Moses: Exploiting Cross-device Transferable Features for On-device Tensor Program Optimization

Zhihe Zhao[†], Xian Shuai[†], Neiwen Ling[†], Nan Guan[§], Zhenyu Yan[†], and Guoliang Xing[†,*]
[†]The Chinese University of Hong Kong, Hong Kong SAR, China
[§]City University of Hong Kong, Hong Kong SAR, China

ABSTRACT

Achieving efficient execution of machine learning models on mobile/edge devices has attracted significant attention recently. A key challenge is to generate high-performance tensor programs for each operator inside a DNN model efficiently. To this end, deep learning compilers have adopted auto-tuning approaches such as Ansor. However, it is challenging to optimize tensor codes for mobile/edge devices by auto-tuning due to limited time budgets and on-device resources. A key component of DNN compilers is the cost model that can predict the performance of each configuration on specific devices. However, current design of cost models cannot provide transferable features among different hardware accelerators efficiently and effectively. In this paper, we propose Moses, a simple yet efficient design based on the lottery ticket hypothesis, which fully takes advantage of the hardware-agnostic features transferable to the target device via domain adaptation to optimize the time-consuming auto-tuning process of DNN compiling on a new hardware platform. Compared with state-of-the-art approaches, Moses achieves up to 1.53X efficiency gain in the search stage and 1.41X inference speedup on challenging DNN benchmarks.

CCS CONCEPTS

• **Computer systems organization** → *Real-time System.*

KEYWORDS

Efficient DNN Processing, DNN Compiler, Transfer Learning

1 INTRODUCTION

Efficient inference of deep neural networks (DNN) is of great importance for real-time AI applications such as autonomous driving and augmented reality, which usually run on embedded devices with limited computation resources [18, 26, 27]. Existing approaches rely on hand-optimized acceleration libraries, e.g., NVIDIA cuDNN [22] and Intel MKL [10]. However, they are vendor-specific, and thus cannot support a wide range of diverse hardware devices. Deep learning compilers can enable high-performance code generation for various models on different hardware automatically. During compilation, the deep learning model is first transformed into a computation graph, where each node in the graph represents an operator, and each edge represents the data flow. The computation graph is then partitioned into subgraphs according to the fusion opportunity among DNN kernels [20, 21]. Finally, the automatic code generation process produces high-performance execution codes for each subgraph. Typically, the tensor computations inside most DNN operators such as convolutional kernel are implemented by a set of compute-intensive nested loops. To accelerate kernel inference speed at the for-loop level, we need to maximize the corresponding intra-operator parallelism. One common method is to use cache-blocking strategies, which break the computation down into manageable chunks to enhance the cache hit rate [16]. As an example, we now discuss a simple Matrix Multiplication case, which is iteratively processed inside a convolutional kernel and can be performed on heterogeneous hardware platforms with the same computation pattern. As shown in Figure 1, the data chunk is computed block by block, inside of which the memory accesses are grouped within a small neighborhood to leverage the memory locality. The optimal block size can be automatically determined by DNN compilers such as TVM [6]. Generating such high-performance tensor programs from a given high-level expression is extremely difficult, as the optimal organization and the parameters of the for-loops vary significantly for different devices. Therefore, to accelerate the end-to-end model inference on various hardware platforms, existing DNN compilers first generate a large space of candidate configurations and then search for the optimal one based on on-device profiling [5]. This process is referred to as auto-tuning [29].

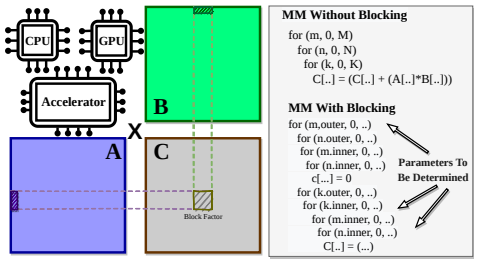

Figure 1: Matrix Multiplication of A and B, and the generated lowering intermediate results corresponding to with/without cache blocking (based on *tvm.lower()* provided by TVM [6]).

Although DNN compilers can produce optimized programs for DNN models on specific platforms to accelerate the end-to-end model inference, *the compilation process of each model requires excessively long search time, especially on mobile and edge devices.* For example, the auto-tuning time for a ResNet-18 of AutoTVM [5] can take up to tens of hours on edge devices like NVIDIA Jetson

*Corresponding email: glxing@cuhk.edu.hk.

