One-Shot Tuner for Deep Learning Compilers

Jaehun Ryu
jaehunryu@postech.ac.kr
Department of Computer Science and Engineering
POSTECH
Pohang, South Korea

Eunhyeok Park
eh.park@postech.ac.kr
Department of Computer Science and Engineering
POSTECH
Pohang, South Korea

Hyojin Sung
hsung@postech.ac.kr
Department of Computer Science and Engineering
POSTECH
Pohang, South Korea

Abstract

Auto-tuning DL compilers are gaining ground as an optimizing back-end for DL frameworks. While existing work can generate deep learning models that exceed the performance of hand-tuned libraries, they still suffer from prohibitively long auto-tuning time due to repeated hardware measurements in large search spaces. In this paper, we take a neural-predictor inspired approach to reduce the auto-tuning overhead and show that a performance predictor model trained prior to compilation can produce optimized tensor operation codes without repeated search and hardware measurements. To generate a sample-efficient training dataset, we extend input representation to include task-specific information and to guide data sampling methods to focus on learning high-performing codes. We evaluated the resulting predictor model, One-Shot Tuner, against AutoTVM and other prior work, and the results show that One-Shot Tuner speeds up compilation by 2.81x to 67.7x compared to prior work while providing comparable or improved inference time for CNN and Transformer models.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Neural networks*; *Discrete space search.*

Keywords: optimizing compilers, autotuning, performance models, deep neural networks

ACM Reference Format:
Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. 2022. One-Shot Tuner for Deep Learning Compilers. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC '22), April 02–03, 2022, Seoul, South Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3497776.3517774

1 Introduction

Deep learning (DL) models have recently emerged as an application field that drives innovations in domain-specific optimization technologies. While many DL programming frameworks [1, 11, 41] rely on hand-optimized libraries such as NVIDIA cuDNN/cuBLAS or Intel oneDNN for their backends, DL compilers [12, 44, 47], which can generate codes specifically optimized for the target model, are exploring the potential as a more flexible and portable solution on increasingly diverse and heterogeneous platforms.

One key challenge for DL compilers lies in generating highly optimized codes with comparable or better performance than hand-tuned libraries. However, conventional rule-driven optimization heuristics and cost models are often too general to fine-tune tensor operations with domain-specific knowledge and/or too rigid to adapt to widely varying performance characteristics of DL hardware [2, 6, 17, 24]. Thus, DL compilers are increasingly adopting a data-driven approach to determine optimization strategies [4, 7, 13, 15, 44]. These "auto-tuning" compilers use statistical cost models to learn the correlation between programs and runtime behaviors from profiling runs. Then apply space exploration algorithms, i.e., simulated annealing or genetic algorithm, on the cost model to predict the candidates for the best performing codes. Profiling the candidates refines the cost model and guides the search to the more promising space. By repeating this process numerous times, DL compilers can search for the ideal compilation configurations.

Although the auto-tuning approach can produce high-performing codes, it incurs hours of compilation time from repeated hardware measurements to explore the huge optimization space even for simple convolution kernels and to train a performance-predicting cost model. Prior work reduced the overhead by using more efficient search mechanisms [3, 36, 60, 61], by improving cost models to guide the search with a higher accuracy [13, 48], or by reducing/accelerating profiling [21], often resulting in locating more optimized codes as well. However, even with all the improvements, the inherent cost of the core mechanism – building the cost model for optimization space through tons of profiling runs – remains significant. The most recent methods still spend hours with hundreds of iterations per layer to get reliable results [60, 61].

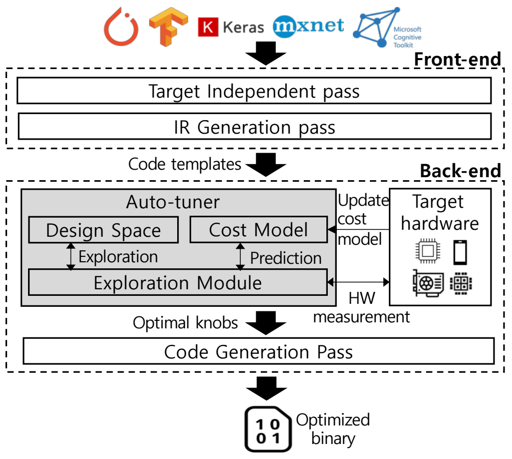


Figure 1. The Structure of DL auto-tuning Compilers

Table 1. Description of knobs in TVM

Knobs	Definition
tile_f tile_y tile_x	Loop tiling parameters on the number of filter, height, and weight of feature maps
tile_rc tile_ry tile_rx	Loop tile reduction parameters on the number of channels, height, and weight of filters
auto_unroll max_step	Guide max unroll iterations
unroll_explicit	Explicitly unroll loops
offset_bgemm	Determine offset value in bgemm
warp_row_tiles warp_col_tiles	Number of WMMA tiles for row and col in one warp
block_row_warps block_col_warps	Number of warps needed for row and col in one block

