TDC: Towards Extremely Efficient CNNs on GPUs via Hardware-Aware Tucker Decomposition

Lizhi Xiang*†
University of Utah
Salt Lake City, UT
xianglizhi456@gmail.com

Miao Yin*
Rutgers University
New Brunswick, NJ
miao.yin@rutgers.edu

Chengming Zhang†
Indiana University
Bloomington, IN
czh5@iu.edu

Aravind Sukumaran-Rajam
Meta and University of Utah
aravindsr@meta.com

P. Sadayappan
University of Utah
Salt Lake City, UT
saday@cs.utah.edu

Bo Yuan
Rutgers University
New Brunswick, NJ
bo.yuan@soe.rutgers.edu

Dingwen Tao‡†
Indiana University
Bloomington, IN
ditao@iu.edu

Abstract

Tucker decomposition is one of the SOTA CNN model compression techniques. However, unlike the FLOPs reduction, we observe very limited inference time reduction with Tucker-compressed models using existing GPU software such as cuDNN. To this end, we propose an efficient end-to-end framework that can generate highly accurate and compact CNN models via Tucker decomposition and optimized inference code on GPUs. Specifically, we propose an ADMM-based training algorithm that can achieve highly accurate Tucker-format models. We also develop a high-performance kernel for Tucker-format convolutions and analytical performance models to guide the selection of execution parameters. We further propose a co-design framework to determine the proper Tucker ranks driven by practical inference time (rather than FLOPs). Our evaluation on five modern CNNs with A100 demonstrates that our compressed models with our optimized code achieve up to 2.21× speedup over cuDNN, 1.12× speedup over TVM, and 3.27× over the original models using cuDNN with at most 0.05% accuracy loss.

CCS Concepts • **Computing methodologies** → **Massively parallel algorithms**; **Neural networks**; • **Software and its engineering** → *Source code generation*;

Keywords Convolutional neural network, inference, GPU, performance, model compression.

*Both authors contributed equally to the paper.
†This work was done when Lizhi Xiang, Chengming Zhang, and Dingwen Tao were with Washington State University.
‡Dingwen Tao is the corresponding author.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577478

1 Introduction

Not only are deep neural networks such as convolutional neural networks (CNNs) critical to traditional AI tasks [6, 15], but their applications in science domains are also growing [4, 20, 26]. However, in recent years, demands to improve the inference performance of CNNs have never been satisfied. Prior works approach faster and more efficient CNNs from different aspects, such as weight pruning [25], quantization [10, 18], and kernel factorization [34, 41].

Pruning techniques are proposed to remove the weight redundancy and produce an irregular sparse model to reduce the computational cost and memory bandwidth requirements of inference. However, this requires specialized sparsity-aware accelerators with significant effort in new hardware design and fabrication. Moreover, some researchers proposed to create a regular sparsity pattern through hardware-aware pruning algorithms [9, 25, 43], but their compression ratio is largely limited by the enforced sparsity patterns.

Quantization is to reduce the number of bits to store the model weights but faces two main challenges: (1) The quantized CNNs usually require arbitrary precisions (e.g., 4-bit weight), while widely used accelerators such as GPUs only support a limited range of precisions (e.g., half precision). (2) The compression ratio provided by quantization is still insufficient, considering many large CNN models.

Due to the above limitations, recent CNN model compression studies have mainly focused on a new direction—building the compact models using kernel factorization such as matrix/tensor decomposition. Specifically, matrix/tensor decomposition can represent a large matrix/tensor with the combination of multiple small matrices/tensors. Correspondingly, the number of the required representation parameters can be significantly reduced. However, matrix decomposition needs to convert high-order tensors to matrices and then perform matrix decomposition, causing loss of spatial information, whereas high-order tensor decomposition achieves much higher accuracy [41]. Thus, various compact CNN models have been developed using different tensor decomposition approaches [11, 19, 39]. Among those efforts, Tucker

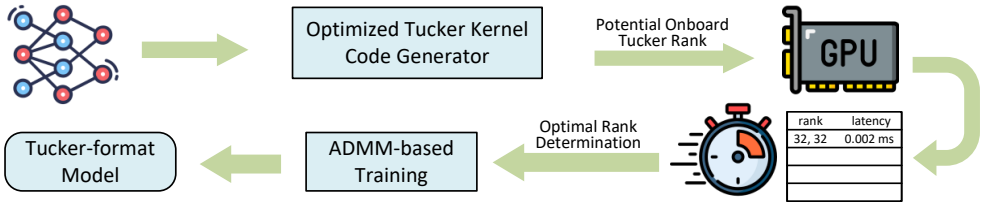

Figure 1. Overview of our TDC framework for generating TKD-compressed CNN models with high-performance inference code on GPUs.

decomposition (TKD) [38] finds its strengths in minimal modification of the hardware support, ultra-high compression ratio, and lower accuracy loss for CNN acceleration.

We identify three major challenges in accelerating TKD-compressed CNNs on state-of-the-art GPUs.

Lack of hardware-aware TKD algorithms for CNN acceleration. Once the target hardware platform is selected, Tucker ranks are the most important parameters impacting the practical latency. Unfortunately, existing TKD compression methods [13, 19] select ranks via either laborious efforts or iterative search, none of which are hardware aware.

Lack of software-aware TKD convolution algorithms for CNN acceleration. Tucker-based convolution decomposition can greatly decrease the computation FLOPs by reducing the number of the input channels and output channels of the original convolution layer. However, it is hard to translate the FLOPS reduction to real performance increment especially for small batch size such as one[1]. The root cause is that Tucker-based convolutions suffer from severe resource under-utilization when executed using existing software like cuDNN. For example, our evaluation (will be shown in Figure 8) illustrates that TKD-compressed ResNet18 model (with 2.7× FLOPs reduction) using cuDNN only achieves 1.47× speedup over the original model on A100 GPU.

Lack of performance model for tuning tucker-format convolutions on GPUs. Popular software for high performance convolutions such as TVM can generate fast convolution code by tuning hyper-parameters via machine learning (ML) approaches on a given set of convolution templates. However, those computation schemes do not work perfectly on the tucker-format convolutions on GPUs (will be detailed in Section 5.1). To solve this issue, an analytical performance model to guide the parameter tuning for tucker-format convolutions on GPUs is urgently needed.

Lack of performance-driven frameworks for highly efficient and accurate CNN inference on GPUs. Existing model compression methods are primarily driven by FLOPs reductions rather than real runtime reductions. However, it is still unclear how to design an end-to-end compression framework (including training highly accurate TKD-compressed models and building optimized inference code) to reduce the model size to the sweet spot where further compression cannot provide any performance benefit.

To this end, we propose an end-to-end framework, called Tucker Decomposition Convolution (TDC), that can generate highly accurate and compact TKD-compressed CNN models with optimized C++/CUDA code, which can be easily deployed on GPUs for high-performance inference, as illustrated in Figure 1. First, we propose an Alternating Direction Method of Multipliers (ADMM) based training algorithm for TKD-format convolutions and incorporate inference performance into the model generation. Second, we propose a new computation scheme for Tucker-format convolutions on GPUs. We also build a performance model to guide the selection of tiling sizes for our convolution kernel rather than performing an exhaustive search of the best tiling sizes. Third, we propose a co-design approach that can automatically determine the proper Tucker ranks based on a target speedup of inference performance. Based on the determined ranks, we can train a highly accurate TKD-compressed model and generate the optimized code for inference on GPUs. The main contributions of this paper are summarized as follows:

- We propose an algorithm-hardware co-designed, TKD-based compression framework that can achieve inference acceleration on GPUs with high model accuracy.
- We redesign the ADMM-based training algorithm for TKD-compressed models.
- We develop a new kernel for Tucker-format convolutions and a performance model for tiling size selection.
- We develop an approach to choose the proper Tucker ranks based on practical inference runtime and plug them into the training phase for model generation.
- Evaluation results on five modern CNN models with ImageNet demonstrate that our solution can speed up the end-to-end inference on A100 by 1.26~2.21× over cuDNN and 1.03~1.12× over TVM on the tested models. Moreover, compared with the original models, our compressed models achieve up to 3.27× speedup with at most 0.05% accuracy loss (even higher accuracy).

In Section 2, we introduce CUDA architecture, TKD-based model compression, and GPU convolution. In Section 4, we present our hardware-aware tucker decomposition for CNN acceleration. In Section 5, we discuss the details of our convolution scheme. In Section 6, we present our new co-design framework. In Section 7, we show our evaluation results. In Section 8, we conclude this work and discuss future work.

[1]A batch size of 1 is important for ML applications that require real-time responses [12], such as navigation systems in autonomous driving. As another example, the ability to auto-fill in suggestions when a retail customer enters a product name into an e-commerce site is required for optimal response.

2 Background

In this section we present the background about CUDA GPU architecture, tensor decomposition for model compression, and high-performance software for GPU convolutions.

2.1 CUDA Architecture

The thread is the basic programmable unit that allows GPU programmers to use massive numbers of CUDA cores. CUDA threads are grouped at different levels such as warp, block, and grid. Specifically, a group of 32 threads is called a *warp*. All threads in the same warp will execute the same instruction. However, if different threads in a warp follow different control paths, some threads are masked from performing any useful work. This situation is called a *warp divergence*, which is one of the fundamental factors that limit the performance of GPUs. Multiple warps are combined to form a thread *block*, and the set of thread blocks is called the *grid*.

2.2 Related Work on Tensor Decomposed DNN

Tensor Decomposition for DNNs. Higher-order tensor decomposition is an advanced technique to explore multi-dimensional linear correlations on the weight kernels via directly decomposing high-order kernels to multiple small tensor cores. Multiple tensor decomposition formats have been successfully applied in CNN compression. In prior work [21, 33], CP decomposition is adopted to factorize weight tensors to reduce computational and storage costs of convolutional layers. Tensor-train (TT) [30] and its variant tensor-ring (TR) [44] decomposition are also commonly used low-rank compression methods for CNNs [11, 28, 42]. In addition, [13, 19] propose to compress and accelerate CNN models via using Tucker decomposition, and [23] further develops a mixed CP and Tucker method to improve the model accuracy. However, these existing works still face several challenges for efficient CNN acceleration.