HotMobile '23, February 22–23, 2023, Newport Beach, CA, USA

ACM ISBN 979-8-4007-0017-0/23/02…$15.00
https://doi.org/10.1145/3572864.3580330

TX2. During the compilation, an auto-tuner needs to run different code combinations on the target hardware to find the optimal tensor program with minimal execution time, which varies a lot among different edge platforms. Hence, *if a model tuned on powerful server-level platforms was directly deployed on embedded devices, the inference speed would be extremely slow.*

As auto-tuning is time consuming, it is impossible to measure the latency of all tensor program candidates on the hardware platform. To reduce the time consumption of on-device measurements, the tensor program optimizer employs a cost model to estimate the performance of the potential candidates in the search space [3]. However, training a cost model offline still requires a large number of measurements on each specific edge platform. For instance, Tenset [30] provides a tensor program performance dataset collected from 6 devices, containing 52 million program performance records. Based on this dataset, the online search time can be reduced without sacrificing the quality of optimized tensor programs. However, when two hardware platforms differ significantly in architecture, such a vanilla fine-tuning approach would fail to learn the run-time behaviors of a new device, and hence performs poorly at run time.

Previous efforts trying to address this challenge mainly focus on either designing a new cost model to search over the large space of tensor program transformation choices [4, 13, 25] or exploring effective search algorithms during auto-tuning [2, 15]. However, these approaches still need large numbers of iterations for hardware-dependent measurements during the search. Additionally, for edge devices, achieving low inference latency should be treated as the first citizen while the search efficiency should also be considered. This poses another challenge: how to minimize the time budget for online learning of the cost model compared to other domain adaptation methods while ensuring or even improving the quality of searched tensor programs for a DNN kernel. Motivated by these challenges, we propose Moses, a novel cost model adaptation framework based on the lottery ticket hypothesis [8], which can *adapt the trained cost model from a source device to a new target device with high efficiency.* A key observation in our paper is that the source feature space of our cross-device cost model transfer learning problem can be divided into hardware-aware and hardware-agnostic domains. This motivates us to design a novel transfer learning method for cross-device program optimization by distilling transferable and non-transferable model parameters to minimize the domain discrepancy in an efficient and adaptive manner. We summarize the contributions of this work as follows:

1) To the best of our knowledge, Moses is the first work that can achieve highly efficient auto-tuning among different hardware platforms based on transfer learning. Moses enables the DNN compiler to generate optimized tensor programs with significantly shorter search time for a new device.
2) We propose a novel approach that can automatically identify the transferable hardware-independent parameters of a pre-trained cost model, and achieve cross-device cost model adaption via fine-tuning.
3) We conduct comprehensive experiments and show that Moses is a general and effective approach for diverse hardware platforms and DNNs.

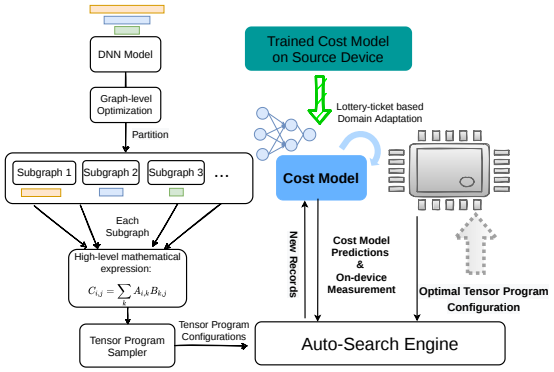

Figure 2: The pipeline of automatic tensor program generation for a given DNN model. The green and blue boxes are the main contributions of Moses.

2 RELATED WORK

DNN Compilers. General DNN compilers optimize the computation flow of DNN tasks on two levels: graph level and tensor level. Some notable compilers are TVM [6], TASO [11], XLA [1] and Halide [17]. These compilers either utilize compiler techniques such as graph substitutions to optimize the intermediate representation (IR) level graph or focus on tensor program optimization using learning-based algorithms. Building on these DNN compilers, recent works treat the optimization process as a black box and propose some advanced searching or cost model training approaches based on run-time information [5][2].

Auto-Tuning with Cost Models. Figure 2 shows the pipeline of a search-based low-level tensor code generation used by TVM [6]. The neural network is first partitioned by the graph-level optimizer. The search algorithm will search for the optimal low-level tensor implementation for each subgraph. Usually, the search space is in the order of millions for CPUs and billions for GPUs, as there are a variety of schedule primitives such as tiling and thread binding [6, 28]. On the one hand, such a large search space enables the automatic tensor compiler to find the program that is better than the hand-optimized implementation. On the other hand, the large search space can incur significant search time, especially on embedded devices. To accelerate the searching process, TVM introduces the ML-based cost model to directly predict the time cost of the innermost non-loop program rather than extensively measuring the program's run-time. In this paper, we adopt the 164-*d* features in Ansor [29] to represent the program.

