ETTE: Efficient Tensor-Train-based Computing Engine for Deep Neural Networks

Yu Gong*
Miao Yin*
Rutgers University
New Jersey, USA
yu.gong@rutgers.edu
miao.yin@rutgers.edu

Lingyi Huang
Rutgers University
New Jersey, USA
lingyi.huang@rutgers.edu

Jinqi Xiao
Rutgers University
New Jersey, USA
jinqi.xiao@rutgers.edu

Yang Sui
Rutgers University
New Jersey, USA
yang.sui@rutgers.edu

Chunhua Deng
ScaleFlux, Inc.
California, USA
chunhua.deng518@gmail.com

Bo Yuan
Rutgers University
New Jersey, USA
bo.yuan@soe.rutgers.edu

ABSTRACT

Tensor-train (TT) decomposition enables ultra-high compression ratio, making the deep neural network (DNN) accelerators based on this method very attractive. TIE, the state-of-the-art TT based DNN accelerator, achieved high performance by leveraging a compact inference scheme to remove unnecessary computations and memory access. However, TIE increases memory costs for stage-wise intermediate results and additional intra-layer data transfer, leading to limited speedups even the models are highly compressed.

To unleash the full potential of TT decomposition, this paper proposes *ETTE*, an algorithm and hardware co-optimization framework for **E**fficient **T**ensor-**T**rain **E**ngine. At the algorithm level, ETTE proposes new tensor core construction and computation ordering mechanism to reduce stage-wise computation and storage cost at the same time. At the hardware level, ETTE proposes a lookahead-style across-stage processing scheme to eliminate the unnecessary stage-wise data movement. By fully leveraging the decoupled input and output dimension factors, ETTE develops an efficient low-cost memory partition-free access scheme to efficiently support the desired matrix transformation.

We demonstrate the effectiveness of ETTE via implementing a 16-PE hardware prototype with CMOS 28nm technology. Compared with GPU on various workloads, ETTE achieves 6.5× – 253.1× higher throughput and 189.2× – 9750.5× higher energy efficiency. Compared with the state-of-the-art DNN accelerators, ETTE brings 1.1× – 58.3×, 2.6× – 1170.4× and 1.8× – 2098.2× improvement on throughput, energy efficiency and area efficiency, respectively.

*Both authors are co-first authors.

ISCA '23, June 17–21, 2023, Orlando, FL, USA

ACM 979-8-4007-0095-8/23/06…$15.00
https://doi.org/10.1145/3579371.3589103

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**.

KEYWORDS

tensor decomposition, neural networks, low rank, accelerator

ACM Reference Format:
Yu Gong, Miao Yin, Lingyi Huang, Jinqi Xiao, Yang Sui, Chunhua Deng, and Bo Yuan. 2023. ETTE: Efficient Tensor-Train-based Computing Engine for Deep Neural Networks. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23), June 17–21, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3579371.3589103

1 INTRODUCTION

Deep neural networks (DNNs) have been widely adopted in many important artificial intelligent (AI) applications. Considering in practice the sizes of large-scale neural networks typically need to be reduced to fit for the edge and mobile applications[42–46, 56], *model compression* strategy has been popularly performed to generate the low-cost and lightweight networks. To date, many model compression techniques [1, 5, 6, 12, 15, 19, 20, 23, 26, 27, 35, 37, 38, 40, 50, 57, 59, 61]have been widely adopted for the practical DNN deployment.

Among various compression techniques, tensor-train (TT) decomposition is a very unique and attractive solution. Fundamentally different from pruning and quantization, TT decomposition aims to minimize the network structure-level redundancy via exploring the low-tensor-rankness of the entire models. Rooted in the mathematically rigid tensor theory, such low-tensor-rank-based compression philosophy, by its nature, can provide very high compression ratio. As reported in various NeurIPS/ICML/CVPR works [14, 33, 34, 48, 51], TT decomposition enables 100× ~ 50000× size reduction for LSTM and CNN models in video, audio and image domains. Very recently, AI researchers further discover the promising benefits of applying TT decomposition to the emerging large-scale models. For instance, Meta [54] (best paper award in MLSys'21) performs TT decomposition to their ultra-large DLRM models in the recommendation systems, enjoying 100× model size reduction with negligible performance loss. Amazon [55] (KDD'22) further applies TT decomposition for compressing large embedded tables in graph

neural networks, reducing the size by up to 81,362×. Considering the language models in NLP continues to scale, NLP researchers also use TT decomposition to compress NLP transformers, bringing more than 40× parameter reduction [39] (ICLR'22).

Such outstanding compression performance and unique computing scheme of TT-format models, by their nature, call for the efficient architectural support. More specifically, the accelerator design of TT-oriented DNN is motivated by three main reasons. First, the promising compression performance of TT decomposition, if properly supported by underlying hardware, can be translated to very high hardware performance and outperform the other compression approaches (e.g., pruning) and their corresponding accelerators. For instance, as reported in Fig. 11, our proposed TT decomposed DNN accelerator shows significant performance improvement than sparsity-aware Transformer accelerators. Second, because TT-format processing scheme has its own unique computing pattern and challenges, the micro-architecture of the existing general-purpose GPU and DNN accelerators, which are designed for uncompressed model and other compression formats such as sparse DNNs, cannot well support the efficient execution of TT-format models or fully unleash their potential. Third, because a DNN weight layer, no matter its specific type, is mathematically a tensor, TT decomposition, as a powerful tool directly operated on tensor, unifies the computations of different DNN types in the same format. Therefore, a TT-oriented accelerator naturally supports a wide spectrum of DNN models, e.g., LSTM, Transformer, DLRM, etc. As reported in Fig. 11, our proposed TT-oriented hardware can provide acceleration of different DNN types; while many existing DNN accelerators are customized for a specific type.

To date the most representative TT-oriented accelerator is TIE [9] (ISCA'19), and its architecture is further adopted in a DNN ASIC chip [17] (ISSCC'21). By customizing dataflow and memory access to TT-decomposed DNNs, TIE significantly improving energy efficiency and execution speed than the straightforward implementation. However, TIE is still not an ideal hardware solution for accelerating TT decomposed DNNs because of *three limitations*. First, TIE only provides limited speedup for the highly compressed TT-format models. The computational saving brought by TT decomposition is not as promising as the storage reduction. In other words, there exists a gap between the compression ratio and FLOPs reduction ratio for a TT-format model. Consequently, though TIE indeed eliminates the unnecessary computations that the straightforward design suffers, its *necessary computational cost* is still not as low as the corresponding storage cost, limiting the practical speedup it can achieve.

Second, TIE consumes increasing memory costs for intermediate results. To be specific, when TIE executes a TT decomposed FC or CONV *layer*, it essentially splits the entire computation into multiple *stages*, where the output size of each stage can be much larger than the size of the final output of this layer. For instance, for the VGG-FC7 layer evaluated in TIE, the final output is only a length-4096 vector; while the intermediate outputs of its stage-wise computations are six 1024-by-16 matrices. Because now the intra-layer results have much larger sizes than the inter-layer results, on-chip memory cost significantly increases, degrading original ultra-low storage benefits brought by TT decomposition.

Third, TIE causes additional intra-layer data transfer. In general, for a d-stage TT decomposed layer, the intermediate results output from each stage need to be written to the memory and then read out again for d times; while for the uncompressed layer only one-time read/write access to the output data is needed. In other words, though TIE indeed significantly reduces the data movement for model weights, it meanwhile brings additional memory access to intermediates results. Evidently, such extra intra-layer data transfer limits the overall improvement on energy efficiency.

To address these challenges and fully unleash the potential of TT decomposition, this paper propose *ETTE*, an Efficient Tensor-Train Engine for real-time low-power DNN acceleration. ETTE is built on algorithm and hardware co-optimization to systemically overcome the limitations of TIE. At the algorithm level, we first analyze and identify the root cause for the drawbacks of the existing TT-format DNN execution – the tensor core construction format and computing order. Based on this key observation, we develop a new tensor core construction mechanism and computing scheme, simultaneously reducing the FLOP count and memory costs for intermediate results. At the hardware level, we develop the corresponding architecture to fully reap these algorithmic benefits. We first propose a lookahead-style across-stage processing scheme to eliminate the unnecessary stage-wise data movement. Built on the top of smaller size of intermediate results, this new dataflow brings very significant reduction in the intra-layer data transfer. Then we develop a memory partition-free access scheme to efficiently realize the complicated matrix transformation operation.

We implement a 16-PE ETTE design example with CMOS 28nm technology. Operated on 1000MHz, this hardware prototype has 1.25mm^2 area consumption and 135.6mW power consumption. Compared with GPU on various workloads, ETTE achieves 6.5× – 253.1× higher throughput and 189.2× – 9750.5× higher energy efficiency. Compared with the state-of-the-art DNN accelerators, ETTE brings 1.1× – 58.3×, 2.6× – 1170.4× and 1.8× – 2098.2× improvement on throughput, energy efficiency and area efficiency, respectively.

2 BACKGROUND

2.1 TT-format DNN Computation

In general, TT decomposition[33][57] aims to explore the structural low-tensor-rankness of DNNs to remove model redundancy. To be specific, given the weight matrix $W \in \mathbb{R}^{I \times O}$ of a layer with input vector $x \in \mathbb{R}^I$ and output vector $y \in \mathbb{R}^O$ where $I = \prod_{n=1}^{d} I_n$ and $O = \prod_{n=1}^{d} O_n$, TT decomposition factorizes the original weight matrix W into a set of small *tensor cores* $\{\mathcal{G}_n\}_{n=1}^{d}$ with $\{R_n\}_{n=1}^{d}$ as *TT-ranks*, and the corresponding TT-format computation can be performed as:

$$Y(i_1, \ldots, i_d) = \sum_{j_1, \ldots, j_d} \mathcal{G}_1[i_1, j_1]\mathcal{G}_2[i_2, j_2], \ldots, \mathcal{G}_d[i_d, j_d]X(j_1, \ldots, j_d), \quad (1)$$

where $Y \in \mathbb{R}^{O_1 \times O_2, \ldots, \times O_d}$ and $X \in \mathbb{R}^{I_1 \times I_2, \ldots, \times I_d}$ are the reshaped high-order tensor format of vector y and x, respectively. Meanwhile, $\mathcal{G}_n[i_n, j_n]$ represents the R_n-by-R_{n+1} element matrix of the 4-order $\mathcal{G}_n \in \mathbb{R}^{R_{n-1} \times I_n \times O_n \times R_n}$.

