposed into S Tucker-formats. Each Tucker-format contains a transfer tensor $\boldsymbol{\mathcal{U}}^{(s)}\in\mathbb{R}^{r_1\times\cdots\times r_d}$ and d 3rd-order tensors $\boldsymbol{\mathcal{G}}^{(s,k)}\in\mathbb{R}^{r_k\times m_k\times n_k}$, where $s\in[1,S], k\in[1,d]$.

In general, TT is considered the simplest decomposition method because of its simple and regular structure, and it can be supported by most of the existing TNN accelerators [29], [31]. HT, TR, and BT are more complex decomposition methods that rely on complex structures to achieve a higher compression ratio or a better accuracy. Besides the methods mentioned above, novel and improved decomposition methods, e.g., KCP [25], have been proposed. Generally speaking, the decomposed weight tensors have relatively small parameter sizes compared to the original weight matrix. Table II summarizes the key properties of TT, HT, TR, and BT decomposition methods, and the number of parameters they require, where $r=\max(r_k), m=\max(m_k)$, and $n=\max(n_k)$. The rank hyperparameters used in decomposition methods, namely r, S, are key factors that determine the compression ratio and model accuracy after decomposition.

B. Tensor Network Graph

A tensor network graph can be used to describe a TNN inference. The input tensor is represented by an input node and the decomposed weight tensors are represented by weight nodes

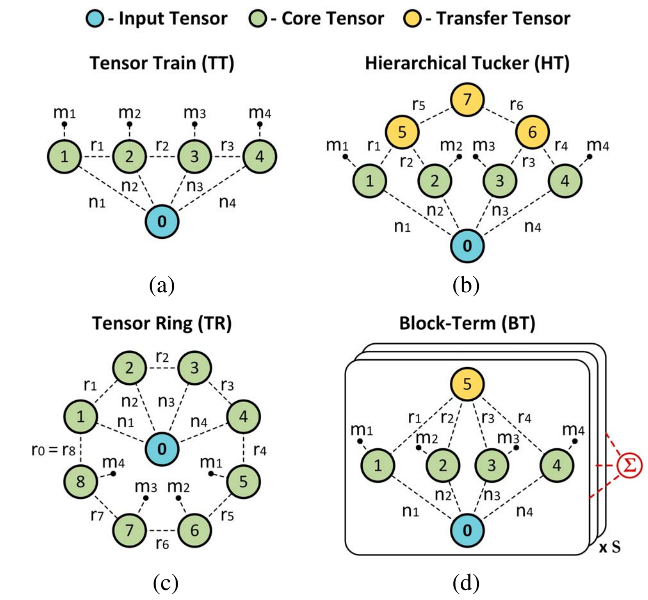


Fig. 2. Tensor network graph representation of TNN inference. The four methods depicted are: (a) TT, (b) HT, (c) TR, and (d) BT. In each representation, the input tensor $\in\mathbb{R}^{n_1\times n_2\times n_3\times n_4}$ and the output tensor $\in\mathbb{R}^{m_1\times m_2\times m_3\times m_4}$. r_i and S are the rank hyperparameters.

connected following the topology defined by the decomposition method. In a tensor network graph, every tensor dimension (greater than 1) is represented by an edge that is connected to a node. Two nodes are connected by an edge if they have one or more shared dimensions that can be contracted via a tensor contraction. A loose edge from a node that is not attached to other nodes represents a free dimension which cannot be contracted with any other node. Fig. 2 illustrates tensor network graphs for TT, HT, TR, and BT methods.

A TNN inference is done via a sequence of tensor contractions. When contracting two tensors, the two nodes are merged into a single node that inherits all edges from both sides except the edge connecting them. After all contractions are done, the final node represents the output tensor and the loose edges represent the output dimensions. Fig. 3 walks through an example of inference of a TT layer with 3 weight nodes. First, the input node (0) is contracted with the weight node (3) to form an intermediate node (03) that inherits all loose and connected edges. Then, (03) is contracted with weight node (2) to form (023). Lastly, (023) is contracted with weight node (1) to form the output node with the output dimensions m_1, m_2, and m_3.

C. Tensor Contraction and Tensor Shaping

In practice, a tensor contraction is converted into a matrix-matrix multiplication (MMM) for processing on hardware [32]. For example, given $\boldsymbol{\mathcal{A}}\in\mathbb{R}^{n_1\times n_2\times n_3}$ and $\boldsymbol{\mathcal{B}}\in\mathbb{R}^{m_1\times m_2\times m_3}$ and $n_2=m_3$, the tensor contraction between $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ is performed in three steps: 1) tensor shaping is first applied to $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ to obtain matrix $\boldsymbol{A}\in\mathbb{R}^{(n_1\times n_3)\times n_2}$ and matrix $\boldsymbol{B}\in\mathbb{R}^{m_3\times(m_1\times m_2)}$, respectively; 2) the MMM between $\boldsymbol{A}$ and $\boldsymbol{B}$ is performed to produce matrix $\boldsymbol{C}\in\mathbb{R}^{(n_1\times n_3)\times(m_1\times m_2)}$; and 3) another tensor shaping is applied to $\boldsymbol{C}$ to put the output tensor in the designated format, e.g., $\boldsymbol{\mathcal{C}}\in\mathbb{R}^{n_1\times m_1\times m_2\times n_3}$.

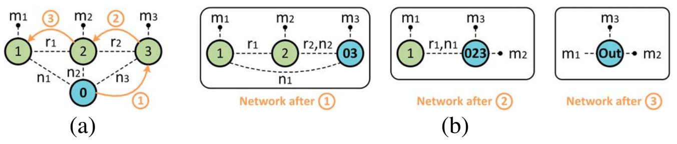


Fig. 3. (a) Illustration of a contraction sequence for a TT workload, and (b) the step-by-step example of tensor contractions.

Tensor shaping includes tensor reshape, permute, and transpose operations. **Reshape** merges multiple contiguous dimensions into one dimension or splits one dimension into multiple contiguous dimensions, e.g., tensor $\boldsymbol{\mathcal{A}}$ from above can be reshaped by merging the n_1 and n_2 dimensions into one dimension to make $\boldsymbol{\mathcal{A}}' \in \mathbb{R}^{(n_1 n_2)\times n_3}$. **Permute** changes the locations of dimensions except for the last dimension, e.g., $\boldsymbol{\mathcal{A}}$ can be permuted by swapping n_1 and n_2 locations to make $\boldsymbol{\mathcal{A}}' \in \mathbb{R}^{n_2\times n_1\times n_3}$. **Transpose** swaps the location of the last dimension with another dimension, e.g., $\boldsymbol{\mathcal{A}}$ can be transposed by swapping n_1 and n_3 locations to make $\boldsymbol{\mathcal{A}}' \in \mathbb{R}^{n_3\times n_2\times n_1}$. As a convention, the last (the inner-most) dimension is stored as one word in memory. Since reshape and permute do not touch the last dimension, they can operate on memory words, whereas transpose touches the last dimension, and thus requires operating on the individual elements in a memory word. The word-level reshape and permute can be coalesced with computation to hide the latency. The sub-word-level transpose costs additional hardware and latency and is more expensive than reshape and permute. Tensor shaping that involves transpose is referred to as **complex tensor shaping** and should be avoided.

The tensor shaping operations needed for a contraction depends on the location of the contracted dimensions in the input tensors (e.g., n_2, m_3 in the above example) and the required dimension ordering of the output tensor (e.g., $n_1 \times m_1 \times m_2 \times n_3$ in the above example). Since the number of dimensions of the input tensors, the locations of contracted dimensions, and the required dimension ordering of the output tensor can all vary, numerous combinations of reshape, permute, and transpose are needed to support versatile TNN inference, making it a challenging hardware design.

D. TNN Inference Accelerator Design Goals

A TNN inference is computed via a sequence of tensor contractions which are broken down into tensor shaping and MMM operations. General-purpose SIMD processors can perform MMMs efficiently. However, when it comes to tensor shaping, additional memory operations are required. Specifically, read-shaping-write operations are needed to organize data from different levels of memory hierarchy in memory buffers, ensuring the data is structured into specific layouts before MMMs can be executed. The memory operations render a SIMD architecture less efficient than a domain-specific accelerator that can coalesce tensor shaping and computation by optimizing the memory access with the datapath design.

Recent TNN inference accelerators [29], [30], [31] were designed for one specific tensor decomposition method and uses one specific tensor contraction sequence, e.g., backward processing of TT-TNN by TIE [29] and customized flow for HT-TNN by FDHT [30]. It is either impractical or highly inefficient to apply them to other tensor decomposition methods that they are not designed for. To make a true a domain-specific TNN inference accelerator, two major requirements need to be met: 1) support of all tensor decomposition methods, and 2) support of optimal contraction sequences.