Cross-Device Cost Model Adaptation. A major practical drawback of the current automatic tensor program optimization approach [5] is the extremely long search time. Recent work such as [2, 24] provides a breakdown of the search time and shows that the time for on-device measurements dominates. Therefore, a solution to shorten the online searching time is to collect a comprehensive tensor program performance dataset offline, and pre-train a cost model that can be directly utilized [30]. However, the cost model is device-specific, as its input features include configuration knobs that are closely related to the hardware architecture such as the size of BlockIdx and ThreadIdx. When two architectures are

significantly different (e.g., edge GPUs and mobile GPUs), the traditional vanilla transfer learning approach may fail. During vanilla transfer learning, excessive unnecessary hardware-aware domain knowledge is involved, which slows down the cost model convergence. MTL-TLP [24] proposes to use multi-tasking techniques to address the unavailability of cross-hardware cost models. The key idea is to use a small amount of target hardware data to train a cost model with high performance. However, the online collected hardware-dependent features are still hand-crafted with heavy feature engineering. They cannot be easily ported to new hardware, especially those with huge hardware architecture gaps.

3 MOSES

3.1 Problem Formulation

The Auto-tuning process in a DNN compiler aims to generate a large search space of tensor program configurations and to find the optimal one based on the on-device performance measurement records [6]. During the search process in auto-tuning, the compiler picks the top-k (where k can be one) programs with the performance predictions from the cost model for each task. The process can utilize mixed on-device measurements and cost model predictions. If the on-device measurement cost is large, the process can completely rely on the cost model.

Formally, we let Function *Perf* denotes the hardware run-time measurement on latency (i.e., the performance); Ψ the set of possible tensor program configurations of a task which consists of a combination of parameters called *knobs*; t the transformation pass representing the tensor program; g the tensor programs generating functions with knobs and the transformation pass as inputs. Thus, given an input subgraph, the objective of the auto-tuning process is finding the optimal combination of knobs ψ^* to maximize the performance defined as $\psi^* = \text{argmin}_{\psi \in \Psi} Perf(g(\psi, t))$. Precise estimation of the performance of the tensor program can effectively reduce the time-consuming on-device measurements, especially for embedded or mobile devices. For example, on an NVIDIA TX2, the total on-device measurement time of a VGG16 model can be up to 10 hours. Thus, the objective of the cost model is to minimize the difference between the predictions and the real-world measurements, which is:

$$\Theta = \underset{\theta}{\text{argmin}} \, \|C(g(\psi, t)|\theta) - Perf(g(\psi, t))\|, \psi \in \Psi \qquad (1)$$

Here, function C denotes the cost model predictions on code performance and θ represents the weights of the trained cost model.

In this paper, we aim to find a new weight $\Theta^{\dagger}$ of the cost model for the target device based on the existing Θ obtained on source devices. In an adaptive manner, the cross-device cost model adaptation can guide the auto-search engine to generate more optimized codes of each subgraph more efficiently, for the target device.

3.2 Design of Moses

We propose Moses, a novel cross-device cost model transfer approach based on the lottery ticket hypothesis [8]. As shown in Figure 3, we transfer the cost model trained on source devices (e.g., Server GPUs) to target devices (e.g., mobile GPUs) by only fine-tuning part of the model parameters while keeping the rest of the parameters deactivated. The rationale of our design is two-fold.

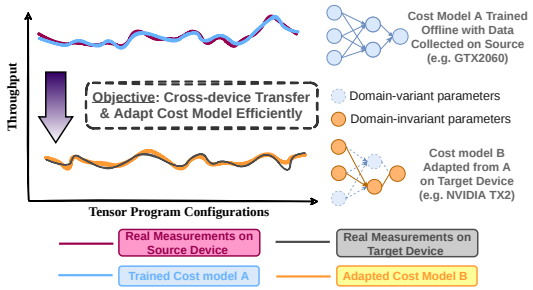

Figure 3: Given a cost model A trained on the source device, we aim to obtain a cost model B that can accurately predict the throughput of the target device under various tensor program configurations.