Recent work [13, 48] showed that the auto-tuning time can be reduced by conditioning the cost model through knowledge reuse. The cost model, pre-trained by transfer learning or meta learning with performance distributions for kernels of different input/output shapes or different operation types, more quickly adapts to the search space of a compiled kernel and more effectively guides the search than a model trained from scratch. The potential for a predictor model that generalizes for unseen data in distinct search spaces suggested by the results leads us to ask the following question: *If we train a cost model with a subset of random codes and their execution times prior to compilation, can we replace the auto-tuning process with a one-shot validation with the model?*

Thus, we propose **One-Shot Tuner**, a pre-trained performance predictor model that can practically eliminate the repeated iterations of hardware measurement and cost model update in the auto-tuning process. Inspired by recent research in the neural architecture search (NAS) domain [55], we train the One-Shot Tuner model with a set of randomly generated code samples to learn correlations between optimized tensor programs and their runtime performance, and to identify high-performing codes during compilation.

However, pre-training a model to predict performance distributions of unseen tensor operations with varying types and input/output shapes, without any online adaptation, poses a unique challenge in designing the sampling process of training data and input representation for One-Shot Tuner. In this paper, we propose the following to address the challenge: (1) input code representation extended to include explicit task-specific features, (2) a task-sampling mechanism that exploits layer type and size distributions of existing models to improve random sampling efficiency, and (3) an exploration-based code sampling mechanism that enables the predictor model to focus on learning high-performing codes.

We implemented One-Shot Tuner as a Transformer-based cost model for the AutoTVM-based auto-tuner [13], and compared its compilation time and compiled model performance with those of existing TVM-based auto-tuning solutions. The evaluation result shows that One-Shot Tuner is 2.81x to 67.7x faster than all prior work in generating optimized models, which provide comparable or improved inference time with for a range of CNN and Transformer models showing that a single iteration of search and validation with One-Shot Tuner is sufficient for locating high-performing codes.

The rest of the paper is organized as follows. Section 2 provides background information for DL compiler frameworks with auto-tuning support and a NAS solution that inspired One-Shot Tuner. Section 3 presents motivations and challenges for One-Shot Tuner. Section 4 describes the design and implementation of One-Shot Tuner, focusing on how we design input features and sample data to train the proposed predictor model. Section 5 and 6 present and analyze the evaluation result of One-Shot Tuner in terms of end-to-end auto-tuning time and inference time of compiled models. Section 7 discusses the cost and feasibility of One-Shot Tuner. Section 8 describes related work, then Section 9 concludes the paper.

2 Background

Deep learning compilers. DL compiler frameworks provides programming infrastructure to design and implement DL models and code-generating back-ends to either translate the high-level models into a sequence of hand-tuned library calls or optimized binary codes. Popular DL frameworks [1, 11, 29, 41] provide different levels of programmability, portability, and compilation support. While some frameworks have dedicated compilers (e.g., XLA [34] for TensorFlow [1] and Glow [47] for PyTorch [41]), general compiler back-ends such as TVM [12] and Tensor Comprehensions [52] can support multiple front-ends.

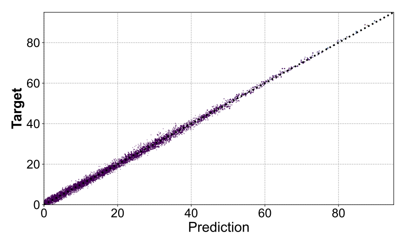


Figure 2. Predicted vs. target accuracy of a regression model pre-trained with data samples for VGG-16.

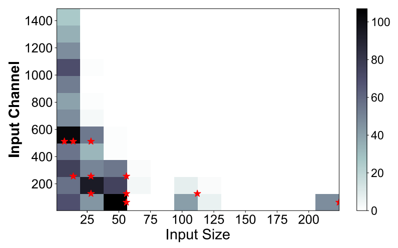


Figure 3. Visualization of the occurrence frequency of input and channel sizes crawled from the online model zoo.

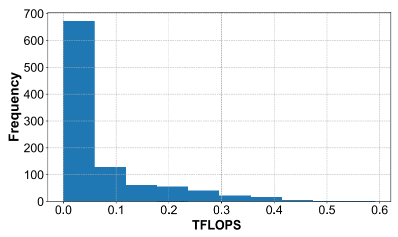


Figure 4. Histogram of inference time for codes sampled by random knob sampling for *ResNet-18*.