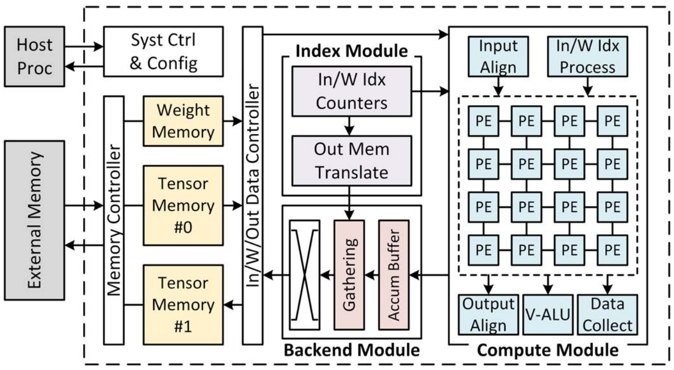


Fig. 4. TetriX system architecture.

In this work, we design a domain-specific TNN inference accelerator called TetriX that provides flexible tensor shaping to enable efficient support of all tensor decomposition methods using any contraction sequence. We also present a methodology that selects the optimal contraction sequence to minimize the latency of processing a given TNN inference by taking advantage of TetriX's unique hybrid mapping scheme.

III. TetriX Architecture and Dataflow

A tensor is stored in a 2D format in memory where memory words are accessed for MMMs. The MMM can be performed via an inner product (IP) or outer product (OP) based on the tensor layout in the memory. Existing accelerators are designed for either IP or OP and use a specific tensor layout in memory. However, due to the diverse TNN workloads and versatile tensor shaping, the resulting tensors may be stored in different memory layouts, causing challenges and overheads to existing designs. To address this, TetriX adopts a unified architecture that supports both IP and OP.

A. Unified IP and OP Architecture

The TetriX system architecture is presented in Fig. 4. TetriX consists of a control module, a compute module, an index module, and a backend module. In the control module, a system controller receives instructions and configurations from the host processor, and coordinates the other modules. The compute module is configurable to perform MMMs using both IP and OP with weight stationary (WS) and output stationary (OS) dataflows, respectively. A PE array is used for the core computation. Pre-processing and post-processing units, i.e., input/output align, index processing, data collect, are used to support input streaming and output collection.

A weight memory stores all the decomposed weight tensors for a TNN layer and two tensor memories are used alternately

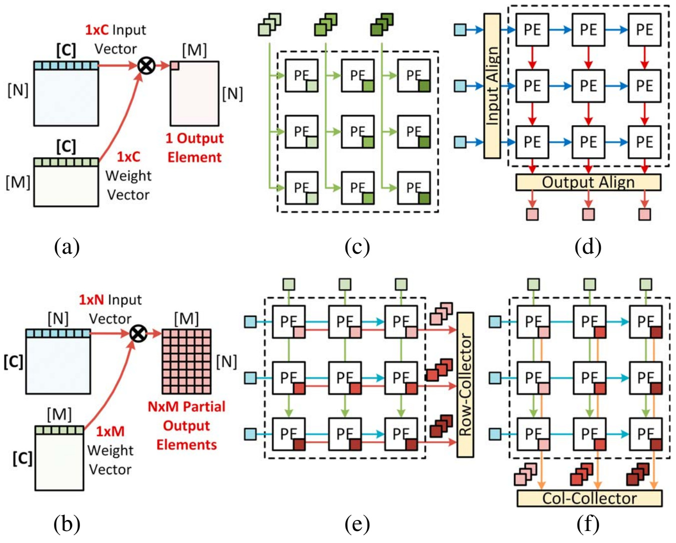


Fig. 5. (a) Illustration of IP, (b) illustration of OP, (c) input streaming in the WS data flow, (d) output collection in the WS dataflow, and (e) row collection in the OS dataflow, (f) column collection in the OS dataflow.

to store the input and output tensor. The memories are multi-banked to provide the flexibility to hold various input and output tensor shapes. The index module performs index translation to convert high dimensional input and weight tensor indices into 2D output memory addresses. The backend module accumulates the partial sums (psums), performs output gathering to form output data words, and coordinates the writeback of data words into the output memory bank.

B. *TetriX Mapping and Dataflow*

Fig. 5(a) and 5(b) illustrate the IP and OP for MMM between an input and weight tensor. The two mappings mainly differ in the data layout in memory and the accumulation pattern. TetriX is designed to support both IP and OP using WS and OS dataflows, respectively, on its unified PE array. The steps of IP and OP in TetriX are illustrated in Fig. 5(c)–5(f).

IP Mapping: this mapping follows the data layout in memory shown in Fig. 5(a) to allow vectors across the C dimension (the dimension to be contracted) to be fetched for computation. The IP between a pair of input and weight vectors results in an output element.

TetriX adopts the WS dataflow for IP where the compute module acts as a systolic array [37]. The weight vectors are first sent to the PE array columns and cached locally in each PE for reuse (Fig. 5(c)). The input vectors are then streamed in to the PE rows from left to right, and propagate in a systolic fashion. The outputs are collected from the bottom PE array row and aligned to form output vectors (Fig. 5(d)).

OP Mapping: this mapping follows the data layout in memory shown in Fig. 5(b) to allow vectors across the non-contractable dimensions, N and M, to be fetched for processing. The OP between a pair of input and weight vectors results in $N \times M$ output psums.

For OP, TetriX adopts the OS dataflow [38] where the compute module acts as a spatial array [29]. The input and weight vectors are broadcast across the PE array horizontally and vertically, respectively, and the psums are accumulated temporally in the PEs. Once the accumulation completes, the outputs are stored in an output data FIFO for output collection.

We propose two output collection modes: 1) row collection and 2) column (col) collection. In row collection, a row collector arbitrates between rows of PEs that are ready for collection, and reads the output data and memory addresses from the granted row (Fig. 5(e)). Col collection works in similar way but on columns (Fig. 5(f)). The two output collection modes provide more flexibility and tensor shaping options.

The unified IP and OP architecture allows TetriX to perform MMMs for different tensor layouts in memory. The WS and OS dataflows employ different schemes for data propagation and psum accumulation, leading to variations in PE utilization, memory traffic, and control overhead. For instance, for an MMM of $(256 \times 64) \times (64 \times 50)$ (input matrix $\times$ weight matrix), the IP mapping achieves 100% PE utilization, while OP mapping reaches only 78%. Conversely, for an MMM of $(256 \times 50) \times (50 \times 64)$, the PE utilization is 78% for IP mapping and 100% for OP mapping. By utilizing pre-determined tensor layouts in a TNN inference, TetriX can support the mapping scheme that delivers the best performance and efficiency.

IV. HYBRID INNER-OUTER MAPPING

A TNN inference is performed through a series of tensor contraction steps referred to as a contraction sequence. After a contraction step, tensor shaping needs to be done to convert the memory layout of the output tensor of one step to make it compatible with IP or OP for the next contraction step. In conventional hardware that offers only IP or OP, the required tensor shaping can be expensive. We take advantage of TetriX's flexibility using a hybrid mapping scheme that alternates between IP and OP as needed to completely eliminate the need for transpose, the most expensive tensor shaping operation.

A. *Tensor Layout Notations*

We introduce tensor layout notations to capture how tensor dimensions are ordered and how tensors are stored in memory for IP and OP mapping. The notation $[X] \times [Y]$ represents the tensor layout in memory with X spanning the memory's rows, and Y spanning the columns. The dimensions in Y are flattened to be stored in one memory word; and the dimensions in X are flattened to be stored across memory rows. We follow the convention that if X, Y contain multiple dimensions, the rightmost dimension is considered the inner-most dimension for storage. For example, if $Y = \{y_1, y_2, y_3\}$, y_3 is considered the inner-most dimension. The storage ordering is $Y = \{0, 0, 0\}$ first, followed by $Y = \{0, 0, 1\}$, and so on.

