TKD: An Efficient Deep Learning Compiler with Cross-Device Knowledge Distillation

1st Yiming Ma
School of Integrated Circuits
Southeast University
Nanjing, China
mayiming@seu.edu.cn

2nd Chaoyao Shen
School of Integrated Circuits
Southeast University
Nanjing, China
shen_chaoyao@seu.edu.cn

3rd Linfeng Jiang
School of Integrated Circuits
Southeast University
Nanjing, China
linfengjiang@seu.edu.cn

4th Tao Xu
School of Integrated Circuits
Southeast University
Nanjing, China
xu_tao@seu.edu.cn

5th Meng Zhang
School of Integrated Circuits
Southeast University
Nanjing, China
zmeng@seu.edu.cn

***Abstract*—Generating high-performance tensor programs on resource-constrained devices is challenging for current Deep Learning (DL) compilers that use learning-based cost models to predict the performance of tensor programs. Due to the inability of cost models to leverage cross-device information, it is extremely time-consuming to collect data and train a new cost model. To address this problem, this paper proposes TKD, a novel DL compiler that can be efficiently adapted to devices that are resource-constrained. TKD reduces the time budget by over 11x through an adaptive tensor program filter that eliminates redundant and unimportant measurements of tensor programs. Furthermore, by refining the cost model architecture with a multi-head attention module and distilling transferable knowledge from source devices, TKD outperforms state-of-the-art methods in prediction accuracy, compilation time, and compilation quality. We conducted experiments on the edge GPU, NVIDIA Jetson TX2, and the results show that compared to TenSet and TLP, TKD reduces compilation time by 1.58x and 1.16x, while achieving 1.40x and 1.27x speedups of the tensor programs, respectively.**

***Index Terms*—Deep Neural Networks; Tensor Compiler; Cost Model; Knowledge Distillation; Active Learning**

I. Introduction

As the complexity and scale of deep neural networks (DNNs) increase, these models demonstrate remarkable performance in various fields such as computer vision [1]–[3] and natural language processing [4]–[6]. The rapid advancement of DNNs has driven the emergence of specialized artificial intelligence hardware (e.g., Google TPU [7], Cambricon NPU [8]). With the efforts of researchers, DNNs can be efficiently deployed on hardware such as CPUs [9], MCUs [10], and Edge GPUs [11]. Optimizing the inference time of DNN models on hardware is crucial for real-time tasks that require high processing speed and low memory/computation resource usage. However, traditional DL frameworks find it difficult to achieve these goals because they rely on hand-crafted optimizations or vendor-provided kernel libraries such as CuDNN [12], which lack cross-device optimization capabilities. DL compilers (e.g., Halide [13], TVM [14]) can automatically optimize tensor programs and generate code that can be extended to multiple hardware platforms, thus becoming increasingly popular among developers.

DL compilers usually rely on heuristic exploration modules and learning-based cost models to implement automatic tensor optimization. Compared to traditional cost models trained with small datasets collected online [14], [15], recent works have shown that cost models that are pre-trained with offline datasets [16]–[18] outperform them in terms of prediction accuracy and search efficiency. Pre-trained cost models typically adopt a data-driven paradigm: the cost models are pre-trained on large-scale tensor program tuning records and capture the correlation between optimization operations and real runtime behaviors on specific hardware. A representative pre-trained cost model is TenSet-MLP [16], which collects 52 million records from 6 hardware platforms and reduces the compilation time by up to 10× while maintaining the same compile quality.

However, the data-driven paradigm brings several problems while aiming for higher performance. The first problem is that data-driven cost models depend heavily on the quantity of measurement records. Some deep learning-based cost models such as LSTM [19] and Transformer [4] need large-scale datasets for training, millions of data are required to achieve ideal performance. The second problem is that collecting measurement records is extremely time-consuming and resource-intensive. Since the structures vary across different hardware platforms (e.g., different GPUs, CPUs, and FPGAs), the records collected from one device are not applicable to others. Developers have to remeasure the tensor programs on new hardware platforms. To ensure the accuracy of measurements, monopolization and repetition are necessary. This process involves multiple steps,

This work is supported by Key R&D Program of Guangdong Province (No. 2021B1101270006), the Fundamental Research Funds for the Central Universities (Project No. 2242024K30017), Shandong Provincial Natural Science Foundation (No. ZR2023QF056) and China Postdoctoral Science Foundation (No. 2024M751267). Yiming Ma and Chaoyao Shen contribute equally to this work. Chaoyao Shen and Meng Zhang are the corresponding authors.