Limitations of CP-based Method [33]. In general, CP decomposition suffers two limitations for high performance CNN acceleration. At the algorithm level, the CP method, by its nature, has an inherent instability problem, thereby causing inferior model accuracy. Though the state-of-the-art (SOTA) CP method aims to mitigate this challenge, its accuracy is still unsatisfying. At the hardware level, the ranks of different modes in CP are always identical when decomposing a kernel tensor, which hinders the adjustment of the read-write loads in memories by changing the ranks ratio.

Limitations of TT-based Method [42]. The state-of-the-art TT compression [42] has two inherent drawbacks. First, as stated in [11], to compress a 4-D weight tensor, TT needs to first convert the original convolution to a huge matrix-matrix multiplication, and then uses TT to transform the multiplication to a TT-FC layer [28]. In such a conversion, important spatial information in filter dimensions (RxS) is lost and the convolution process no longer exists, thereby causing non-negligible accuracy drop. Second, according to tensor theory, when decomposing a 4-D weight tensor, with the similar rank setting TT brings lower computational cost reduction than other methods such as Tucker.

Limitations of Tucker-based Method [13]. The state-of-the-art Tucker work [13] still has unsatisfactory accuracy performance when compressing modern CNNs (e.g., ResNet) on ImageNet dataset (will be shown in Table 3). Moreover, the rank selection of [13] is not performed in a hardware-friendly way, thus limiting its runtime performance.

Tucker Method for RNN Compression. Tucker decomposition can also be used for compressing matrix-vector-multiplication-centered models, e.g., recurrent neural networks (RNNs). To that end, we can directly reshape the weight matrix to a high-order tensor, then decompose it into Tucker format. Under the Tucker format, the original matrix-vector multiplication is converted to a series of matrix multiplications in the execution. Considering that matrix multiplication is well optimized in existing software stacks and the Tucker-based CNN kernel optimization is still under-investigated, we focus on CNNs in this work.

2.3 High-Performance Convolutions on GPUs

Software for high-performance convolutions can be categorized into two classes: (1) library-based implementations and (2) code generator-based implementations. The most famous convolution library is cuDNN [5], which is widely used in open-source frameworks such as TensorFlow [1] and PyTorch [32]. Specifically, a library like cuDNN can take arbitrary convolution problem size as input and is very easy to use. However, the performance of library-based implementation is often less than the code generator-based approaches. This is because library-based implementations are optimized to handle a range of problem sizes, whereas the code generator can be targeted to optimize a given problem size, thus potentially achieving higher performance. Since the problem size is known, the code generator can precisely control factors that affect performance, such as tile sizes, thread coarsening levels, and memory placement strategies. For example, TVM [3] is a well-known code generation framework that generates fast convolution code by tuning hyper-parameters on given convolution templates. However, unlike TVM, which relies on heavy auto-tuning and ML processes, our code-generation-based approach employs analytical performance models to efficiently select the best execution parameters for both high performance and accuracy.

3 Basics of TKD-based Convolution

In this paper, bold calligraphic letters, bold capital letters and bold lowercase letters represent high-order tensors, matrices and vectors, respectively, e.g., $\boldsymbol{\mathcal{A}}$, $\boldsymbol{A}$, $\boldsymbol{a}$; non-bold letter with indices $\mathcal{A}(i_1, \cdots, i_k)$ denotes the $(i_1, \cdots, i_k)$-th entry

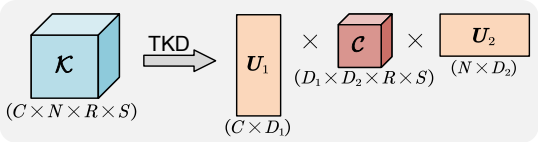

Figure 2. The original 4D convolution kernel is converted to three components in a Tucker-format convolution layer.

of a k-th order tensor $\mathcal{A}$; $A(i_1, i_2)$ and $a(i)$ denote the corresponding entry of the matrix $\boldsymbol{A}$ and vector $\boldsymbol{a}$, respectively. All the notations are summarized in Table 1.

Table 1. Notation summary.

C	# i/p channels	N	# o/p channels
H	image height	W	image width
R	stencil height	S	stencil width
$\mathcal{X}$	input tensor	$\mathcal{Y}$	output tensor
$\mathcal{K}$	kernel tensor		

Give a k-th order tensor $\mathcal{A} \in \mathbb{R}^{N_1 \times \cdots \times N_k}$, with Tucker decomposition, it can be generally represented as

$$\mathcal{A}(i_1, \cdots, i_k) = \sum_{d_1, \cdots, d_k}^{D_1, \cdots, D_d} C(d_1, \cdots, d_k) \cdot U_1(i_1, d_1) \cdots U_k(i_k, d_k).$$

$\boldsymbol{C} \in \mathbb{R}^{D_1 \times \cdots \times D_d}$ is called *core tensor*, $\{\boldsymbol{U}_i \in \mathbb{R}^{N_i \times D_i}\}_{i=1}^{k}$ are called *factor matrices*, $[D_1, \cdots, D_d]$ are called *Tucker ranks*.

For compressing a convolutonal layer with kernel tensor $\mathcal{K} \in \mathbb{R}^{C \times N \times R \times S}$ using Tucker decomposition, to preserve the spacial information, we only decompose the first mode and the second mode, i.e., the output and the input channel. Hence, kernel $\mathcal{K}$ can be decomposed as

$$\mathcal{K}(c, n, r, s) = \sum_{d_1, d_2}^{D_1, D_2} C(d_1, d_2, r, s) U_1(c, d_1) U_2(n, d_2), \quad (1)$$

where $\boldsymbol{C} \in \mathbb{R}^{C \times N \times D_1 \times D_2}, \boldsymbol{U}_1 \in \mathbb{R}^{C \times D_1}, \boldsymbol{U}_2 \in \mathbb{R}^{N \times D_2}$ are the compressed components to store and compute in the Tucker-format convolution layer. The decomposed components are illustrated in Figure 2.

With the above Tucker-format decomposed kernel, a convolution layer can be performed with three small consecutive convolutions. Specifically, for an input tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{H \times W \times C}$, the output tensor $\boldsymbol{\mathcal{Y}} \in \mathbb{R}^{H' \times W' \times N}$ is computed by

$$\mathcal{Z}_1(h, w, d_1) = \sum_{c=1}^{C} \mathcal{X}(h, w, c) U_1(c, d_1), \quad (2)$$

$$\mathcal{Z}_2(h', w', d_2) = \sum_{r=1}^{R} \sum_{s=1}^{S} \sum_{d_1}^{D_1} \mathcal{Z}_1(h, w, d_1) C(d_1, d_2, r, s), \quad (3)$$

$$\mathcal{Y}(h', w', n) = \sum_{d_2}^{D_2} \mathcal{Z}_2(h', w', d_2) U_2(d_2, n), \quad (4)$$

where $h = h' + r - 1, w = w' + s - 1$.

The above Tucker-format convolution is mathematically equivalent to the original convolution. Figure 3 illustrates the Tucker-format convolution. The first operation Eq. (2) is essentially a channel-wise 1×1 convolution that transforms original channels C to the smaller latent channels D_1. Then

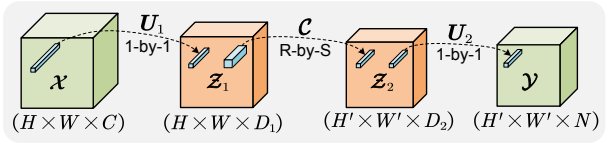

Figure 3. Illustration of Tucker-format convolution.

the computation Eq. (3) can be considered as the regular 2D convolution with the same kernel size R, S from channels D_1 to D_2 (called *core convolution*). The last computation Eq. (4) is also a channel-wise 1×1 convolution that transforms the latent channels D_2 to the original output channels N.

Benefits from hardware perspective. Note that regular R-by-S convolution is generally more sensitive to size than one-by-one channel-wise convolution. With this Tucker-format convolution, one can customize the second convolution by defining D_1 and D_2 as the hardware such as GPU friendly sizes, then use channel-wise convolution to complement the unbalanced custom channel sizes.

Computation and storage complexity. The overall storage complexity is number of parameters of factor matrices $\boldsymbol{U}_1, \boldsymbol{U}_2$ and core tensor $\boldsymbol{C}$, i.e., $C \cdot D_1 + N \cdot D_2 + R \cdot S \cdot D_1 \cdot D_2$. The overall computation complexity is also the sum of FLOPs of the three convolutions, i.e., $H \cdot W \cdot C \cdot D_1 + H' \cdot W' \cdot R \cdot S \cdot D_1 \cdot D_2 + H' \cdot W' \cdot N \cdot D_2$. The Tucker ranks D_1, D_2 are the main hyper-parameters that decide the overall complexity. Compared with the original convolution layer, the parameters and FLOPs reduction ratio γ_P, γ_F are given by:

$$\gamma_P = \frac{C \cdot N \cdot R \cdot S}{C \cdot D_1 + R \cdot S \cdot D_1 \cdot D_2 + N \cdot D_2}, \quad (5)$$

$$\gamma_F = \frac{H' \cdot W' \cdot R \cdot S \cdot C \cdot N}{H \cdot W \cdot C \cdot D_1 + H' \cdot W' \cdot D_2 \cdot (R \cdot S \cdot D_1 + N)}. \quad (6)$$

4 Hardware-aware Tucker Decomposition

In this section, we describe our proposed hardware-aware tucker decomposition approach for CNN model compression.

