TIE: Energy-efficient Tensor Train-based Inference Engine for Deep Neural Network

Chunhua Deng*  
Rutgers University  
chunhua.deng@rutgers.edu

Fangxuan Sun*  
Nanjing University  
fxsun@mail.nju.edu.cn

Xuehai Qian  
University of Southern California  
xuehai.qian@usc.edu

Jun Lin  
Nanjing University  
jlin@nju.edu.cn

Zhongfeng Wang  
Nanjing University  
zfwang@nju.edu.cn

Bo Yuan  
Rutgers University  
bo.yuan@soe.rutgers.edu

ABSTRACT
In the era of artificial intelligence (AI), deep neural networks (DNNs) have emerged as the most important and powerful AI technique. However, large DNN models are both storage and computation intensive, posing significant challenges for adopting DNNs in resource-constrained scenarios. Thus, model compression becomes a crucial technique to ensure wide deployment of DNNs.

This paper advances the state-of-the-art by considering tensor train (TT) decomposition, an extremely high compression ratio. However, the challenge is the inference on the TT-format DNN model and incurs massive amount of redundant computations, causing significant energy consumption. Thus, the straightforward application of TT decompression is not feasible.

To address this fundamental challenge, this paper develops a computation-efficient inference scheme for TT-format DNN, which enjoys two key merits: 1) it achieves theoretical limit of number of multiplications, thus eliminating all redundant computations; and 2) the multi-stage processing scheme reduces the intensive memory access to all tensor cores, bringing significant energy saving.

Based on the novel inference scheme, we develop TIE, a TT-format compressed DNN-targeted inference engine. TIE is highly flexible, supporting different types of networks for different needs. A 16-processing elements (PE) prototype is implemented using CMOS 28nm technology. Operating on 1000MHz, the TIE accelerator consumes 1.74mm² and 154.8mW. Compared with EIE, TIE achieves 7.22×~10.66× better area efficiency and 3.03×~4.48× better energy efficiency on different workloads, respectively. Compared with CtrCNN, TIE achieves 5.96× and 4.56× higher throughput and energy efficiency, respectively. The results show that TIE exhibits significant advantages over state-of-the-art solutions.

*Both authors contributed equally to this research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
In the emerging artificial intelligence (AI) era, deep neural networks (DNNs) [25] have become the most important and popular machine learning (ML) technique. Thanks to their unique large-scale structure that consists of thousands of neurons and millions of connections, DNNs are able to obtain very strong learning and representation capability. Thus, DNNs are able to deliver and guarantee very high task accuracy in many real-life intelligence-demand applications [5, 15, 44, 56, 73].

However, the unprecedented high task performance of DNNs does not come for free, — executing these large-scale DNNs normally demands large storage space and powerful computing resources. This is because, different from many other ML approaches, the performance of DNNs are strongly related to their scale: both theoretical analysis and empirical experiments have suggested that deepening or widening DNN models can bring better representation capability and higher classification accuracy [29, 79]. Motivated by this promising property, today’s AI researchers usually tend to use and scale up large-scale DNN models to achieve the required high accuracy in various AI tasks. However, the consequence of the trend is that the state-of-the-art DNNs are all very storage-intensive and computation-intensive. In particular, it causes severe challenges on the efficient implementation of DNNs in many resource-constrained embedded and mobile systems.

In order to facilitate and promote the widespread deployment of DNNs in boarder scope of application scenarios, both ML and hardware communities have conducted extensive investigations on compressing DNNs with affordable accuracy loss. Specifically, due to the well recognized and verified redundancy of DNN models [24, 27, 28], different compression approaches, such as pruning [27],...
clustering [24], low rank decomposition [46] and low bit-width [47] etc., have been proposed and adopted to reduce the redundancy on structure, layer, weight or number precision of DNN models. Correspondingly, several compression-oriented DNN accelerators [6, 26, 75] have also been customized for those compression approaches to achieve high hardware performance.

Among various DNN compression techniques, tensor-train (TT) decomposition [52] is unique due to its extremely high compression ratios. For instance, experiments in [50] show that applying TT decomposition to the fully-connected (FC) layers of VGG-16 [50] on ImageNet dataset [17] can bring record-breaking 50000x compression ratio, while other compression approaches typically only achieve much less compression on FC layers. Moreover, due to the generality of TT decomposition, this approach can also be applied to compressing convolutional (CONV) layers via decomposing the weight tensors of CONV layers [23]. To date, ML community [23, 50, 74, 77] has successfully verified the effectiveness of TT decomposition and its variant (TT ring) [81] in several representative types of DNNs, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs).

From the perspective of tensor theory, the impressive compression capability of TT decomposition comes from its unique tensor factorization scheme. As illustrated in Fig. 1, TT decomposition can decompose a d-dimensional tensor to 3-dimensional n1 × n2 × ... × nd tensor A to d 3-dimensional r_{k-1} × n_k × r_k tensor cores, where r_k is the preset rank value. Thanks to this special representation scheme, only \[ \sum_{k=1}^{d} n_k r_{k-1} r_k \] parameters need to be stored in the TT format while conventionally \[ \prod_{k=1}^{d} n_k \] parameters were required for an explicit representation. Since in practice r_k is typically small, the compression ratio, as defined as \[ \frac{\prod_{k=1}^{d} n_k}{\sum_{k=1}^{d} n_k r_{k-1} r_k} \], can be very significant and hence brings orders of magnitude reduction in storage cost.

Due to the promising advantages of TT decomposition on model compression, exploiting efficient DNN hardware architecture based on TT decomposition (referred as TT-DNN) is very attractive. Considering the exceedingly high compression ratios that TT decomposition can bring, such specialized architecture, if properly developed, has the potential to execute the state-of-the-art CNN and RNN models using very small memory resource, thereby leading to highly area and energy efficient solutions for today’s resource-constrained but memory and compute intensive DNN accelerator design.

However, realizing a high-performance TT-DNN accelerator is far from trivial, and needs to overcome the severe challenge on the inefficient inference scheme based on a TT-format DNN model. Specifically, the current TT-format inference scheme contains a massive amount of redundant computations, leading to significantly higher computational cost in the inference phase. Moreover, those inherent redundant computations also incur intensive memory accesses, — the tensor cores need to be frequently accessed when calculating each element of output tensor, thereby causing high energy consumption. As a result, despite the high compression ratios, the inherent inefficiency of TT-format inference scheme directly impedes the potential deployment of TT-DNN accelerator in energy-constrained applications.