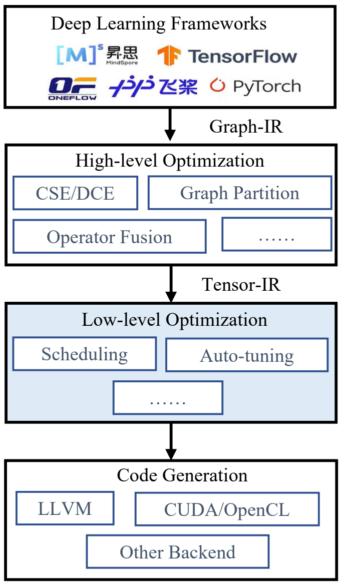

Fig. 1. The workflow of deep learning compilers

including compilation and repetitive execution. When it comes to devices with restricted memory and computational resources, it can take up to several months to complete data collection [17]. The third problem is the imbalance and redundancy of data. We analyze the data distribution of TenSet and find that high-performance tensor programs account only 20% on CPUs and 10% on GPUs. Imbalanced training data will reduce the accuracy of the cost model, making it difficult to identify high-performance tensor programs.

These problems obstruct the implementation of DL compilers on different devices and reduce their universality. In this paper, we propose a novel cross-device DL compiler framework to efficiently compile tensor programs on target devices with extremely low pre-training overhead. The main contributions of this paper are as follows:

- We analyze the data distribution of TenSet and propose an adaptive tensor program filter to eliminate redundant and low-performance generated tensor programs and reduce the time budget to pre-train a cost model by 11×.
- We refine the cost model architecture with a multi-head attention module and propose a cross-device knowledge distillation method to leverage transferable knowledge from known devices. The experimental results demonstrate that our cost model achieves improved accuracy compared to the state-of-the-art work.
- We propose an efficient DL compiler called TKD by integrating the adaptive tensor program filter and cross-device knowledge distillation method into Ansor. We conduct end-to-end experiments on NVIDIA Jetson TX2, and TKD outperforms TenSet-MLP and TLP on both compilation time and compile quality.
- We release the dataset collected from NVIDIA Jetson TX2 on https://github.com/yimingma/TKD.

II. Background

A. Deep Learning Compiler

With the emergence of DL frameworks (e.g., PyTorch [20], TensorFlow [21], MXNet [22]) and hardware (e.g., AMD CPUs, Intel CPUs, NVIDIA GPUs), engineers can choose various combinations to develop and deploy DL models according to their application scenarios. However, it takes considerable human engineering effort to transform DL models into hardware-compatible code. DL compilers serve as a bridge to convert DL models from development phase to deployment phase. Fig. 1 shows the workflow of DL compilers. DL compilers usually consist of three stages: 1) High-level optimization that converts DL models to graph intermediate representations (IR) and performs graph-level optimizations such as graph partitioning and operator fusion. 2) Low-level optimization that converts graph IR to tensor IR and optimizes the computation schedule using unified rules or auto-tuning. 3) Code generation, where the optimized tensor IR is translated to hardware code that can run on target devices.

TVM is a representative DL compiler that provides an end-to-end optimization stack. It translates tensor programs to a tensor expression language called Relay IR and uses schedule primitives to support automatic code generation. TVM supports loop transformation, cache location, and parallel cooperation via schedule primitives. AutoTVM [23] and Ansor [15] are two auto-tuning frameworks based on TVM; they perform auto-tuning to generate optimal codes for tensor programs. These frameworks utilize exploration algorithms to search for possible combinations of schedule primitives and predict the real performance using a learning-based cost model.

B. Active Learning Methods

TenSet released a large-scale tensor program tuning dataset as a standard benchmark and pre-training dataset for a learning-based cost model, it has collected more than 51 million tensor program measurement records on 6 hardware platforms containing CPUs and GPUs.

However, data imbalance and redundant low-performance samples in the TenSet dataset make it challenging for cost models to identify high-performance programs precisely. To address this problem, many works adopt active learning methods [24] to filter the redundant and low-performance tensor programs to adjust the data distribution. BALTO [25] uses a diversity-based active learning method to remove redundant data. ALT [26] leverages an active learning method to sample more informative data to be labeled. To improve the quality of training data and obtain a high-performance cost model, we will adopt an active learning approach and adaptively adjust the data distribution based on the training results.