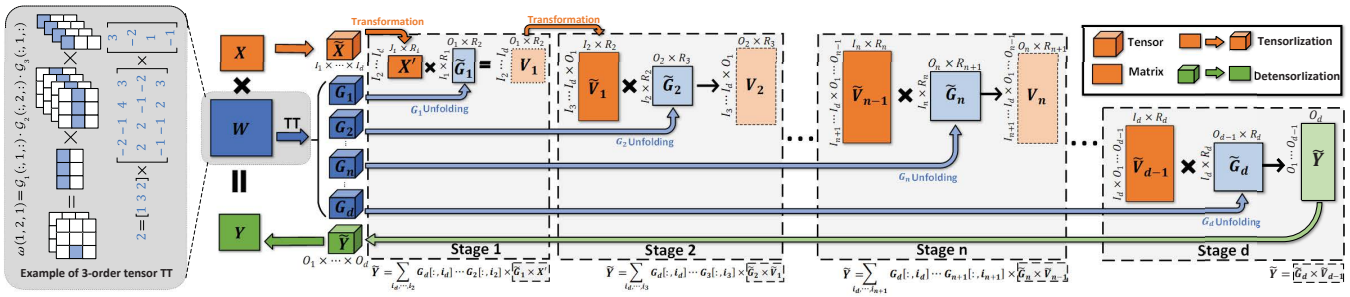


Figure 1: Example computing scheme of TIE. The sizes of $\mathcal{V}_n$, $X^{'}$ and $\widetilde{Y}$ are $R_{n+1}\prod_{i=n+1}^{d} I_i \prod_{j=1}^{n} O_j$, $\prod_{i=1}^{d} I_i$ and $\prod_{j=1}^{d} O_j$, respectively. When I_i and O_j are close, the size of $\mathcal{V}_n$ can be much larger (R_{n+1} times) than that of $X^{'}$ and $\widetilde{Y}$.

2.2 TIE: Compact TT-format Computation

In general, using TT decomposition significantly reduces the model size of DNN from $\prod_{n=1}^{d} I_n O_n$ to $\sum_{n=1}^{d} I_n O_n R_n R_{n+1}$. However, one key drawback of the original TT-format computing scheme (Eq. 1) is the challenging computational redundancy problem. As analyzed in TIE, when the execution on TT-format layer is performed as the multi-dimensional summation of the products of $\mathcal{G}_n[i_n, j_n]$ (see Eq. 1), it has to consume multiple times of consecutive multiplications among the same $\mathcal{G}_n[i_n, j_n]$, causing massive amount of unnecessary computational redundancy. For instance, to execute FC6 layer of VGG-16 model with $d = 6$ and $R_n = 4$, the Eq. 1-based straightforward implementation consumes more than 1000× multiplications over the theoretical minimum for calculating all $Y(i_1, \ldots, i_d)$'s.

To overcome this challenge and improve computation efficiency, TIE proposes and develops a compact TT-format computing scheme that can eliminate all the unnecessary computational redundancy. As indicated in Fig. 1, by fully paralleling the computations involved with the same $\widetilde{\mathcal{G}}_n$, as the matricized $\mathcal{G}_n$, the original repeated data fetching and multiplications with the same $\mathcal{G}_n[i_n, j_n]$ can now be avoided. To be specific, a customized transformation mechanism for the data layout of the intermediate results ($\mathcal{V}_n$) is delicately designed to form a multi-stage consecutive matrix multiplication scheme. As verified by theoretical analysis, the number of multiplications consumed by TIE matches the minimum computational cost for calculating all $Y(i_1, \ldots, i_d)$'s, leading to a redundant computation-free processing scheme.

2.3 Limitations of TIE

While TIE efficiently eliminates the unnecessary redundant computations, it still has three main drawbacks. First, with the presence of promising high compression ratio enabled by TT decomposition, the reduction in the *necessary* computation is not as high as storage saving. For instance, for a TT-format VGG-FC6 layer with [1,4,4,4,4,4,1] rank setting, which means 50972× compression ratio, the compact computing scheme of TIE only brings 28.2× FLOPs reduction. Even with the more aggressive compression effort, e.g., 61021× model size reduction, the corresponding computation reduction only slightly increases to 30.0×. Evidently, such huge gap between compression ratio and speedup ratio severely limits the performance of TT-oriented DNN hardware.

Second, though TIE enjoys the benefit of low storage cost of weight parameters, it consumes high memory overhead for intermediate results. Specifically, as illustrated in Fig. 1, TIE executes one DNN layer in a multi-stage processing manner. In such scenario, the sizes of these *intra-layer* intermediate results from each stage, i.e., $\mathcal{V}_n$'s, are typically much larger than that of the transformed *inter-layer* intermediate result ($\widetilde{Y}$) of the current layer. To store those intermediate results incurred by stage-wise computations, much more on-chip memory resource has to be budgeted than before, and in many cases such extra memory consumption even exceeds the demand from weight parameters. For instance, the hardware implementation of TIE only needs 16KB weight SRAM for models; but it consumes 768KB working SRAM to store larger-size intermediate results.

Third, another even more challenging problem incurred by the multi-stage processing scheme is the high data transfer for the stage-wise intermediate results. Recall that in the conventional DNN hardware there only exists inter-layer data movement for $\widetilde{Y}$ and W; in addition to them, TIE also consumes the unique intra-layer data movement for $\mathcal{V}_n$'s. In particular, consider 1) a d-stage layer has $d-1$ stage outputs $\mathcal{V}_n$'s and only one layer output $\widetilde{Y}$; and 2) as analyzed before, $\mathcal{V}_n$ typically has much larger size than $\widetilde{Y}$, such frequent stage-wise data transfers inevitably cause very high overhead on memory access, significantly limiting the overall energy efficiency.

3 ETTE: ALGORITHM

3.1 Analysis of Limited Performance of TIE

Before moving to the details of ETTE, we first analyze the *cause* of the architectural limitations of TIE – insufficient speedup, intermediate results overhead, and extra data transfer. Essentially, all these three drawbacks either completely or at least partially are incurred by the *phenomenon of large-size intermediate results ($\mathcal{V}_n$'s)*. Specifically, as illustrated in Fig. 1, at each computing stage-n TIE obtains $\widetilde{\mathcal{V}}_{n-1}$ (the transformed $\mathcal{V}_{n-1}$) from memory, multiplies it with $\widetilde{\mathcal{G}}_n$, and then stores the calculated $\mathcal{V}_n$ back to memory. Evidently, the performance of TIE is very sensitive to the size of $\mathcal{V}_n$ – the larger $\mathcal{V}_n$ means more storage requirement, higher computational cost and increasing memory access. Unfortunately, $\mathcal{V}_n$ in TIE is typically very huge, sometimes even several times larger than layer input/output ($X^{'}$, $\widetilde{Y}$), causing inefficient hardware performance.

Upon discovering this phenomenon, a naturally raised question is why $\mathcal{V}_n$ is so huge in TIE processing. In general, two underlying factors, namely as *tensor core construction format* and *tensor core*

computing order, jointly causes this problem. More specifically, the construction format (e.g., order and shape) of tensor core $\mathcal{G}_n$, which is involved with the calculation of $\mathcal{V}_n$ as $\mathcal{V}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{\mathcal{G}}_n$, has direct impact on the size of $\mathcal{V}_n$. Meanwhile, for a fixed left-to-right TT-format execution on a series of $\mathcal{G}_n$, the choice of the computing sequence, which is essentially equivalent to the placement of each $\mathcal{G}_n$ in the entire "train" of tensor cores, also plays an important role for determining the overall costs.

3.2 New Computing Scheme of ETTE

Based on the above observation and analysis, next we develop the efficient computing scheme for TT-format DNN via exploring new tensor core construction format and computing order. To that end, we first consider the *generalized TT format*, in which the order of each factorized tensor core is not limited to 4. In other words, a large matrix $W \in \mathbb{R}^{I\times O}$ is decomposed to d tensor cores $\mathcal{G}_n \in \mathbb{R}^{R_n\times I_{j_{n-1}+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\ldots O_{k_n}\times R_{n+1}}$, where $I = \prod_{i=1}^{d_a} I_i$, $O = \prod_{o=1}^{d_b} O_o$, $j_d = d_A$ and $k_d = d_B$. As shown in Fig. 2, here each tensor core $\mathcal{G}_n$ is now relaxed to having arbitrary order instead of 4. Notice that when performing the multi-stage computation over these generalized TT cores, similar to the case for the 4-D $\mathcal{G}_n$, the arbitrary-dimensional $\mathcal{G}_n$ is also first flatten to 2-D format $\widetilde{\mathcal{G}}_n$, and then it is multiplied with the transformed input $\widetilde{\mathcal{V}}_{n-1}$ to generate the intermediate result $\mathcal{V}_n$ for the current stage.

Within the framework of the generalized TT format, we next explore the cost-aware tensor core construction format and computing order. Notice that because our ultimate goal is to reduce the incurred computation/memory cost for the inference on the decomposed W, the selection of d is also flexible as long as the information carried by I_i and O_o is preserved in the $\mathcal{G}_n$ after the factorization, i.e., the shape of $\mathcal{G}_n$ contains dimension factors I_i and O_o. Based on this observation, we propose to identify the efficient $\mathcal{G}_n$ construction format and computing order in a recursive way. More specifically, given the original $\mathcal{G}_n \in \mathbb{R}^{R_n\times I_{j_{n-1}+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\ldots O_{k_n}\times R_{n+1}}$ where $n = 1, 2, \ldots d$, we factorize it to two new tensor cores and check the corresponding computational cost change. Notice that such factorization is essentially to explore different formats of $\mathcal{G}_n$, since d, j_n and k_n are set to general values. To maximize the scope of the to-be-explored tensor core construction settings, the factorization process can be repeated.

Factorize $\mathcal{G}_n$ to $\mathcal{G}_{n,2}$. As analyzed before, such $\mathcal{G}_n$-level factorization does not alter the information preservation if all the dimension factors of $\mathcal{G}_n$ are contained in the new tensor cores. For simplicity, let $\mathbb{A} \triangleq \{j_{n-1}+1, \ldots, j_n\}$ and $\mathbb{B} \triangleq \{k_{n-1}+1, \ldots, k_n\}$ denote the index of factor I_a and O_a of $\mathcal{G}_n$ respectively, and $\mathbb{A}\backslash l$ and $\mathbb{B}\backslash l$ represent the subsets where element l is excluded from $\mathbb{A}$ and $\mathbb{B}$, respectively. Then FLOPs saving can be obtained with two factorization schemes:

Proposition #1: Factorizing $\mathcal{G}_n$ to two new tensor cores, where $A_{n,1} \in \mathbb{R}^{R_n\times I_l\times R'}$ or $B_{n,1} \in \mathbb{R}^{R_n\times O_l\times R'}$ serves as left or right components, respectively, reduces FLOP count.