4.1 Training Tucker-format Models with ADMM

Limitation of direct training. The capacity of the Tucker-format model is lower and its depth becomes larger than the original model, thus it is more sensitive to hyper-parameters settings. As a result, directly training Tucker-format models from scratch suffers from significant performance degradation (Table 2). Another way to obtain a Tucker-format model is to decompose a pre-trained model, and then retrain. In this case, the pre-trained model is generally full rank, and decomposing it to Tucker format leads to a non-negligible approximation error. Thereby, the accuracy cannot be recovered in the Tucker-format model even after a long retraining.

Table 2. Accuracy comparison between directly training and our ADMM-based compression for ResNet-20 on CIFAR-10.

Method	Top-1 (%)	FLOPs↓
Baseline	91.25	N/A
Direct Compression	87.41	60%
ADMM-based	91.02	60%

Inspired by the previous optimization work [42], we propose an ADMM-based training technique to obtain high-accuracy Tucker-format models by gradually imposing low-Tucker-rank properties onto the original models during training process. The objective for compressing a baseline model using optimization-based training is given as

$$\min_{\mathcal{W}} \ \ell(\mathcal{K}), \ \text{s.t. rank}(\mathcal{K}) \leq [D_1^*, D_2^*], \tag{7}$$

where ℓ is the loss function of the DNN model, rank$(\cdot)$ returns the latent Tucker ranks of the kernel tensor $\mathcal{K}$, and D_1^*, D_2^* are the desired Tucker ranks.

Since rank$(\cdot)$ is non-differentiable, the above training objective Eq. (7) cannot be solved by a regular optimizer. Fortunately, Alternating Direction Method of Multipliers (ADMM) algorithms [2] can efficiently solve this non-convex problem via alternatively solving two simple sub-problems. As a result, the training is converted to iterative optimization.

First, with scaled augmented Lagrangian form, the original problem Eq. (7) can be rephrased as

$$\min_{\mathcal{K}, \widehat{\mathcal{K}} \in \mathcal{Q}} \max_{\mathcal{M}} \ \ell(\mathcal{K}) + \frac{\rho}{2}\|\mathcal{K} - \widehat{\mathcal{K}} + \mathcal{M}\|_F^2 - \frac{\rho}{2}\|\mathcal{M}\|_F^2, \tag{8}$$

where $\widehat{\mathcal{K}}$ is an introduced variable whose shape is identical to $\mathcal{K}$, $\mathcal{Q}$ is the set where kernel tensors satisfy the target constraints, i.e., $\mathcal{Q} = \{\widehat{\mathcal{K}} | \text{rank}(\widehat{\mathcal{K}}) \leq [D_1^*, D_2^*]\}$, $\mathcal{M}$ is the dual multiplier, and ρ is the penalty coefficient.

Then, with ADMM, the alternate problem Eq. (8) can be split into two sub-problems which can be independently solved, and all the variables are iteratively updated in the training process.

$\mathcal{K}$-update. The first optimization step is to update the original kernel tensor, which is to solve

$$\min_{\mathcal{K}} \ \ell(\mathcal{K}) + \frac{\rho}{2}\|\mathcal{K} - \widehat{\mathcal{K}} + \mathcal{M}\|_F^2. \tag{9}$$

The above objective is essentially minimizing the original loss function with L_2-regularization. Hence, $\mathcal{K}$ can be updated via standard mini-batch SGD as

$$\mathcal{K} \leftarrow \mathcal{K} - \alpha\left(\frac{\partial \ell(\mathcal{K})}{\partial \mathcal{K}} + \rho(\mathcal{K} - \widehat{\mathcal{K}} + \mathcal{M})\right), \tag{10}$$

where $\frac{\partial \ell(\mathcal{K})}{\partial \mathcal{K}}$ is the gradient in the back-propagation and α is the learning rate.

$\widehat{\mathcal{K}}$-update. The following optimization step is to let $\widehat{\mathcal{K}}$ satisfy the desired Tucker ranks, i.e.,

$$\min_{\widehat{\mathcal{K}}} \ \|\mathcal{K} - \widehat{\mathcal{K}} + \mathcal{M}\|_F^2, \ \text{s.t.} \ \widehat{\mathcal{K}} \in \mathcal{Q}. \tag{11}$$

According to [2], this sub-problem can result in an analytical solution by directly projecting $\mathcal{K} + \mathcal{M}$ to set $\mathcal{Q}$. Hence, $\widehat{\mathcal{K}}$ can be explicitly updated via

$$\widehat{\mathcal{K}} \leftarrow \textbf{proj}(\mathcal{K} + \mathcal{M}), \tag{12}$$

where **proj** is the truncated-HOSVD that truncates the smallest singular values of mode-1 and mode-2 matricization to satisfy the rank constraints. To be specific, suppose $\mathcal{T} = \mathcal{K} + \mathcal{M} \in \mathbb{R}^{C \times N \times R \times S}$, thus the mode-1 and mode-2 matricization of $\mathcal{T}$ is $\boldsymbol{T}_{(1)} \in \mathbb{R}^{C \times NRS}$ and $\boldsymbol{T}_{(2)} \in \mathbb{R}^{N \times CRS}$, respectively. Performing matrix SVD on $\boldsymbol{T}_{(1)}$ and $\boldsymbol{T}_{(2)}$, we have $\boldsymbol{U}_1\Sigma_1\boldsymbol{V}_1 = \boldsymbol{T}_{(1)}$ and $\boldsymbol{U}_2\Sigma_2\boldsymbol{V}_2 = \boldsymbol{T}_{(2)}$, where Σ_1 and Σ_2 are diagonal matrices with singular values in descending orders. To project $\mathcal{T}$ with target Tucker ranks, truncating smallest singular values in Σ_1 and Σ_2, with TKD we can obtain $\widehat{\boldsymbol{U}}_1 \in \mathbb{R}^{C \times D_1^*}$, $\widehat{\boldsymbol{U}}_2 \in \mathbb{R}^{C \times D_2^*}$, and $\widehat{\mathcal{C}} \in \mathbb{R}^{C \times N \times D_1^* \times D_2^*}$. Correspondingly, we can use Eq. 1 to recover back to the projected tensor $\widehat{\mathcal{K}}$ with the truncated $\widehat{\boldsymbol{U}}_1$, $\widehat{\boldsymbol{U}}_2$ and $\widehat{\mathcal{C}}$.

$\mathcal{M}$-update. The update step for dual multiplier $\mathcal{M}$ can be considered as the optimization for the dual problem (the maximizing problem described in Eq. 8) by gradient descent with fixed step size $\frac{1}{\rho}$, i.e., $\mathcal{M} \leftarrow \mathcal{M} + \mathcal{K} - \widehat{\mathcal{K}}$.

In summary, the above three updates are alternatively performed during the optimization-incorporated training.

4.2 Plug-in Hardware-Aware Constraints

Even though one can obtain a highly accurate Tucker-format compressed model using the introduced optimization-based training technique in the previous subsection, it is still challenging to get the desired on-device speedup. This is because (1) the decomposed model architecture is determined by the target Tucker ranks in Eq. (7), which are set empirically. Thus, it difficult to fit all practical hardware devices; (2) the compressed Tucker-format models are not well supported by existing convolution software (e.g., cuDNN) and deep learning framework (e.g., PyTorch), thereby leading to undesired latency in terms of theoretical FLOPs reduction.

To address this challenge, we reformulate the optimization objective Eq. (7) such that the rank setting with practical GPU latency can be considered as plug-in constraints, i.e.,

$$\min_{\mathcal{W}} \ \ell(\mathcal{K}), \ \text{s.t. rank}(\mathcal{K}) \leq \mathcal{P}_{\text{device}}. \tag{13}$$

Specifically, we first generate a benchmark with GPU performance for our designed kernel. Then, we find the proper Tucker ranks that can obtain the best latency and satisfy the overall compression budget. Correspondingly, we plug these rank settings as $\mathcal{P}_{\text{device}}$ in the above Eq. (13).

We will introduce TDC in detail in Section 6 after we present our design for efficient Tucker-format convolution.

5 Our Convolution Kernel Design

In this section, we first discuss the limitations of TVM's convolution scheme and then propose our convolution scheme with a performance analysis.

5.1 Discussion on TVM's Convolution Scheme

TVM follows direct convolution to design its scheme [3]. Specifically, the threads in each thread block are responsible for computing output positions with continuous coordinates on the same panel. Thus, the threads responsible for nearby positions have some overlapping from the perspective of the

input tensor. Moreover, all threads in the same thread block require the same kernel weight elements. Each thread block keeps two shared memory buffers: one is for the input tensor and the other one is for the weight tensor. After loading the input to the shared buffer, each thread starts its own computation. Listing 1 shows its pseudo code.

Limitations of TVM's convolution scheme. First, TVM introduces two thread synchronizations because its scheme uses the shared memory to achieve data reuse from both kernel tensor and input tensor (lines 1 and 2) at each iteration of C (input channel). Second, TVM's scheme splits the workloads on height and width dimensions for the threads. Hence, the tiles of input tensor are different for each thread. Accordingly, there exists overlapping of the tensor tile for each thread. To load all the required tiles of input tensor to the shared memory will potentially limit the achievable GPU occupancy. Moreover, since the tiles of input tensor are different across threads, they cannot load all tiles across C at one time and hence need to traverse the C loop with additional thread synchronizations (line 6). Third, TVM's scheme does not split workloads across the input channels. However, convolutions in Tucker format have relatively small ranks for input/output channels, resulting in a smaller workload. Thus, not splitting across input channels leads to the under-utilization of resources.

```
1  //Input: Input tensor 𝒳, Conv kernel 𝒦
2  //Output: Output tensor 𝒴
3  shared float shared_input[]
4  shared float shared_kernel[]
5  float local_input[]
6  float local_kernel[]
7  float local_compute[]
8  for c = 0 to C:
9    threads_synchronization()
10   load input to shared_input
11   load kernel to shared_kernel
12   threads_synchronization()
13   load local_input
14   load local_kernel
15   for n =0 to N:
16      local_compute
17 write_out
```

Listing 1. Pseudocode of TVM convolution design.