C. Cross-device Adaption

Although TenSet has collected data from 6 hardware platforms, we find that there is still a lack of data collected from edge devices. Due to the limited computational and storage resources of edge devices, data collection requires a significant amount of time and computational resources, which impedes

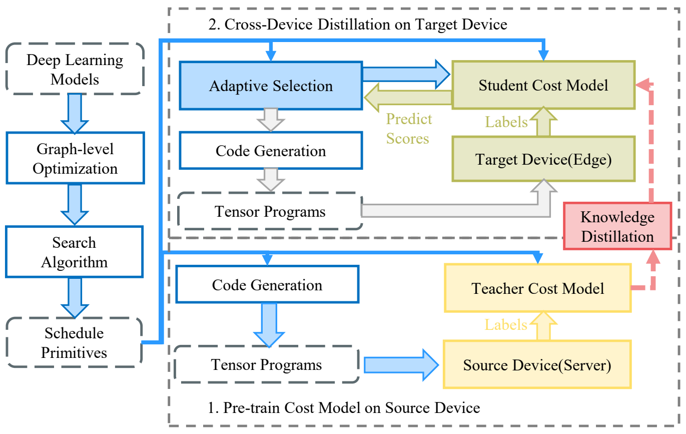

Fig. 2. The overview workflow of proposed cross-device compiler.

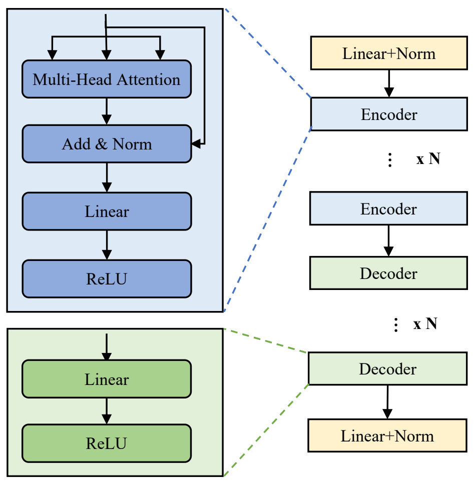

Fig. 3. Model architecture description of proposed cost model.

the extension of the learning-based cost model to the edge devices.

Therefore, a cost model that can reach ideal performance based on a small amount of data is necessary. In order to improve the performance of the cost model, many works adopt transfer learning methods. Moses [17] optimize the time-consuming process of training cost model on new hardware platform by domain adaptation. TLP [18] extends TenSet on Intel i7 CPU and employs a multi-task learning method to address the cross-hardware unavailability of a pre-trained cost model.

III. Methodology

A. Overview

Fig. 2 illustrates the workflow of TKD. TKD consists of two parts: high-level optimization and low-level optimization. For high-level optimization, the deep learning model undergoes graph-level optimizations (e.g., operator fusion, constant folding, and data layout transformations) to generate multiple sub-graphs. For low-level optimization, the exploration module will automatically search for possible schedule primitives of computation sub-graphs and the cost model will predict the performance score of schedule primitives. We use Ansor to realize high-level optimization and exploration module and propose a two-phase efficient cross-device framework to implement other low-level optimizations. Our framework dramatically improves the efficiency of compiling DL models on embedded devices with limited computing resources.

In the first phase, we train a teacher cost model on the source device. The auto-tuner combines the computational sub-graphs with schedule primitive to generate optimized tensor programs. We extract features directly from schedule primitives referred to the method of TLP. The latency of optimized tensor programs is measured on the source device, serving as the training label.

In the second phase, we train a student cost model on the target device. The limited hardware resources make it time-consuming to measure all optimized tensor programs. So TKD introduces an adaptive tensor program filter, using an active learning method to sample high-performance and important tensor programs to measure. The sampled tensor programs will be measured on the target hardware, providing labels for training the student cost model. Additionally, to improve cost model accuracy and sample quality, cross-device knowledge distillation is employed to transfer features learned from the source device to the student cost model.

Finally, when deploying a deep learning model on the target device, the auto-tuner receives the computational sub-graphs generated by the graph-level optimizations and utilizes the pre-trained student cost model and search algorithm to efficiently generate optimized tensor programs with minimized latency on the target hardware.