I-Layout for IP Mapping: For IP mapping (described in Section III.B), the input and weight need to be in the I-Layout where the contractable dimension, C, spans the memory columns. An example of input and weight tensor layout for IP is listed below.

$$\begin{aligned} \text{Input} &: [i_1, i_2, i_3, i_4] \times [C], \\ \text{Weight} &: [w_1, w_2, w_3, \boldsymbol{w}_4] \times [C], \\ \text{Output} &: \{\Phi(i_1, i_2, i_3, i_4, w_1, w_2, w_3), \boldsymbol{w}_4\}, \end{aligned} \tag{2}$$

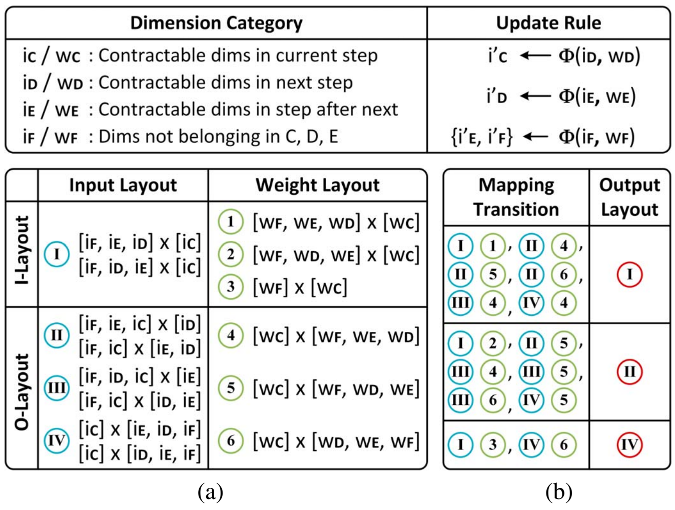
where $i_1, \ldots, i_4$ and C are the input tensor dimensions; $w_1, \ldots, w_4$ and C are the weight tensor dimensions.

Based on the IP mapping described by Fig. 5(c) and 5(d), the C dimension is contracted after the IP, leaving the remaining inner-most dimension of the weight tensor, or $\boldsymbol{w}_4$ in the above example, to be the inner-most dimension of the output tensor. Other than this inner-most dimension, the rest of the dimensions in the output tensor can be reshaped and permuted, as indicated by the notation $\Phi(\cdot)$. The notation $\{.\}$ signifies that the output tensor can be placed in any 2D memory layout $[X] \times [Y]$ as long as the dimension ordering is kept.

O-Layout for OP Mapping: For OP mapping (described in Section III.B), the input and weight need to be in the O-Layout where the contractable dimension, C, spans the memory rows. An example of the tensor layout for OP is listed below.

$$\begin{aligned} \text{Input} &: [i_1, i_2, C] \times [i_3, \boldsymbol{i}_4] \\ \text{Weight} &: [w_1, w_2, C] \times [w_3, \boldsymbol{w}_4] \\ \text{Output-Row} &: \{\Phi(i_1, i_2, i_3, i_4, w_1, w_2, w_3), \boldsymbol{w}_4\} \\ \text{Output-Col} &: \{\Phi(i_1, i_2, i_3, w_1, w_2, w_3, w_4), \boldsymbol{i}_4\} \end{aligned} \tag{3}$$

Based on the OP mapping described by Fig. 5(e) and 5(f), the C dimension is contracted after the OP, leaving the inner-most dimension of in the input tensor or the weight tenor, i.e., $\boldsymbol{i}_4$ or $\boldsymbol{w}_4$ in the above example, to be the inner-most dimension of the output tensor (depending on whether the col or the row collection mode is used). Again, other than the inner-most dimension, the remaining dimensions in the output tensor can be reshaped and permuted, and the output tensor can be placed in any 2D memory layout $[X] \times [Y]$ as long as the dimension ordering is kept.



Dimension Category	Update Rule
i_C / w_C : Contractable dims in current step	$i'_C \longleftarrow \Phi(i_D, w_D)$
i_D / w_D : Contractable dims in next step	$i'_D \longleftarrow \Phi(i_E, w_E)$
i_E / w_E : Contractable dims in step after next	
i_F / w_F : Dims not belonging in C, D, E	$\{i'_E, i'_F\} \longleftarrow \Phi(i_F, w_F)$

	Input Layout	Weight Layout
I-Layout	I $[i_F, i_E, i_D] \times [i_C]$ $[i_F, i_D, i_E] \times [i_C]$	1 $[w_F, w_E, w_D] \times [w_C]$; 2 $[w_F, w_D, w_E] \times [w_C]$; 3 $[w_F] \times [w_C]$
O-Layout	II $[i_F, i_E, i_C] \times [i_D]$ $[i_F, i_C] \times [i_E, i_D]$	4 $[w_C] \times [w_F, w_E, w_D]$
	III $[i_F, i_D, i_C] \times [i_E]$ $[i_F, i_C] \times [i_D, i_E]$	5 $[w_C] \times [w_F, w_D, w_E]$
	IV $[i_C] \times [i_E, i_D, i_F]$ $[i_C] \times [i_D, i_E, i_F]$	6 $[w_C] \times [w_D, w_E, w_F]$

(a)

Mapping Transition	Output Layout
I 1, II 4, II 5, II 6, III 4, IV 4	I
I 2, II 5, III 4, III 5, III 6, IV 5	II
I 3, IV 6	IV

(b)

Fig. 6. (a) Illustration of the possible input and weight layouts, and (b) the mapping transitions of all possible parings and their resulting output layouts.

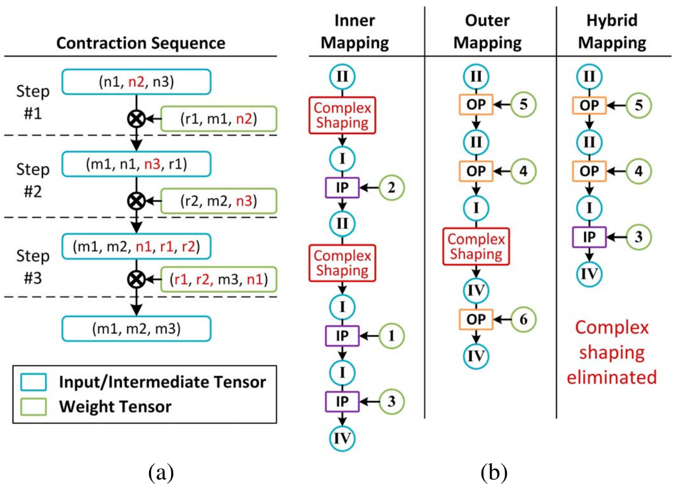


Fig. 7. (a) Illustration of a contraction sequence example with the contractable dimensions in each step highlighted in red, and (b) comparison of hybrid and single-mode mapping.

B. Hybrid Inner-Outer Product Mapping

In a sequence of contraction steps, the output from one step becomes the input to the next step. The tensor layout needs to be transitioned from one step to the next as required by the next step's processing mode (IP or OP). To plan the optimal mapping of a contraction sequence, we need to look at a tensor's contractable dimensions in the current step as well as look ahead at the contractable dimensions in future steps.

To formalize the procedure, we categorize a tensor's dimensions into four sets: C, D, E (represent the set of contractable dimension(s) in the current, the next, and the one after next contraction step, respectively) and F (represents the set of remaining dimensions). Note that we consider the current step and two future steps because the two-step lookahead provides nearly optimal results in the contraction sequence search while keeping a manageable complexity. Using this notation, i_C and w_C represent an input tensor's and a weight tensor's contractable dimensions in the current step; i_D and w_D represent an input tensor's and a weight tensor's contractable dimension in the next step and so on. After each contraction step, the sets are updated.

Using the dimension categorization, we list the input and weight tensor layouts in Fig. 6. The table captures common and meaningful tensor layouts and the list is not meant to be exhaustive. We also list possible pairings of input and weight tensor layouts for contraction and derive the output layouts as shown in Fig. 6(b).

TetriX allows IP and OP to be used as needed to avoid complex tensor shaping. Fig. 7(a) shows an example of a contraction sequence with three contraction steps. The input tensor and the weight tensor at each step are categorized into possible input tensor layouts and weight tensor layouts. The output tensor layouts are then derived based on the transition table to complete the mapping for the step. Fig. 7(b) shows the results of IP-only, OP-only, and IP/OP hybrid mapping. In a single-mode mapping, if an input tensor is not in I-layout for IP or O-Layout for OP, a costly tensor transpose must be performed to change the memory layout. In contrast, the hybrid mapping alternates between IP and OP to completely eliminate expensive transpose operations, and coalesce the remaining tensor shaping operations with the computation.

V. Mapping-Aware Contraction Sequence Search

A tensor network graph with K nodes can have $O(K!)$ possible contraction sequences, and each contraction sequence has distinct computation, memory storage, and tensor shaping requirements. We propose a mapping-aware contraction sequence search (MCSS) that utilizes hybrid mapping and layout transitions to determine the optimal contraction sequence. MCSS maximizes TetriX's compute utilization and minimizes its latency.