5.2 Proposed Tucker-format Convolutions

Our proposed scheme for Tucker-format convolutions is also a direct-convolution-based scheme. In our design, we first tile the input tensor across its height (H), width (W), and input channel (C). There are $\frac{H}{TH} \times \frac{W}{TW} \times \frac{C}{TC}$ thread blocks in total, and each tile with the size of $TH \times TW$ is mapped to a thread block. We then assign every thread block N threads (where N is the number of output channels) and map each thread to one output channel. After that, each thread block will load a cube with the size of $(TH + R - 1) \times (TW + S - 1) \times TC$ of the input tensor to its shared memory at the very beginning. After loading the tile of the input tensor to the shared memory, each thread will compute its contribution to the input tensor and save the intermediate result to a temporary array (in register) with the size of $TH \times TW$. In summary, each thread performs the following computation.

$$\mathcal{Y}(n, th, tw) = \sum_{c=0}^{TC-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathcal{I}(c, th - r, tw - s) \times \mathcal{K}(n, c, r, s)$$

Finally, after finishing the computation, each thread will write the final output result from its temporary array to its mapped output channel. Hence, the input tensor is reused across all the output channels in our design. In other words, all the threads inside the thread block need the same tile of the input tensor. Note that our proposed scheme splits the workload across the input channels, which not only provides a higher degree of parallelism but also enables us to load the input tensor at once. This can help avoid thread synchronizations which occur in the TVM's scheme. Furthermore, since each thread is responsible for a different output channel, there is no atomic writing required inside a thread block.

Still, this scheme has a limitation—although splitting the workload across all output channels helps us get a fully reused input tensor and avoids thread synchronizations, the data volume in the kernel is high, and the kernel data loaded for each thread is not usable for other threads. Thus, the data reuse rate in this scheme is lower than TVM's scheme.

To mitigate this issue, we use the format of CRSN for the kernel tensor to reduce the time overhead of loading the kernel data. This is because by using the CRSN format, the kernel tensor loading will be fully coalesced, which can reduce the time overhead. Note that the kernel tensor format conversion can be completely done offline once, which will not affect the inference performance. More details about our proposed convolution scheme can be found in Listing 2.

```
//Input: Input tensor 𝒳, Conv kernel 𝒦
//Output: Output tensor 𝒴
shared input_tile[TC][(TH+R-1)*(TW+S-1)]
float temp_result[TH][TW], kernel[R][S]
unsigned int tile_tc_id = blockId/(H/TH * W/TW)
unsigned int tile_id = blockId%(H/TH * W/TW)
unsigned int tile_h_id = tile_id/(W/TW)
unsigned int tile_w_id = tile_id%(W/TW)
unsigned int output_n = threadIdx.x
//copy tiled input tensor from global to shared
copy(input_tile, 𝒳)
syncthreads() //synchronize all threads in a thread block
for c = 0 to TC:
  copy(kernel, 𝒦,n,c+tile_tc_id*TC)
  for (v,h,w) in (input_tile):
    for r = 0 to R
      for s = 0 to S
        y_out = h - r
        x_out = w - s
        if y_out<0 or x_out< 0 or y_out>TH or x_out>TW:
            continue
        result = v * kernel[r][s]
        temp_result[y_out*TW+x_out] += result
// Write the output back to memory
for th to TH:
  for tw to TW:
    y = tile_id/(W/TW)*TH+th
    x = tile_id%(W/TW)*TW+tw
    atomicAdd(Y[H*W*N+y*W*N+x*N+n], temp_result[th*TW+tw])
```

Listing 2. Pseudocode of our core convolution design.

5.3 Computation latency analysis

In our convolution kernel for Tucker core tensors, tiling plays an important role to determine the data movement volume and computation resource utilization. In the following discussion, we will model computation and memory latency based on a given tiling size and convolution shape. Based on these models, we will propose our tiling size selection.

From the perspective of resource utilization: the total number of thread blocks can be denoted as $\text{num\_blks} = \frac{H}{TH} \times \frac{W}{TW} \times \frac{C}{TC}$. Since each thread block has N threads, the total number of threads in our core convolution kernel is $\text{Num\_ths} = \text{Num\_blks} \times N$. The number of FLOPs for each thread block is $\text{flops\_blk} = 2 \times (TH + R - 1) \times (TW + S - 1) \times TC \times N \times R \times S$. The peak performance for each thread block is $\text{blk\_peak} = \text{GPU\_peak} \times \frac{N}{\text{GPU\_ths}}$. Thus, the computation latency for each thread block can be estimated as $\text{comp\_latency\_blk} = \frac{\text{FLOPs\_blk}}{\text{Blk\_peak}}$. If we combine all the equations above, we can derive the computation latency for a thread block as $\text{comp\_latency\_blk} =$

$$\frac{2 \cdot (TH + R - 1) \cdot (TW + S - 1) \cdot TC \cdot \text{GPU\_ths} \cdot R \cdot S}{\text{GPU\_peak}}$$

When launching a GPU kernel, if the number of threads requested is greater than the total threads that a GPU can provide, the computation will be divided into multiple waves, and the kernel can be completed only after the last wave finishes. We formulate the number of GPU waves as

$$\begin{aligned} \text{comp\_waves} &= \left\lceil \frac{\text{Num\_ths}}{\text{GPU\_ths} \times \text{Occupancy}} \right\rceil \\ &= \left\lceil \frac{\frac{H}{TH} \times \frac{W}{TW} \times \frac{C}{TC} \times N}{\text{GPU\_ths} \times \text{Occupancy}} \right\rceil. \end{aligned} \tag{14}$$

Eq. (14) illustrates that the total number of GPU waves is determined by the convolution problem shape (H, W, C, N), tiling sizes (TH, TW, TC) and the GPU hardware metrics (GPU_ths). Note that the occupancy can be estimated by the hardware metrics such as shared memory size, register file size along with the given tiling sizes (TH, TW, TC); or simply put we can obtain it by querying via the NVCC compiler to get the exact occupancy for specific tiling sizes.

The computation latency for each wave is equal to the computation latency of a thread block as all the thread blocks in a wave have the same FLOPs for a dense convolution. Hence, the total computation latency for the kernel is

$$\text{comp\_latency} = \text{comp\_waves} \cdot \text{comp\_latency\_blk}. \tag{15}$$

When only considering the computation latency, an optimized tiling configuration (TH, TW, TC) for a given convolution shape (H, W, C, N) is to minimize *Comp_latency* under the constraint of $TH \le H, TW \le W, TC \le C$.

5.4 Memory latency analysis

On the GPU, besides computation latency, memory latency also plays an important role in the overall latency. Different tiling sizes affect not only the computation latency but also the data movement, which further influences the memory latency. Different tiling selections will affect the data movement from different GPU memory hierarchies such as shared memory, register, and global memory. Since global memory is the slowest memory, we only analyze the data movement from the perspective of global memory. For a given tiling size (TH, TW, TC) and convolution shape (H, W, C, N), the data movement volume of convolution kernel can be denoted as

$$volume_k = \left\lceil \frac{H}{TH} \right\rceil \times \left\lceil \frac{W}{TW} \right\rceil \times C \times N. \tag{16}$$

The input tensor volume can be denoted as

$$\begin{aligned} volume_x = \left\lceil \frac{H}{TH} \right\rceil \times \left\lceil \frac{W}{TW} \right\rceil \times C \times \\ (TH + R - 1) \times (TW + S - 1). \end{aligned} \tag{17}$$

The output tensor volume can be denoted as

$$volume_y = H \times W \times N \times \frac{C}{TC}. \tag{18}$$

Thus, the total data-movement volume can be denoted as

$$volume\_total = volume_x + volume_k + volume_y. \tag{19}$$

Based on Eq. (16), (17), (18), we can also find an optimized tiling option to minimize the overall data movement latency.

5.5 Analytical model for tiling selection

The previous analyses show that the tiling option has a significant effect on both computation latency (*comp_latency*) and memory latency (*memory_latency*). An ideal tiling option will minimize both latencies. However, the best tiling option for the computation latency may not be the best option for the memory latency, vice versa. Hence, the best tiling option should minimize the overall latency.

We note that a previous work [31] demonstrates that dense convolution is likely to be compute bound. Thus, in our tiling analytical model, we put the computation latency in prior to the memory latency. Specifically, our tiling analytical model first computes the *comp_latency* for all the tiling size options. Based on the results, we sort the tiling configurations in increasing order. Then, we select the top (5%, 15%) of tiling configurations (A100, 2080Ti) as our candidates. For those selected candidates, we pick the one which has the minimum *memory_latency* as our final tiling selection. Our tiling analytical model allows to generate high-performance convolution code on GPUs for a given shape without going through a costly parameter tuning process required by TVM.

In addition to our analytical modeling, we also provide an auto-tuning script for core convolutions as part of our framework. The auto-tuning process is based on an exhaustive search of tiling size. For a give core convolution shape, we run all tiling options (the number of tiling options is $H \times W \times C$) and obtain the best one based on their actual latency. The exhaustive search guarantees to produce the best tiling option for a given convolution shape. However, it requires a

high time overhead of offline tuning like TVM. The code generated by our analytical model achieves a slightly worse performance compared with exhaustive search (called "oracle"), i.e., ~25% performance drop on both A100 and 2080Ti machines, while it is still 1.5× faster than TVM on average. The performance comparison of the code generated with "oracle" and "model" approaches will be shown later.