DL compiler frameworks consist of language front-ends and code-generating back-ends, as shown in Figure 1. The front-end translates input models into high-level internal representation (IR; often graph-based) and applies target-independent optimizations such as operator fusion and data layout transformation, whereas the back-end goes through target-dependent optimization passes that further transform IR to better exploit target hardware features. TVM [12], an open-source DL compiler, provides an auto-tuning framework as a back-end optimization pass.

Template-based auto-tuning in TVM. In general, auto-tuning approaches the optimization problem as exploring a search space of all optimized codes to locate the best-performing version with hardware measurements [22, 30, 43]. For efficient navigation of the search space, a cost model function is often used to predict code performance and guide the search. With a code-generating function ρ that uses optimization parameters ϕ (e.g., tile size), tensor operation σ (e.g., 2D convolution) and operation options c (e.g., input/output sizes) to create a unique search space D_{σ_c} of optimized codes and a cost model function f that predicts performance of codes in D_{σ_c}, an auto-tuning "task" solves a problem formulated as

$$\phi^* = \text{argmax}_{\phi} f(\rho(\phi|\sigma, c)). \tag{1}$$

In AutoTVM [13], ρ relies on code templates pre-defined for each tensor operation on a given platform, and ϕ consists of tunable parameters (*knobs*) that determine how the code template is optimized. Thus, an auto-tuning task for σ_c finds an optimal combination of knobs (Table 1) to maximize the performance predicted by the cost model f. AutoTVM adopts a machine-learning based predictor model for f, and for each auto-tuning task, f is dynamically trained with hardware measurements of codes in D_{σ_c}. One-Shot Tuner does not change the problem itself, but proposes to solve it with f' pre-trained with samples from a set of D's to deliver ϕ with competitive performance for an unseen D.

Neural Predictor. Neural architecture search (NAS) is a method that uses machine learning algorithms to locate the best model architecture in a given search space [37, 45, 63]. NAS performs search space exploration to identify the architecture that maximizes an objective function, e.g., test accuracy or computation cost. However, measuring the model accuracy is prohibitively expensive, because it can be done only after the network is generated and trained. Thus, the key challenge for efficient NAS is to precisely predict the accuracy of a given architecture configuration, and there has been active research to minimize the cost of NAS algorithm [8, 14, 19, 35, 42, 55, 57, 58, 64]. Recently, Neural Predictor-based NAS [8, 19, 55] shows promising results with high sampling efficiency with minimal architecture evaluations. Neural Predictor trains a performance regression network as a cost model to predict the target accuracy for a given model architecture. The predictor training is expensive but this is a one-time cost, and a tiny fraction of random samples over the input space can produce a representative model [55]. The final NAS result is obtained by performing architecture search guided by the prediction of the trained predictor model. The approach reduces the exploration overhead without any degradation of the resulting model accuracy. In the rest of the paper, we present how we developed the idea of Neural Predictor to drastically reduce the overhead of the auto-tuning process.

3 Motivations and Challenges

To establish the feasibility of a neural predictor for performance prediction, we conducted preliminary experiments. Figure 2 shows that a regression model trained with a small set of data samples can predict the outcomes of codes optimized with unseen knob values. However, per-task predictor models are not realistic, because numerous tasks are dynamically generated with different tensor operation types and input/output shapes. We identify three key challenges that guide our cost model design in Section 4.

1. Current input data and cost models are not designed to learn (task, knob, performance) triplets. While prior work [3, 13, 36, 61] had to train and adapt a cost model to learn (knob, performance) correlation only in a given search space of a compiled kernel, a multi-task cost model should be able to consider multiple search spaces as well, i.e., to predict how the same optimization will behave for kernel