To address the fundamental challenges, this paper exploits the hardware-friendly inference scheme for TT-DNN. By carefully reviewing and investigating the root reason and mechanism of redundant computation of current TT-format inference scheme, we derive the theoretical limit for minimum number of multiplications needed for TT-format inference. Then, leveraging the same methodology in the theoretical analysis, we develop a computation-efficient inference scheme for TT-DNN. The proposed TT-format inference scheme has two benefits: 1) it is very compact. The required number of multiplications of this scheme is identical to the theoretical limit, thus eliminating all the unnecessary redundant computations; and 2) based on its multi-stage processing style, the computing engine only needs to access one tensor core in each stage, thereby leading to significant saving in memory access.

Based on the proposed inference scheme, we develop TIE, the TT-DNN Inference Engine, a novel specialized hardware architecture based on TT-DNN. TIE is designed to fully reap the benefits of our proposed hardware-friendly inference scheme and achieves high computation efficiency as well as simple memory access. Also, TIE is highly flexible and can be adapted to various network types, values of ranks, number of tensor dimensions, and combinations of factorization factors, thereby making itself well suited for various application scenarios and tasks.

We implement an prototype TIE design using CMOS 28nm technology. With 16 processing elements (PES) operating on 1000MHz, the TIE accelerator consumes 1.74mm\textsuperscript{2} and 154.8mW. Compared to the state-of-the-art compressed DNN-oriented accelerators using other compression methods, such as sparsification (EIE [26]) and structured matrices (CirCNN [18]), the TT decomposition-based TIE exhibits significant advantages in hardware performance. Compared with EIE, TIE achieves 7.22× ~ 10.66× better area efficiency and 3.03× ~ 4.48× better energy efficiency on different workloads, respectively. Compared with CirCNN, TIE achieves 5.96× and 4.56× higher throughput and energy efficiency, respectively.

2 TT-BASED DNN COMPRESSION

2.1 TT Decomposition & TT Format

As introduced in Section 1, TT decomposition is an efficient compression approach to reduce DNN model sizes. In general, the key idea of TT decomposition is to decompose a large-size multi-dimensional tensor into a set of small-size 3-dimensional tensors. Specifically, for a d-dimensional tensor A, after TT decomposition A is stored in the TT format using d tensor cores \[ G_k \in \mathbb{R}^{r_{k-1} \times n_k \times r_k} \], where \( k = 1, 2, \ldots, d \), and each element in A can be reconstructed as follows [50]:

\[
A(j_1, \ldots, j_d) = G_1[j_1] \times G_2[j_2] \times \cdots \times G_d[j_d],
\]

where \( G_k[j_k] \in \mathbb{R}^{r_{k-1} \times r_k} \) is the \( j_k \)-th slice of the \( k \)-th tensor core \( G_k \) with \( j_k = 1, 2, \ldots, n_k \), and \( r_k \) is the rank of a tensor core. Accordingly, the number of parameters to represent A is reduced from \( \prod_{k=1}^{d} n_k \) to \( \sum_{k=1}^{d} n_k r_{k-1} r_k \). It is worth noting that, though the value of \( r_k \) can vary since TT-decomposition for arbitrary tensor is not unique, \( n_0 \) and \( r_d \) are always set as 1 to satisfy the boundary condition.

In practical use, the value of \( r_k \) is typically set as a small value, so that the parameter saving resulting from TT decomposition can
be very significant. Consequently, leveraging TT decomposition to perform efficient compression on DNN models is very attractive since the fully-connected (FC) and convolutional (CONV) layers of DNNs are in the format of matrix and tensor, which can be decomposed and represented in the TT format. As suggested in [50], in order to maintain high test accuracy, the TT decomposition is typically not directly applied to the 2D weight matrix or 4D weight tensor but to their reshaped format. For instance, as illustrated in Fig. 1, in order to store a 5×12 weight matrix in the TT format, the weight matrix is first reshaped to a 3-dimensional tensor as $d=3$, and then it is decomposed and stored in the three tensor cores ($G_1 \sim G_3$).

2.2 TT-Format Inference & Training on DNNs

In general, when a DNN model is stored in the TT format, the corresponding inference and training schemes need to be re-formulated since the underlying representation for weight matrix and tensor of FC and CONV layers have been changed.

**Inference on TT-format FC layers.** Conventionally, with weight matrix $W \in \mathbb{R}^{M \times N}$, input vector $x \in \mathbb{R}^N$ and output vector $y \in \mathbb{R}^M$, the inference procedure on FC layer is $y=Wx$. In the scenario of representing weight matrix in the TT format, such inference scheme is re-formulated as follows [52]:

$$y(i_1, \ldots, i_d) = \sum_{j_1, \ldots, j_d} G_1[i_1, j_1] G_2[i_2, j_2] \ldots G_d[i_d, j_d] x(j_1, \ldots, j_d).$$

Figure 1: Represent a matrix via using the TT decomposition of its reshaped tensor.

Figure 2: TT-format inference scheme.

where $y \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_d}$ and $x \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ are the reshaped $y$ and $x$ in the tensor format with $M = \prod_{k=1}^d m_k$ and $N = \prod_{k=1}^d n_k$, respectively. For weight matrix $W$, it is first reshaped into a $d$-dimensional tensor $G$, and then $G$ is decomposed and represented in the TT format with $d$ tensor cores $G_k$, where $k = 1, 2, \ldots, d$. Notice that here $G_k \in \mathbb{R}^{r_k \times m_1 \times m_2 \times \ldots \times m_d}$ is different from the representation in Eqn. (1) in Section 2.1. This is because, as indicated in [50], this 4D representation of tensor cores is better for describing the matrix-vector multiplication in the TT format\(^2\). Accordingly, as illustrated in Fig. 2, $G_k$ can be viewed as a 2D $m_k$-by-$n_k$ array, where each element ($G_k[i_k, j_k]$) in Eqn. (2)) of this array is an $r_k$-by-$r_k$ matrix.