Proof. The FLOPs of original computing scheme ($\widetilde{\mathcal{V}}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{\mathcal{G}}_n$) at the stage-$n$ can be calculated as:

$$FLOP_{org} = \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o\Big(\prod_{i\in\mathbb{A}} I_i \prod_{o\in\mathbb{B}} O_o R_n R_{n+1}\Big). \quad (2)$$

Now consider to decompose a $(j_n - j_{n-1} + k_n - k_{n-1} + 2)$-order $\mathcal{G}_n$ to a 3-order tensor core[1] and a $(j_n - j_{n-1} + k_n - k_{n-1} + 1)$-order tensor core denoted as $\mathcal{G}_{n,2}$. Since the shape of a 3-order TT tensor core must contain two rank values (as defined by TT), there exist four possible factorization schemes:

Case-1. Factorize $\mathcal{G}_n$ to $A_{n,1} \in \mathbb{R}^{R_n\times I_l\times R'}$ and $\mathcal{G}_{n,2} \in \mathbb{R}^{R'\times I_{j_{n-1}+1}\times\cdots\times I_{l-1}\times I_{l+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\times\cdots\times O_{k_n}\times R_{n+1}}$, where $l \in \mathbb{A}$. As shown in Fig. 3, in such case $A_{n,1}$ is placed at the left side of $\mathcal{G}_{n,2}$ after decomposition. Then the FLOP count for $\widetilde{\mathcal{V}}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{A}_{n,1}\widetilde{\mathcal{G}}_{n,2}$ is

$$FLOP_1^1 = \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o\Big(\prod_{i\in\mathbb{A}} I_i R_n R' + \prod_{i\in\mathbb{A}\backslash l} I_i \prod_{o\in\mathbb{B}} O_o R' R_{n+1}\Big). \quad (3)$$

Comparing Eq. 3 and Eq. 2, it is seen that because $I_l \geq 2$, $\prod_{o\in\mathbb{B}} O_o \geq 2$, and $R' < R_n$, we have $FLOP_1^1 < FLOP_{org}$, indicating computational saving.

Case-2. Factorize $\mathcal{G}_n$ to $A_{n,1} \in \mathbb{R}^{R'\times I_l\times R_{n+1}}$ and $\mathcal{G}_{n,2} \in \mathbb{R}^{R_n\times I_{j_{n-1}+1}\times\cdots\times I_{l-1}\times I_{l+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\times\cdots\times O_{k_n}\times R'}$. In such case $A_{n,1}$ is essentially placed at the right side of $\mathcal{G}_{n,2}$. Then the FLOP count for $\widetilde{\mathcal{V}}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{\mathcal{G}}_{n,2}\widetilde{A}_{n,1}$ is:

$$FLOP_1^2 = \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o\Big(\prod_{i\in\mathbb{A}} I_i \prod_{o\in\mathbb{B}} O_o R_n R' + \prod_{o\in\mathbb{B}} O_o R' R_{n+1} I_l\Big). \quad (4)$$

Since $R_{n+1} \leq I_l$ and $R_n \geq 2$, $FLOP_1^1 < \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o \prod_{i\in\mathbb{A}\backslash l} I_i (\prod_{i\in\mathbb{A}} I_i \prod_{o\in\mathbb{B}} O_o R_n) R' < FLOP_1^2$, FLOP count of Case-2 is higher than that of Case-1.

Case-3. Since the newly factorized 3-order tensor core can also contain O_o dimension factor instead of I_i. In other words, we can factorize $\mathcal{G}_n$ to $B_{n,1} \in \mathbb{R}^{R'\times O_l\times R_{n+1}}$ and $\mathcal{G}_{n,2} \in \mathbb{R}^{R_n\times I_{j_{n-1}+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\times\cdots\times O_{l-1}\times O_{l+1}\times\cdots\times O_{k_n}\times R'}$. In such case, $B_{n,1}$ is essentially placed at the right side of $\mathcal{G}_{n,2}$. Then the FLOP count for $\widetilde{\mathcal{V}}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{\mathcal{G}}_{n,2}\widetilde{B}_{n,1}$ is:

$$FLOP_1^3 = \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o\Big(\prod_{i\in\mathbb{A}} I_i \prod_{o\in\mathbb{B}\backslash l} O_o R_n R' + \prod_{o\in\mathbb{B}} O_o R' R_{n+1}\Big). \quad (5)$$

Comparing Eq. 5 and Eq. 2, $O_l \geq 2$, $\prod_{i\in\mathbb{A}} I_i \geq 2$ and $R' < R_{n+1}$, so Case-3 brings fewer FLOPs ($FLOP_1^3 < FLOP_{org}$).

Case-4. Factorize $\mathcal{G}_n$ to $B_{n,1} \in \mathbb{R}^{R_n\times O_l\times R'}$ and $\mathcal{G}_{n,2} \in \mathbb{R}^{R'\times I_{j_{n-1}+1}\times\cdots\times I_{j_n}\times O_{k_{n-1}+1}\times\cdots\times O_{l-1}\times O_{l+1}\times\cdots\times O_{k_n}\times R_{n+1}}$. As shown in Fig. 3, $B_{n,1}$ is placed at the left side of $\mathcal{G}_{n,2}$. Then the FLOP count for $\widetilde{\mathcal{V}}_n = \widetilde{\mathcal{V}}_{n-1}\widetilde{B}_{n,1}\widetilde{\mathcal{G}}_{n,2}$ is

$$FLOP_1^4 = \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o\Big(\prod_{i\in\mathbb{A}} I_i O_l R_n R' + \prod_{o\in\mathbb{B}} O_o \prod_{i\in\mathbb{A}} I_i R' R_{n+1}\Big). \quad (6)$$

Since $R_{n+1} \leq O_l$ and $R_n \geq 2$, $FLOP_1^3 < \prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_{n-1}} O_o(\prod_{i\in\mathbb{A}} I_i O_l R_n R') < FLOP_1^4$, Case-4 has more FLOP counts than Case-3 ($FLOP_1^4 > FLOP_1^3$). □

Remark #1. Summarizing the four cases, it is seen that: 1) Case-1/3 have fewer FLOPs than original computing scheme; and 2) Case-2/4 have more FLOPs than Case-1/3. Hence in order to achieve the guaranteed and higher FLOPs reduction, $A_{n,1}$ or $B_{n,1}$ should be placed at the left or right side of $\mathcal{G}_{n,2}$.

[1] Based on the definition of TT formats, a tensor core must contain at least 3 dimensions. So each time factorizing $\mathcal{G}_n$ with a 3-order tensor core minimize the granularity of the exploration for possible tensor core construction format, maximizing the searching space.

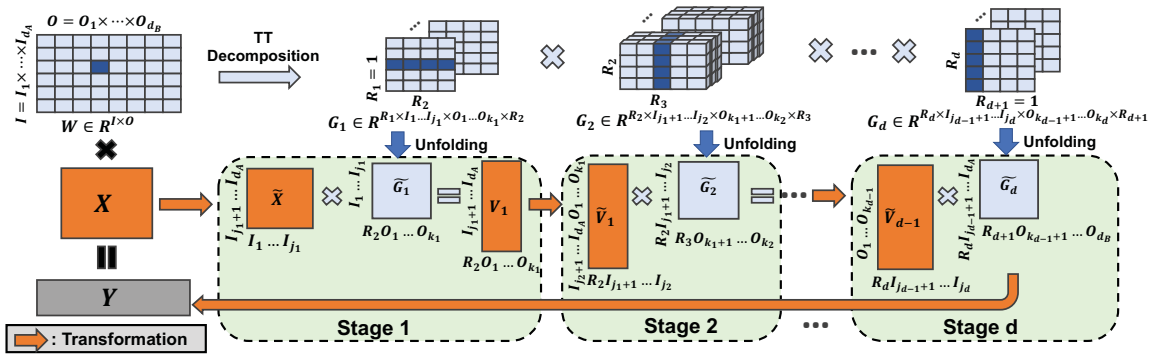


Figure 2: General TT-format computing scheme without optimization on FLOPs and memory costs. Each $\mathcal{G}_n$ is now relaxed as a high-order tensor core (order can be larger than 4). Fig. 1 is essentially a special case of Fig. 2.

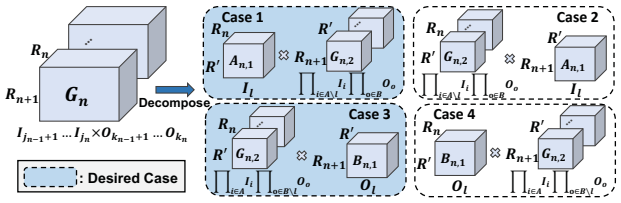


Figure 3: Factorize one new tensor core ($A_{n,1}$ or $B_{n,1}$) from $\mathcal{G}_n$. Here Case-1 and Case-3 can bring the guaranteed and higher FLOPs reduction. Hence after factorization $A_{n,1}$ or $B_{n,1}$ must be located at the left or right side of $\mathcal{G}_{n,2}$, respectively.

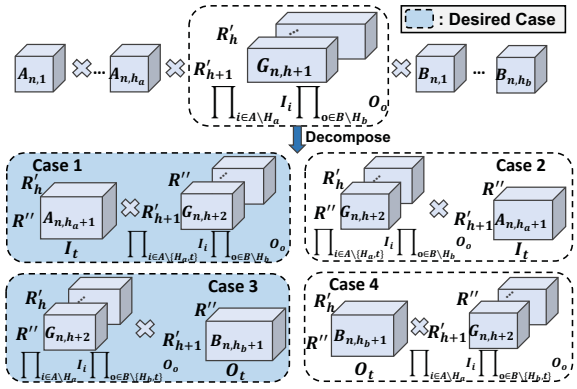


Figure 4: Recursively factorize one new tensor core (A_{n,h_a+1} or B_{n,h_b+1}) from $\mathcal{G}_{n,h+1}$. Here $A_{n,i}$'s and $B_{n,o}$'s have been placed at the left and right side of $\mathcal{G}_{n,h+1}$ after previous recursive factorization, i.e., intra-G_n-level splitting. Again, Case-1 and Case-3 can bring the guaranteed and higher FLOPs reduction. Hence after current factorization A_{n,h_a+1} or B_{n,h_b+1} must be located at the left or right side of $\mathcal{G}_{n,h+d}$, respectively.

Factorize $\mathcal{G}_{n,h+1}$ to $\mathcal{G}_{n,h+2}$. Proposition #1 shows that factorizing $\mathcal{G}_n$ to two new tensor cores can bring computational saving in some cases. Next we show that this also holds in general condition, i.e., when $\mathcal{G}_n$ has already been factorized to multiple tensor cores.