First, to accelerate the online search instead of collecting a dataset for every new device offline, we need to take advantage of the cost model pre-trained on source devices rather than training a new model from scratch. Second, as vanilla fine-tuning approaches may fail due to substantial architecture changes, we have to utilize the prior knowledge from the pre-trained cost model wisely.

In our problem, the feature space is independent of hardware architecture while the outputs of the cost model (throughput predictions for different tensor program configurations) are dependent, shown as:

$$H\{\mathcal{X}\} \equiv H\{\mathcal{X}_{Ind}\} + H\{\mathcal{X}_{Dep}\} \qquad (2)$$

where H maps to the hidden feature space representations, $\mathcal{X}_{Ind}$ and $\mathcal{X}_{Dep}$ present the decoupled feature space: hardware-independent information such as program unrolling factor and hardware-dependent information such as used memory of a non-loop innermost statement, respectively [29].

3.3 Lottery-Ticket-Based Cross-Device Adaptation

In order to solve this problem, we propose to leverage the following lottery ticket hypothesis in this paper: only part of the parameters in the trained cost model on source devices are essential for learning hardware-independent knowledge. In other words, Only part of the information needs to be adapted to the target device while other parameters tend to fit the domain. Based on this hypothesis, we can transform our problem into the following question: *How to learn domain invariant parameters to minimize the domain discrepancy brought by the hardware difference?*

The training data for cost model updating is collected online during the search, which makes the search very time-consuming due to inevitable on-device measurements. We denote by x the tensor program optimization knobs, y the corresponding throughput of x. Thus, we can define a set of labeled tensor program records on source device D_S by $S = \{(x_s^i, y_s^i)\}$, where i is the i_{th} record. Now given a target unlabeled program performance records set $T = \{(x_T^j)\}$ with different configurations of program knobs x sampled from target device D_T, in which a small records set $T^{'}$ can be collected by on-device measurements (which are conducted typically under a given time budget). Formally, The expected domain adaptation error on the target device can be defined as $\varepsilon(h(\Theta^{\dagger}))$

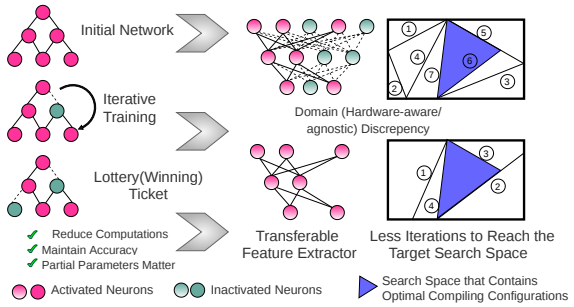

Figure 4: Illustration of the lottery ticket hypothesis and cross-device auto-tuning optimization. The left part shows the original hypothesis applied to model compression. The middle part shows how Moses is motivated by the hypothesis, and the right part shows the decrease in searching iterations due to the reduction of search space.

where h is a hypothesis learned from the target or the source domain. Thus the following inequity holds:

$$\varepsilon(h_{Target}(\Theta^{\dagger})) \leq \varepsilon(h_{Source}(\Theta^{*})) + dist(D_S(\mathcal{X}), D_T(\mathcal{X})), \quad (3)$$

We adopt the same notation in [19] to define *dist* as the distribution discrepancy distance between the cross-device domains. The feature representations are influenced by hardware differences in our problem. To achieve the cross-device domain adaptation, instead of minimizing the distance between feature representations and their resulting data distribution discrepancy, we show the effectiveness of the bound minimization strategy in solving the cross-device domain adaptation. We propose to find the bound limitation by optimizing the labeling black-box functions. Such an approach is inspired by the lottery ticket hypothesis, which was originally proposed in the context of model compression, showing that only part of the parameters is fit for model generalization [8]. We show in this paper that the same hypothesis is applicable to our problem (see Section 4). That is, *there exists a super-subnet, named winning ticket, with a set of essential parameters of the trained cost model on source devices, which would be the domain invariant information.* In other words, training from a super-subnet on the target device would achieve the optimal transfer performance in our cross-device domain adaptation problem. To answer the question of why the lottery-ticket hypothesis works, the key insight here is that the factors that affect the overall performance of DNN compiling can be divided into two categories: hardware-agnostic factors and hardware-aware factors. As illustrated in Figure 4, the idea of iterative pruning to find the "winning ticket" model structure in the lottery ticket hypothesis can be leveraged to address our problem, that is, finding the domain discrepancy between two auto-tuning search processes on the two hardware platforms. For each iteration of the tensor program sampling-search of optimization knobs, the cost model with more portions of domain transferable features is more easily to step close to the target search space, as shown in the right part of Figure 4.