**Inference on CONV layers in the TT format.** Due to its generality for arbitrary tensor, TT decomposition can also enable efficient inference on CONV layer that is affiliated with a weight tensor. Specifically, there are two methods to represent the conventional 4D weight tensor of CONV layer in the TT format. The first one is to directly apply TT decomposition to the 4D tensor and obtain the corresponding tensor cores. However, as indicated in [23], such method is not very efficient for CONV layer with small kernel sizes (e.g., 1×1 convolution). Therefore, another method [23] is to reshape the 4D weight tensor to 2D matrix, and then use the same procedure for inference on FC layer to perform inference on CONV layer. As illustrated in Fig. 3, such transform is mathematically rigid since the 2D convolution between the 3D input tensor and 4D weight tensor is equivalent to the matrix-by-matrix multiplication [33]. Consequently,

\(^2\) We can still use 3D-based tensor cores to describe inference scheme. The drawback is the complicated indexing. The 4D representation can be viewed as folding the original 3D tensor.
both the inference on FC layers and CONV layers can be executed on the same TT-format inference engine.

![Figure 3: Converting computation on CONV layer to matrix multiplication. Here $H' = H - f + 1$ and $W' = W - f + 1$.](image)

**Training TT-format DNN models.** In general, after the sizes of tensor cores $G_k$ have been determined, a DNN model in the TT format can be either trained from the scratch or obtained from a pre-trained non-TT-format model. Specifically, the train-from-scratch strategy assigns initial values for each tensor core and then performs backward propagation scheme in [50] to update them. On the other hand, if converting a non-TT-format trained model to the TT format is needed, the standard TT decomposition in [52] is first applied to the weight matrix/tensor of the FC/CONV layer of model to form the initial values of tensor cores. Then backward propagation-based fine-tuning process is performed to retain original high accuracy.

### 2.3 Compression & Accuracy Performance

Based on the training and inference described in Section 2.2, the TT-format DNN models can be trained and tested. Table 1 - Table 3 list the test accuracy and compression ratio (CR) of different types of DNN models (convolutional neural network (CNN) and recurrent neural network (RNN)) on different datasets. Here the CR is measured as the reduction of number of parameters of the model. Specifically, the experimental settings are as follows:

#### Table 1: FC-dominated CNN on ImageNet.

<table>
<thead>
<tr>
<th>FC-dominated CNN [50] (NIPS’16)</th>
<th>Accuracy (%)</th>
<th>CR for FC layers</th>
<th>CR for overall network</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG-16 (baseline)</td>
<td>69.1</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>TT-VGG-16</td>
<td>67.8</td>
<td>30.5x</td>
<td>7.4x</td>
</tr>
</tbody>
</table>

#### Table 2: CONV-dominated CNN on CIFAR-10.

<table>
<thead>
<tr>
<th>CONV-dominated CNN [23] (NIPS’17)</th>
<th>Accuracy (%)</th>
<th>CR for CONV layers</th>
<th>CR for overall network</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN (baseline)</td>
<td>90.7</td>
<td>1x</td>
<td>1x</td>
</tr>
<tr>
<td>TT-CNN</td>
<td>89.3</td>
<td>3.5x</td>
<td>3.27x</td>
</tr>
</tbody>
</table>

**Table 3: RNN on Youtube Celebrities Face Data.**

<table>
<thead>
<tr>
<th>RNN [77] (ICML’17)</th>
<th>CR for FC layers</th>
<th>CR for overall network</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM (baseline)</td>
<td>33.2</td>
<td>1x</td>
</tr>
<tr>
<td>TT-LSTM</td>
<td>75.5</td>
<td>15283x</td>
</tr>
<tr>
<td>GRU (baseline)</td>
<td>34.2</td>
<td>1x</td>
</tr>
<tr>
<td>TT-GRU</td>
<td>80.0</td>
<td>11683x</td>
</tr>
</tbody>
</table>

**FC-dominated CNN [50]:** Two FC layers (FC6 and FC7) are in the TT format, where $d=6$, $m_1 \sim m_6=4$, $r_1 \sim r_3=4$. For FC6 and FC7, $n_1 \sim n_6=[2,7,8,8,7,4]$ and $[4,4,4,4,4,4]$, respectively.

**CONV-dominated CNN [23]:** The 2nd ~ 6th CONV layers are in the TT format, where $d=4$, $m=[4,4,4,4]$ and $[3,4,8,4]$ for the 2nd and the 3rd ~ 6th layers, respectively, $n=[4,4,4,4]$ and $[3,4,8,4]$ for the 2nd ~ 3rd and the 4th ~ 6th layers, respectively, $r_1 \sim r_3=[22,20,20,22,22,22]$, $[27,22,22,22,23,23]$ for the 2nd, the 3rd and the 4th ~ 6th layers, respectively.

**LSTM or GRU-based RNN [77]:** All the input-to-hidden layers are in the TT format, where $d=4$, $m=[4,4,4,4]$, $n=[4,20,20,36]$ and $r_2 \sim r_3=4$.

From Table 1 - Table 3 we can see that TT decomposition enables significant reduction in the number of parameters of the decomposed layers and the entire DNN model sizes. Meanwhile, it preserves high task accuracy on different datasets, making it very attractive for practical deployment of DNN models. However, its inherent drawback of low computational efficiency in the inference phase, which will be elaborated in Section 3.1, impedes its wide adoption in practical systems.

### 3 EFFICIENT TT-FORMAT INERENCE: CHALLENGE & SOLUTION

#### 3.1 Challenge of TT-format Inference

As described in Eqn. (2) of Section 2.2, the inference on the TT-format layers of DNN models can be performed via multi-dimensional summation of the products of slices of different tensor cores. This implementation, though classical and straightforward, incurs severe challenge that leads to low computational efficiency.

In general, the low computational efficiency of TT-format inference scheme comes from its inherent redundant computations. Recall that in Eqn. (2), calculating a specific element of output tensor ($Y(i_1, \ldots, i_d)$) requires consecutive multiplications of $G_k[i_k, j_k]$ over all $j_k$’s. Since each $Y(i_1, \ldots, i_d)$ always has the partially same indices $i_k$ with many other $Y(i_1, \ldots, i_d)$’s, calculating those indices-shared elements inherently contains multiple times of consecutive multiplication among the same $G_k[i_k, j_k]$, thereby causing unnecessary computational redundancy. Fig. 4 illustrates the existence of such redundancy in the calculation of 3-dimensional tensor. We see that, the calculation procedure of $Y(0,0,0)$ and $Y(1,0,0)$ have two identical matrix-vector multiplication stages out of all three stages. Notice that Fig. 4 only shows the computational redundancy for these two specific output tensor elements. In general, such replicated multiplications in Eqn. (2) always exist for any pair of tensor elements that shares part of the indices.

To further quantify the computational redundancy, we perform an analytic evaluation on the total number of multiplications consumed in Eqn. (2) and the minimum required number of multiplications.
for calculating all $\mathcal{Y}(i_1, \ldots, i_d)$, respectively. For simplicity, only multiplication is counted for computational cost.