A. Baseline Breadth-First Approach

Several approaches [39], [40] were proposed to search the optimal contraction sequence with the minimum computation cost. In this work, we adopt the breadth-first (BF) approach [39] as the baseline. Fig. 8(a) illustrates a BF search on a tensor network graph of 4 weight nodes and an input node. The approach constructs a series of node-merging sets Set_k, from $k = 0$ to $k = 4$. Set_0 contains only the input tensor node 0. Next, the input node is contracted with each contractable weight node in the graph to create merged nodes (01) and (02) to be put in Set_1. Next, the merged nodes in Set_1 are contracted with each contractable weight node to create merged nodes to put in Set_2, and so on.

For each merged node in Set_k, we find all possible splits into a predecessor node (pred) from Set_{k-1} and a contractable weight node to obtain the merged node. For instance, the merged node (012) in Set_2 can have 2 splits: (01|2), and (02|1). The minimum computation cost for the merged node is found as the minimum among all possible splits following

$$\text{cost(merged)} = \min_{\forall \text{ splits}} \left(\text{cost(pred)} + \text{cost}_C(\text{pred}, \text{weight})\right),$$

where $\text{cost}_C(A, B)$ represents the cost of computing the contraction between tensor node A and B. The lowest cost is recorded for every merged node. Once the cost of Set_K is completed, backward tracing is done to find the sequence of the lowest computation cost.

B. Mapping-Aware Breadth-First Approach

We extend the baseline BF approach to incorporate TetriX's hybrid mapping to find the sequence of the minimum computation latency. While the computation cost in prior works [39], [40] is primarily determined based on the MAC count, our approach differs. We consider latency as the cost in our optimization, which encompasses both the MAC count and compute utilization for each contraction during the forward exploration.

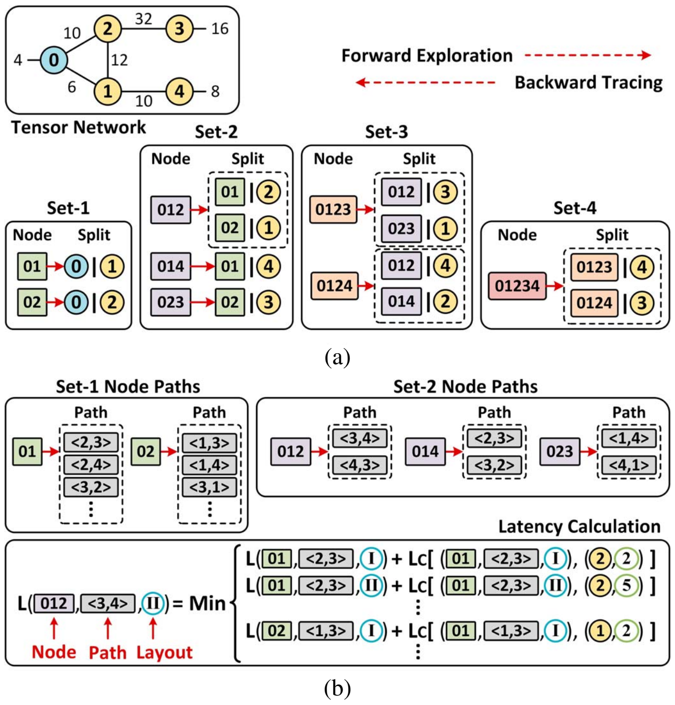


Fig. 8. Illustration of the optimal contraction sequence search on a tensor network example using two approaches: (a) the baseline BF approach, and (b) the mapping-aware BF approach. The mapping-aware BF approach takes into account candidate paths and mapping layout transitions in the latency calculation.

With the proposed mapping-aware BF approach, for each merged node in a set, we look ahead to identify candidate paths for the next two contraction steps in a sequence. Fig. 8(b) shows the paths of the merged nodes in Set_1 and Set_2 of the previous example. For instance, the path $\langle 3, 4\rangle$ of the merged node (012) indicates that the merged node (012) contracts with weight node 3 and 4 in the next two contraction steps. With the path information, we use the layout transition analysis in Fig 6 to identify the IP and OP mapping for each contraction step to estimate the compute utilization and then latency. Similar to Eq. (4), the computation latency of a merged node associated with a path and an input format is calculated from all possible splits, paths, and layout transitions. Fig. 8(b) shows an example of calculating the latency (L) of the merged node (012) on path $\langle 3, 4\rangle$ and input form Ⓘ. After the forward exploration of the minimum latency, backward tracing is done to find the optimal sequence of the shortest latency.

C. Contraction Sequence Analysis

MCSS identifies the optimal sequence considering the hardware cost, including the total number of MAC operations, the MMM PE utilization and latency, and the required memory size to hold all intermediate tensors during inference.

We use an HT-TNN layer example from [18] to demonstrate the advantage of MCSS. The tensor network graph of the optimal contraction sequence (Optimal) from MCSS is compared to the two fixed contraction patterns (Pattern-1 and Pattern-2) proposed by prior TNN works [9], [18] in Fig. 9(a). In

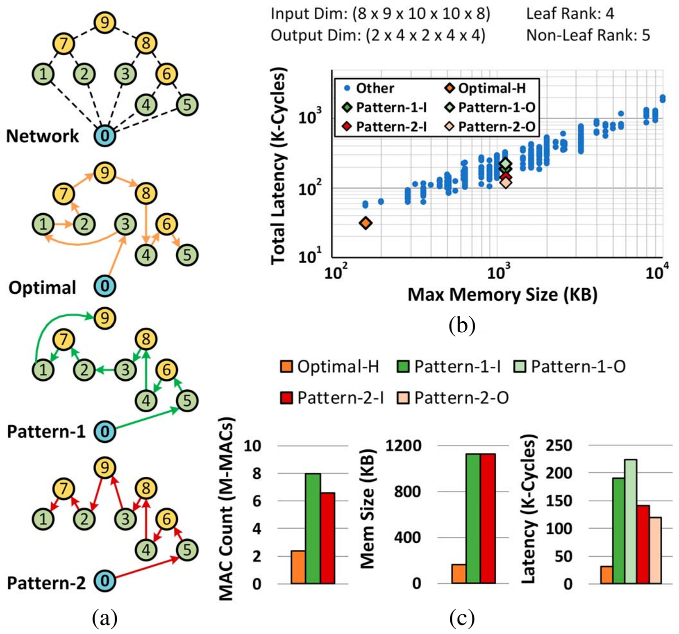


Fig. 9. Illustration of the contraction sequences for an HT-TNN layer inference example: (a) the optimal contraction sequence (optimal) and fixed contraction patterns used in previous works (pattern-1, pattern-2); (b) contraction sequence space is represented in terms of total latency and required memory size; (c) comparison of contraction sequences in total MAC operations, required memory size, and total latency.

Fig. 9(b), we illustrate the quality of the optimal sequence (using hybrid mapping) alongside Pattern-1, Pattern-2 (using IP or OP mapping), and the entire search space in terms of the end-to-end latency and the required memory size to hold every intermediate tensor. The quality metrics of the Optimal, Pattern-1, and Pattern-2 are detailed in Fig. 9(c). The optimal sequence obtained from MCSS requires 2.7× to 3.3× fewer MAC operations, 7× less memory, and 3.8× to 7.2× lower latency compared to the fixed contraction sequences Pattern-1 and Pattern-2.

VI. Flexible Tensor Shaping Support

With the hybrid mapping, TetriX only needs to support reshape and permute operations in tensor shaping, which are generally less expensive than transpose and can be coalesced with the computation. We present two mechanisms in TetriX to support flexible reshape and permute of tensor dimensions: 1) index translation to calculate the output address for every output of the compute module, and 2) output gathering to group contiguous output data into words for efficient writeback.

A. Index Translation

Fig. 10 illustrates TetriX's index translation to convert tensor indices into 2D output memory addresses. Two groups of index counters keep track of the input and weight tensor indices during processing. The output memory translation unit converts the tensor indices of each output data into the corresponding output memory address.

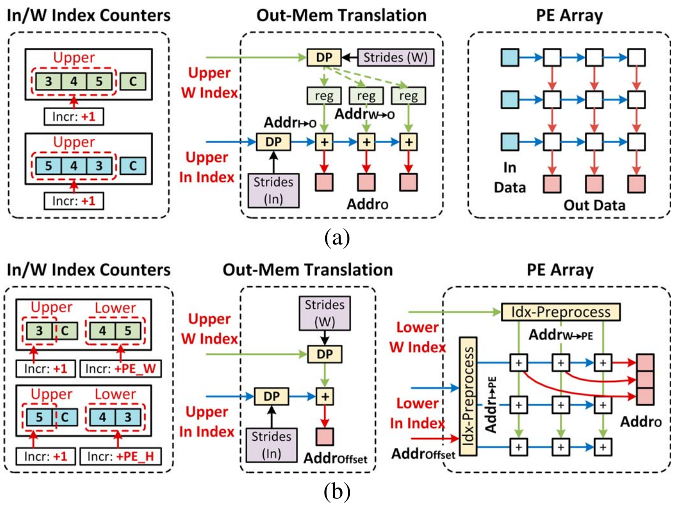


Fig. 10. Index translation mechanism for (a) WS dataflow and (b) OS dataflow.