We refer to parameters in these super-subnets as *transferable parameters* and the remaining set of parameters as *untransferable parameters*. A key question is how to distill these transferable parameters during each subgraph auto-tuning stage. We identify the distilling boundary criterion $\xi(ph)$ as $|w(ph) * \nabla w(ph)|$ where ph is the tuning phase of the subgraphs, $w(ph)$ represents the parameter weights and its gradient is $\nabla w(ph)$. If $\xi(ph)$ is larger than a certain threshold ϑ (e.g., 0.5), the corresponding $w(ph)$ can be viewed as transferable parameters. In contrast, $w(ph)$ would be regarded as domain-variant parameters if $\xi(ph)$ is small(e.g., close to zero). We iteratively update the boundary of domain-invariant parameters as well as variant parameters and update these invariant ones during each online training epoch to learn invariant representations to achieve minimization of bound limitation. Lastly, we adopt the adversarial loss training objective function inspired from [9], in such a way, the parameters with higher gradient flows, representing more benefits to the domain-invariant information learning process, are considered. As for the parameters that represent domain variant information, We update these parameters with a weight decay mechanism as a penalty, which is defined as:

$$w_v(ph+1) \rightarrow w_v(ph) - \alpha(wd(w_v(ph))) \quad (4)$$

where α is the learning rate, $w_v(ph)$ represents the domain-variant parameters and function wd represents the weight decay process for each updating phase.

3.4 Adaptive Tuning Data Partition

To gather on-device measurements efficiently and maintain the performance of the online domain adaptation cost model, we use an adaptive controller (AC) module to early terminate the hardware tuning data collection stage. The basic idea of AC is to statistically analyze the certainty of the online training cost model. For a given subgraph s which is to be tuned, we initially divide the total tuning tasks into *tuning tasks for online training* t_{train} with hardware measurement data collection and *tuning tasks for cost model predictions* t_{pred} with a ratio of p. We further divide t_{train} into $q \in 1, 2, .., q$ batches and collect both on-device measurement records $C(t_{train}(s))$ and $(t_{train}(s))$. Then, we use the coefficient of variations (the standard deviation divided by the mean) formulated as $CV = \frac{\sigma(C(t_{train}(s))_1, C(t_{train}(s))_2..C(t_{train}(s))_q)}{\mu(C(t_{train}(s))_1, C(t_{train}(s))_2..C(t_{train}(s))_q)}$ to dynamically estimate the certainty of the existing cost model, and terminate the hardware measurement phase in advance if the value is smaller than a certain value. We empirically set these hyper-parameters based on multiple trials in our experiments.

3.5 Putting Everything Together

In previous sections, we described the design details of the lottery-ticket-based cross-device cost model transfer and the adaptive online training data partition mechanism. We now put these components together and summarize the working flow of Moses.

Step 1. Pre-training a cost model on the source device: We pre-train a cost model offline using the dataset of on-device measurement records from the source device. This dataset includes randomly generated tensor programs for wide deep learning models.

Step 2. Transferring the trained model to the target device: The learned cost model from the source device is directly transferred to the target device in this step, which guides the search stage during auto-tuning.

Step 3. Adaptive training data partition with the AC module: We directly apply the learned cost model from the source

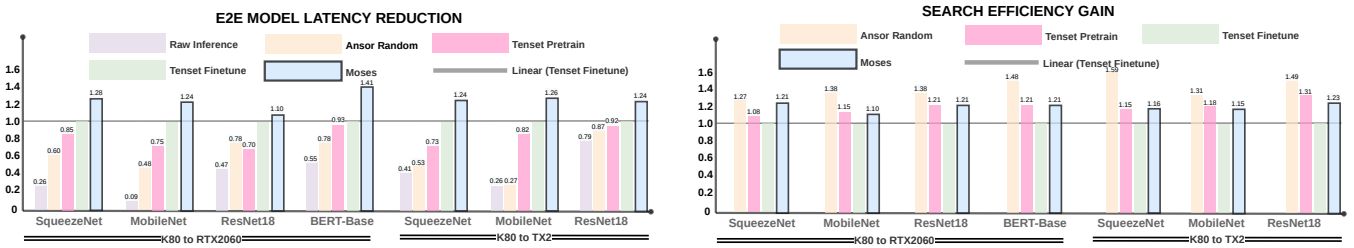

Figure 5: End-to-end DNN model inference latency reductions and auto-tuning search efficiency gain comparisons among MobileNet, ResNet18, BERT-Base, and SqueezeNet over two domain adaptation baselines.

device to the target device, to guide the search stage during auto-tuning. For each tuning task, we dynamically control the hardware measurement costs by using the AC module, where the portion of on-device measurements can be adjusted based on the evaluation results of cost model performance in that epoch.