Our analytical modeling provides the user an option to generate a fast Tucker-friendly convolution kernel for quick deployment; while if the user looks for extreme performance and does not care about the costly offline tuning process, using the exhaustive search is recommended.

6 Our Proposed Co-design Framework

In our Tucker decomposition, an original convolution kernel will be two 1×1 convolutions and one core convolution. The filter of our core convolution has the same height (R) and width (S) compared with the original convolution. The input channels (D_1) and output channels (D_2), i.e., the ranks for the core convolution (C, N), for our core convolution are smaller than the original convolution. The smaller (D_1, D_2), the more FLOPs we can reduce. Theoretically, the ranks (D_1, D_2) for our core convolution can be reduced to any size. Greedily, we can reduce them until (1,1). However, if the ranks are too small, it causes two problems: (1) It would cause a significant drop on model accuracy (hard to recover). (2) It would decrease the potential for pruning the next convolution layers.

The fundamental idea of our proposed co-design framework is only considering the reduced ranks that can provide a performance/runtime benefit. In Section 5, we have demonstrated that different convolution shapes may end up to with the same computation latency by adjusting tiling size. Figure 4 shows the overall latency changing trend for two convolutions with their C, H, W fixed and N changes from 32 to 256 on 2080Ti machine. The blue line has $C = 64, H = 28, W = 28$ and the red line has $C = 64, H = 14, W = 14$. Both of their running time trend lines are in a monotonic staircase when the output channels (N) increase. It shows the fact that the overall latency may remain the same when the FLOPs changes. The most important reason is that an optimized tiling strategy can increase the parallelism level for a larger problem to reduce the overall latency. Intuitively, we can employ this to guide our decomposition and training process to avoid the "over rank reduction" situation.

For a given trained CNN, users need to first carefully determine a decomposition budget B (i.e., the target FLOPs reduction ratio). An aggressive budget may lead to accuracy drop, so we recommend to choose B based on state-of-the-art FLOPs reduction ratios. While in the scenario where there is no FLOPS reduction ratio for reference, we recommend to start the budget from 10%; we can further increase the budget as long as the current budget meets the accuracy requirement after training for some epochs.

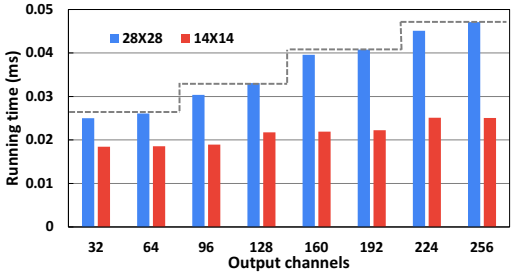

Figure 4. Runtime of convolutions with different numbers of output channels (the number input channels is fixed to 64).

After that, we select our decomposition ranks for retraining based on B. The entire process of our rank selection is shown in Figure 5. Specifically, the first step is to generate the micro-kernel CUDA code for all possible decomposed convolutions for each original convolution in the model by applying their optimized tiling size (as described in Section 5), and empirically run them to get a performance table T. Theoretically, for a convolution layer with shape (C, N, H, W), there exist $C \times N$ decomposition candidates because the decomposed input channel can be any number from $\{1, 2, 3..., C-1\}$ and the output channel can be any number from $\{1, 2, 3..., N-1\}$. However, reducing the input and output channels by one at a time is unnecessary for two reasons: (1) it reduces FLOPs by very small amounts, and (2) it may cause idle threads in some convolution schemes as a GPU warp performs as a group of 32 threads. To this end, we reduce the input channels and output channels by 32 each time. Hence, there exist $\frac{C}{32} \times \frac{N}{32}$ decomposition candidates.

For each convolution layer, assuming the number of input channels is C and the number of output channels is N. With a budget of B, we choose the (D_1, D_2) such that the overall FLOPs in the Tucker-format model is less than the budget, i.e., $\mathcal{P}(D_1, D_2) \lessapprox B$, where D_1 and D_2 are the Tucker ranks that determine the input channel and output channel of the core convolution. Note that since Tucker decomposition allows to decrease both C and N, there will be multiple (D_1, D_2) candidates. Thus, we further look up the performance table T to select (D_1, D_2) based on its performance/latency.

Considering that the Tucker decomposition brings two more 1×1 convolutional layers, and the extra two 1×1 of kernel launch time may even cause the performance worse than the original convolutional layer, we use a threshold θ

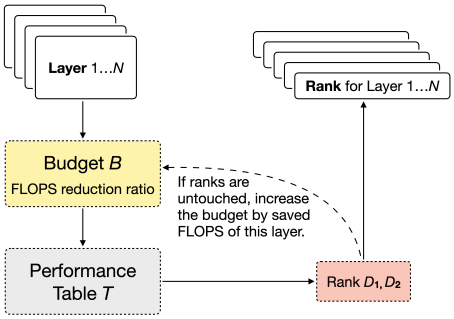

Figure 5. Workflow of proposed hardware-ware rank selection.

Algorithm 1: TDC: Proposed co-design framework for hardware-aware Tucker decomposed training.

Input: Pre-trained CNN kernel $\mathcal{K}$, decompose budget B, optimized benchmark table T, training epochs E;
Output: Hardware-aware Tucker decomposed kernels $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, $\boldsymbol{C}$.
$D_1^*, D_2^* = \max\{\arg\min_{\mathcal{P}(D_1,D_2)\leq B} T(D_1, D_2)\}$; //Minimize the overall latency while maximize ranks under the budget
Plug D_1^*, D_2^* to Equation (13);
$\widehat{\mathcal{K}} \leftarrow \mathcal{K}$, $\mathcal{M} \leftarrow 0$;
//ADMM-incorporated training
while $e < E$ **do**
  $\mathcal{K} \leftarrow \mathcal{K} - \alpha\left(\frac{\partial \ell(\mathcal{K})}{\partial \mathcal{K}} + \rho(\mathcal{K} - \widehat{\mathcal{K}} + \mathcal{M})\right)$;
  $\widehat{\mathcal{K}} \leftarrow \textbf{proj}(\mathcal{K} + \mathcal{M})$;
  $\mathcal{M} \leftarrow \mathcal{M} + \mathcal{K} - \widehat{\mathcal{K}}$;
end
$\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{C} = \text{TKD}(\mathcal{K})$; //Decompose to Tucker format
Fine-tune $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, $\boldsymbol{C}$ in the Tucker-format model.

to determine whether (D_1, D_2) should be selected. Specifically, assuming (D_1, D_2) has a latency of t_1 and the original (C, N) convolution layer has a latency of t_2, we will not apply Tucker decomposition on this layer if $t_1 \geq (1-\theta) \times t_2$. For simplicity, we choose $\theta = 15\%$ in our experiments. Furthermore, if this convolution layer is unchanged, it will save $C \times R \times S \times H \times W \times N \times B$ FLOPs in budget which can be further utilized; in other words, we can increase B by the saved FLOPs reduction to the remaining convolution layers.

Finally, after going through all convolution layers, we can obtain (D_1^*, D_2^*) as our target ranks for Tucker decomposition and plug them into Equation (13) to perform the ADMM-based training to fine-tune for several epochs, obtaining the high-performance model with the lowest latency. Algorithm 1 shows the detail of our co-designed training process.

7 Performance Evaluation

7.1 Experimental Setup

We use two platforms for evaluation: (1) Nvidia GTX 2080 Ti (68 SMs, 11 GB) machine running Ubuntu 20.04 LTS and, and (2) Nvidia Ampere A100 (108 SMs, 80GB) machine running Ubuntu 18.04.4 LTS. We use CUDA 11.3.1 paired with cuDNN 8.2.0 on the 2080Ti platform and CUDA 11.6.1 paired with cuDNN 8.2.1 on the A100 platform. These platforms represent two different GPU architectures (Ampere and Turing), which are designed for two different user groups (consumer and enterprise). On both of the platforms, we use TVM 0.7.0 paired with LLVM 10.0.0. For cuDNN's convolution operations, regarding the layerwise performance evaluation, we compare our micro-kernel with three different cuDNN convolution methods (i.e., GEMM-based convolution `IMPLICIT_GEMM` [7], WINOGRAD-based convolution `WINOGRAD` [29], and FFT-based convolution `FFT` [29]) and TVM; regarding the end-to-end performance evaluation, we fix the cuDNN convolution method as `IMPLICIT_GEMM` [7] since it has the best performance compared with the other two cuDNN convolution methods on the 2080Ti machine. Moreover, in the experiment, we run 1,000 inferences with different inputs for each model and report the average time.

Table 3. Performance comparison between TDC and existing SOTA compression methods. MD denotes matrix decomposition. DN-1/-2 denote DenseNet-121/-201. Most existing tensor decomposition works do not report results for ResNet-50, VGG-16 and DenseNet on ImageNet.