**Analysis on total number of multiplications in Eqn. (2):**

First we examine the total required number of multiplications consumed in Eqn. (2). As indicated in Fig. 4, computing one $\mathcal{Y}(i_1, \ldots, i_d)$ needs $d$ stages of matrix-vector multiplication between a length-$r_i$ and $r_j$-by-$r_{j-1}$ matrix. Therefore, total number of multiplications to calculate all $M\mathcal{Y}(i_1, \ldots, i_d)$’s is:

$$\text{MUL}_{\text{naive}} = MN \sum_{i=1}^{d} r_ir_{i-1}. \quad (3)$$

**Analysis on minimum number of multiplications for $\mathcal{Y}(i_1, \ldots, i_d)$:**

Next we analyze the minimum number of required multiplications for calculating all $\mathcal{Y}(i_1, \ldots, i_d) \in \mathcal{Y}$. The general procedure is to first determine the computational cost for $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ when $i_1 \sim i_{d-1}$ are specific. Based on that, we then determine the number of non-redundant multiplications for calculating $\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}$ when $i_1 \sim i_{d-2}$ are specific. Notice that here all the computations involved with $G_{d}[i_d,i_d]$ is not considered since they have been included before, thereby avoiding counting repeated computation. After that, we can perform the similar analysis from the $(d-2)$-th dimension to the 1-st dimension of $\mathcal{Y}$, and finally we can obtain the minimum required number of multiplications for calculating all $\mathcal{Y}(i_1, \ldots, i_d)$’s as $\mathcal{Y}(i_1, \ldots, i_d)$. In general, such recursive counting method ensures that all the multiplications involved with the calculation of all $\mathcal{Y}(i_1, \ldots, i_d)$’s are included and meanwhile those multiplications are not counted repeatedly.

Specifically, the detail of above analysis procedure is described as follows. First let us consider the computational cost for $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ (referred as stage-1). Recall that Eqn. (2) indicates calculating one $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ requires all $\mathcal{X}(j_1, \ldots, j_{d-1},:)^{4}$’s and $G_k[i_k,j_k]$’s where $k = 1, 2, \ldots, d$. Also, notice that as illustrated in Fig. 4, $\mathcal{X}(j_1, \ldots, j_{d-1})$ only shares one common index $j_k$ with one slice of tensor core $G_k[i_k,j_k]$. Based on these two observations, in order to facilitate the analysis on $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ we further partition the counting procedure into $d$ steps, where in the $k$-th step we consider the involvement of $\mathcal{X}(j_1, \ldots, j_{d-k},:)^{4}$ and $G_k[i_k,j_k]$ for calculating $\mathcal{Y}(i_1, \ldots, i_{d-k},:)^{4}$. The computational cost is $r_d-r_k n_d$. Notice that inherently this cost corresponds to parallelising the $d$-th dimension of input $\mathcal{X}(i_1, \ldots, i_{d-1},:)^{4}$ (as shown in stage-1 of Fig. 5). Interestingly, though such parallelized input scheme does not reduce any computational redundancy involved with $G_k[i_k,j_k]$ (stage-1 in Fig. 5), it saves the computation involved with $G_{d-1}[i_{d-1},i_{d-1}]$ (stage-2 in Fig. 5). This is because now only one, instead of $n_d$ length-$r_i$ vector is multiplied with $G_{d-1}[i_{d-1},i_{d-1}]$. Besides, since all the computations involved with $G_{d-1}[i_{d-1},i_{d-1}]$ has been considered before, the additional computational cost for $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ with specific $i_{d-1}$ is $r_{d-1} r_{d-1} n_d + r_{d-2} r_{d-2} n_d$. Therefore, the number of multiplications of calculating $\mathcal{Y}(i_1, \ldots, i_{d-1},:)^{4}$ with $\mathcal{X}(i_1, \ldots, i_{d-2},:)^{4}$ is $(r_d-r_k n_d + r_{d-2} r_{d-2} n_d) m_{d-1}$. By recursively applying this analysis, we can then derive the number of multiplications for calculating $\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}$ with $\mathcal{X}(i_1, \ldots, i_{d-2},:)^{4}$ as:

$$\text{MUL}_{\text{extra}}(i_1, \ldots, i_{d-2},:) = m_d \sum_{i=1}^{d} (r_ir_{i-1}) \prod_{l=1}^{i} n_l. \quad (4)$$

Next we consider the additional computational cost for calculating $\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}$ (referred as stage-2). Similar to the previous analysis on recursive computation, when computing $\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}$, the computation involved with $\mathcal{Y}(i_1, \ldots, i_{d-2},i_{d-1},:)^{4}$ has already been considered and should not be re-counted again. Therefore, the additional number of multiplications for calculating $\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}$ with $\mathcal{X}(i_1, \ldots, i_{d-2},:)^{4}$ is:

$$\text{MUL}_{\text{extra}}(\mathcal{Y}(i_1, \ldots, i_{d-2},:)^{4}) = (m_{d-1} m_d - m_d) \sum_{i=1}^{d-1} (r_ir_{i-1}) \prod_{l=1}^{i} n_l. \quad (5)$$
By generalizing Eqn. 5, we can derive the additional number of multiplications for calculating \( \mathbf{Y}(i_1, \ldots, i_l, \ldots) \) with \( \mathbf{X}(\cdot, \cdot, \cdot) \) in stage-1 as:

\[
MUL_{\text{extra}} \text{ for } \mathbf{Y}(i_1, \ldots, i_l, \ldots) = \left( \prod_{j=1}^{d} m_j - \prod_{j=l+1}^{d} m_j \right) \sum_{i=1}^{l} (r_{i} r_{i-1}^{-1}) \prod_{t=1}^{i} n_t )
\]

\[
= (m_l - 1) \prod_{j=l+1}^{d} m_j \sum_{i=1}^{l} (r_{i} r_{i-1}^{-1}) \prod_{t=1}^{i} n_t )
\]

(Eqn. 7) gives the analytical result of minimum number of multiplications for performing TT-format inference. Comparing this theoretical limit with Eqn. 3, we can find that the conventional TT-format scheme contains significant computational redundancy. For instance, for the FC-6 layer in VGG-16 with \( d=6 \) and \( r_1 = 4 \), the number of multiplications consumed in Eqn. 3 is 1073 times than that in Eqn. 7. Obviously, such huge redundancy in multiplication caused very low computational efficiency.