Step 4. Online updating of the cost model with iterative pruning: For each tuning task, we divide the parameters of the cost model into domain-invariant ones and domain-variant ones based on the calculation of $\xi(i)$, and update the domain-invariant parameters with gradient decent while letting the rest gradually decrease to zero through weight decay.

During the auto-tuning process, Moses keeps updating the cost model in an adaptive and iterative manner based on the collected hardware measurement records, while the search algorithms keep querying the newest cost model for efficient explorations of optimal program configurations.

4 EXPERIMENTS

In this section, we evaluate the effectiveness and efficiency of Moses, with our proposed lottery-ticket based cost model adaptation method. We implement Moses as a plug-in cross-device cost model adaptation tool in TVM auto-tuning [6]. Specifically, the cost model fine-tuning is integrated with the tensor programs random sampling and an evolutionary search algorithm [29]. The training of the cost model is implemented in PyTorch. We set the max epoch to 30. We set the initial learning rate *alpha* to 0.001, and the distilling boundary criterion threshold ϑ to 0.5. Most of these parameters we set are adopted from Ansor [29] and Tenset [30] except the transferable parameter ratio. We conduct each experiment 10 times and record the corresponding results. The dataset and codes in this paper are available at https://github.com/Kyrie-Zhao/Moses.

4.1 Experimental Settings

Our experiments are conducted on NVIDIA GeForce GTX 2060 and NVIDIA Jetson TX2 with Pascal GPU architecture with 256 NVIDIA CUDA cores. We include four DNN models in our experiments: ResNet-18, MobileNet, BERT-base, and SqueezeNet. We use the default settings for other hyper-parameters provided by Ansor [29]. As for the backbone of the cost model, we choose the representative one used in Ansor, which is an MLP with two hidden layers, with 512 neurons for each. The two domain adaptation tasks we validate are $K80 \rightarrow 2060$ and $K80 \rightarrow TX2$. Considering BERT-base is usually deployed in cloud service due to its large model size, we do not include it in the $K80 \rightarrow TX2$ experiment. We use the end-to-end latency/throughput and the end-to-end search efficiency of auto-tuning as evaluation metrics. Specifically, we measure the obtained speedups of tuned tensor programs and the reductions of searching time of an input DNN model over other baselines including the state-of-art cost model transfer method provided in Tenset. We also introduce a concept named Cost Model & Auto-tuning Efficiency Gain Score (CMAT) to evaluate the cost model influence on the end-to-end inference performance at the same time, defined as: *CMAT = (Gain on Search Efficiency * Reduction on Tuned Model Latency-1)*100%*. As CMAT considers both search efficiency and end-to-end inference latency, it is an effective metric for evaluating the overall cross-device cost model adaptation performance.

Table 1: Comparisons of CMAT under different trials. S, R, M, and B refer to DNNs mentioned in Fig. 5.

CMAT (%)	2060-S	2060-R	2060-M	2060-B
Small Trials (200)	57.2	19.6	105	66.7
Large Trials (20000)	48.1	32.7	45.8	87.4

CMAT (%)	TX2-S	TX2-R	TX2-M
Small Trials (200)	28.7	66.4	64.5
Large Trials (20000)	44.7	53.1	45.9

4.2 Results

We compare Moses with four baselines: 1). Raw: inference based on vendor-supplied libraries (e.g. NVIDIA cuDNN). 2). Ansor-Random [29]: randomly initialize the cost model and train it from scratch during the auto-tuning. 3). Tenset-Pretrain: pre-train a cost model on Tenset dataset and directly apply it to the target device without fine-tuning. 4). Tenset-Finetune: utilize the cost model pre-trained on Tenset dataset and then perform the vanilla online fine-tuning.