In the IP dataflow, the output translation unit receives the upper input and weight tensor indices to calculate the output addresses, as illustrated in Fig. 10(a). First, in loading a weight vector to a PE array column, the upper weight indices are multiplied by the corresponding strides to compute $Addr_{W \to O}$. These are stored for future reuse. Then during processing, as the inputs are streamed in, the upper indices of every input vector are multiplied by the corresponding strides to compute $Addr_{I \to O}$. Subsequently, these values are propagated and summed with $Addr_{W \to O}$ to finally obtain the output addresses $Addr_O = Addr_{I \to O} + Addr_{W \to O}$.

In the OP dataflow, the output addresses are computed in two steps. First, the upper input and weight indices (excluding the contractable dimension C) are sent to the output translation unit to calculate $Addr_{\text{Offset}} = Addr_{I \to O} + Addr_{W \to O}$, as illustrated in Fig. 10(b). Second, the lower input and weight indices, along with $Addr_{\text{Offset}}$ are sent to the pre-processing units in the PE array to calculate the partial output addresses, namely $Addr_{I \to PE}$ and $Addr_{W \to PE}$. These partial addresses are broadcast across the array, allowing each PE to sum the partial output addresses to obtain the final output addresses, $Addr_O = Addr_{I \to PE} + Addr_{W \to PE}$.

B. Output Gathering

After index translation, the output data can be written back into the output memory. For an efficient writeback and memory usage, output data of contiguous memory addresses are grouped into memory words before being sent to the output memory bank, which we refer to as output gathering. Due to the versatile tensor dimensions across TNN workloads, the output gathering needs to flexibly adapt to all possible cases.

We propose a hierarchical approach for output gathering as illustrated in Fig. 11. The hierarchical approach performs gathering over multiple stages. Take as an example a 2-stage output gathering shown in Fig. 11(a), 8 data elements A to H (with associated (row, col) indices) are gathered in two stages.

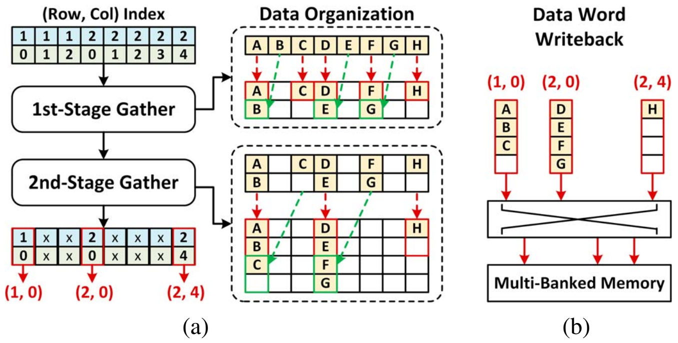


Fig. 11. Illustration of the hierarchical output gathering mechanism using (a) two gathering stages followed by (b) a switching stage.

In the first stage, we gather data into groups of up to 2: A and B are grouped together because they belong to the same row and take consecutive col addresses, and similarly D and E, F and G are grouped together. In the second stage, we gather data into groups of up to 4: A, B, C are grouped together, and similarly D, E, F, G. The gathered memory word and address pairs are written back to memory as shown in Fig. 11(b).

The index translation followed by output gathering mechanism provides the needed flexibility in TetriX. The mechanism also allows TetriX to operate on wider memory words compared to previous works [29], leading to more efficient memory accesses. The design is scalable and implementation friendly, with negligible impact on area and power consumption.

VII. Benchmarking and Evaluation

To assess performance, area, and efficiency, a TetriX prototype is designed in a 28nm CMOS technology. The prototype is employed to evaluate the hybrid mapping scheme against baseline schemes across various TNN workloads. Furthermore, TetriX is compared to state-of-the-art TNN accelerators [29], [30] to demonstrate its advantages in reduced computation cost, better performance, and flexibility.

A. Evaluation Methodology

To evaluate TetriX, we collected the specifications of various TNN workloads from existing TNN literature, including [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [22], [23], [24]. This collection contains more than 100 distinct TNN workloads from TT, HT, TR, BT, and Tucker, with each workload consisting between 3 and 13 tensor contractions. To reflect the diverse forms of model weights obtained from hyperparameter search [41], [42], we augment the collected workloads with synthetic ones. For each collected workload, synthetic workloads are generated by reordering the tensor dimensions of the input and output, and selecting random ranks within a meaningful range. These combined workloads provide a comprehensive setup for evaluating the performance and effectiveness of TetriX and MCSS across different tensor decomposition methods.

To accurately simulate and analyze the performance of TetriX, cycle-accurate models have been developed. For a fair comparison with the hybrid mapping scheme, baseline designs using single-mode mapping schemes, i.e., Base-I and Base-O, are evaluated using MCSS on the same dataset. For benchmarking, cycle-accurate models are implemented for TIE [29] and FDHT [30], which are state-of-the-art inference accelerators for TT and HT.

A TetriX prototype is implemented in RTL and synthesized in a 28nm CMOS technology. The prototype occupies 1.34 mm^2 and contains 256 MACs and 548 KB of on-chip memory. It operates at a 1.0 GHz clock frequency. Similar to prior works [29], [30], the TetriX prototype adopts a 16b fixed-point quantization for both input and weight values. Each tensor index is tracked by an 8b counter that can support dimension sizes from 2 to 255. If a dimension size exceeds 255, the index may be split and counted using spare index counters. The prototype design achieves a peak compute throughput of 512 GOPS (one 16b MAC counts as 2 OPs) and consumes 197.7 mW and 180.0 mW in running the IP and OP dataflows, respectively.

TABLE III
Area and Power Breakdown of TetriX

	Area (mm^2)	Area (%)	Power^a (mW)	Power^a (%)
Compute	0.26	19.6	116	64.4
Index	0.07	5.4	6.8	3.8
Backend	0.03	2.6	11	6.1
Memory	0.97	72.5	46.2	25.7
Total	1.34	100	180	100

[a]Power based on the OP mapping.

Table III presents the area and power breakdown of TetriX. The memory module occupies 72.5% of the total area and consumes 25% of the power. The index and backend modules, i.e., the overhead to support flexible tensor contraction, account for less than 10% of the area and power.

B. MCSS Efficacy and Performance Analysis

The speedup of TetriX over single-mode mapping baselines, Base-I and Base-O, on TT, TR, HT, and BT workloads is shown in Fig. 12. The baseline designs use fixed-pattern contraction sequences for TNN inference. Specifically, fixed contraction sequences proposed by [29] and [30] are used for TT and HT, respectively. For TR and BT, the contraction sequence begins from the first to the last, following the examples shown in Fig. 2(b) and 2(d), such as 1, 2, …, 8 for TR and 1, 2, …, 5 for BT. When compared to TetriX that uses MCSS with hybrid mapping, the baseline approaches, Base-I and Base-O, suffer from redundant MAC computation and complex tensor shaping that involves non-coalesced read-shaping-write operations. As a result, these baseline designs incur additional latency overhead. Due to space limitation, we only list the top and the last 10 results in speedup, and the geometric mean for all collected and synthetic workloads.

In Fig. 12(a), for TT workloads, TetriX shows a speedup of 1.4× and 1.3× over Base-I and Base-O, respectively. The improvement is relatively modest, mainly due to the

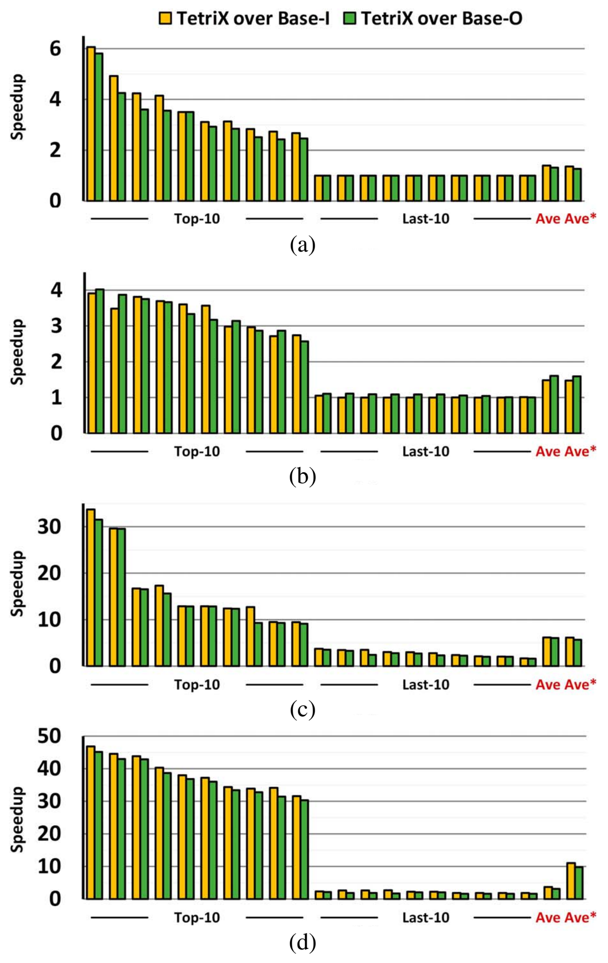


Fig. 12. Speedup in performance achieved by TetriX with hybrid mapping compared to the baselines with inner-only mapping (base-I) and outer-only mapping (base-O) for various workloads: (a) TT, (b) TR, (c) HT and (d) BT. The terms "Ave" and "Ave*" refer to the average speedup across collected and synthetic workloads, respectively.