	Model	Compression Method	Top-1/Drop (%)	FLOPs↓
ResNet-18	Original [14]	No compr.	69.75/-0.00	N/A
	FPGM [16]	Pruning	68.41/-1.34	42%
	DSA [27]	Pruning	68.61/-1.14	40%
	SCOP [37]	Pruning	68.62/-1.13	45%
	TRP [40]	MD	65.51/-4.24	60%
	Stable [33]	CPD	69.06/-0.69	65%
	Opt. TT [42]	TTD	69.29/-0.46	60%
	Std. TKD [19]	TKD	66.65/-3.10	60%
	MUSCO [13]	TKD	69.28/-0.47	58%
	TDC	TKD	69.70/-0.05	63%
ResNet-50	Original [14]	No compr.	76.13/-0.00	N/A
	FPGM [16]	Pruning	75.59/-0.54	42%
	HRank [24]	Pruning	74.98/-1.15	44%
	TDC	TKD	77.46/+1.33	40%
	Stable [33]	CPD	74.66/-1.47	60%
	TDC	TKD	76.42/+0.29	60%
VGG-16	Original [14]	No compr.	71.59/-0.00	N/A
	CC [22]	MD	68.81/-2.78	50%
	TDC	TKD	71.62/+0.03	80%
DN-1	Original [14]	No compr.	74.43/-0.00	N/A
	TDC	TKD	76.33/+1.90	10%
DN-2	Original [14]	No compr.	76.88/-0.00	N/A
	TDC	TKD	76.92/+0.04	10%

We choose five CNN models including one VGG [35], two DenseNets [17], and two ResNets [14] on large-scale dataset ImageNet [8]. These three types of networks represent the architecture of modern CNNs. However, modern CNNs also grow in width. For example, there are networks which are not deep but are wide such as GoogleNet [36] and NasNet [45]. In those CNNs, there are multiple convolutions computed at the same time at each convolution stage. We leave those wide CNNs to our future work due to two factors: (1) It is difficult to develop a scheme that minimizes the latency for multiple concurrent convolutions. (2) It is challenging to determine the ranks for the concurrent convolutions in the same stage.

7.2 Evaluation on Model Accuracy

We evaluate TDC and compare it with other state-of-the-art CNN model compression approaches including pruning based approaches such as FPGM [16], DSA [27], SCOP [37], the matrix decomposition based approach such as TRP [40], and other tensor decomposition approaches. Table 3 shows that the test accuracy and FLOPs reduction with different methods. For the budget B, we choose 65%/60% for ResNet-18/-50, since state-of-the-art works have achieved these FLOPs reductions; similarly, we choose 50% for VGG-16 at the beginning, leading to a much higher accuracy than the state-of-the-art work CC [22], so we increase the budget to 65% and further to 80%; for DN-1/-2, due to no existing work to refer to FLOPs reduction ratio, we choose 10% as the budget. It is seen that our compression method provides the

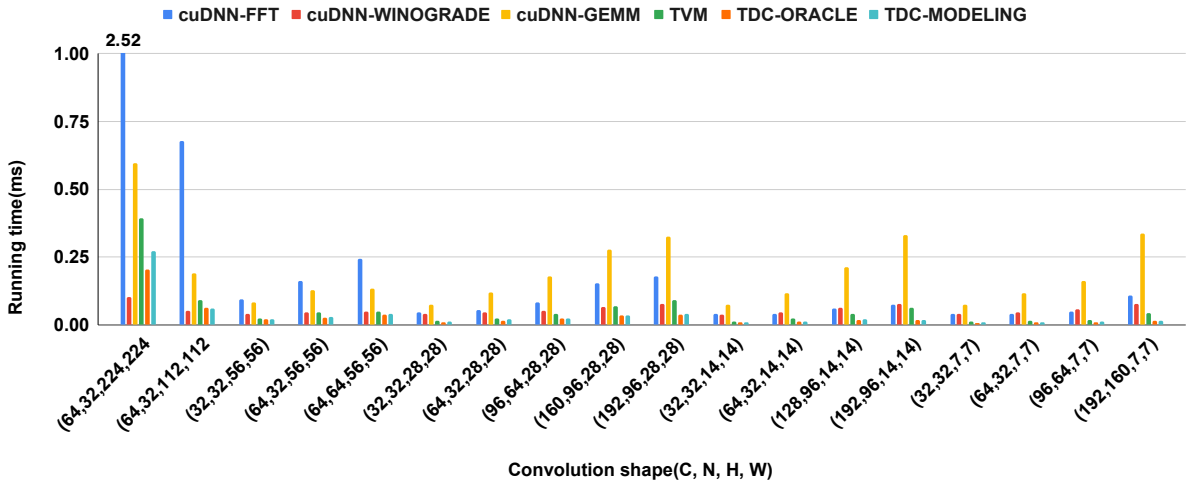

Figure 6. Performance comparison of our proposed kernel with TVM and cuDNN on all convolution shapes in the tested CNN models on A100.

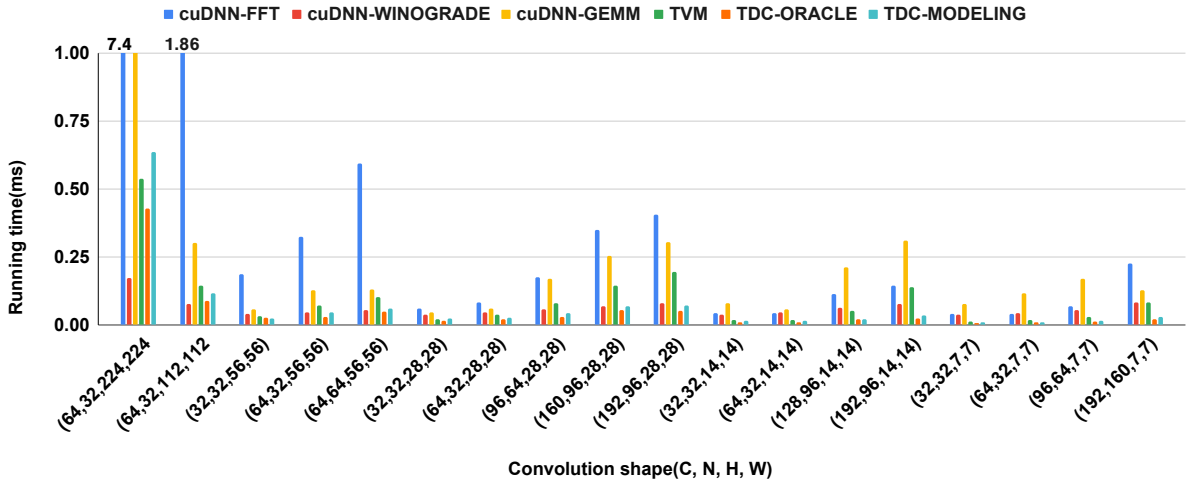

Figure 7. Performance comparison of our proposed kernel with TVM and cuDNN on all convolution shapes in the tested CNN models on 2080Ti.

highest test accuracy even with the highest FLOPs reduction for all tested DNN models. Specifically, our TKD-compressed models reduce the FLOPs by up to 63% with only 0.05% accuracy drop compared with the original non-compressed models on the tested CNNs. Note that except ResNet-18, the inference accuracy of our TKD-compressed models even has higher accuracy than that of the original non-compressed models, thanks to the proposed training algorithm.

We also evaluate the impact of different target budgets on accuracy. We choose 65%, 70%, 75%, and 80% as our target budgets for ResNet-18. The achieved accuracies are 69.70%, 67.86%, 66.59%, and 64.81%, respectively, and the achieved FLOPs reductions are 66%, 70%, 76%, and 80%, respectively. It illustrates that aggressive target budgets lead to significant accuracy drops. Thus, we recommend users choose the budget B based on existing work (e.g., 65% in ResNet-18 from [22]) or starting from 10% if no prior knowledge.

7.3 Evaluation on Convolution Kernel Performance

To demonstrate the effectiveness of our proposed convolution kernel, we compare the runtime of our kernel against the convolution kernel generated by TVM code generator and provided by cuDNN. We use the data type of float32 (single precision), the filter size of 3 (for core convolution), and the batch size of 1 (for inference).

Figure 6 and Figure 7 show the speedup of our scheme compared to TVM and cuDNN across all the core convolution shapes in the networks. On A100, our convolution with oracle/modeling approaches achieve a 5.38×/4.91× speedup over cuDNN-FFT, 3.12×/2.92× speedup over cuDNN-WINOGRAD, 8.95×/8.63× speedup over cuDNN-GEMM, and 1.81×/1.72× over TVM on average; on 2080Ti, our convolution with oracle/modeling approaches achieves 8.17×/6.21× speedup over cuDNN-FFT, 2.75×/2.12× speedup over cuDNN-WINOGRAD, 5.84×/5.38× speedup over cuDNN-GEMM, and 2.35×/1.81× over TVM on average.

From Figure 6 and Figure 7, we can observe that there are two shapes (64, 32, 224, 224) and (64, 32, 112, 112) from VGG16 where our solution is actually slower than or similar to TVM and cuDNN. The main reason is that our scheme splits the workload across the output channels inside a thread block, while TVM splits the workload across height/width.

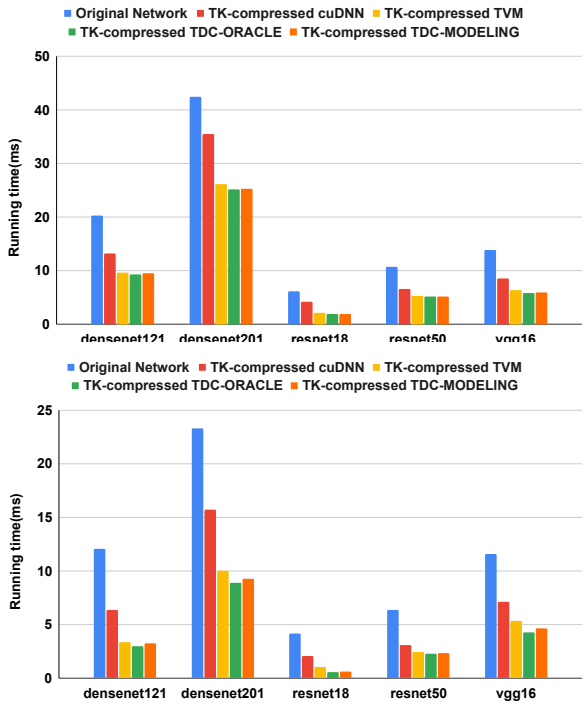

Figure 9. End-to-end evaluation of inference performance on 2080Ti.