4.2.1 Inference Time & Search Efficiency Comparisons. Fig. 5 shows the comparison of the final end-to-end inference time of the input DNN model optimized on each baseline. Moses achieves up to 41.1% faster inference speed over Tenset-Finetune and up to 53% higher speed over Tenset-Pretrain on the $K80 \rightarrow 2060$ baseline. Moses also achieves up to 26.2% over Tenset-Finetune and up to 52% over Tenset-Pretrain on the $K80 \rightarrow TX2$ baseline, respectively. Overall, Moses yields the best inference performance among all other configurations and algorithms. Fig. 5 shows the auto-tuning search efficiency gains comparisons over these baselines. Moses also

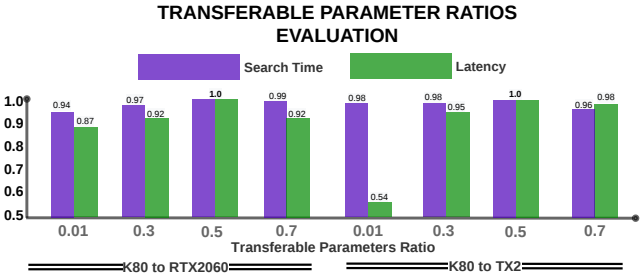

Figure 6: An illustration of Moses's performance with a wider ratio of transferable parameters.

outperforms all other baselines for both $K80 \rightarrow 2060$ and $K80 \rightarrow TX2$ settings. It can also be observed that, for some input DNN models such as SqueezeNet and MobileNet, Ansor-Random and Tenset-Pretrain could be more efficient than Moses. This is because these baselines provide no online learning during the auto-tuning. Therefore, the corresponding end-to-end model inference latency of these models can be greatly lower than Moses. The evaluation results show that the search efficiency gain of the $K80 \rightarrow 2060$ setting can be up to 47.8% while up to 58.5% for the $K80 \rightarrow TX2$ settings. This is because the on-device data collection costs on TX2 are much higher than on RTX2060.

4.2.2 CMAT Score Comparisons. Table. 1 shows the superior comprehensive performance of Moses over both small (200) and large (20000 for 2060, 5000 for TX2) numbers of trials across all input DNN models. As mentioned above, although Tenset achieves 15% auto-tuning efficiency gain on MobileNet based on the $K80 \rightarrow 2060$ setting, which is better than Moses (9.6%), the corresponding CMAT is -14.75% over Tenset fine-tuning, which is much worse than Moses (up to 45.8%). We can observe that for some cases (e.g. 2060-S), the CMAT gain under the small-trial setting can even be better than the large-trial one, due to the characteristics of the heuristic searching algorithm embedded in the auto-tuning component in TVM. The base performance of Tenset-Finetune can be extremely low with no prior knowledge during the transfer process of the cost model.

4.2.3 Ratio of Transferable Parameters. Here we provide the analysis of the ratio of transferable parameters. Fig. 6 shows the results of Moses on a wider setting of transferable parameters ratio: $\{0.01, 0.3, 0.5, 0.7\}$. According to the end-to-end performance results, we can observe that the optimal performance can be around 0.5. Generally speaking, the *std* value for settings of $\{0.3, 0.5, 0.7\}$ ratio is not large. The results illustrate that the optimal performance produced by different ratios is not sensitive to the ratio setting when it is ranging from 0.3 to 0.7.

5 DISCUSSION AND FUTURE WORK

Extension to More Hardware Platforms. With the emergence of a huge number of new hardware platforms, cross-device DNN compiling acceleration is becoming increasingly important [23, 24]. Our experiments employ only two edge platforms (GTX2060 and Jetson TX2) and leave a possible extension of auto-tuning optimizations on *mobile devices* as our future works. Another natural idea is to enable the compiling knowledge transfer among devices with larger hardware difference gaps, for example, the transfer between server-level GPUs to CPUs or FPGAs.

Auto-Tuning Search Space Pruning. To further accelerate the auto-tuning process on the target device, another possible research direction is to offer accurate search space pruning during compiling, especially on mobile and edge devices, since there are more hardware limitations such as the number of total ALUs (Arithmetic logic units) and DRAM bandwidth that would become the potential bottlenecks for fast auto-tuning on these devices, which naturally restricts the search space [12]. To strategically shrink the search space, we plan to explore *Bayesian Non-parametric Space Partition (BNSP)* [7] to adaptively guide the subspace pruning during each auto-tuning iteration. BNSP models provide a flexible and geometrically interpretative way to describe the implicit and complex relationships among different covariates (features), with which we can partition the N-dimensional feature data space (tensor programs) into a set of blocks, thus exponentially decreasing the numbers of search iterations.