### 3.2 Compact TT-format Inference Scheme

To address the challenge of low computational efficiency, we propose a novel computation-efficient TT-format inference scheme. With carefully designed computation procedure, our approach, namely compact inference scheme, calculates all the elements of output tensor \( \mathbf{Y} \) in parallel without any redundant computations, thereby significantly improving computational efficiency over the conventional TT-format inference scheme.

In general, the design of this compact inference scheme is inspired by the theoretical analysis on the minimum required number of computations in Section 3. Recall that in the previous analysis the minimum number of multiplications is counted based on the assumption that all the computations involved with \( \mathbf{G}_k[l_k, j_k] \) are not included for the future computations involved with \( \mathbf{G}_{k-1}[l_{k-1}, j_{k-1}] \).

To achieve this, in our design we perform the computation on different \( \mathbf{G}_k \)’s one by one. In other words, different from Eqn. 2 that calculates one output tensor element using \( d \mathbf{G}_k[l_k, j_k] \)'s where \( k = 1, 2, \ldots, d \), our scheme performs computation using all \( m_k n_k \mathbf{G}_k[l_k, j_k] \)'s with only one specific \( k \), and then will never use them in the future for other \( k \)’s. Consequently, such computing arrangement breaks the original data dependency and eliminates all the potential computational redundancy.

**Algorithm 1: Compact TT-format Inference Scheme.**

**Input:** \( \mathbf{X}, \mathbf{G}_1, \ldots, \mathbf{G}_d, m = [m_1, \ldots, m_d], n = [n_1, \ldots, n_d], r = [r_0, r_1, \ldots, r_d] \)

**Output:** \( \mathbf{Y} \)

1. \( \mathbf{X}' = \text{Reshape} (\mathbf{X}, [n_d, -1]) \)
2. \( \mathbf{V}'_{d+1} = \mathbf{X}' \)
3. for \( h = d \) to 1 do
4.   \( \mathbf{V}_h = \text{MatMul} (\mathbf{G}_h , \mathbf{V}'_{h+1}) \)
5.   \( \mathbf{V}'_h = \text{Transform}(\mathbf{V}_h, h) \)
6. **Function Transform**(\( \mathbf{V}, h \))
7. \( \mathbf{V}' = \text{Transpose}(\mathbf{V}) \)
8. \( \mathbf{V}' = \text{Reshape}(\mathbf{V}', [n_{h-1}, -1]) \)
9. // split
10. \( \mathbf{t}' = \text{new} [n_{h-1}, r_{h-1}] \)
11. \( \mathbf{T}' = \text{new} [\prod_{k=1}^{h-2} n_k, \prod_{k=h}^{d} m_k] \mathbf{t}' \)
12. for \( j = 1 \) to \( \prod_{k=1}^{h-2} n_k, \prod_{k=h}^{d} m_k \) do
13.  \( \mathbf{T}'[j] = \mathbf{V}'[:,(j-1)*r_{h-1} + r_{h-1}] \)
14.  \( \mathbf{T}'[j] = \text{Reshape}(\mathbf{T}'[j], [n_{h-1}, r_{h-1}]) \)
15. // assemble
16. \( \mathbf{V}' = \text{new} [n_{h-1}, r_{h-1}, \prod_{k=1}^{h-2} n_k, \prod_{k=h}^{d} m_k] \)
17. for \( j = 1 \) to \( \prod_{k=1}^{h-2} n_k, \prod_{k=h}^{d} m_k \) do
18.  \( \mathbf{V}'[:,j] = \mathbf{T}'[j] \)
19. Return \( \mathbf{V}' \)

Fig. 6 illustrates the key idea of the proposed compact inference scheme. Recall that as analyzed in Section 3.1 and shown in
Fig. 5, partial parallel input multiple $X(j_1, \ldots, j_{d-1}, j_d)$s as $X(j_1, \ldots, j_{d-1}, j_d)$ reduces redundant computations involved with $G_{d-1}[j_{d-1}, j_d]$. To be consistent with this, we further fully parallel the computation involved with all the input $X(j_1, \ldots, j_{d-1}, j_d)$s (as $X(\ldots, \ldots)$) and all the slices of $G_d$ (see Fig. 6(a)). As revealed in Fig. 6, such parallel computation corresponds to a compact matrix-format multiplication that replaces the original summations in Eqn. 2 over index $j_d$. Different from Fig. 5, the computations in stage-1 of Fig. 6(a) are now for every $X(j_1, \ldots, j_{d-1}, j_d)$ and every slice of $G_d$; therefore the input tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ needs to be properly transformed to a new matrix format to guarantee the functional validity and suited for matrix multiplication. In general, such transform converts a tensor $X \in \mathbb{R}^{n_1 \times n_2 \times \ldots \times n_d}$ to a matrix $X' \in \mathbb{R}^{n_d \times \prod_{i=1}^{d-1} n_i}$, and the mapping principle for this transform is as follows:

$$X(j_1, \ldots, j_{d-1}, j_d) \rightarrow X'(p, q). \quad (8)$$

where $p = j_d$ and $q = \sum_{i=1}^{d-1} j_i \prod_{i=1}^{d-1} n_i$. Then, a compact matrix-format multiplication can be performed as:

$$V_d = \bar{G}_d X'. \quad (9)$$

where $V_d$ is the intermediate matrix to be sent to stage-2, and $\bar{G}_d$ is the matrix format of unfolded $G_d$ (see Fig. 6(a)).

It should be noted that Fig. 6(a) only performs compact matrix-format computation in stage-1; while in stage-2 and stage-3 $G_2[j_2, j_3]$ and $G_3[j_1, j_2]$ are still fetched for processing in serial, thereby causing redundant computations. As analyzed in Section 3.1, avoiding the redundant computations in each computing stage demands the involvement of all the output values from previous stages (e.g. $V_3$) and the matrix format of the unfolded tensor core (e.g. $\bar{G}_2$). However, as illustrated in Fig. 6, $G_2$ and $V_3$, or $G_1$ and $V_2$, cannot be simply multiplied because 1) their sizes do not fit for direct matrix multiplications; and 2) the elements of $V_h$ as the intermediate values from stages-$(d - h + 1)$, and $\bar{G}_{d-1}$, as the matrix format of unfolded tensor core in the stage-$(d - h + 2)$, are not in the correct format for operation. Therefore, these operations can be fully parallelized as presented in Fig. 6(b).
To address this problem, we propose to transform $V_h$ before it is multiplied with $\tilde{G}_{h-1}$. In general, in the stage-$(d - h + 1)$ such transform is to convert $V_h \in \mathbb{R}^{(m_1 \times n_1 - 1) \times (n_2 - 1) \times \cdots \times (m_{d-k} - 1)}$ to $V'_h \in \mathbb{R}^{(n_1 - 1) \times (n_2 - 1) \times \cdots \times (n_{d-k} - 1)}$, and the mapping principal for this transform is as follows:

$$V'_h(p, q) = \sum_{k=2}^{d-k+1} \sum_{g=2}^{d-h+1} \sum_{i=2}^{h-2} \sum_{j=1}^{i-1} m_d \cdots m_2 m_1,$$

where $p = i_k h_k h_{k-1} + h_{k-1} + 1$, $q' = i_k h_k h_{k-1} + 1$, $\gamma = \sum_{l=2}^{h-1} \sum_{j=1}^{i-1} n_i \prod_{k=1}^{d-h} m_{d-k+1}$.

According to this scheme, two working SRAMs are used to avoid the multiplication between $V_h$ and corresponding $\tilde{G}_{h-1}$. By using this scheme, all the elements of final output tensor $_f$ can be obtained simultaneously at the output end of stage-1 without any redundant computations. Notice that for practical implementation, the transformation described in Eqn.2 can be equivalently achieved by performing 4-step matrix-wise multiplications (see Transform in Fig. 6(b)). Putting all together, the proposed compact TT-format inference scheme is described in Algorithm 1.

Notice that at each stage of compact TT-format inference scheme, the intermediate values should be buffered on-chip for the processing of next stage. Needed extra storage capacity to store the intermediate values from state-$(d - h + 1)$ is max$(\gamma_{h-1} \prod_{k=1}^{d-h} n_k) \prod_{k=h}^{d} m_k$, where $h = 1 \ldots d$. Consider that both the input and output of each stage should be stored. The overall storage overhead is $2 \times \max(\gamma_{h-1} \prod_{k=1}^{d-h} n_k) \prod_{k=h}^{d} m_k$, where $h = 1 \ldots d$. However, consider that the activation size is much less than weights size in typical, the storage overhead brought by compact TT-format inference scheme can be accepted.

## 4 HARDWARE ARCHITECTURE

### 4.1 Data Mapping and Processing Scheme

Fig. 6 shows the overall computing flow of the proposed TIE. In general, it contains two types of operation: reshaping the inputs ($x$ to $x'$ and $V_h$ to $V'_h$), and multiplying $V'_h$ and $\tilde{G}_{h-1}$. Considering reshaping $x$ to $x'$ can be prepared offline and reshaping $V_h$ to $V'_h$ can be performed by carefully designed memory access scheme in Section 4.4, the datapath of TIE is mainly responsible for executing matrix multiplication. Fig. 10 illustrates the detailed data mapping and processing scheme of an example 2-PE datapath for the multiplication between $3 \times 2 \tilde{G}_{h-1}$ and $2 \times 4 V'_h$ matrices. Here, each PE is equipped with $3 \times$ multiply-accumulate (MAC) units. In each clock cycle, one column of $\tilde{G}_{h-1}$ is broadcast to all the PEs, where each multiplier of PE receives one element of the column. Meanwhile, two elements in the same row of $V'_h$ are sent to two PEs, respectively; and each of these elements in $V'_h$ is broadcast to all the multipliers of their corresponding PEs. After finishing the computation in the current cycle, in the next cycle PEs will move on to process the next column of $\tilde{G}_{h-1}$ and next row of $V'_h$. In general, with $N_{PE}$ PEs equipped with $N_{MAC}$ MAC units, the proposed processing scheme can produce a $N_{MAC} \times N_{PE}$-size sub-block of the result matrix $V'_h \Rightarrow \tilde{G}_{h-1} V'_h$ in $N_{col}$ cycles, where $N_{col}$ is the number of columns of $\tilde{G}_{h-1}$. Notice that when the number of rows of $\tilde{G}_{h-1}$ (as $N_{row}$) is larger than $N_{MAC}$ or number of columns of $V'_h$ (as $N_{col}$) is larger than $N_{PE}$, it takes PEs multiple $N_{col}$ cycles to calculate the entire $V'_h$ (see Fig. 7).

### 4.2 Overall Architecture

Based on the data mapping and processing scheme described above, the overall architecture of TIE is shown in Fig. 8. For the inference task on one layer, the datapath of TIE performs $d$-stage matrix multiplications between matrix $V'_h$ read from the working SRAM and $\tilde{G}_{h-1}$ read from the weight SRAM. During the computation in each stage, as indicated in Section 4.1, part of the results matrix $V_{h-1} \Rightarrow \tilde{G}_{h-1} V_{h-1}$ are already calculated in the PEs, these sub-block of $V_{h-1}$, once available, will be written to another working SRAM. According to this scheme, two working SRAMs are used to avoid potential read/write conflict. After the entire $V_{h-1}$ has been written to one working SRAM, in order to continue the next stage of computation correctly, this SRAM will then output $V'_{h-1}$ as the reshaped $V_{h-1}$, to the datapath via a specifically designed memory read scheme (described in Section 4.4). Therefore, the two working SRAMs act as source and destination memories, respectively, and exchange their roles for every stage. Notice that during the last stage of computations for $V_1$, the calculated elements of $V_1$ need to be sent to activation units first and then written to working SRAM.

### 4.3 Datapath & Weight SRAM

**Datapath.** As shown in Fig. 8, TIE consists of an array of $N_{PE}$ PEs that perform matrix multiplication. Specifically, each PE contains $N_{MAC}$ MAC units and $N_{MAC}$ activation units. Notice that by using the processing scheme in Section 4.1, the $N_{MAC} N_{PE}$ elements of result matrix are available simultaneously in the registers of all PEs every $N_{col}$ cycles, and then they will be written to working SRAM in parallel.

**Weight SRAM.** The weight SRAM of TIE stores the weight parameters of tensor cores $\tilde{G}_h$’s. Notice that though each layer of TT-format DNN models is affiliated with $d$ tensor cores, the sequential access to different $\tilde{G}_h$’s in different computation stages, which is described in Section 3.2, enables the simple storing strategy of locating all $\tilde{G}_h$’s in the same weight SRAM sequentially from $h = 1$ to $d$. However, different from such sequential placement for the consecutive $\tilde{G}_h$’s, the data allocation within the same $\tilde{G}_h$ may be not always sequentially in the weight SRAM. For instance, as illustrated in Fig. 9, when the number of rows of $\tilde{G}_h$’s is larger than the number
of PEs, in order to be consistent with processing scheme described in Section 4.1, the elements in the same column of $G_h$ need to be stored in the different row of weight SRAM via an interleaved way. In short, the entire data allocation of weight SRAM is sequential at the inter-$G_h$ level and interleaved at the intra-$G_h$ level.