B. Deep Learning-based Cost Model

During the auto-tuning phase in DL compilers, the auto-tuner generates a large search space of tensor program configurations based on the search algorithm and searches for the optimal tensor program for the target hardware. The auto-tuner leverages a cost model during the search process to select the tensor program with the best performance. The effectiveness of the cost model is crucial to determine whether the best tensor program can be selected. During auto-tuning, computational sub-graphs are abstracted into high-level expressions first. High-level expressions lack the specific details in low-level implementations, such as loop order, memory scope, and threading unspecified. For a given high-level expression, many functionally equivalent low-level codes can be generated. The possible scheduling space from the high-level expression exp to the low-level code is denoted as S_{exp}. For any $sch \in S_{exp}$, the corresponding low-level code is represented as $x = g(exp, sch)$, where g is the code generator of the DL compiler that produces low-level code from the high-level representation exp and its associated schedule primitive sch. Our goal is to minimize $f(x)$, which is the real runtime cost of the hardware. It is important to note that $f(x)$ does not have an analytical expression. We use a cost model $\widetilde{f}(x)$ to approximate the output of $f(x)$. Therefore, the optimization objective can be formalized as:

$$\arg\min_{sch \in S_{exp}} \widetilde{f}(g(exp, sch)) \tag{1}$$

To accurately select the optimal configurations (schedule primitives), we designed an offline cost model based on an encoder-decoder architecture, achieving precise predictions. The training dataset consists of tensor programs from the TenSet dataset and a small number of tensor programs collected from the Jetson TX2. The input features extracted from different schedule primitives will be padded or clipped to the same size. The architecture of our cost model is shown in Fig. 3, the input features $x_0, x_1, ..., x_n$ are mapped into embedding $emb_0, emb_1, ..., emb_n$ through a linear layer, and then pass through self-attention layers to capture the relationships between features and focus on the important ones. We use MLP as a decoder to establish the mapping relationship between the embeddings and the performance scores. For a tensor program, the physical latency measured on the device is normalized to 0-1 according to the minimum latency in the same scheduling space S:

$$score = \frac{\min\limits_{j \in S} latency_j}{latency} \tag{2}$$

Due to the huge gap in hardware architecture and configuration, the actual latency of one tensor program varies on different devices(e.g. the latency of a Conv2d operation on the order of 10^{-4} second on server GPU, but is on the order of 10^{-3} second on embedded GPU). The cost model will fit the absolute value of latency and learn the bias of a specific device if we use mean squared error(MSE) as a loss function. To increase the generality of our cost model, we use the LambdaRank [25] as a loss function. LambdaRank loss can evaluate a model's capacity to correctly predict the rank of the tensor programs by their performance score and places more emphasis on high-performance programs, making the cost model learn which tensor programs can run fastest on the device.

C. Adaptive Tensor Program Filter

The performance of the cost model heavily depends on the quality of its training data. However, our analysis of the distribution of the most popular tensor program dataset, TenSet, reveals a significant data imbalance. High-performance tensor programs account for less than 10% and 20% on average for GPUs and CPUs, respectively, leading to a large redundancy of low-performance programs. This not only increases the time-consuming measurement cost but also negatively impacts the performance of the cost model.

To address the data imbalance and redundancy and reduce unnecessary measurements on the target device, we designed an adaptive tensor filter based on diversity-based and representative-based active learning(AL) methods to select the tensor programs that are most beneficial for improving latency prediction accuracy. Diversity-based sampling selects the sample that is farthest from the currently selected set by evaluating the distances between samples, ensuring that the selected set covers the entire sample space as thoroughly as possible. We use the predicted scores of the cost model $\widetilde{f}$ to represent the distances between samples. If x is an unlabeled tensor program and D_l is the labeled programs set, we calculate the distance of x and D_l as follows:

Algorithm 1 Hybrid sampling algorithm

```
Input: $D_T$, $\widetilde{f}$, $\alpha$, $\beta$
Output: $D_l$
 1: $D_l \leftarrow \varnothing$
 2: while $D_T \neq \varnothing$ do
 3:   $D_b \leftarrow RandomSelect(D_T)$
 4:   $D_T \leftarrow D_T - D_b$
 5:   $L = length(D_b)$
 6:   $C = 0$
 7:   while $C < \alpha L$ do
 8:     if $C < \alpha\beta L$ then
 9:       $s = \arg\max_{x \in D_b} KDE(f(x))$
10:     else
11:       $dist(x) = \min_{x \in D_T, y \in D_l} \|f(x) - f(y)\|$
12:       $s = \arg\max_{x \in D_T} f(x) \cdot dist(x)$
13:     end if
14:     $C = C + 1$
15:     $D_b = D_b - s$
16:     $D_l \leftarrow D_l \cup s$
17:   end while
18: end while
19: return $D_l$
```

$$Dist(x, D_l) = \min_{y \in D_l} \left| \widetilde{f}(x) - \widetilde{f}(y) \right| \tag{3}$$

Each time, we select one sample with the maximum $Dist(x, D_l)$ from the unlabeled set and add it to the labeled set, forming a diverse sample set.

To address the data shift, we combined representative-based sampling. We use *Kernel Density Estimation(KDE)* to fit the distribution density of sample performance, and take the distribution density as the representative. The hybrid sampling method is detailed in Algorithm 1, each time a batch of unlabeled programs D_b is randomly selected from the entire dataset D_T, the batch size is N_b. We conduct representative-based sampling to select $\alpha\beta N_b$ samples in line 8-9 and conduct diversity-based sampling to select $\alpha(1-\beta)N_b$ samples in line 11-12, where α is the sample ratio and β is the coefficient to balance diversity and representativeness.

D. Cross-device Knowledge Distillation

With ample computational and storage resources on the source device, we can train a large-scale cost model using a massive labeled dataset. However, when operating on a target device with limited resources, collecting large amounts of data becomes time-consuming. Although we use an adaptive tensor program filter to enhance the sample quality, the accuracy of the cost model remains suboptimal due to the small size of the training data.

To address this issue, we employ a cross-device knowledge distillation approach. As shown in Fig. 4, the teacher cost model was pre-trained on the source device, and its parameters are

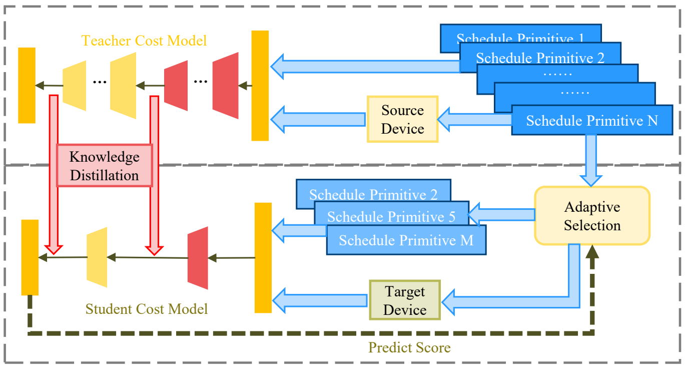

Fig. 4. Cross-device knowledge distillation between source device and target device.

frozen. For the student cost model, we reduce the number of encoders and decoders and distill the knowledge of high importance from the teacher cost model to avoid overfitting and improve prediction accuracy. The distillation is implemented by transferring the attention distribution from the teacher cost model to the student cost model. We use attention score to denote the attention distribution, and assume that $A \in \mathbb{R}^{L \times D}$ is the output feature map of the last encoder/decoder layer, attention score $AS \in \mathbb{R}^{1 \times D}$ is calculated as follows:

$$AS = \sum_{i=1}^{n} \frac{A_{ij}^2}{\|A_{ij}^2\|_2} \tag{4}$$

So the attention distillation loss can be described as follows:

$$AD_{Loss} = \frac{\beta}{2} \sum_{i \in Enc, Dec} \|AS_{stu}^i - AS_{tch}^i\|_2 \tag{5}$$

Where β is the scale factor to adjust the weight of attention loss, AS_{stu}^{Enc} is the attention score of the output feature map of the last encoder of the student cost model and AS_{stu}^{Dec} is that of the last decoder of student cost model, the meaning of the upper and lower labels is the same for the teacher model.

The general loss function is the sum of Lambda Rank loss and attention distillation loss, the student cost model can extract embeddings of the features of high importance by mimicking the output of the teacher cost model and formulate the correct mapping relationship between embedding and performance score on the target device by minimizing rank loss function.