TABLE IV
PERFORMANCE COMPARISON OF DECOMPOSITION METHODS ON TETRIX

	In Dim×Out Dim	Rank	Weight (Compression)	Acc.
Base [20]	57600×256	-	59.0 M	69.7%
TT [14]	(8,20,20,18)×(4,4,4,4)	4[a]	3,232 (18,250×)	79.6%
HT [18]	(8,10,10,9,8)×(4,4,2,4,2)	5 | 4[b]	1,245 (47,375×)	87.2%
TR [17]	(4,2,5,8,6,5,3,2)×(4,4,2,4,2)	10 | 5[c]	1,725 (34,193×)	86.9%
BT [20]	(8,20,20,18)×(4,4,4,4)	1 | 4[d]	3,387 (17,414×)	85.3%

[a] $r_0 = 1$, $r_{\neq 0} = 4$. [b] $r_{\text{leaf}} = 4$, $r_{\neq \text{leaf}} = 5$. [c] $r_0 = 10$, $r_{\neq 0} = 5$. [d] $S = 1$, $r = 4$.

	MAC Count (Reduction)	Util (%)	Max Mem Size (KB)	Latency[e] (Speedup[f])
TT	1,912,832 (7.7×)	98.7	112.5	9,367 (6.1×)
HT	2,383,872 (6.2×)	30.7	160	33,168 (1.7×)
TR	2,547,900 (5.8×)	32.9	351.6[g]	30,263 (1.9×)
BT	2,387,968 (6.2×)	85.1	112.5	12,768 (4.5×)

[e] Measured in processing cycles. [f] Compared to the latency for base matrix with peak compute throughput. [g] External memory access needed.

simplicity of TT decomposition, where the contraction sequence follows a regular pattern with minimal variation, and transpose is infrequent.

Unlike the TT decomposition, TR, HT, and BT decompositions exhibit more complex structures, resulting in a larger search space for MCSS. Therefore, TetriX achieves larger speedups over the two baselines, as shown in Fig. 12(b), 12(c), and 12(d). TetriX provides speedups of 1.5×, 6.1×, and 9.5× over Base-I for TR, HT, and BT workloads, respectively; and speedups of 1.6×, 5.7×, and 8.3× over Base-O for TR, HT, and BT workloads, respectively. For these decomposition methods, MCSS is effective in identifying better contraction sequences that require fewer MAC operations and memory sizes than fixed pattern contraction sequences. On average, MCSS reduces the MAC operations by 1.6×, 7.9×, and 11.7× for TR, HT, and BT, respectively, when compared to the fixed contraction sequences.

Overall, TetriX consistently achieves performance improvements across all TNN types and tensor dimensions, demonstrating the advantages of its unified IP/OP architecture and MCSS with the hybrid mapping scheme. With MCSS, TetriX outperforms Base-I and Base-O by up to 33× and 46× for HT and BT workloads, respectively.

C. Decomposition Method Performance Analysis

Table IV presents an illustrative example of different decomposition methods, TT, HT, TR, and BT [14], [17], [18], [20], applied to the input-to-hidden layer of an LSTM model for video task classification. All methods provide higher accuracy compared to the baseline LSTM. In terms of model size, TT and BT achieve compression ratios of approximately 18,000×, and TR and HT achieve even higher compression ratios, exceeding 34,000×.

The decomposed models are evaluated on TetriX to compare the performance of these different methods. Optimal contraction sequences for each method are obtained using MCSS. The TT decomposed model achieves the best latency, closely followed by the BT decomposed model. Both TT and BT methods produce simple and uniform ranks of 4, which can be easily mapped onto TetriX's 16×16 compute array. On the other hand, the HT and TR methods produce better compression, but also result in non-uniform ranks and smaller input and output dimensions. Consequently, the utilization of TetriX's compute array may be suboptimal. By leveraging the MAC reduction provided by MCSS, all methods achieve a speedup of over 1.7× compared to the base matrix.

D. Accelerator Performance Comparison

Fig. 13 provides a comparison between TetriX, TIE [29], and FDHT [30], which are state-of-the-art TNN accelerators for TT and HT workloads, respectively. TetriX uses MCSS with hybrid mapping to find and map the optimal contraction sequence. In contrast, both TIE and FDHT employ a fixed customized contraction pattern [29], [30]. For the purpose of this comparison, we evaluated the workloads from the dataset that could be fully accommodated by TetriX.

TIE is specifically designed for TT workloads. TetriX can achieve comparable performance to TIE in cases where the optimal TT contraction sequence aligns with the TIE's fixed

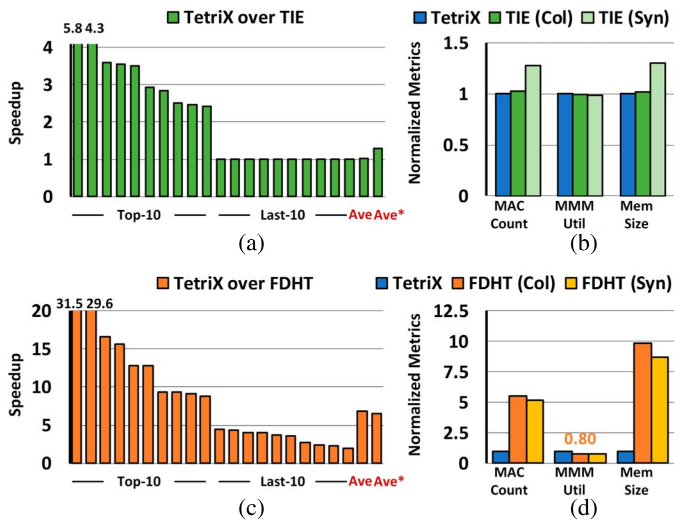


Fig. 13. (a) Comparison of speedup achieved by TetriX over TIE across TT workloads, (b) comparison of resource utilization by TetriX over TIE, (c) comparison of speedup achieved by TetriX over FDHT across HT workloads; and (d) comparison of resource utilization by TetriX over FDHT. The terms “Ave” and “Ave*” refer to the average speedup across collected (Col) and synthetic (Syn) workloads, respectively.

TABLE V
COMPARISON TO EXISTING ACCELERATOR WORKS

	TetriX	TIE	FDHT	DNPU
TNN Supported Contraction Sequence	**TT/HT/TR/BT/etc.** **Arbitrary**	TT Only Fixed Pattern	HT Only Fixed Pattern	No Fixed Pattern
Technology (nm)	28	28	28	65
Frequency (MHz)	1000	1000	1000	200
Compute Unit (MACs)	256	256	256	64
Precision (bit)	16	16	16	7
On-Chip Mem. (KB)	548	784	1809.2	10
Area (mm^2)	1.34	1.40[a]	2.96[b]	8[c]
Power (mW)	IP: 197.7 OP: 180	104.8[a]	160.4[b]	21
Comp Efficiency[d,e] (TOPS/W)	TT: 1.8 | 7.5	3.3 | 10.7	-	1.1
	HT: 1.4 | 2.2	-	1.3 | 0.4	
Inf. Throughput[e] (K-Inf./s)	TT: 215.9 | 875.7	173.8 | 567.4	-	1.8[c]
	HT: 25.6 | 40.1	-	3.9 | 1.2	3.1[c]

[a]Synthesis results reported in [29]. [b]Layout results reported in [30]. [c]Projected from results reported in [28]. [d]A MAC is counted as 2 OPs. [e]Evaluated on collected and synthetic workloads, nominal|effective numbers are reported.

contraction pattern. The comparison in Fig. 13(a) shows that when the optimal TT contraction sequence deviates from TIE’s pattern, TetriX outperforms TIE with a speedup of up to 5.8×. On average, across all collected (Col) and synthetic (Syn) TT workloads, TetriX achieves speedups of 1.03× and 1.2× over TIE, respectively. This shows that TetriX’s flexibility enables better adaptation to varying tensor dimensions in synthetic workloads compared to TIE. For TT workloads where TetriX selects a different contraction sequence than TIE, TetriX provides an average speedup of 1.6× over TIE, while also reducing MAC operations by 1.6×, decreasing memory requirements by 1.6×, and achieving 3% higher PE utilization.