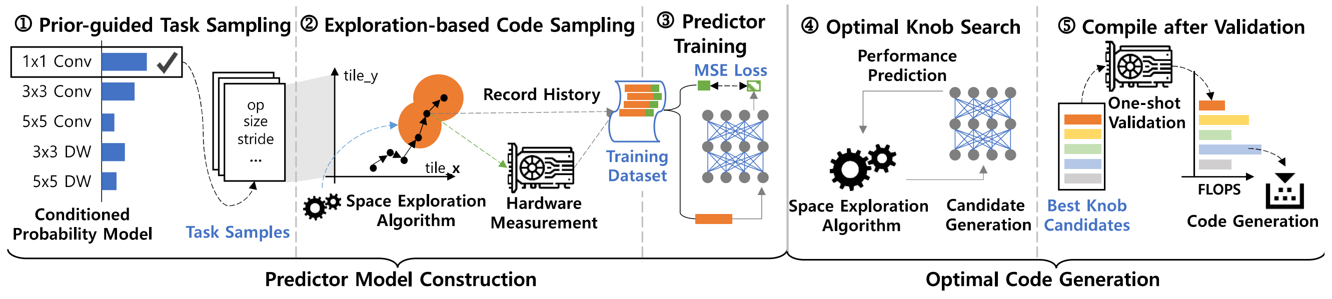


Figure 5. The Overview of One-Shot Tuner

instances with different performance distributions. However, template-based code representation in TVM is designed with an assumption of a single search space; the representation does not include task features such as input/output shapes or filter size, and only implicitly embeds them as loop iteration counts. This makes it very challenging for a single predictor model to accurately encode performance distributions of multiple tasks and generalize across tasks.

2. The task sampling method dictates the generality of the cost model. The search space of *optimized codes* is large but bounded, and knob values are constrained by allowed value ranges or types of external parameters, but the search space of plausible *tasks* is technically unbounded. If we sample tasks directly from a small set of popular models, the resulting predictor model will not work for unseen tasks that are still plausible but have shapes and sizes not in the range of sampled tasks. If we randomly sample tasks from a more general and sparse set of models such as [25] or synthesized tasks, we will need a prohibitively large number of samples for the cost model to see enough samples for tasks from known models (e.g., red stars for tasks in *ResNet-18* and *VGG-16* in Figure 3) so that it can make learned predictions. This calls for a task sampling method that balances the trade-off between generality and sampling efficiency.

3. Random sampling of hardware measurements can result in skewed performance distributions. We measured the fraction of zero or close-to-zero hardware measurements for AutoTVM as shown in Figure 4. 5.6% of the total samples have zero FLOPS due to compilation or runtime errors (e.g., out of memory), and about 44.9% of samples have very low, close to zero FLOPS (<1% of the maximum FLOPS measured). Some prior work also observed this phenomenon of many invalid measurements [3, 36, 59]. [3, 36] reported "numerous invalid configurations" and "lots of zero points (invalid knobs) in the search space", and [59] reported that 27.8% of samples are invalid. While some unstable (zero FLOPS) results due to invalid knob values can be filtered out to an extent by search algorithms [61], a non-uniform search space can arise when a tensor kernel with a certain type or data sizes does not respond well to optimizations. This phenomenon does not severely affect the accuracy of the online cost model, because low-performing samples are not used for cost model adaption, although they can slow down the search. However, if a predictor model is pre-trained only using data samples that have such distributions as in our proposal, it cannot properly model and predict high-performing codes.

4 Design and Implementation

In this section, we present the design and implementation of One-Shot Tuner focusing on how we address the challenges identified in Section 3. As a pre-trained predictor model, One-Shot Tuner goes through a one-time model construction phase prior to compilation. This phase reduces the traditional auto-tuning phase with thousands of iterations during compilation into a single-search and single-validation tuning phase. The following subsections provide a detailed description of each phase, **the predictor model construction phase** and **the optimal code generation phase**.

4.1 Predictor Model Construction

The accuracy of the predictor model heavily depends on the quality of the training dataset. However, our preliminary analysis in Section 3 reveals that the combined search space for multiple tasks is not only huge and sparse, but that each search space has a distinct and often highly skewed distribution. This makes the straightforward random sampling method unfeasible in our case as it will have a very low sampling efficiency. With random sampling, we would need an unrealistically large number of data samples to achieve reliable prediction accuracy.

Figure 5 shows the execution steps of the model construction phase that feature task and code sampling methods to maximize sampling efficiency. For the dataset to include a sufficient number of representative samples from diverse tasks, we guide both ① task sampling (determined by tensor operation type and input/output shapes) and ② code sampling (determined by knob values) methods to exploit prior model knowledge and focus on promising samples. Sampled codes are executed on the target device to measure inference time in FLOPS. The training data set generated with

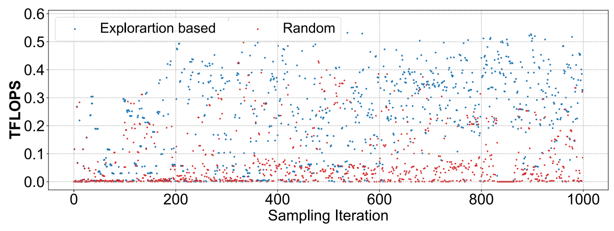


Figure 6. Scatter plot of sampling data as the sampling progresses.

input features for the sampled codes and their performance in FLOPS as labels is then used to ③ pre-train the predictor model for the optimal code generation phase.