In general, suppose that $\mathcal{G}_n$ has been factorized to $(h+1)$ components with h_a of $A_{n,i}$, h_b of $B_{n,o}$ and one $\mathcal{G}_{n,h+1}$, where $i \in \mathbb{H}_a$, $o \in \mathbb{H}_b$ and $h = h_a + h_b$. Then FLOPs saving can be obtained with the following scheme:

Proposition #2: Factorizing $\mathcal{G}_{n,h+1}$ to two new tensor cores, where A_{n,h_a+1} or B_{n,h_b+1} serves as left or right components, respectively, reduces FLOP count.

Proof. Considering the generality of n, h_a and h_b, we aim to use mathematical induction to prove the above conclusion. Since the case of $h = 0$ has been verified in Proposition #1, we now show the case of $h+1$ holds if the case of h holds, i.e., all the previously decomposed $A_{n,i}$ and $B_{n,i}$ are placed on the left and right side of $\mathcal{G}_{n,h+1}$, respectively. Then, similar to the analysis in Proposition #1, when decomposing $\mathcal{G}_{n,h+1}$ to two new tensor cores, Fig. 4 shows that there exist four possible cases, i.e., Case-1/2: A_{n,h_a+1} on the left/right side of $\mathcal{G}_{n,h+2}$, Case-3/4: B_{n,h_b+1} on the right/left side of $\mathcal{G}_{n,h+2}$. Then the FLOPs incurred by $\mathcal{G}_{n,h+2}$, A_{n,h_a+1} and B_{n,h_b+1} is:

$$FLOP_h^{org} = \prod_{i\in\mathbb{A}\setminus\mathbb{H}_a} I_i \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o R_h^{'} R_{h+1}^{'}, \tag{7}$$

$$FLOP_{h+1}^{1} = \prod_{i\in\mathbb{A}\setminus\mathbb{H}_a} I_i R_h^{'} R^{''} + \prod_{i\in\mathbb{A}\setminus\{\mathbb{H}_a,t\}} I_i \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o R^{''} R_{h+1}^{'}, \tag{8}$$

$$FLOP_{h+1}^{2} = \prod_{i\in\mathbb{A}\setminus\mathbb{H}_a} I_i \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o R_h^{'} R^{''} + \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o I_t R^{''} R_{h+1}^{'}, \tag{9}$$

$$FLOP_{h+1}^{3} = \prod_{i\in\mathbb{A}\setminus\mathbb{H}_a} I_i \prod_{o\in\mathbb{B}\setminus\{\mathbb{H}_b,t\}} O_o R_h^{'} R^{''} + \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o R^{''} R_{h+1}^{'}, \tag{10}$$

$$FLOP_{h+1}^{4} = \prod_{i\in\mathbb{A}\setminus\mathbb{H}_a} I_i O_t R_h^{'} R^{''} + \prod_{i\in\mathbb{A}\setminus\{\mathbb{H}_a,t\}} I_i \prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o R^{''} R_{h+1}^{'}. \tag{11}$$

Comparing Eq. 8 - Eq. 11 with Eq. 7 it is seen that, because $I_t \geq 2$, $O_t \geq 2$, $\prod_{o\in\mathbb{B}\setminus\mathbb{H}_b} O_o \geq 2$, $\prod_{o\in\mathbb{A}\setminus\mathbb{H}_a} I_i \geq 2$ and $R^{''} < R_h^{'}$, we can observe $FLOP_{h+1}^{1} < FLOP_h^{org}$, $FLOP_{h+1}^{1} < FLOP_h^{2}$, $FLOP_{h+1}^{3} < FLOP_h^{org}$ and $FLOP_{h+1}^{3} < FLOP_h^{4}$. Case-1/3 bring guaranteed and higher FLOPs reduction. □

Remark #2. The generality of Proposition #2 shows that the factorization of $\mathcal{G}_{n,h+1}$ can be applied in a recursive way to further reduce FLOP count. Finally, $\mathcal{G}_{n,h+1}$ is decomposed to a group of 3-order $A_{n,i}$ on the left side and a group of 3-order $B_{n,o}$ on the right side, bringing significant FLOPs reduction (as illustrated in Fig. 4).

From $\mathcal{G}_n$ to $\{\mathcal{G}_n\}$. The above described factorization and splitting strategy for $\mathcal{G}_n$ can be further extended to be applied to $\{\mathcal{G}_n\}$. In other words, each $\mathcal{G}_n$ is decomposed to a sequence of $A_{n,i}$ and

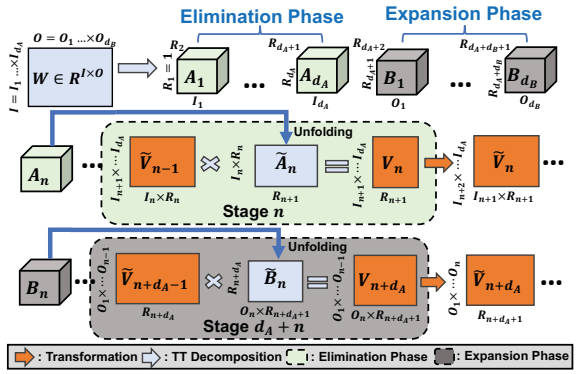


Figure 5: Extending the strategy in Fig. 4 for one $\mathcal{G}_n$ to all $\mathcal{G}_n$'s. All $A_{n,i}$'s and $B_{n,o}$'s factorized from different $\mathcal{G}_n$'s are placed at the left and right side, respectively, showing inter-$\mathcal{G}_n$-level splitting. During inference, the size of intermediate result $\mathcal{V}_n$ first decreases at the beginning d_A stages where A_n's are multiplied with $\mathcal{V}_{n-1}$ (i.e., Elimination Phase). Then, the size of $\mathcal{V}_n$ increases at the following d_B stages where B_n's are multiplied with $\mathcal{V}_{n+d_A-1}$ (i.e., Expansion Phase).

$B_{n,o}$ to reduce the FLOPs for TT-format execution. Moreover, because the entire $\{\mathcal{G}_n\}$ can be interpreted as a "mega" tensor, its factorized tensor cores $\{A_{n,i}\}$ and $\{B_{n,o}\}$ should also be split to the "left" and "right" groups (as shown in Fig. 5). Notice that this inter-$\mathcal{G}_n$-level split strategy does not only reduce computational cost (addressing the first drawback of TIE), but also saves memory cost for intermediate results (addressing the second drawback of TIE).

Proposition #3: Splitting $\{A_{n,i}\}$ and $\{B_{n,o}\}$ to two groups reduces the storage costs of intermediate result $\mathcal{V}_n$.

Proof. For original computing scheme as illustrated in Fig. 2, the size of the intermediate result at stage-n is $size_{org} = R_{n+1}\prod_{i=j_n+1}^{d_A} I_i \prod_{o=1}^{k_n} O_o$. On the other hand, after applying the computing scheme shown in Fig. 5, the size of the intermediate result at stage-n is either $size^A_{new} = R_{n+1}\prod_{i=n}^{d_A} I_i$ or $size^B_{new} = R_{n+1}\prod_{o=1}^{n-d_A} O_o$, which represent the size of $\mathcal{V}_n$ that is involved with the $A_{n,i}$ or $B_{n,o}$, respectively. Evidently, both $size^A_{new}$ and $size^B_{new}$ are smaller than $size_{org}$, leading to lower storage cost of intermediate result $\mathcal{V}_n$. □

Remark #3. When applying our proposed new TT-format execution scheme (shown in Fig. 5), the size of intermediate result $\mathcal{V}_n$ at stage-n first decreases, as revealed by $size^A_{new} = R_{n+1}\prod_{i=n}^{d_A} I_i$, and then increases, as revealed by $size^B_{new} = R_{n+1}\prod_{o=1}^{n-d_A} O_o$. We name these two different phases as *elimination* phase and *expansion* phase, respectively.

Factorization/Computing Order for A_n and B_n. Following prior TT works [34, 51, 53], given the target compression ratio, ETTE selects $\{I_n\}$, $\{O_n\}$ and $\{R_n\}$ to approach uniform distribution, e.g., make $I_1 \approx I_2 \approx ...$, $O_1 \approx O_2 \approx ...$, $R_2 \approx R_3 \approx ...$ as much as possible. Under such setting, as long as 1) $\{A_n\}$ and $\{B_n\}$ are split into two groups and 2) the computation involved with the entire $\{A_n\}$ is performed earlier than the entire $\{B_n\}$, the change of FLOPs incurred by different factorization/computing orders for specific A_i's and B_i's in Fig. 5 is not significant, since the shape of different A_i's are quite similar (also for different B_i's). For instance, for a TT-decomposed VGG-FC6 with $I_n = [16, 14, 8, 14]$, $O_n = [16, 16, 16]$ and $R_n = [1, 4, 4, 4, 4, 4, 4, 1]$, the different factorization/computing orders only bring up to 10% FLOPs difference. Considering the limited impact caused by different factorization/computing orders, ETTEs simply performs consecutive computation with A_i's and B_i's with descending order of their sizes, e.g., $size(A_1) \geq size(A_2) \geq ...$, $size(B_1) \leq size(B_2) \leq ...$, avoiding the offline exhaustive search.

4 ETTE: HARDWARE ARCHITECTURE

Based on the proposed new computing scheme, in this section we develop the corresponding ETTE hardware accelerator. Fig. 6 shows the overall architecture. Here the datapath consists of a 1-D PE array, where each of PE contains multiple multiplier and accumulator (MAC) units. The inputs of datapath are from weight SRAM and activation SRAM via unitcasting and broadcasting manner, respectively. Notice that the activation SRAM does not only store inter-layer immediate results (activation), but also store intra-layer immediate results output from each stage. Also, two copies of activation SRAM are used to serve as ping-pong buffer. Next, we describe two architectural optimization on ETTE hardware accelerator.

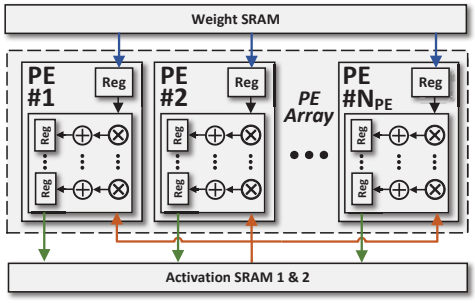


Figure 6: Overall Architecture of ETTE.

4.1 Optimizing Intra-layer Data Transfer

Recall that in addition to the normal layer-level data transfer for $\widetilde{Y}$, TIE needs extra stage-level data transfer for $\mathcal{V}_n$ due to its multi-stage processing scheme, which is also adopted by ETTE. Although the computing scheme in ETTE can reduce the size of $\mathcal{V}_n$ (Proposition #3), because each layer in ETTE consists of $d_A + d_B$ stages, the overall amount of stage-wise data movement is still considerable.