FDHT is specifically designed for HT workloads. However, TetriX goes a step further by taking advantage of the large contraction sequence search space available in HT workloads. This enables TetriX to achieve a significant speedup compared to FDHT. When compared to FDHT, TetriX outperforms FDHT by achieving a speedup of up to 31.5× across all HT workloads, with an average speedup of 6.9× for collected workloads. The performance improvement is shown in Fig. 13(c). Furthermore, TetriX requires 5.5× fewer MAC operations, uses 9.8× less memory, and achieves 1.3× higher PE utilization for collected workloads, as shown in Fig. 13(d). These results highlight the performance and efficiency advantages of TetriX in comparison to FDHT.

TetriX is able to complete inferences more quickly than previous works due to reduced number of MAC operations and a higher throughput. Additionally, TetriX’s reduced memory size allows for full mapping of more end-to-end TNN workloads on the chip. This means these workloads can be processed without the need for external memory access.

Table V provides a comparison of TetriX with TIE, FDHT, and DNPU – a conventional DNN accelerator [28]. TetriX, TIE, and FDHT are all designed in 28 nm technology, clocked at the same frequency, and have an equal number of MACs. On the other hand, DNPU is fabricated in 65 nm technology and operates at 200 MHz. In terms of on-chip memory, TetriX requires only 256 KB for each tensor memory, smaller than the 384 KB and the 896 KB tensor memory in TIE and FDHT, respectively. TetriX’s silicon area is comparable to TIE, while being 2.2× smaller than FDHT. The smaller footprint is achieved even with the inclusion of additional index and backend modules in TetriX.

TIE [29] and FDHT [30] were evaluated using a limited set of workloads, specifically 4 and 2 workloads, respectively. These evaluations used uniform ranks (leaf/non-leaf ranks for HT). However, these workloads do not fully represent the diversity of TNN workloads. In our work, we evaluated across a much larger set of TT and HT workloads from our collected and synthetic dataset. These workloads have an average compression ratio of 5.3× for TT and 6.9× for HT. Across the TT workloads, TetriX achieves a MAC reduction of 4.1×, while TIE achieves a reduction of 3.3×. For the HT workloads, TetriX achieves a MAC reduction of 1.6×, whereas FDHT only achieves a reduction of 0.3×. In fact, the contraction pattern for FDHT actually requires more MACs than the original MMM computation. DNPU, on the other hand, is unable to support the tensor shaping required for on-chip TT or HT decomposition. Therefore, DNPU is evaluated on the original weight matrix.

For the TNN accelerators, we report both nominal and effective performance. Nominal performance reflects the actual number of MACs performed. Effective performance takes into consideration the number of MACs in the original DNN layer. In terms of nominal performance, TetriX’s compute efficiency is lower than TIE’s due to TetriX’s higher power consumption. However, TetriX achieves 1.2× higher inference throughput than TIE by employing optimal contraction sequences that reduce latency. Compared to FDHT, TetriX demonstrates 1.1× higher compute efficiency and 6.5× higher inference throughput. In terms of effective performance, TetriX benefits from a larger MAC reduction from MCSS, enabling it to surpass TIE in inference throughput and outperform FDHT both in efficiency and inference throughput. TetriX also outperforms DNPU in both throughput and efficiency owing to its significant MAC reduction for the TT and HT workloads.

In summary, TetriX uses configurable dataflows, index translation, and output gathering to provide the necessary flexibility for supporting a wide range of TNN workloads. Additionally, TetriX demonstrates superior performance and efficiency over the existing state-of-the-art designs by employing MCSS.

VIII. Related Work

Tensor decomposition is commonly applied to specific layers in a network such as FC, LSTM, CONV, and attention layers, resulting in a combination of normal layers and TNN layers. TetriX is primarily designed for TNN layers but can also handle normal layers by using the systolic array [37] and disabling the index translation and output gathering. When it comes to CONV layers, there is no standard decomposition method due to the sliding window operation [8], [9], [11], [12]. However, TetriX overcomes this by supporting a CONV-TNN layer, wherein the sliding window operation is converted into a matrix form, similar to SIMD architectures. Furthermore, TetriX supports large embedding layers using tensor decomposition [23], [24] by partitioning the workload and accessing each partition from external memory.

In the past, the application of optimal contraction sequence search has mainly focused on reducing computation and memory costs in fields such as quantum many-body systems, quantum chemistry, and data analytics [39], [40]. However, the aspect of workload mapping and compute utilization on specialized hardware has not been considered until now.

Existing NN accelerators [27], [28] and general matrix multiply architectures [37] are efficient for workloads with known orders such as CONV, FC, and MMM, as they rely on constrained computation sequence search spaces. Additionally, previous research on flexible dataflow and mapping [43] has been primarily focused on specific workload types where tensor orders and contraction patterns are known in advance, albeit with varying tensor dimensions. However, these designs are not well-suited for TNNs due to the arbitrary tensor orders and dimensions involved, resulting in a significant degree of freedom for contraction sequence search.

FPGA design compilation frameworks, e.g., PolySA [44], AutoSA [45], can generate systolic array designs for a specific workload. However, these frameworks require prior knowledge of the workload to determine the best design, making them unsuitable for TNN inference with high workload versatility. Additionally, PolySA and AutoSA primarily focus on exploring the dataflow and mapping spaces, without optimizing the contraction sequence. This limitation can lead to redundant MAC operations and increased memory footprint.

ETTE [46], a subsequent work to TIE, introduces a TT variant that further decomposes weight tensors into 3rd-order (compared to 4th-order in TT), similar to the TR method. It uses a look-ahead contraction sequence that processes tensor and weight chunks and stores intermediate tensors in PE registers to avoid unnecessary memory access. However, it is important to note that the look-ahead mechanism and architecture of ETTE are specifically designed for the proposed TT variant method, limiting its ability to support other common decomposition methods. Additionally, the fixed contraction ordering imposed by the look-ahead processing may result in redundant MAC operations during processing. In contrast, MCSS supports all tensor decomposition methods, including the proposed TT variant, to identify the optimal contraction sequence with minimal MAC operations.

IX. Conclusion

We present TetriX, a co-design that combines a flexible domain-specific architecture with optimal workload mapping to efficiently process TNNs. TetriX is capable of flexibly adapting to the structure, order, and dimensions of a TNN layer, and leverages optimal contract sequences to achieve the best performance and energy efficiency for inference. The design of TetriX includes a configurable dataflow to support both IP and OP, as well as index translation and output gathering, enabling flexible tensor shaping for versatile TNN workloads. To address the complexity of tensor shaping in an arbitrary contraction sequence, we propose a hybrid mapping scheme that alternates between IP and OP mappings. Furthermore, we introduce MCSS, an approach that identifies the optimal contraction sequence on TetriX. The contraction sequences obtained using MCSS require considerably less computation (3.3×), memory (7×), and exhibit shorter latency (7.2×) compared to patterns used in previous works. When compared to the baseline accelerator design that uses fixed contraction sequences and mapping, MCSS achieves substantial speedups of 1.4×, 1.6×, 6.1×, and 9.5× for TT, TR, HT, and BT workloads, respectively. Additionally, it significantly reduces MAC operations and memory footprint. TetriX is the first design to support all tensor decomposition methods in existing TNNs, accommodating different orders, dimensions, and ranks. Taking HT workloads as an example, compared to the state-of-the-art accelerator, TetriX provides 6.5× higher inference throughput and 1.1× higher compute efficiency.

Acknowledgment

The authors would like to thank Shang-En Huang, Miao Yin, and Salar Latifi for the valuable discussions.

References

[1] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.
[2] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64270–64277, 2018.
[3] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 1135–1143.
[4] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2014, pp. 1269–1277.