Figure 9: Data allocation in weight SRAM.

Algorithm 2: Data Read Scheme for Working SRAM.

Input: Memory[$N_g$, $N_r$, $M$]. $N_g$ component SRAMs are divided into $N_g$ groups. Each group contains $N_r$ component SRAMs. Each component SRAM contains $M$ elements. $N_{PE}$ denotes the number of PE.

Output: $Data$

//Number of total cycles for reading data
$N_c = \lfloor \frac{N_r M}{N_{PE} N_g} \rfloor$

while ($k < N_c$) do
  for $j = 1$ to $\lfloor \frac{M}{N_{PE} N_g} \rfloor$ do
    for $i_g = 1$ to $\frac{N_r}{N_{PE}}$ do
      $Data = \text{Read}(\text{Memory}(i_r, j * \lfloor \frac{M}{N_{PE} N_g} \rfloor : (j + 1) * \lfloor \frac{M}{N_{PE} N_g} \rfloor))$
      $Data = \text{ReArrange}(Data)$
  end for
  $k++$
end while

Function ReArrange($Data$)

$Data' = \text{new}[N_g * \lfloor \frac{M}{N_{PE} N_g} \rfloor]$

for $j = 1$ to $\lfloor \frac{M}{N_{PE} N_g} \rfloor$ do
  for $i_g = 1$ to $N_g$ do
    $Data'[j * N_g + i_g] = Data(i_g, j)$
  end for
end for

Return $Data'$

4.4 Working SRAM

As indicated in Algorithm 1, a transform from $V_h$ to $V_0^*$ is required in each stage of computation to guarantee the functional correctness of the proposed inference scheme. Conventionally, such transform, including matrix reshape and transpose, demands extra memory resource to implement those matrix operations [9], thereby significantly degrading the hardware performance of entire TIE design on both area efficiency and power efficiency.
To address such problem, we carefully design efficient read and write schemes for working SRAMs to achieve zero-cost matrix transform. Our general methodology is to ensure that the datapath always reads the required elements of $V_h'$ from the working SRAM that stores $V_h$, thereby enabling on-the-fly matrix transform for $V_h$. To achieve this, the key idea is to partition working SRAM to multiple groups with well-designed data selection mechanism. Next we describe the proposed schemes in detail.

**Writing scheme.** Recall that in the proposed computing scheme (Section 4.1) each PE calculates $N_{MAC}$ elements in the same column of $V_h$ after every $N_{Gcol}$ cycles. To make data allocation in the working SRAM consistent with the corresponding matrix format of $V_h$, different PEs assemble the calculated elements in the same positions of MAC units together and write them to one row of component SRAMs. In general, during the writing phase the calculated elements in the $t$-th MAC units among different PEs form one row of data to be written to memory. Notice that as mentioned before, each of the two working SRAMs is partitioned to multiple groups, where each group contains multiple component SRAMs. Based on this type of memory organization, multiple columns of $V_h$ can be written to multiple component SRAMs simultaneously without access conflict.

**Reading scheme.** As mentioned before, the matrix transform operation on $V_h$ is performed during the reading phase. To achieve that, we design a partitioned group-based data selection mechanism. Algorithm 2 describes the proposed mechanism in detail. Here the key idea of this mechanism is to utilize the indices of SRAM groups, component SRAMs and element to locate and read the targeted element of $V_h'$ in a mathematically equivalent manner.

Fig. 10 illustrates the working SRAM reading scheme based on the proposed data selection mechanism. It is seen that in each cycle the required elements of $V_h'$ can be located and read from the rows of different component SRAMs of memory group. As shown in Fig. 10, after being assembled, these data form the row vector of targeted $V_h'$, and then they will be distributed to their corresponding PEs for calculating $V_{h+1} = \mathcal{G}_{h+1}V_h$.

Notice that besides the transform from $V_h$ to $V_h'$, the inference on entire DNN models also require transform from $V_1$ of this layer to $X'$ of next layer. Interestingly, our mathematical analysis shows that such inter-layer transform is identical to the intra-layer transform described before. Therefore, when the TIE is performing the computation between two consecutive layers, it will still utilize the proposed working SRAM read scheme.

**5 Evaluation**

**5.1 Experimental Methodology**

**Simulation and CAD Tools.** The high-level functional behavior of TIE was modeled by a bit-accurate cycle-accurate simulator. Based on that, we developed the RTL model using Verilog and verified its functional validity. Then, the verified RTL model was synthesized using Synopsis Design Compiler with CMOS 28nm library. Here the gate-level netlist was annotated with toggle rate that was obtained from the extracted switching activity during the simulation. After that we used Synopsis IC Compiler to perform place and route and generate layout (see Fig. 11). Then, Synopsis Prime-Time PX was used to estimate power consumption. Notice that the area and power of memory part were reported by Cacti.

**Benchmarks.** To evaluate the performance of TIE on different tasks, we choose several workloads from two models used in image classification and video classification tasks, respectively. Here the same-size layers with different TT-decomposition setting are viewed as different workloads. Table 4 lists the information of four benchmark layers, including the size, TT-decomposition settings ($d$, $n$, $m$, and $r$) and compression ratio.

**5.2 Hardware Performance**

**Design Configuration.** Table 5 shows the configuration information of the TIE hardware. The entire design consists of 16 PEs with 16-bit quantization. For each PE, it is equipped with 16 MACs and 16 activation units, where each MAC contains one 16-bit width multiplier and one 24-bit width accumulator. Regarding the memory, a 16 KB Weight SRAM is used to store up to 8192 16-bit weights on the chip. According to Section 2.3, such budgeted capacity for weight SRAM is sufficient for most TT-DNN models. For working SRAM, it contains two copies acting as ping-pong buffer, where each copy has the capacity as 384KB. Therefore the total capacity of working RAM is $384 \times 2 = 768$KB.

**Hardware Resources and Performance.** Fig. 11 shows the overall hardware source and performance of TIE design. Operating on 1000 MHz, the 16-PE TIE occupies 1.74mm$^2$ silicon area and consumes 154.8mW power. Notice that all the memory used in TIE are on-chip SRAM due to the high compression ratio brought by TT decomposition. The area and power breakdown is shown in Table 6.