To further alleviate this challenging problem, we propose to optimize the intra-layer processing scheme to reduce the incurred data transfer. Our key idea is that once a part of $\mathcal{V}_n$ in stage-n are calculated on the fly, it is not stored back to memory but remains in the PE to serve for the calculation of $\mathcal{V}_{n+1}$ in stage-$(n+1)$. Evidently, such *lookahead* processing style can efficiently reduce the data transfer for intermediate results via increasing their locality. More specifically, considering this across-stage computing schedule highly depends on the underlying computation pattern and data dependency, which exhibit different behaviors in the elimination and expansion phases, we next describe the corresponding processing schemes in these two phases, respectively.

Elimination Phase-specific Schedule. Fig. 7 illustrates an example when performing the lookahead-style schedule in the

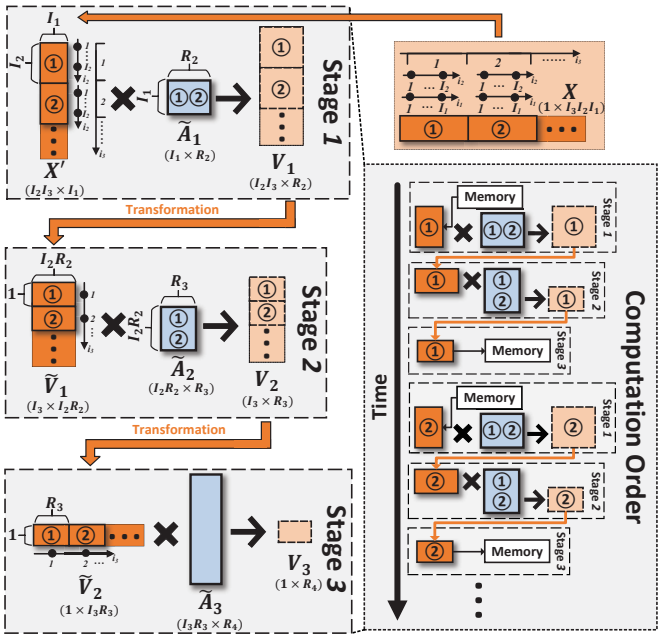


Figure 7: Lookahead processing in the elimination phase. Different numbers at the same stage represent the processing order, and different tiles with the same numbers have data dependency.

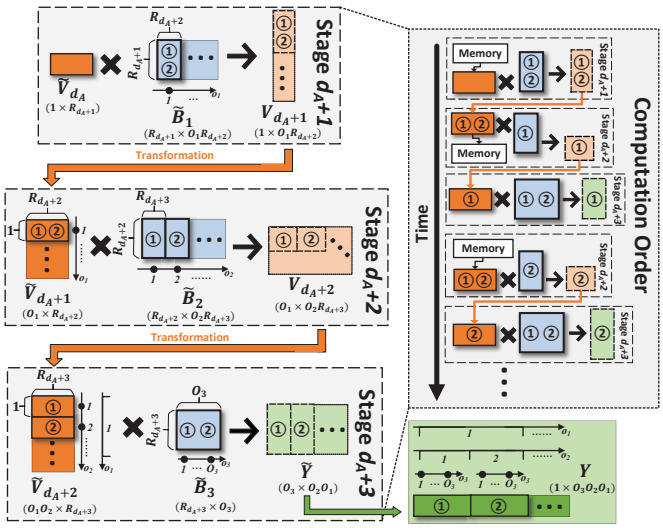


Figure 8: Lookahead processing in the expansion phase. Different numbers at the same stage represent the processing order, and different tiles with the same numbers have data dependency.

elimination phase of ETTE with $d_A = 3$. At stage-1, only one tile of $X^{'}$ is brought from memory and participates in the computation for $\mathcal{V}_1$. The entries of this partially calculated $\mathcal{V}_1$ are then directly transformed to the corresponding part of $\widetilde{\mathcal{V}}_1$, which continues to be multiplied with $\widetilde{A}_2$ to generate one tile of $\mathcal{V}_2$. After that, another tile of $\widetilde{\mathcal{V}}_2$ is read from memory for the similar lookahead computation of $\mathcal{V}_3$. As marked in Fig. 7, the tiles of $\mathcal{V}_n$ and A_n with data dependency across two stages are marked with the same number. Therefore, because 1) each tile of $\mathcal{V}_n$ at stage-n is on-the-fly calculated; and 2) it has data dependency with only one tile of $\mathcal{V}_{n+i}$ at stage-$(n+i)$, this tile of $\mathcal{V}_n$ can be then discarded without data transfer once the next-stage computation consumes it.

It should be noted that this lookahead-style processing is not limited to only cross two stages but can be applied to multiple stages. In general, for a d_A-stage elimination phase, such lookahead computation can be maximally performed across $d_A - 1$ stages. This is because as shown in Fig. 7, the calculation of $\mathcal{V}_{d_A}$, even for only one entry, needs the entire information of $\widetilde{\mathcal{V}}_{d_A-1}$ instead of part of it. Meanwhile, multiple lookahead-style processing schedules can co-exist in the same elimination phase. For instance, the processing for $d_A = 5$ can be either first performed across 3 stages followed by another 2-stage lookahead, or being performed across 2 stages followed by another 3-stage lookahead. The design space for these different processing schedules will be explored in Section 5.3.

Algorithm 1 describes the general lookahead-style processing procedure in the elimination phase. Notice that the stage-wise matrix transformation, which is essentially the mapping principle from the entries of $\mathcal{V}_n$ to $\widetilde{\mathcal{V}}_n$, is also described here. The efficient hardware mapping for such critical yet costly operation will be discussed in Section 4.2.

Algorithm 1: The processing schedule of elimination phase with lookahead computation.

input : $X = I_d \times I_{d-1} \times \ldots \times I_1, \tilde{A_1}, \ldots, \tilde{A_{d_A}}$,
$R = [R_1, R_2, \ldots, R_{d_A+1}]$
output: Y

$X^{'}$ = Reshape(X, [I_1, −1]);
//Lookahead computation order in stage 1 *to* $d_A - 1$;
$L = new[R_{d_A}, I_{d_A}]$;
for $k = 0$ **to** $I_{d_A} - 1$ **do**
  $\tilde{\mathcal{V}} = X^{'}[:, \prod_{m=2}^{d_A-1} I_m * k : \prod_{m=2}^{d_A-1} I_m * (k-1)]$;
  for $j = 1$ **to** $d_A - 1$ **do**
    $\tilde{\mathcal{V}}$ = Transform (Matmul ($\tilde{A_j}, \tilde{\mathcal{V}}$), j)
  $L[:, k] = \tilde{\mathcal{V}}$
Y = Transform (Matmul ($\tilde{A_{d_A}}, L$), d_A)*//Stage* d_A;
Function Transform($\mathcal{V}$, j):
  $\mathcal{V}$ = Transpose($\mathcal{V}$) ;
  h, w = size($\mathcal{V}$);
  $\tilde{\mathcal{V}}$ = new[$h/I_{j+1}, R_j * I_{j+1}$];
  for $i = 0$ **to** $h/I_{j+1} - 1$ **do**
    T = new[1, $R_j \times I_{j+1}$] ;
    for $k = 0$ **to** $I_{j+1} - 1$ **do**
      $T[k * R_j : (k+1) * R_j] = \mathcal{V}[i * I_{j+1} + k, :]$
    $\tilde{\mathcal{V}}[i+1, :] = T$;
  $\tilde{\mathcal{V}}$ = Transpose($\tilde{\mathcal{V}}$) ;

Expansion Phase-specific Schedule. Fig. 8 shows an example of applying the lookahead-style processing in the 3-stage expansion phase. This example, together with the general processing procedure described in Algorithm 2, shows that the lookahead computation in the expansion phase exhibits two differences from its counterpart in the elimination phase. First, because of the unique computing pattern in the expansion phase, one tile of $\mathcal{V}_{d_A+n}$ in the previous stage has data dependency with multiple tiles of $\mathcal{V}_{d_A+n+i}$ in the later stages. Consequently, the intermediate results $\widetilde{\mathcal{V}}_{d_A+n-1}$ cannot be simply discarded after the generation of one tile of $\mathcal{V}_{d_A+n}$. Instead, it has to be retained in the memory for the future use of calculating another dependent tile of $\mathcal{V}_{d_A+n}$. Notice that though this multi-dependency phenomenon prevents the reduction in data transfer, the lookahead-style schedule can still reduce the memory consumption for $\mathcal{V}_{d_A+n}$. As indicated in Fig. 8, during the entire phase for each stage only one tile of $\mathcal{V}_{d_A+n}$ needs to be stored in the on-chip memory. Notice that this benefit also exists in the elimination phase. In a nutshell, the lookahead-style processing reduces both memory consumption and data transfer for $\mathcal{V}_n$ in the elimination phase; while it only reduces memory consumption of $\mathcal{V}_{d_A+n}$ in the expansion phase.

Second, unlike in the elimination phase, the lookahead-style processing can be performed up to the last $((d_A + d_B)$-th) stage of expansion phase. As illustrated in Fig. 8, the calculation of one tile of $\widetilde{Y}$ in the stage-$(d_A + d_B)$ can be realized with using one tile of $\mathcal{V}_{d_A+n}$ in each of the previous stages. Therefore, the lookahead computation is able to cover the last stage now. Consider multiple across-stage processing schedules can also co-exist in the expansion phase, including last stage into lookahead computation can further increase the exploration space for performance optimization. In addition, another promising benefit for this phenomenon is that it unlocks the possibility of lookahead-style processing *across the adjacent layers*. This is because each time the first stage of the elimination phase in the next layer only needs one tile of input, which can be just provided by the last stage of the current layer. Consequently, the lookahead-style processing schemes of the two adjacent layers can be possibly combined to further reduce data transfer if needed. Notice that in this paper we do not focus on exploiting this inter-layer optimization opportunity while leaving it for future work.

The difference of lookahead-style processing in the two phases can also be understood from the perspective of *exploration priority*. Fig. 9 uses a depth-d tree to depict the computing procedure of d-stage phase ($d_A = d_B = d = 3$). Here the leaf (m, n), as the m-th node at the n-the level of the tree, represents the m-th tile of $\widetilde{\mathcal{V}}_{n-1}$ at the n-th stage that will be used for multiplication. Meanwhile, similar to Fig. 7 and 8, the number associated with the arrow here also denotes the computing sequence and data dependency between the tiles. With such notation, it is clear that computing procedure in the elimination and expansion phases explore this index tree in the breadth-first and depth-first manners, respectively.