Hence, for the convolution with large height and width, our scheme suffers a large kernel volume (referred to Equation (16)). However, we argue that this type of shape is becoming rare in modern CNN architecture designs. This is because the height (H) and width (W) of convolution layers decrease rapidly when the model goes deeper (mainly due to pooling layers), and hence the input channel (C) and output channel (N) increase. After an input goes into the model, after one or two layers, H/W decrease to a medium size, and the convolution shapes are likely to have very few large H/W but many medium H/C with large C/N. For example, in the CNNs released in recent years such as DenseNet, ResNet, and GoogleNet, the largest H/W is 56 for all the convolution layers (except the first convolution layer for the input). VGG is still considered one of the modern CNNs, however, it is older than the other tested models with lower accuracy.

7.4 Evaluation on End-to-End Inference Time

To fully demonstrate that our proposed co-design framework is effective not only for core convolutions but also for the entire network, we present the end-to-end performance (involving all layers) on the tested CNN models. To better understand the performance improvement, we implement the five models using C++/CUDA code to eliminate the overhead of Python deep learning framework. For example, in PyTorch, padding operation is done on the CPU side which requires additional memory copy between device and host.

Figures 8 and 9 show the inference time comparison among original models, TKD-compressed models with cuDNN, and TKD-compressed models with our optimized convolution operations. Specifically, the blue bars show the inference time of the original models on A100 and 2080Ti. For the implementation of original models, we call the cuDNN library for all the layers including the convolution layers. The red bars represent the inference time of TKD-compressed models which also call the cuDNN library for all the layers. The yellow bars represent the inference time of TKD-compressed models using TVM generated kernel for the core convolutions. The green bars represent the inference time of TKD-compressed models using our optimized kernel for the core convolutions. Note that for a fair comparison, we use cuDNN to implement other layers (including 1×1 convolution, pooling layers, etc.) and the *NCHW* data layout (i.e., TVM's best-fit data layout) for both our solution and the TVM solution.

On A100, our five TKD-compressed models (DenseNet121, DenseNet201, ResNet18, ResNet50, and VGG16) using our optimized core convolution kernel with oracle/modeling achieves speedup of 2.14×/2.11×, 1.7×/1.68×, 3.27×/3.24×, 2.07×/ 2.04×, and 2.37×/2.33×, respectively, compared with the original non-compressed models using cuDNN. The speedup over the TKD-compressed models using cuDNN is 1.41×/1.38×, 1.42×/1.40×, 2.21×/2.18×, 1.26×/1.25×, and 1.45×/1.43×, respectively. The speedup over the TKD-compressed models using TVM is 1.03×/1.01×, 1.04×/1.03×, 1.12×/1.11×, 1.02×/1.01×, and 1.09×/1.08×, respectively.

On 2080Ti, the compressed models using our optimized core convolution kernel achieves speedup of 4.15×/3.80×, 2.62×/2.55×, 7.3×/6.55×, 2.83×/2.75×, and 2.73×/2.45×, respectively, compared with the non-compressed models implemented with cuDNN. The speedup over the compressed models using cuDNN is 2.16×/1.97×, 1.81×/1.74×, 3.71×/3.25×, 1.38×/1.35×, and 1.68×/1.53×, respectively. The speedup over the compressed models using TVM is 1.13×/1.04×, 1.13×/1.08×, 1.91×/1.69×, 1.09×/1.06×, 1.25×/1.14×, respectively.

8 Conclusion and Future Work

In this paper, we propose an efficient end-to-end framework to generate highly accurate and compact CNN models via Tucker decomposition and optimized C++/CUDA code for GPU inference. We incorporate hardware constraints into the model generation and develop a new scheme for Tucker-format convolutions with high-performance GPU kernels. Evaluation on modern CNNs with A100 demonstrates that our compressed models with optimized code achieve up to 2.37× speedup over cuDNN and 1.10× speedup over TVM.

In the future, we plan to extend our work to cover wide CNNs such as GoogleNet and NasNet by developing a scheme that can determine the ranks for multiple concurrent convolutions and minimize the latency.

Acknowledgement

The research was supported by the NSF awards 2232120, 2034169, 2042084, 1937403, 1955909, 2018016, and 2009007. We thank Jacques Pienaar for helping us improve the paper quality.

References

[1] Martín Abadi. Tensorflow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pages 1–1, 2016.

[2] Stephen Boyd, Neal Parikh, and Eric Chu. *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc, 2011.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation*, pages 578–594, 2018.

[4] Mathew J Cherukara, Tao Zhou, Youssef Nashed, Pablo Enfedaque, Alex Hexemer, Ross J Harder, and Martin V Holt. Ai-enabled high-resolution scanning coherent diffraction imaging. *Applied Physics Letters*, 117(4):044103, 2020.

[5] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[6] Michael Collins and Nigel Duffy. Convolution kernels for natural language. *Advances in neural information processing systems*, 14, 2001.

[7] cuDNN v2: Higher Performance for Deep Learning on GPUs. https://developer.nvidia.com/blog/cudnn-v2-higher-performance-deep-learning-gpus/.

[8] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[9] Peiyan Dong, Siyue Wang, Wei Niu, Chengming Zhang, Sheng Lin, Zhengang Li, Yifan Gong, Bin Ren, Xue Lin, and Dingwen Tao. Rtmobile: Beyond real-time mobile acceleration of rnns for speech recognition. In *2020 57th ACM/IEEE Design Automation Conference*, pages 1–6. IEEE, 2020.

[10] Boyuan Feng, Yuke Wang, Tong Geng, Ang Li, and Yufei Ding. Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2021.

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

[12] Groq. The Challenge of Batch Size 1: Groq Adds Responsiveness to Inference Performance. https://groq.com/wp-content/uploads/2020/04/GROQP002_groq_whitepaper_V1-DB-1.pdf.

[13] Julia Gusak, Maksym Kholiavchenko, Evgeny Ponomarev, Larisa Markeeva, Philip Blagoveschensky, Andrzej Cichocki, and Ivan Oseledets. Automated multi-stage compression of neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[15] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 639–648, 2020.

[16] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.

[17] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[18] Sian Jin, Sheng Di, Xin Liang, Jiannan Tian, Dingwen Tao, and Franck Cappello. Deepsz: A novel framework to compress deep neural networks by using error-bounded lossy compression. In *Proceedings of the 28th international symposium on high-performance parallel and distributed computing*, pages 159–170, 2019.

[19] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *International Conference on Learning Representations*, 2016.

[20] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 649–660. IEEE, 2018.

[21] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *International Conference on Learning Representations*, 2015.

[22] Yuchao Li, Shaohui Lin, Jianzhuang Liu, Qixiang Ye, Mengdi Wang, Fei Chao, Fan Yang, Jincheng Ma, Qi Tian, and Rongrong Ji. Towards compact cnns via collaborative compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6438–6447, 2021.

[23] Chia-Chun Liang and Che-Rung Lee. Automatic selection of tensor decomposition for compressing convolutional neural networks a case study on vgg-type networks. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 770–778. IEEE, 2021.

[24] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1529–1538, 2020.

[25] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5117–5124, 2020.

[26] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. Cosmoflow: Using deep learning to learn the universe at scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829. IEEE, 2018.

[27] Xuefei Ning, Tianchen Zhao, Wenshuo Li, Peng Lei, Yu Wang, and Huazhong Yang. Dsa: More efficient budgeted pruning via differentiable sparsity allocation. In *European Conference on Computer Vision*, 2020.

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

[29] Nvidia. NVIDIA Deep Learning cuDNN Documentation. https://docs.nvidia.com/deeplearning/cudnn/api/index.html.

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

[31] Jongsoo Park, Sheng Li, Wei Wen, Ping Tak Peter Tang, Hai Li, Yiran Chen, and Pradeep Dubey. Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*, 2016.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance

deep learning library. *Advances in neural information processing systems*, 32, 2019.

[33] Anh-Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavskỳ, Valeriy Glukhov, Ivan Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. In *European Conference on Computer Vision*, pages 522–539. Springer, 2020.

[34] Daniel Povey, Gaofeng Cheng, Yiming Wang, Ke Li, Hainan Xu, Mahsa Yarmohammadi, and Sanjeev Khudanpur. Semi-orthogonal low-rank matrix factorization for deep neural networks. In *Interspeech*, pages 3743–3747, 2018.

[35] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[37] Yehui Tang, Yunhe Wang, Yixing Xu, Dacheng Tao, Chunjing XU, Chao Xu, and Chang Xu. Scop: Scientific control for reliable neural network pruning. In *Advances in Neural Information Processing Systems*, volume 33, pages 10936–10947, 2020.

[38] Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

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

[40] Yuhui Xu, Yuxi Li, Shuai Zhang, Wei Wen, Botao Wang, Yingyong Qi, Yiran Chen, Weiyao Lin, and Hongkai Xiong. Trp: Trained rank pruning for efficient deep neural networks. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 977–983, 2020.

[41] Miao Yin, Siyu Liao, Xiao-Yang Liu, Xiaodong Wang, and Bo Yuan. Towards extremely compact rnns for video recognition with fully decomposed hierarchical tucker structure. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12085–12094, 2021.

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

[43] Chengming Zhang, Geng Yuan, Wei Niu, Jiannan Tian, Sian Jin, Donglin Zhuang, Zhe Jiang, Yanzhi Wang, Bin Ren, Shuaiwen Leon Song, and Dingwen Tao. Clicktrain: Efficient and accurate end-to-end deep learning training via fine-grained architecture-preserving pruning. In *Proceedings of the ACM International Conference on Supercomputing*, pages 266–278, 2021.

[44] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[45] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.

A Artifact Description/Evaluation

A.1 Overview

The artifacts contain all code, scripts, and datasets that are necessary for reproducing our results. The artifacts can be obtained from https://github.com/black-cat-sheriff/TDC-PPOPP and https://doi.org/10.5281/zenodo.7439434.