[5] T. N. Sainath, B. Kingsbury, V. Sindhwani, E. Arisoy, and B. Ramabhadran, "Low-rank matrix factorization for deep neural network training with high-dimensional output targets," in *Proc. Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, 2013, pp. 6655–6659.
[6] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, "Tensorizing neural networks," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 442–450.
[7] H. Huang and H. Yu, "LTNN: A layerwise tensorized compression of multilayer neural network," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 5, pp. 1497–1511, May 2019.
[8] V. Aggarwal, W. Wang, B. Eriksson, Y. Sun, and W. Wang, "Wide compression: Tensor ring nets," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 9329–9338.
[9] B. Wu, D. Wang, G. Zhao, L. Deng, and G. Li, "Hybrid tensor decomposition in neural network compression," *Neural Netw.*, vol. 132, pp. 309–320, Sep. 2020.
[10] G. Li, J. Ye, H. Yang, D. Chen, S. Yan, and Z. Xu, "BT-Nets: Simplifying deep neural networks via block term decomposition," 2017, *arXiv:1712.05689*.
[11] J. Ye, G. Li, D. Chen, H. Yang, S. Zhe, and Z. Xu, "Block-term tensor neural networks," *Neural Netw.*, vol. 130, pp. 11–21, Jun. 2020.
[12] 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," 2016, *arXiv:1511.06530*.
[13] A. Tjandra, S. Sakti, and S. Nakamura, "Compressing recurrent neural network with tensor train," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2017, pp. 4451–4458.
[14] Y. Yang, D. Krompass, and V. Tresp, "Tensor-train recurrent neural networks for video classification," in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2017, pp. 3891–3900.
[15] A. Tjandra, S. Sakti, and S. Nakamura, "Tensor decomposition for compressing recurrent neural network," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2018, pp. 1–8.
[16] C. C. Onu, J. E. Miller, and D. Precup, "A fully tensorized recurrent neural network," 2020, *arXiv:2010.04196*.
[17] Y. Pan et al., "Compressing recurrent neural networks with tensor ring for action recognition," in *Proc. Conf. Artif. Intell. (AAAI)*, 2019, pp. 4683–4690.
[18] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Compressing recurrent neural networks using hierarchical Tucker tensor decomposition," 2020, *arXiv:2005.04366*.
[19] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Towards extremely compact RNNs for video recognition with fully decomposed hierarchical Tucker structure," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2021, pp. 12085–12094.
[20] J. Yeet al., "Learning compact recurrent neural networks with block-term tensor decomposition," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 9378–9387.
[21] H. P. Minh, N. N. Xuan, and S. T. Thai, "TT-ViT: Vision transformer compression using tensor-train decomposition," in *Proc. Int. Conf. Comput. Collective Intell. (ICCCI)*, 2022, pp. 755–767.
[22] X. Ma et al., "A tensorized transformer for language modeling," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2019, pp. 2232–2242.
[23] O. Hrinchuk, V. Khrulkov, L. Mirvakhabova, E. Orlova, and I. Oseledets, "Tensorized embedding layers for efficient model compression," 2019, *arXiv:1901.10787*.
[24] C. Yin, B. Acun, C.-J. Wu, and X. Liu, "TT-Rec: Tensor train compression for deep learning recommendation models," in *Proc. Conf. Mach. Learn. Syst. (MLSys)*, 2021, pp. 448–462.
[25] D. Wang et al., "Kronecker CP decomposition with fast multiplication for compressing RNNs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 34, no. 5, pp. 2205–2219, May 2023.
[26] X. Liu and K. K. Parhi, "Tensor decomposition for model reduction in neural networks: A review," *IEEE Circuits Syst. Mag.*, vol. 23, no. 2, pp. 8–28, 2nd Quart. 2023.
[27] B. Moons, R. Uytterhoeven, W. Dehaene, and M. Verhelst, "14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm FDSOI," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2017, pp. 246–247.
[28] D. Shin, J. Lee, J. Lee, J. Lee, and H.-J. Yoo, "DNPU: An energy-efficient deep-learning processor with heterogeneous multi-core architecture," *IEEE Micro*, vol. 38, no. 5, pp. 85–93, Sep./Oct. 2018.
[29] 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. Comput. Archit. (ISCA)*, 2019, pp. 264–277.
[30] Y. Gong, M. Yin, L. Huang, C. Deng, and B. Yuan, "Algorithm and hardware co-design of energy-efficient LSTM networks for video recognition with hierarchical Tucker tensor decomposition," *IEEE Trans. Comput.*, vol. 71, no. 12, pp. 3101–3114, Dec. 2022.
[31] R. Guo et al., "A 5.99-to-691.1TOPS/W tensor-train in-memory-computing processor using bit-level-sparsity-based optimization and variable-precision quantization," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2021, pp. 242–244.
[32] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, 2009.
[33] I. V. Oseledets, "Tensor-train decomposition," *SIAM J. Sci. Comput.*, vol. 33, no. 5, pp. 2295–2317, 2011.
[34] W. Hackbusch and S. Kühn, *Springer J. Fourier Anal. Appl.*, vol. 15, no. 5, pp. 706–722, 2009.
[35] Q. Zhao, G. Zhou, S. Xie, L. Zhang, and A. Cichocki, "Tensor ring decomposition," 2016, *arXiv:1606.05535*.
[36] L. De Lathauwer, "Decompositions of a higher-order tensor in block terms—part II: Definitions and uniqueness," *SIAM J. Matrix Anal. Appl.*, vol. 30, no. 3, pp. 1033–1066, 2008.
[37] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 1–12.
[38] Z. Du et al., "ShiDianNao: Shifting vision processing closer to the sensor," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2015, pp. 92–104.
[39] L. Liang et al., "Fast search of the optimal contraction sequence in tensor networks," *IEEE J. Sel. Topics Signal Process.*, vol. 15, no. 3, pp. 574–586, Apr. 2021.
[40] C.-C. Lam, P. Sadayappan, and R. Wenger, "On optimizing a class of multi-dimensional loops with reduction for parallel execution," *World Sci. Parallel Process. Lett.*, vol. 7, no. 2, pp. 157–168, 1997.
[41] M. Yin, Y. Sui, S. Liao, and B. Yuan, "Towards efficient tensor decomposition-based DNN model compression with optimization framework," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2021, pp. 10674–10683.
[42] F. Sedighin, A. Cichocki, and A.-H. Phan, "Adaptive rank selection for tensor ring decomposition," *IEEE J. Sel. Topics Signal Process.*, vol. 15, no. 3, pp. 454–463, Apr. 2021.
[43] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks," in *Proc. Int. Symp. High-Perform. Comput. Archit. (HPCA)*, 2017, pp. 553–564.
[44] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *Proc. Int. Conf. Comput.-Aided Des. (ICCAD)*, 2018, pp. 1–8.
[45] J. Wang, L. Guo, and J. Cong, "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA," in *Proc. Int. Symp. Field Programmable Gate Arrays (FPGA)*, 2021, pp. 93–104.
[46] Y. Gong et al., "ETTE: Efficient tensor-train-based computing engine for deep neural networks," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2023, pp. 1–13.

Jie-Fang Zhang (Member, IEEE) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2015, and the M.S. degree in computer science and engineering and the Ph.D. degree in electrical and computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018 and 2022, respectively. He joined NVIDIA in 2022 as a Deep Learning Architect focusing on GPU performance analysis, modeling, and optimization for deep learning models. His research interests include energy-efficient hardware architecture and accelerator design for machine learning, computer vision, and robotics applications.

Cheng-Hsun Lu (Graduate Student Member, IEEE) received the B.S. degree in electrical engineering and the M.S. degree from the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan, in 2016, and 2019, respectively. He is currently working toward the Ph.D. degree in electrical and computer engineering with the University of Michigan, Ann Arbor, MI, USA. His research interests include energy-efficient, high-performance VLSI circuits and systems for machine learning, DSP algorithms, and forward error correction.

Zhengya Zhang (Senior Member, IEEE) received the B.A.Sc. degree in computer engineering from the University of Waterloo, in 2003, and the M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley (UC Berkeley), in 2005 and 2009, respectively. He has been a Faculty Member with the University of Michigan, Ann Arbor, MI, USA, since 2009, where he is currently a Professor with the Department of Electrical Engineering and Computer Science. His research interests include low-power and high-performance VLSI circuits and systems for computing, communications, and signal processing. He was a recipient of the University of Michigan College of Engineering Neil Van Eenam Memorial Award in 2019, the Intel Early Career Faculty Award in 2013, the National Science Foundation CAREER Award in 2011, and the David J. Sakrison Memorial Prize from UC Berkeley in 2009. He has served on the Technical Program Committee of the IEEE Custom Integrated Circuits Conference (CICC) since 2019. He served as an Associate Editor for IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS from 2015 to 2022, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART I: REGULAR PAPERS from 2013 to 2015, and IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: EXPRESS BRIEFS from 2014 to 2015. He served on the Technical Program Committee of IEEE VLSI Symposium on Technology and Circuits from 2019 to 2022. He is an IEEE Solid-State Circuits Society Distinguished Lecturer from 2023 to 2024.