Difference from Layer Fusion. Readers who are familiar with convolutional neural network (CNN) accelerators may find that the proposed lookahead-style processing shares the similar philosophy of layer fusion [2]. Both of these two techniques aim to reduce the data transfer incurred by intermediate results via passing the output of the current computation to the next one. However, our proposed approach has three important differences. First, unlike layer fusion that is specially designed for CNN, our method works for a broad spectrum of DNN types, including MLP, RNN, transformers and CNN. This is because after TT decomposition, both the FC and CONV layers are unified in the same TT format. Second, our processing schedule supports more complicated computing pattern, e.g., stage-wise matrix transformation and two split computing phases; while layer fusion can only work for transformation-free and single-phase processing. More specifically, layer fusion takes the kernel size as the unit to fuse layers in a pyramid-style processing scheme; therefore when the matrix transformation is involved, the fusion is not able to continue to the next layer until the entire row or column is calculated, even most of the data is not used in the next tile of calculation. Instead, our approach takes the $I_n R_n$ and R_{d_A+n} as basic tiles for elimination and expansion phases in a breadth-first and depth-first searching manner, respectively, as shown in Fig. 9. The intermediate result of each tile is enough for partially matrix transformation for the next layer. Third, there exists data overlapping in each tile of CNN computation, which need additional computation or memory overhead; while our proposed processing scheme has no overlapping as shown in Fig. 7 and 8.

Difference from TPU/TIE on Fully Storing Intermediate Data. Modern DNN accelerators, e.g. TPU, use sufficient on-chip SRAM to fully store intermediate data. Such strategy is also adopted in TIE design. Though $\mathcal{V}_n$ is also stored on chip in ETTE, the key goal and benefit of the processing scheme in ETTE is very different from TPU/TIE. First, the sufficient SRAM budgeted in TPU/TIE can only reduce *off-chip DRAM access*, while the lookahead-style processing in ETTE further reduces *on-chip SRAM access*. As analyzed in Section 2.3 and the first paragraph of Section 4.1, executing TT-decomposed models requires additional intra-layer multi-stage processing, increasing the movement of intermediate results. Using large SRAM can only avoid data access to DRAM, but the data access to SRAM actually increases since $\mathcal{V}_n$ of stage-n is entirely stored in SRAM and then it is fetched for stage-$n+1$. On the other hand, the proposed lookahead-style processing calculate part of $\mathcal{V}_n$ on the fly and store that small chunk of data in the register for the following computation, reducing the data access to SRAM. As reported in Fig. 12, lookahead-style processing schemes can reduce data movement of $\mathcal{V}_n$ by up to 3×. Second, lookahead-style processing also reduces SRAM size. As shown in Fig. 1, the inter-stage intermediate result $\mathcal{V}_n$ can be $R_{n+1}\times$ larger than inter-layer result $X^{'}$ and $\tilde{Y}$, causing expensive memory cost when fully buffered on chip. Though the proposed decomposition algorithm reduces the huge $\mathcal{V}_n$, lookahead-style processing further reduces the required SRAM size by computing only one part of $\mathcal{V}_n$ and passing it to next stages. As reported in Section 5, under the same compression ratio for various workload, the activation SRAM size of ETTE (256KB) is 3× smaller than TIE (768KB).

Difference from Loop Blocking. Both the proposed lookahead-style processing and Loop blocking[4, 36] perform computation at the granularity of data tile. However, lookahead computation differs from loop blocking in two aspects. First, loop blocks adopts an in-order processing style, making the computation of the current stage fully complete first. On the other hand, lookahead scheme

adopts out-of-order processing style via taking advantage of the data dependency across different stages. Second, loop blocking stores the whole $\mathcal{V}_n$ stage by stage; while lookahead-style processing only stores one small part of $\mathcal{V}_n$. Since in TT-format model the rank value R_n is typically smaller than output size O_n, lookahead computation requires smaller SRAM than loop blocking for storing intermediate result. For instance, in expansion phase of VGG-FC6, with $O_n = [16, 16, 16]$, $R_n = [1, 4, 4, 4, 1]$ and 16-bit quantization, the storage requirement for intermediate result in lookahead-style processing is 8 Byte; while loop blocking-based scheme needs 2K Byte budget (256× increase).

Algorithm 2: The processing schedule of expansion phase with lookahead computation.

input : $\tilde{\mathcal{V}} = Y, \tilde{B_1}, \ldots, \tilde{B_{d_B}}, R = [R_{d_A+1}, R_{d_A+2}, \ldots, R_{d_A+d_B+1}]$
output: Z

for $o_1 = 0$ **to** $O_1 - 1$ **do**
  $\tilde{\mathcal{V}} = \texttt{Transform}(\texttt{Matmul}(\tilde{B_1}[o_1 * R_{d_A+2} : (o_1+1) * R_{d_A+2}, :], \mathcal{V}), 1)$;
  for $o_2 = 0$ **to** $O_2 - 1$ **do**
    ...
    for $o_{d_B} = 0$ **to** $O_{d_B} - 1$ **do**
      $\tilde{\mathcal{V}} = \texttt{Transform}(\texttt{Matmul}(\tilde{B_{d_B}}[o_{d_B} * R_{d_A+d_B+1} : (o_{d_B}+1) * R_{d_A+d_B+1}, :], \tilde{\mathcal{V}}), d_B)$;
      $Z[\sum_{h=1}^{d_B} o_h \prod_{k=h+1}^{d_B} O_k] = \tilde{\mathcal{V}}$

Function Transform($\mathcal{V}$, j):
  $\mathcal{V}$ = Transpose($\mathcal{V}$) ;
  h, w = size($\mathcal{V}$);
  $\tilde{\mathcal{V}}$ = new$[h * w/R_{d_A+j+1}, R_{d_A+j+1}]$;
  for $i = 0$ **to** h **do**
    $\mathcal{T}$ = new$[w/R_{d_A+j+1}, R_{d_A+j+1}]$;
    for $k = 0$ **to** w/R_{d_A+j+1} **do**
      $T[k+1, :] = \mathcal{V}[i, k * R_{d_A+j+1} : (k+1) * R_{d_A+j+1}]$
    $\tilde{\mathcal{V}}[i * w/R_{d_A+j+1} : (i+1) * w/R_{d_A+j+1}, :] = T$;
  $\tilde{\mathcal{V}}$ = Transpose($\tilde{\mathcal{V}}$) ;

4.2 Optimizing Matrix Transformation

Memory Partition in TIE. As illustrated in Fig. 7 and 8, the matrix transformation from $\widetilde{\mathcal{V}}_{n-1}$ to $\mathcal{V}_n$ is an important operation in the computing scheme of ETTE to ensure functional validity. To avoid memory access conflict, the conventional solution is to use an extra copy of memory for storing intermediate results, causing both area and power overhead. To address this problem, TIE adopts *memory partition* technique to enable the simultaneous access to the original conflicted data groups, realizing efficient matrix transformation. However, this memory bank-based solution is not free. This is because different workloads have different shapes (I_n, O_n and R_n), which bring various demands for the desired partition granularity. Consequently, it becomes quite challenging to achieve high reconfigurability in practice.

Memory Partition-free Solution. We develop a very efficient memory read/write scheme to support matrix transformation used in ETTE. Our key observation is that because of the unique computing pattern of ETTE, its desired matrix transformation is much simpler than the counterpart in TIE. As illustrated in Fig. 9, a transformation from $\mathcal{V}_1 \in \mathbb{R}^{I_2 I_3 \times O_1 R_2}$ to $\widetilde{\mathcal{V}}_1 \in \mathbb{R}^{O_1 I_3 \times I_2 R_2}$ is needed in the stage-1 of TIE. This means that the dimension factors I_2 and O_1 need to be *switched* after this transformation. On the other hand, for ETTE, the corresponding transformation is from $\mathcal{V}_1 \in \mathbb{R}^{I_2 I_3 \times R_2}$ to $\widetilde{\mathcal{V}}_1 \in \mathbb{R}^{I_3 \times I_2 R_2}$ (elimination phase) and from $\mathcal{V}_{d_A+2} \in \mathbb{R}^{O_1 \times O_2 R_{d_A+3}}$ to $\widetilde{\mathcal{V}}_{d_A+2} \in \mathbb{R}^{O_1 O_2 \times R_{d_A+3}}$ (expansion phase); where only one factor needs to be *moved* from one dimension to another. In principle, this phenomenon essentially results from the decoupled I_n and O_n in ETTE – because at each stage only one type of dimension factor (I_n or O_n) now exists in the shape of $\mathcal{V}_n$, such switching-free transformation pattern becomes very natural. By leveraging this simplicity, we then develop the corresponding efficient memory read and write schemes for matrix transformation used in ETTE. As illustrated in Fig. 9, without performing any memory partition, our proposed memory access scheme directly realizes the desired matrix transformation in the elimination and expansion phases.

5 EVALUATION

5.1 Algorithmic Performance

Table 1 summarizes the algorithmic performance of ETTE across various practical applications. It is seen that with similar compression ratio, e.g., for video, image and audio tasks, ETTE can bring much higher FLOPs reduction than original TT decomposition. Meanwhile, ETTE can also achieve both higher model size reduction and FLOPs reduction than original TT with the similar accuracy performance when compressing DNN models used in the recommendation systems and NLP.

5.2 Hardware Performance

Experimental Setting. We model the behavior of ETTE via building a high-level functional simulator. A cycle-accurate RTL model is then developed with Verilog via Visual Studio Code and synthesized using Synposis Design Compiler with CMOS 28nm library. The area and power consumption for non-memory part is reported from synthesize results; and the hardware performance of memory part is reported from Cacti.

Design Configuration. We implement a 16-PE design example of ETTE hardware architecture. Each PE is equipped with 16 multipliers and accumulators (MACs) and the corresponding activation units. Thanks to the ultra-high compression capability of TT decomposition, one copy of 128 KB weight SRAM and two copies of 128 KB activation SRAM (totally 256 KB) are budgeted in this design, which are sufficient for most TT-format DNN models. With 28nm CMOS technology, a 16-PE ETTE occupies 1.25mm^2 silicon area and has 135.6mW power consumption under 1000 MHz clock operating frequency.