A.2 Package Requirements

- cmake(>=3.10)
- cuDNN(>=8)
- CUDA >=11 (A100)
- CUDA >=10 (2080Ti)
- python3(>=3.7)
- tensorly(0.7.4, pip install tensorly==0.7.0)
- timm(0.5.4, pip install timm==0.5.4)
- torch(>=1.4)
- torchvision
- numpy

Note that we require the version of timm to be 0.5.4.

A.3 SETUP

```
export CUDA_PREFIX='/usr/local/cuda'
export CUDA_INCLUDE=$CUDA_PREFIX/include
export CUDA_LIB64=$CUDA_PREFIX/lib64
export LD_LIBRARY_PATH=$CUDA_LIB64:$LD_LIBRARY_PATH
export PATH=$CUDA_PREFIX/bin:$PATH
```

Please change `/usr/local/cuda` to the path to your CUDA library. We assume that cuDNN is installed in `CUDA_PREFIX`, which means that cuDNN header files are in `CUDA_INCLUDE` and cuDNN library files are in `CUDA_LIB64`. Also, please note that some cuDNN versions only have `lib` folder rather than `lib64` folder, please change `CUDA_LIB64` accordingly.

A.4 TDC TRAINED MODEL EVALUATION

A.4.1 Tucker-format model accuracy evaluation Table 3.

We provided 6K images to demonstrate our models, but you are encouraged to obtain more images from ImageNet.

```
cd inference
python main.py --model tkc_resnet18  \
    --data-path test_images/
python main.py --model tkc_resnet50 \
    --data-path test_images/
python main.py --model tkc_densenet121 \
    --data-path test_images/
python main.py --model tkc_densenet201 \
    --data-path test_images/
python main.py --model tkc_vgg16 \
    --data-path test_images/
```

Note that `test_images` is the path to the folder where the images used for the demo are saved (i.e., 6K images in total).

You will observe the following result.

```
Evaluating...
Test:  [1/2]  eta: 0:00:03  loss: 0.7396 (0.7396)  acc1: 81.6406 (81.6406)  acc5: 95.3125 (95.3125)  time: 1.7198  data: 1.1552  max mem: 1043
Test:  [2/2]  eta: 0:00:00  loss: 0.7396 (0.9786)  acc1: 69.1489 (78.2857)  acc5: 89.3617 (93.7143)  time: 0.8762  data: 0.5776  max mem: 1043
Test: Total time: 0:00:01 (0.8888 s / it)
Evaluation Result: Acc@1 78.286%, Acc@5 93.714%, loss 0.979
```

A.4.2 Performance Evaluation of TDC-generated Core Convolution Layers

Please go to the main folder and run:

```
python3 run_2080Ti.py & python3 run_A100.py
```

It will generate three tables.

A.4.2.1 Comparison among TDC oracle kernel, cuDNN, and TVM. You will get the first table like below, including convolution shapes, convolution schemes, runtimes, and TDC speedups. Please refer to Figure 7.

```
N,C,H,W,FFT(ms),WinoGradNonFuse(ms),GEMM(ms),TVM(ms),TDC Oracle(ms),Speedup VS FFT,Speedup VS WINO, Speedup VS GEMM, Speedup VS TVM
32,64,224,224,7.38394,2.74998,1.04614,0.58608,0.50448,14.6367,5.45113,2.07371,1.16175
32,64,112,112,1.85222,0.711392,0.298976,0.275136,0.133216,13.9039,5.34014,2.2443,2.06534
32,32,56,56,0.188352,0.160192,0.0616,0.041312,0.030272,6.22199,5.29175,2.03488,1.36469
32,64,56,56,0.32432,0.22832,0.137472,0.083584,0.044064,7.3602,5.18155,3.11983,1.89688
64,64,56,56,0.592416,0.270688,0.139712,0.283168,0.057952,10.2225,4.6709,2.41082,4.88625
32,32,28,28,0.059392,0.060032,0.051904,0.033376,0.017792,3.33813,3.3741,2.91727,1.8759
32,64,28,28,0.086592,0.081504,0.06624,0.050496,0.021568,4.01484,3.77893,3.07122,2.34125
64,96,28,28,0.17472,0.11424,0.18192,0.088192,0.031456,5.55443,3.63174,5.78332,2.80366
96,160,28,28,0.352864,0.232864,0.275712,0.361696,0.052288,6.74047,4.43819,5.27295,6.91738
96,192,28,28,0.41344,0.275616,0.335488,0.296352,0.061728,6.69777,4.46501,5.43494,4.80093
32,32,14,14,0.042752,0.039104,0.081376,0.019104,0.013152,3.25061,2.97324,6.18735,1.45255
32,64,14,14,0.044352,0.047744,0.064864,0.0256,0.01504,2.94894,3.17447,4.31277,1.70213
96,128,14,14,0.111808,0.086432,0.224032,0.076992,0.02336,4.7863,3.7,9.59041,3.29589
96,192,14,14,0.149984,0.114144,0.334656,0.108928,0.027424,5.46908,4.16219,12.203,3.972
32,32,7,7,0.042272,0.035616,0.07968,0.019456,0.009824,4.30293,3.62541,8.11075,1.98046
32,64,7,7,0.041504,0.039648,0.123648,0.024896,0.011296,3.67422,3.50991,10.9462,2.20397
64,96,7,7,0.065792,0.041536,0.165488,0.049792,0.012736,5.16583,3.26131,12.9874,3.90955
160,192,7,7,0.223296,0.07952,0.13072,0.150464,0.021408,10.4305,3.7145,6.10013,7.0284
```

A.4.2.2 Comparison among TDC modeling kernel, cuDNN, and TVM. You will get the second table like below, including convolution shapes, convolution schemes, runtimes, and TDC speedups.

```
N,C,H,W,FFT(ms),WinoGradNonFuse(ms),GEMM(ms),TVM(ms),TDC Modeling(ms),Speedup VS FFT,Speedup VS WINO, Speedup VS GEMM, Speedup VS TVM
32,64,224,224,7.41027,2.74349,1.04342,0.584608,0.58752,14.6089,5.46567,2.05593,1.15189
32,64,112,112,1.85763,0.71456,0.301088,0.279456,0.13632,13.627,5.24178,2.20869,2.05
32,32,56,56,0.18704,0.160288,0.06016,0.04256,0.032928,5.68027,4.86783,1.82782,1.29252
32,64,56,56,0.319392,0.223344,0.13552,0.082496,0.048192,6.62749,4.75896,2.81208,1.71182
64,64,56,56,0.59616,0.2688,0.136096,0.286336,0.063488,9.39012,4.23387,2.14365,4.51008
32,32,28,28,0.057984,0.060192,0.05248,0.033632,0.021472,2.70045,2.80328,2.44411,1.56632
32,64,28,28,0.0872,0.082848,0.066304,0.051168,0.026272,3.31912,3.15347,2.52175,1.94762
64,96,28,28,0.17744,0.114464,0.18,0.085504,0.03376,5.25592,3.39052,5.33175,2.5327
96,160,28,28,0.356096,0.231584,0.277824,0.364256,0.055168,6.45476,4.1978,5.03596,6.60267
96,192,28,28,0.406688,0.270496,0.327296,0.287424,0.063136,6.44146,4.27815,5.18398,4.55246
32,32,14,14,0.04288,0.04016,0.082656,0.019104,0.017024,2.5188,2.35902,4.85526,1.12218
32,64,14,14,0.046784,0.047008,0.065024,0.026016,0.018368,2.54704,2.55923,3.54007,1.41638
96,128,14,14,0.10864,0.085856,0.229504,0.0768,0.022912,4.74162,3.74721,10.0168,3.35196
96,192,14,14,0.15072,0.112192,0.328,0.105952,0.027712,5.4388,4.0485,11.836,3.82333
32,32,7,7,0.041152,0.033632,0.079872,0.019168,0.016064,2.56175,2.09363,4.97211,1.19323
32,64,7,7,0.042112,0.039008,0.124288,0.024032,0.016416,2.5653,2.37622,7.57115,1.46394
64,96,7,7,0.065472,0.041856,0.165376,0.05824,0.015648,4.18485,2.67485,10.5685,3.21063
160,192,7,7,0.226208,0.079776,0.132,0.14992,0.021056,10.7432,3.78875,6.269,7.12066
```

A.4.2.3 End-to-end performance comparison among pure cuDNN on original models, pure cuDNN on TK compressed models, and TK compressed models. You will get the third table like below, each model has two rows - the first one is header and the second one is runtime. Please refer to Figure 9.

```
densenet121(Run time),densenet121-tk(Run time),densenet121-tk-tvm(Run time),densenet121-tk-modeling(Run time),densenet121-tk-oracle(Run time)
11.987629(ms),6.285938(ms),3.480946(ms),3.197938(ms),2.940913999999994(ms)
densenet201(Run time),densenet201-tk(Run time),densenet201-tk-tvm(Run time),densenet201-tk-modeling(Run time),densenet201-tk-oracle(Run time)
23.042181(ms),15.507432(ms),9.970216(ms),8.879272(ms),8.911656(ms)
resnet18(Run time),resnet18-tk(Run time),resnet18-tk-tvm(Run time),resnet18-tk-modeling(Run time),resnet18-tk-oracle(Run time)
4.125181(ms),2.046399(ms),1.249023(ms),0.4696319000000013(ms),0.4533749999999986(ms)
resnet50(Run time),resnet50-tk(Run time),resnet50-tk-tvm(Run time),resnet50-tk-modeling(Run time),resnet50-tk-oracle(Run time)
6.328288(ms),3.086317(ms),2.521677000000004(ms),2.285837(ms),2.245805000000004(ms)
vgg16(Run time),vgg16-tk(Run time),vgg16-tk-tvm(Run time),vgg16-tk-modeling(Run time),vgg16-tk-oracle(Run time)
11.498522(ms),7.088114(ms),6.08149(ms),4.36149(ms),4.324946(ms)
```