IV. Experimental Results

In this section, we evaluate the effectiveness and efficiency of TKD by two types of experiments: dataset-based and on-device experiments. First, we evaluate the performance of the proposed cost model and then perform adaptive sampling and cross-device knowledge distillation on the TenSet dataset to evaluate the effectiveness of TKD. Then, we adapt the cost model to the Jetson TX2 using TKD, and integrate it into Ansor to evaluate the end-to-end compiling efficiency and quality.

A. Cost Model Performance Evaluation

Before the evaluation, we first split the dataset into three parts: test set, training set and validation set. We hold out a test set that consists of five deep learning networks, ResNet-50, MobileNet-V2, ResNext-50, BERT-tiny, and BERT-base. The test set contains most of the common operation types in computer vision and natural language processing, which is sufficient to demonstrate the generality of our cost model. The remaining dataset is split into 9:1 for the training set and validation set. For each hardware platform in TenSet, we will first train a large-scale cost model using all labeled data to evaluate the effectiveness of the proposed model architecture. We evaluate the performance of the cost model using the top-k score according to the metric of TenSet. The top-k score is calculated as follows:

$$top_k = \frac{\sum_{m,s} min\_latency_{m,s} \cdot weight_{m,s}}{\sum_{m,s} \min_{i \in [1,k]} p\_latency_{m,s,i} \cdot weight_{m,s}} \tag{6}$$

$min\_latency_{m,s}$ denotes the minimum latency among all tensor programs of subgraph s in model m, $weight_{m,s}$ as the frequency of subgraph s appearing in model m, $p\_latency_{m,s,i}$ is the i-th min latency ranked by the predicted score of the cost model. Top-k score reflects whether the model can accurately identify the most optimal scheduling.

To enable batch training, the input to the model is first reshaped to the same size. The input data size is 40×20 for GPUs and 25×22 for CPUs, which has been verified as the optimal size in TLP. In order to determine the optimal hyper-parameters of the model, a large number of experiments were conducted. We evaluate the different combinations of encoder number, MLP layer number, multi-attention head and hidden dim, experiment results show that the cost model achieves the highest when the encoder number and MLP layer are 3, multi-attention head is 8 and hidden dim is 128 for all the encoders and decoders.

Benefiting from the encoder-decoder architecture and self-attention module, our cost model can extract hidden features on a higher dimension and identify features that have a greater impact on the latency, thus exhibiting improved versatility and prediction accuracy. As shown in Table I, our cost model

TABLE I
Top-1 and Top-5 scores of proposed cost model on all hardware platforms of TenSet.

		ResNet-50	MobileNet-V2	ResNext-50	BERT-tiny	BERT-base
platinum-8272	**Top-1 Score**	0.8981	0.7921	0.9198	0.9565	0.7724
	Top-5 Score	0.9572	0.9201	0.9525	0.9627	0.9101
e5-2673	**Top-1 Score**	0.8271	0.7124	0.8557	0.9360	0.9489
	Top-5 Score	0.9388	0.9209	0.9733	0.9817	0.9906
epyc-7452	**Top-1 Score**	0.8618	0.7455	0.9106	0.8928	0.9079
	Top-5 Score	0.9769	0.9084	0.9725	0.9471	0.9873
graviton2	**Top-1 Score**	0.7713	0.7488	0.7278	0.8822	0.8659
	Top-5 Score	0.8770	0.8891	0.9199	0.9551	0.9536
NVIDIA T4	**Top-1 Score**	0.8307	0.8999	0.8738	0.9597	0.9076
	Top-5 Score	0.9075	0.9554	0.9499	0.9781	0.9564
NVIDIA K80	**Top-1 Score**	0.9201	0.8667	0.9158	0.8775	0.8689
	Top-5 Score	0.9617	0.9312	0.9681	0.9587	0.9798

TABLE II
Performance comparison with TenSet-MLP and TLP.