Hardware Performance Comparison. Fig. 11 shows the performance improvement of ETTE over other hardware solutions on various DNN workloads (LSTM, Transformer, DLRM) for different applications (audio, video, language, recommendation system). Compared with NVIDIA RTX 3090 GPU working on dense models, ETTE achieves 36.2× – 253.1× higher throughput and 1204.2× – 8504.5× higher energy efficiency, respectively. Compared with GPU running on TT-decomposed models, ETTE enjoys 6.5× – 63.3× throughput increase and 189.2× – 9750.5× energy

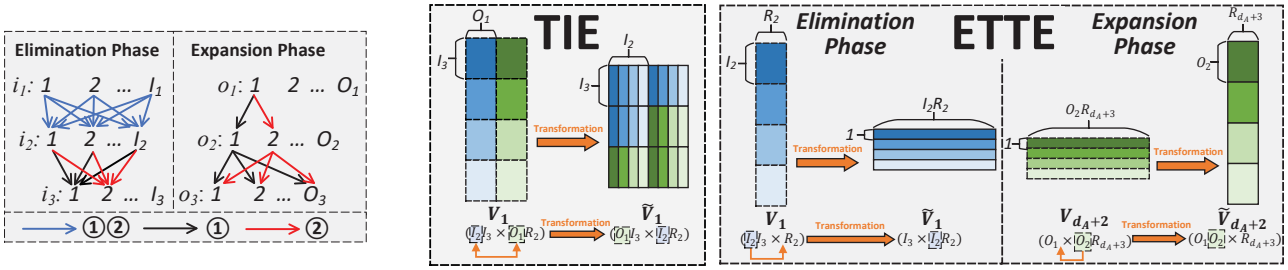


Figure 9: (Left) Interpretation of processing schedule in the elimination and expansion phases over the index tree. Here the node (m, n), as the m-th node at the n-the level of the tree, represents the access to the m-th tile of $\widetilde{\mathcal{V}}_{n-1}$ at the n-th stage. (Right) Different movements of the dimension factors I_n and O_n in the matrix transformation of TIE and ETTE.

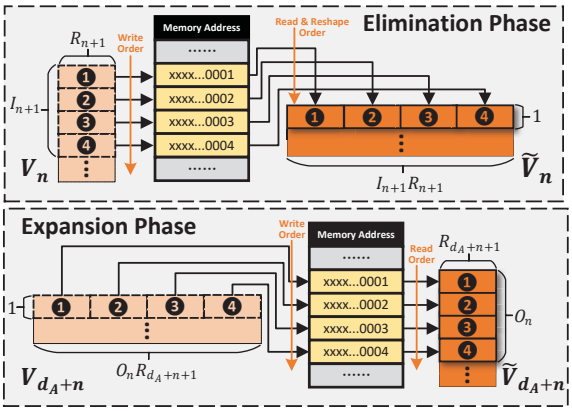


Figure 10: Memory partition-free read/write for matrix transformation in the elimination and expansion phases.

efficiency increase, respectively. Compared with the state-of-the-art DNN accelerators, including TIE, sparsity-aware EIE, sparsity-aware SpAtten[47], sparsity-aware LEOPARD[28], sparsity-aware Centaur[25] and RecPipe[18], ETTE enjoys performance improvement with 1.1× – 58.3× higher throughput, 2.6× – 1170.4× higher energy efficiency and 1.8× – 2098.2× higher area efficiency, respectively. Notice that unlike several DNN accelerators that customized for a specific type of model, e.g., SpAtten and LEOPARD for NLP transformer and Centaur and RecPipe for DLRM, ETTE can widely support the acceleration of various DNN types, demonstrating its generality.

Method	Acc. (%)\ PPL	Params.↓	FLOPs↓
VGG-FC on ImageNet (Image Classification)			
Uncompressed	69.1	–	–
Org. TT[33] (NeurIPS)	67.8	37732×	24.8×
ETTE (Ours)	68.1	38887×	759.5×
Video LSTM on UCF-11 (Video Recognition)			
Uncompressed	69.7	–	–
Org. TT[51] (ICML)	79.6	17554×	4.7×
ETTE (Ours)	89.0	17862×	237.3×
Video LSTM on Youtube Celebrities (Video Recognition)			
Uncompressed	33.2	–	–
Org. TT[51] (ICML)	75.5	17388×	2.7×
ETTE (Ours)	88.3	17554×	170.0×
Audio LSTM [62] on TIMIT (Speech Recognition)			
Uncompressed	79.5	–	–
Org. TT[29]	79.6	7315×	14.4×
ETTE (Ours)	79.6	9280×	448.5×
DLRM [32] on Kaggle (Recommendation System)			
Uncompressed	78.8	–	–
Org. TT[54] (MLSys)	78.68	174.8×	1.4×
ETTE (Ours)	78.68	179.1×	2.2×
Transformer-XL [7] on WikiText103 (Language Translation)			
Uncompressed	24.34	–	–
Org. TT[24] (EMNLP)	28.04	15.1×	0.93×
ETTE (Ours)	29.24	1686×	7.3×
BERT [11] on GLUE (Language Understanding)			
Uncompressed	82.7	–	–
Org. TT[39] (ICLR)	80.0	47.78×	5.23×
ETTE (Ours)	79.8	1671.8×	44.79×

Table 1: Algorithmic performance of ETTE and original TT.

5.3 Analysis & Discussion

Data Movement incurred by $\mathcal{V}_n$. As analyzed in Section 3.2 and 4.1, ETTE can reduce the data transfer of $\mathcal{V}_n$ via reducing its size and access frequency. Fig. 12 shows the overall data movement incurred by $\mathcal{V}_n$ on different workloads. It is seen that the proposed new computing scheme, which essentially shrinks the size of $\mathcal{V}_n$, brings 83× – 2272× reduction in data movement. Then the lookahead-style processing, which reduces memory access, further enables additional 2× – 3× reduction in the amount of data transfer. Overall, joint use of these two techniques brings 200× – 6667× reduction in the data movement for $\mathcal{V}_n$, significantly improving energy efficiency.

Lookahead Strategy. As mentioned in Section 4.1, for the same d_A-stage or d_B-stage elimination or expansion phase, there exist different lookahead-style processing schedules that have different impacts on memory consumption and access frequency. To identify the best-suited lookahead strategy and maximize hardware performance, we explore the design space of lookahead scheme in both elimination and expansion phases. As illustrated in Fig. 13, among different lookahead-style processing options, maximizing lookahead effort, i.e., performing lookahead computation as much as possible, is the best solution since it achieves the minimum memory consumption and data transfer for $\mathcal{V}_n$ in the elimination phase. It is also seen that though in the expansion phase different lookahead schemes have similar impact on memory access incurred by $\mathcal{V}_n$, maximizing the lookahead effort can still brings lowest memory consumption for $\mathcal{V}_n$.

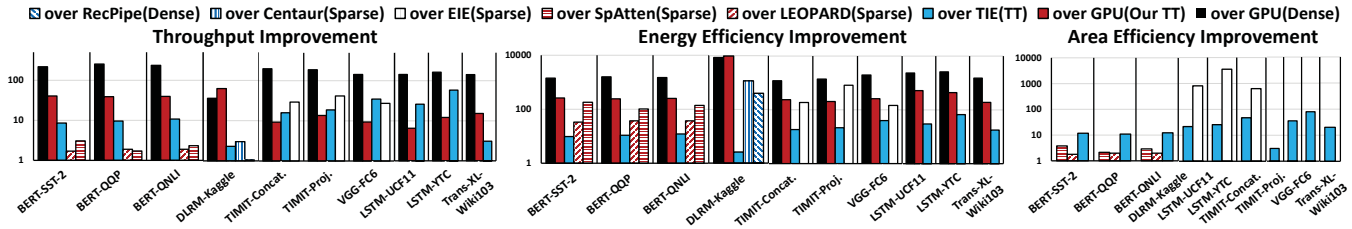


Figure 11: Hardware performance of different DNN hardware solutions. Here some accelerators are customized for a specific DNN type (e.g., NLP transformer); while GPU, TIE and ETTE can widely support all the workloads. Area efficiency of GPU is not reported because of the limited access to GPU area measurement.

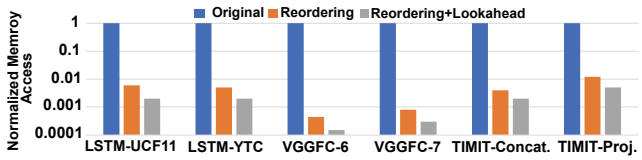
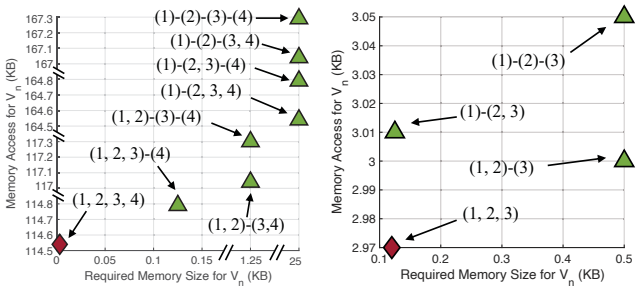


Figure 12: Data movement of V_n before and after using proposed computing scheme and lookahead-style processing.



(a) Elimination Phase. **(b) Expansion Phase.**

Figure 13: SRAM access and size incurred by $\mathcal{V}_n$ with different lookahead strategies on UCF11 (d_A=4 and d_B=3). $(m_i \dots m_j)$ means the lookahead computation from stage-m_i to stage-m_j.

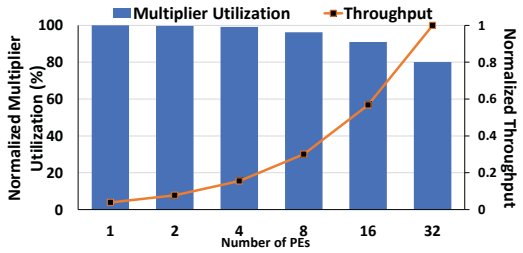


Figure 14: Normalized Multiplier utilization and throughput with different PEs on LSTM-YTC. Each PE has 16 multipliers.

Scalability & Utilization. Fig. 14 illustrates the processing throughput and multiplier utilization of ETTE with various PE settings. Here each PE is equipped with 16 multipliers. As shown in Fig. 14, ETTE shows high scalability performance. Also, it is observed that increasing the number of PEs gradually decreases utilization. This is because for a given shape of input matrix, it becomes more challenging to make the number of PEs be the exact factor of shape dimension with more PEs. Therefore proper trade-off between desired throughput demand and high PE utilization is needed. Our design sets PE amount as 16 to achieve good balance.