Energy-aware DNN Compiling for Edge Devices. Edge AI applications demand both low power consumption and real-time responses. Unfortunately, most high-performance DNN kernels generated by DNN compilers are aimed to improve computational efficiency while the memory efficiency, which affects the on-device power consumption, has gained little attention [14]. Most of these deep-learning compilers adopted loop-oriented scheduling primitives and designed auto-tuning frameworks on top of them. It is possible to insert more hardware-centric schedule spaces into the compiling process to generate energy-aware kernels. As a simple example, it is possible to add $PeakPower(TensorProgram)$ or $AveragePower(TensorProgram)$ into the cost model. The key challenge falls in how we can accurately measure the energy value for each tensor program record efficiently since the time duration for each tuning record is usually at a millisecond level.

Meta Learning for Fast Adaptation. Another research direction is to investigate meta-learning-based approaches to accelerate cross-device auto-tuning [23]. We will design a cost model with meta-learned parameters which enables to leverage of previously learned knowledge and experiences with similar programs, which can enhance the fast cross-device compiling knowledge transfer and thus reduce the high optimization overhead.

6 CONCLUSION

We present Moses, a new framework to optimize the auto-tuning process in the DNN compiler, and thus enable fast compiling knowledge transfer among mobile and edge devices. Our approach achieves cross-device adaptation of a trained cost model by updating the domain invariant parameters during online learning, which greatly improves the efficiency of the DNN compiling process and the end-to-end throughput of tuned tensor programs on the target device.

ACKNOWLEDGEMENT

The work described in this article was supported by the Research Grants Council (RGC)-General Research Fund under Grant No. 14209619.

REFERENCES

[1] Martín Abadi et al. Tensorflow: A system for large-scale machine learning. *CoRR*, abs/1605.08695, 2016.

[2] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. *CoRR*, abs/2001.08743, 2020.

[3] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. A deep learning based cost model for automatic code optimization. *CoRR*, abs/2104.04955, 2021.

[4] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman P. Amarasinghe. A deep learning based cost model for automatic code optimization. *CoRR*, abs/2104.04955, 2021.

[5] Tianqi Chen et al. Learning to optimize tensor programs. *CoRR*, abs/1805.08166, 2018.

[6] Tianqi Chen 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)*, October 2018.

[7] Xuhui Fan et al. Bayesian nonparametric space partitions: A survey. *arXiv preprint arXiv:2002.11394*, 2020.

[8] Jonathan Frankle et al. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

[9] Zhongyi Han, Haoliang Sun, and Yilong Yin. Learning transferable parameters for unsupervised domain adaptation. *arXiv preprint arXiv:2108.06129*, 2021.

[10] Intel. Intel mkl-dnn. https://oneapi-src.github.io/oneDNN/v0/index.html, 2022.

[11] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile gpu workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 190–205, 2021.

[13] Samuel J Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. *arXiv preprint arXiv:2008.01040*, 2020.

[14] Skanda Koppula, Lois Orosa, A Giray Yağlıkçı, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. Eden: Enabling energy-efficient, high-performance deep neural network inference using approximate dram. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 166–181, 2019.

[15] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. Adatune: Adaptive tensor program compilation made efficient. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 14807–14819. Curran Associates, Inc., 2020.

[16] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. The deep learning compiler: A comprehensive survey. *CoRR*, abs/2002.03794, 2020.

[17] Tzu-Mao Li et al. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), July 2018.

[18] Neiwen Ling, Kai Wang, Yuze He, Guoliang Xing, and Daqi Xie. Rt-mdl: Supporting real-time mixed deep learning tasks on edge platforms. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, 2021.

[19] Yishay Mansour et al. Domain adaptation: Learning bounds and algorithms. *arXiv preprint arXiv:0902.3430*, 2009.

[20] Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[21] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[22] NVIDIA. Nvidia cudnn. https://docs.nvidia.com/deeplearning/cudnn/api/index.html, 2022.

[23] Jaehun Ryu and Hyojin Sung. Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. *CoRR*, abs/2102.04199, 2021.

[24] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. *arXiv preprint arXiv:2211.03578*, 2022.

[25] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. Dynatune: Dynamic tensor program optimization in deep neural network compilation. In *International Conference on Learning Representations*, 2021.

[26] Zhihe Zhao, Zhehao Jiang, Neiwen Ling, Xian Shuai, and Guoliang Xing. Ecrt: An edge computing system for real-time image-based object tracking. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 394–395. ACM, 2018.

[27] Zhihe Zhao, Kai Wang, Neiwen Ling, and Guoliang Xing. Edgeml: An automl framework for real-time deep learning on the edge. In *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, IoTDI '21, page 133–144, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Zheng et al. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Lianmin Zheng et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[30] Lianmin Zheng 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.