	TenSet-MLP		TLP		Ours	
	Top-1 score	Top-5 score	Top-1 score	Top-5 score	Top-1 score	Top-5 score
platinum-8272	0.8748	0.9527	0.9194	0.9710	0.9091	0.9524
e5-2673	0.8332	0.8977	0.8941	0.9633	**0.8960**	**0.9723**
epyc-7452	0.8510	0.9175	0.9055	0.9494	0.8884	**0.9607**
graviton2	0.7799	0.9049	0.8207	0.9226	**0.8245**	**0.9329**
NVIDIA T4	0.9083	0.9629	0.9059	0.9741	**0.9139**	0.9579
NVIDIA K80	0.8757	0.9528	0.8847	0.9250	**0.8898**	**0.9629**

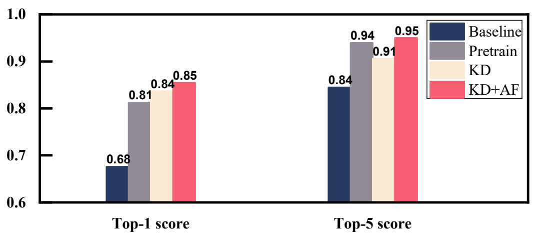

Fig. 5. Ablation study results of TKD.

is valid for different networks and hardware platforms, the average top-1 and top-5 score are 0.8795 and 0.9546 on CPU, 0.9019 and 0.9602 on GPU respectively. To fully illustrate the progressiveness of our cost model, we replicated the most advanced works and compared the results with our cost model. We deploy all the cost models by PyTorch 1.7.0 and train each model by 50 epochs on NVIDIA Tesla V100, and select the model with the lowest loss to evaluate. The comparison is shown in Table II, our cost model achieves improved top-1 score on all platforms by TenSet-MLP and 4 of 6 platforms by TLP.

B. *Cross-device Evaluation*

To demonstrate the effectiveness of the adaptive tensor program filter and cross-device knowledge distillation, ablation study is conducted. Consider that measuring tensor programs is extremely time-consuming on Jetson TX2, which will take several months if we measure the full dataset. To balance the overhead and the performance of the cost model, we measured 9% of data and conducted four experiments to evaluate the results: 1) randomly select 9% data and train the cost model, this result is performed as *baseline*; 2) pre-train the cost model on source device and fine-tune on the target device; 3) train the cost model with cross-device knowledge distillation only; 4) train the cost model with adaptive tensor program filter and cross-device knowledge distillation.

Fig. 5 shows the experimental results, the source device is NVIDIA Tesla K80 and the target device is NVIDIA Tesla T4. The results demonstrate that our method is more effective than baseline and fine-tuning, we reduce the training data and training time by 11× and dramatically increase the prediction accuracy of the cost model. Compared to the baseline, our cost model attains 26.3% and 12.5% improvement on top-1 and top-5 score. Furthermore, our method outperforms fine-tuning by 5.13% and 1.13% on top-1 and top-5 score respectively. By using cross-device knowledge distillation only, the cost model can better identify the schedule primitives with the highest performance, and by adding the adaptive tensor program filter, the versatility of the cost model improved.

C. *End-to-end Evaluation*

To evaluate the end-to-end compilation results, we integrate TKD into Ansor. and conduct on-device auto-tuning. The experimental hardware platform is an edge GPU device, 8GB NVIDIA Jetson TX2, the python version is 3.6.9 and torch version is 1.6.0. We evaluate TKD on three small-scale DNN models: ResNet-18, MobileNet-V2 and MobileNet-V3, the batch size is set to 1 and the input image size is $3 \times 224 \times 224$. We use the pre-trained cost model on the NVIDIA Tesla T4 dataset as the teacher cost model and conduct cross-device adaptation between NVIDIA Tesla T4 and NVIDIA Jetson TX2 using TKD, the number of encoder and decoder layers are reduced by 30% in the student cost model.

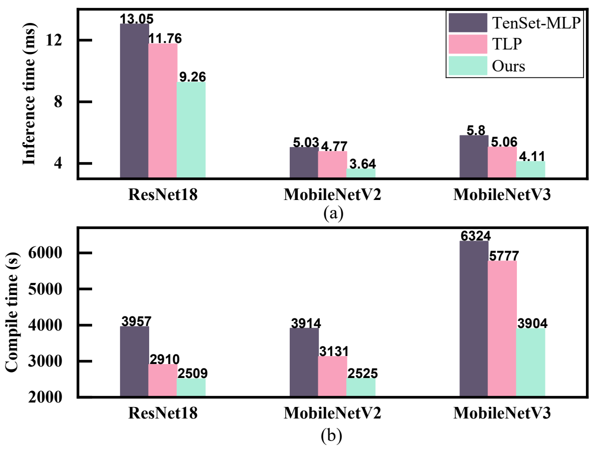

Fig. 6. End-to-end compile result on NVIDIA Jetson TX2.