6 RELATED WORKS

DNN hardware accelerators have been extensively studied in recent years [3, 4, 10, 13, 16, 31, 49]. In particular, different types of compression model-oriented design, including sparse-aware [8, 21, 30, 41, 52] and quantization-aware [22, 60] architecture, have been proposed in the literature. Among these existing solutions, TIE [9] is the state-of-the-art TT-oriented DNN accelerator. On the algorithm aspect, TT decomposition has demonstrated its promising compression performance in several different practical DNN applications. However, the key drawbacks of existing TT-format DNN models are the insufficient reduction for computational costs and data movement. To the best of our knowledge, to date only [58] investigates to reduce the computation of TT-format CNN. ETTE enjoys three main advantages over [58]. First, the solution adopted in [58] focuses on accelerating TT-format CNNs, whose model redundancy is typically quite limited for TT compression; while ETTE aims for much broader spectrum of DNN types, including large-scale MLP, RNN, Transformer, which are the most commonly used scenarios for performing TT decomposition. Second, the method proposed in [58] is heuristic without theoretical guarantee; while ETTE can assure significant reduction of the computational and memory costs with rigid mathematical derivation and proof. Third, [58] is an algorithmic work that does not consider multi-stage data movement and efficient matrix transformation; while ETTE performs algorithm and hardware co-optimization to address all the three challenges of existing TT-format DNNs.

7 CONCLUSION

This paper develops ETTE, an efficient TT engine for high-performance DNN acceleration. By performing algorithm and hardware co-optimization efforts, ETTE brings significant reduction in computational cost, storage demand and data movement over the state-of-the-art DNN hardware solutions.

ACKNOWLEDGMENTS

This work was partially supported by National Science Foundation under Grant CCF-1937403 and CCF-1955909.

REFERENCES

[1] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 382–394.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[3] Xuyi Cai, Ying Wang, Xiaohan Ma, Yinhe Han, and Lei Zhang. 2022. DeepBurning-SEG: Generating DNN Accelerators of Segment-Grained Pipeline Architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1396–1413.

[4] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 367–379.

[5] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[6] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*. 3123–3131.

[7] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).

[8] Chunhua Deng, Yang Sui, Siyu Liao, Xuehai Qian, and Bo Yuan. 2021. GoSPA: an energy-efficient high-performance globally optimized sparse convolutional neural network accelerator. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1110–1123.

[9] Chunhua Deng, Fangxuan Sun, Xuehai Qian, Jun Lin, Zhongfeng Wang, and Bo Yuan. 2019. TIE: energy-efficient tensor train-based inference engine for deep neural network. In *Proceedings of the 46th International Symposium on Computer Architecture*. 264–278.

[10] Chunhua Deng, Miao Yin, Xiao-Yang Liu, Xiaodong Wang, and Bo Yuan. 2019. High-performance hardware architecture for tensor singular value decomposition. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–6.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2019. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 293–302.

[13] Hongxiang Fan, Thomas Chau, Stylianos I Venieris, Royson Lee, Alexandros Kouris, Wayne Luk, Nicholas D Lane, and Mohamed S Abdelfattah. 2022. Adaptable Butterfly Accelerator for Attention-based NNs via Hardware and Algorithm Co-design. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 599–615.

[14] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry Vetrov. 2016. Ultimate tensorization: compressing convolutional and fc layers alike. *arXiv preprint arXiv:1611.03214* (2016).

[15] Ruihao Gong, Xianglong Liu, Shenghu Jiang, Tianxiang Li, Peng Hu, Jiazhen Lin, Fengwei Yu, and Junjie Yan. 2019. Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 4852–4861.

[16] Yu Gong, Miao Yin, Lingyi Huang, Chunhua Deng, and Bo Yuan. 2022. Algorithm and Hardware Co-Design of Energy-Efficient LSTM Networks for Video Recognition with Hierarchical Tucker Tensor Decomposition. *IEEE Trans. Comput.* 71, 12 (2022), 3101–3114.

[17] Ruiqi Guo, Zhiheng Yue, Xin Si, Te Hu, Hao Li, Limei Tang, Yabing Wang, Leibo Liu, Meng-Fan Chang, Qiang Li, et al. 2021. 15.4 a 5.99-to-691.1 tops/w tensor-train in-memory-computing processor using bit-level-sparsity-based optimization and variable-precision quantization. In *2021 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 64. IEEE, 242–244.

[18] Udit Gupta, Samuel Hsia, Jeff Zhang, Mark Wilkening, Javin Pombra, Hsien-Hsin Sean Lee, Gu-Yeon Wei, Carole-Jean Wu, and David Brooks. 2021. RecPipe: Co-designing models and hardware to jointly optimize recommendation quality and performance. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 870–884.

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

[20] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems* 28 (2015), 1135–1143.

[21] Edward Hanson, Shiyu Li, Hai'Helen' Li, and Yiran Chen. 2022. Cascading structured pruning: enabling high data reuse for sparse DNN accelerators. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 522–535.

[22] Yifan Hao, Yongwei Zhao, Chenxiao Liu, Zidong Du, Shuyao Cheng, Xiaqing Li, Xing Hu, Qi Guo, Zhiwei Xu, and Tianshi Chen. 2022. Cambricon-P: A Bitflow Architecture for Arbitrary Precision Computing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 57–72.

[23] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*. 1389–1397.

[24] Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan Oseledets. 2020. Tensorized embedding layers. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 4847–4860.

[25] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. 2020. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 968–981.

[26] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[27] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. 2018. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, Vol. 120.

[28] Zheng Li, Soroush Ghodrati, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Mingu Kang. 2022. Accelerating attention through gradient-based learned runtime pruning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 902–915.

[29] Mingshuo Liu, Miao Yin, Kevin Han, Shiyi Luo, Mingju Liu, Ronald F DeMara, Bo Yuan, and Yu Bai. 2021. Algorithm and Hardware Co-Design Co-Optimization Framework for LSTM Accelerator using Fully Decomposed Tensor Train. *DAC (Work-in-Progress)* (2021).

[30] Zhi-Gang Liu, Paul N Whatmough, Yuhao Zhu, and Matthew Mattina. 2022. S2TA: Exploiting Structured Sparsity for Energy-Efficient Mobile CNN Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 573–586.

[31] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. 2022. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 993–1011.

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

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

[34] Yu Pan, Jing Xu, Maolin Wang, Jinmian Ye, Fei Wang, Kun Bai, and Zenglin Xu. 2019. Compressing recurrent neural networks with tensor ring for action recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4683–4690.

[35] Huy Phan, Miao Yin, Yang Sui, Bo Yuan, and Saman Zonouz. 2023. CSTAR: Towards Compact and STructured Deep Neural Networks with Adversarial Robustness. *AAAI* (2023).

[36] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.

[38] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-dcnn: Highly-scalable deep convolutional neural network using stochastic computing. *ACM SIGPLAN Notices* 52, 4 (2017), 405–418.

[39] Yuxin Ren, Benyou Wang, Lifeng Shang, Xin Jiang, and Qun Liu. 2022. Exploring extreme parameter compression for pre-trained language models. *arXiv preprint arXiv:2205.10036* (2022).

[40] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. 2018. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 78–91.

[41] Jong Hoon Shin, Ali Shafiee, Ardavan Pedram, Hamzah Abdel-Aziz, Ling Li, and Joseph Hassoun. 2022. Griffin: Rethinking Sparse Optimization for Deep Learning Architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 861–875.

[42] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. Pipelayer: A pipelined reram-based accelerator for deep learning. In *2017 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 541–552.

[43] Mingcong Song, Jiaqi Zhang, Huixiang Chen, and Tao Li. 2018. Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 66–77.

[44] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. 2018. In-situ ai: Towards autonomous and incremental deep learning for iot systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 92–103.

[45] Yang Sui, Miao Yin, Yi Xie, Huy Phan, Saman Aliari Zonouz, and Bo Yuan. 2021. Chip: Channel independence-based pruning for compact neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 24604–24616.

[46] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. 2017. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 13–26.

[47] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[48] Wenqi Wang, Yifan Sun, Brian Eriksson, Wenlin Wang, and Vaneet Aggarwal. 2018. Wide compression: Tensor ring nets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 9329–9338.

[49] Lizhi Xiang, Miao Yin, Chengming Zhang, Aravind Sukumaran-Rajam, P Sadayappan, Bo Yuan, and Dingwen Tao. 2023. TDC: Towards Extremely Efficient CNNs on GPUs via Hardware-Aware Tucker Decomposition. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 260–273.

[50] Jinqi Xiao, Chengming Zhang, Yu Gong, Miao Yin, Yang Sui, Lizhi Xiang, Dingwen Tao, and Bo Yuan. [n. d.]. HALOC: Hardware-Aware Automatic Low-Rank Compression for Compact Neural Networks. *AAAI* ([n. d.]).

[51] Yinchong Yang, Denis Krompass, and Volker Tresp. 2017. Tensor-Train Recurrent Neural Networks for Video Classification. In *International Conference on Machine Learning*. 3891–3900.

[52] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 744–762.

[53] Jinmian Ye, Linnan Wang, Guangxi Li, Di Chen, Shandian Zhe, Xinqi Chu, and Zenglin Xu. 2018. Learning compact recurrent neural networks with block-term tensor decomposition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 9378–9387.

[54] Chunxing Yin, Bilge Acun, Carole-Jean Wu, and Xing Liu. 2021. Tt-rec: Tensor train compression for deep learning recommendation models. *Proceedings of Machine Learning and Systems* 3 (2021), 448–462.

[55] Chunxing Yin, Da Zheng, Israt Nisa, Christos Faloutsos, George Karypis, and Richard Vuduc. 2022. Nimble GNN Embedding with Tensor-Train Decomposition. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 2327–2335.

[56] Miao Yin, Huy Phan, Xiao Zang, Siyu Liao, and Bo Yuan. 2022. Batude: Budget-aware neural network compression based on tucker decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 8874–8882.

[57] Miao Yin, Yang Sui, Siyu Liao, and Bo Yuan. 2021. Towards efficient tensor decomposition-based dnn model compression with optimization framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10674–10683.

[58] Miao Yin, Yang Sui, Wanzhao Yang, Xiao Zang, Yu Gong, and Bo Yuan. 2022. HODEC: Towards Efficient High-Order DEcomposed Convolutional Neural Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12299–12308.

[59] Miao Yin, Burak Uzkent, Yilin Shen, Hongxia Jin, and Bo Yuan. 2023. GOHSP: A Unified Framework of Graph and Optimization-based Heterogeneous Structured Pruning for Vision Transformer. *AAAI* (2023).

[60] Ali Hadi Zadeh, Mostafa Mahmoud, Ameer Abdelhadi, and Andreas Moshovos. 2022. Mokey: enabling narrow fixed-point inference for out-of-the-box floating-point transformer models. *arXiv preprint arXiv:2203.12758* (2022).

[61] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–12.

[62] Tehseen Zia and Usman Zahid. 2019. Long short-term memory recurrent neural network architectures for Urdu acoustic modeling. *International Journal of Speech Technology* 22, 1 (2019), 21–30.