**5.3 Comparison with EIE, CIRCNN, and Eyeriss**

In this subsection, we compare TIE with two state-of-the-art compressed DNN-targeted accelerators: EIE [26] and CIRCNN [18]. Different from TIE, model compression in EIE and CIRCNN comes from other sources: For EIE, model compression is achieved via network sparsification; for CIRCNN, model compression is from structuring topology. Moreover, to evaluate the performance of TIE on CNN layers, we also compare TIE with representative CNN-oriented work: Eyeriss [12].

**Comparison with EIE.** Table 7 summarizes the design parameters and hardware performance of EIE and TIE. Due to the different technology nodes adopted in the two works, the clock frequency, silicon area and power consumption of EIE are also projected under the same 28nm technology for fair comparison. Such projection is based on the scaling rule used in [26]: linear, quadratic and constant scaling for frequency, area and power, respectively.

Fig. 12 compares the hardware performance of EIE and TIE on two benchmarks (VGG-FC6 and VGG-FC7) in terms of throughput, area efficiency and energy efficiency. We see that TIE can achieve a comparable throughput with EIE. More importantly, thanks to the high compression ratio brought by TT decomposition, TIE achieves $7.22 \times 10.66 \times$ better area efficiency and $3.03 \times 4.48 \times$ better energy efficiency on different workloads, respectively.

**Comparison with CIRCNN.** Table 8 compares the hardware performance of CIRCNN and TIE. Notice that here the listed performance metrics of the two designs are obtained from their synthesis reports for fair comparison since CIRCNN reports synthesis results.
Meanwhile, due to the lack of area information of CIRCNN, we compare the overall throughput (in term of TOPS) and energy efficiency (in term of TOPS/W) of the two designs. After projecting performance of CIRCNN to the same 28nm technology for fair comparison, it is seen that TIE achieves 5.96× and 4.56× higher throughput and energy efficiency than CIRCNN, respectively.

Comparison with Eyeriss. Table 9 summarizes the design parameters and hardware performance of Eyeriss and TIE on CONV layers of VGG. For fair comparison, the clock frequency, silicon area and power consumption of Eyeriss are also projected under the 28nm technology.
5.4 Flexibility
TIE is designed to provide sufficient flexibility to support the needs of different TT models having different layer sizes and decomposition settings. As illustrated in Fig. 13, different workloads with different \( d, m, n \) and \( r \) can be executed on the same TIE accelerator efficiently in a flexible way. In addition, we also investigate the throughput with different \( r \)’s for the same workload (see Fig. 13). Here the change of \( r \) is specifically studied since it is an important metric to provide flexible control the compression and acceleration effect of TT decomposition. From the figure we can see that TIE exhibits great flexibility to support the flexibility for this important TT decomposition parameter.

Table 9: Comparisons of Eyeriss and TIE on VGG CONV layers.

<table>
<thead>
<tr>
<th>Design</th>
<th>Eyeriss</th>
<th>TIE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOS Tech.</td>
<td>65nm (reported)</td>
<td>28nm (projected)</td>
</tr>
<tr>
<td>Freq (MHz)</td>
<td>200</td>
<td>464</td>
</tr>
<tr>
<td>Quantization</td>
<td>16-bit</td>
<td>16-bit</td>
</tr>
<tr>
<td>Area (mm(^2))</td>
<td>12.25</td>
<td>2.27</td>
</tr>
</tbody>
</table>
| Power (mW) | 2.36 | 2.36 | 3 (\%)

Throughput (Frame/s) | 0.8 | 1.86 | 6.72 (3.61x) |

Area Efficiency (Frame/s/W) | 0.065 | 0.82 | 3.36 (4.71x) |

Energy Efficiency (Frame/s/mm\(^2\)) | 3.39 | 7.89 | 39.5 (5.01x) |

We used core area and processing latency of Eyeriss instead of chip area and total latency for fair comparison with TIE.

6 RELATED WORKS
Designing specialized DNN accelerators has become an active research topic of the architecture community. In the last four years a series of DNN processors and the corresponding instruction sets are proposed in Diannao family [10, 11, 19, 42, 80] and Google’s TPU [34]. Also, in [4, 37, 43], efficient and flexible dataflows are investigated and proposed for DNN accelerator to reduce the memory access and hence improve energy efficiency. On the other hand, some works [13, 22, 36, 39, 58, 71] focus on improving the efficiency of memory system in the DNN accelerators.

The performance of DNN hardware can also be improved by designing accelerators [3, 6, 12, 16, 31, 38, 55, 78] utilizing the sparsity of the network model. Besides, some other computation/memory-reducing approaches are investigated in [1, 53, 60, 68] to facilitate the efficient hardware design. Bit-serial computation-based accelerators [2, 20, 30, 37, 57, 63, 76] are another type of energy-efficient DNN hardware that provides scalable performance with different precisions.

Besides the aforementioned ASIC designs, FPGA-based implementations are widely investigated in [40, 48, 49, 54, 62, 64, 65, 82]. Also, analog or mixed signal-based design, proposed in [7, 8, 21, 41, 61, 66, 70], is another potential solutions for energy-efficient DNN accelerators. In particular, [32] proposes a RRAM-based TT-format DNN accelerator, which is the most relevant work of TIE. Recently, considering the training phase of DNN is very time-consuming, [14, 45, 59, 67, 72] investigate efficient hardware solutions for DNN training.

7 CONCLUSION
This paper develops a computation-efficient inference scheme for TT-format DNN, and accordingly develops TIE, a TT-format compressed DNN-targeted inference engine. The highly flexible architecture completely eliminate redundant computation and memory accesses. A 16-processing elements (PE) prototype is implemented using CMOS 28nm technology. Operating on 1000MHz, the TIE accelerator consumes 1.74mm\(^2\) and 154.8mW. Compared with EIE, TIE achieves 7.22\(\times\)~10.66\(\times\) better area efficiency and 3.03\(\times\)~4.48\(\times\) better energy efficiency on different workloads, respectively. Compared with CIRCN, TIE achieves 5.96\(\times\) and 4.56\(\times\) higher throughput and energy efficiency, respectively. The results show that TIE exhibits significant advantages over state-of-the-art solutions.
TIE: Energy-efficient Tensor Train-based Inference Engine for Deep Neural Network

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

REFERENCES