The golden metrics to evaluate the end-to-end performance of DL compilers are compilation time and inference time, compilation time reflects the efficiency of DL compilers and inference time is the total runtime latency of the optimized generated tensor program, which can reflect the effectiveness of DL compilers.

We compare TKD with two state-of-the-art works: TenSet-MLP and TLP. The experimental results are shown in Fig 6, the results demonstrate that TKD outperforms TenSet-MLP and TLP on both compilation time and inference time. TKD reduces compilation time by 1.58x and 1.16x, while achieving 1.40x and 1.27x speedups of tensor programs, respectively. TKD compresses the model scale and reduces the number of parameters while leveraging the cross-device knowledge to enhance its accuracy, thus achieving improved performance on both compilation time and inference time.

V. Conclusion

To address the challenge of efficiently adapting a pre-trained cost model to embedded devices with limited computational resources, this paper proposes an efficient cross-device compiler framework TKD. TKD can effectively reduce the time budget and training cost of adapting cost models to embedded devices by filtering out redundant data and leveraging the cross-device knowledge extracted from similar hardware platforms. Experiments have been conducted to verify that TKD can reach improved or comparable performance than state-of-the-art works.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] J. Hu, L. Shen, and G. Sun, “Squeeze-and-excitation networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 7132–7141.

[3] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

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

[5] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” in *Proceedings of naacL-HLT*, Vol. 1. 2019.

[6] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, “Hierarchical attention networks for document classification,” in *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, 2016, pp. 1480–1489.

[7] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai et al., “Tpuv4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[8] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen et al., “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[9] M. Tan and Q. Le, “Efficientnet: Rethinking model scaling for convolutional neural networks,” in *International conference on machine learning.* PMLR, 2019, pp. 6105–6114.

[10] J. Lin, W. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, “Mcunet: Tiny deep learning on iot devices,” *Advances in neural information processing systems*, vol. 33, pp. 11711–11722, 2020.

[11] Y. Zhang, J. Yu, Y. Chen, W. Yang, W. Zhang, and Y. He, “Real-time strawberry detection using deep neural networks on embedded system (rtsd-net): An edge ai application,” *Computers and Electronics in Agriculture*, vol. 192, p. 106586, 2022.

[12] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro et al., “cudnn: Efficient primitives for deep learning,” 2014, *arXiv:1410.0759*.

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[14] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, et al., “TVM: An automated End-to-End optimizing compiler for deep learning,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[15] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, et al., “Ansor: Generating high-performance tensor programs for deep learning,” in *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, 2020, pp. 863–879.

[16] L. Zheng, R. Liu, J. Shao, T. Chen, J. Gonzalez, I. Stoica, et al., “Tenset: A large-scale program performance dataset for learned tensor compilers,” in *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[17] Z. Zhao, X. Shuai, N. Ling, N. Guan, Z. Yan, and G. Xing, “Moses: Exploiting cross-device transferable features for on-device tensor program optimization,” in *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*, 2023, pp. 22–28.

[18] Y. Zhai, Y. Zhang, S. Liu, X. Chu, J. Peng, J. Ji, et al., “Tlp: A deep learning-based cost model for tensor program tuning,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Volume 2, 2023, pp. 833–845.

[19] Hochreiter S , Schmidhuber J. ”Long Short-Term Memory,” *Neural Computation MIT-Press*, 1997.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, et al., “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, et al., “Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems,” 2015, *arXiv:1512.01274*.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, et al., “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” 2016, *arXiv:1603.04467*.

[23] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, et al., “Learning to optimize tensor programs,” *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[24] K. Konyushkova, R. Sznitman, and P. Fua, “Learning active learning from data,” *Advances in neural information processing systems*, vol. 30, 2017.

[25] J. Bi, X. Li, Q. Guo, R. Zhang, Y. Wen, X. Hu, et al. ”BALTO: fast tensor program optimization with diversity-based active learning.” *The Eleventh International Conference on Learning Representations*, 2022.

[26] X. Zeng, T. Zhi, Z. Du, Q. Guo, N. Sun, Y. Chen, ”ALT: optimizing tensor compilation in deep learning compilers with active learning.” *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020.

[27] X. Wang, C. Li, N. Golbandi, M. Bendersky, and M. Najork, “The lambdaloss framework for ranking metric optimization,” in *Proceedings of the 27th ACM international conference on information and knowledge management*, 2018, pp. 1313–1322.