4.1.1 Prior-Guided Task Sampling (PGS).

Task sampling navigates an unbounded number of performance distributions determined by the operation type, input/output shapes and sizes, and filter size (if it exists) of a kernel. To give a bias to the randomness towards known models and guarantee the generality of sampled tasks at the same time, we devise a task sampling method that exploits prior knowledge. First, we crawl popular models from public repositories such as PyTorch TorchVision [38] and MXNet model zoo [25], then build a probability model that is conditioned on the task distribution of the collected models which is proportional to the frequency of occurrences within the given condition. This PGS method randomly selects training tasks according to the conditioned probability, where the overall probability is estimated by the chain of the conditional probabilities. For instance, we first select a knob for kernel size, where the probability of selecting the kernel size K_i is proportional to the number of occurrences of K_i over entire data. Then, we choose the stride S_j based on the probability, proportional to the number of collected tasks that have both K_i and S_j over the number of samples with K_i. In Algorithm 1, the innermost loop from line 7 to 10 implements the probability chaining by storing sampled knobs in *sampled* and use it to calculate P_{knob} in the next iteration. By repeating this process likewise until the entire knobs are selected, we create a sampled task space that roughly follows the distribution of tasks in known models. PGS allows a predictor to learn about common and probable kernels rather than corner cases so that the prediction accuracy can be accurate for practical candidates.

4.1.2 Exploration-Based Code Sampling (EBS).

Once a task is sampled, we need to generate code samples with different knob values and execute them to model its unique performance distribution with One-Shot Tuner. However, as observed in Section 3, the performance distributions are often very skewed, with many useless or poor-performing codes. Therefore, to focus on sampling and learning about more promising codes, we propose gathering samples *on the trajectory of a space exploration algorithm.* As shown in

Algorithm 1 Dataset Generation

```
Input: Probability Model $M$, Task set size $K$, Max trials $N$,
Space Exploration Algorithm $S$, Cost model $C$
1:  function PRIOR_GUIDED_TASK_SAMPLING($M$,$K$)
2:      $knobs = \{kernel\_size : k, stride : s, ...\}$ ▷ Initialize $knobs$ for tasks
3:      $tasks$ = []
4:      for $i$ in $K$ do
5:          $sampled$ = []
6:          for $knob\_key$ in $knobs.key$ do
7:              for $knob$ in $knobs[knob\_key].items$ do
8:                  $P_{knob} = \frac{M[sampled \cup [knob]].count()}{M[sampled].count()}$
9:              $sampled.append(sample\_knob(P_{knob}))$
10:                 ▷ Sample task knob values with probability $P$
11:         $T$ = create_task($sampled$)
12:         $tasks$.append($T$)
        return $tasks$
13: function EXPLORATION_BASED_CODE_SAMPLING($tasks$,$N$,$S$,$C$)
14:     $Datasets$ = []
15:     for $task$ in $tasks$ do
16:         for $i$ in $N$ do
17:             $Candidates$ = $S$.explore($C$, $task_{knobs}$)
18:                 ▷ Store search results as $Candidates$
19:             for $code$ in $Candidates$ do
20:                 $History$ = $code$.measure_on_hardware()
21:                 $Datasets$.append(History)
22:                     ▷ Collect (code,FLOPS) pairs
23:             $C$.update($Datasets$)      ▷ Update the cost model $C$
        return $Datasets$
```

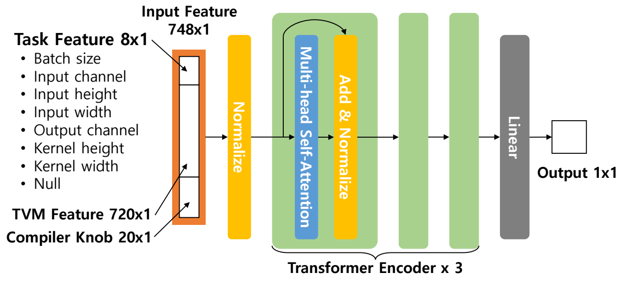


Figure 7. Predictor model architecture

Algorithm 1, once the target tasks are determined by the PGS, we perform a discrete space search from a random starting point for each task. The search algorithm visits many local-optimal points during the exploration process. While these intermediate search results are originally for guiding the search with cost model updates, we create a data set from them.

The intuition behind the exploration-based code sampling is that the promising areas in the search space will be visited increasingly frequently as the search continues, which leads to higher sampling efficiency. The samples obtained by the EBS in Figure 6 appear more in the upper part of the search space with higher FLOPS than randomly sampled data. This difference enables the predictor model trained with these samples to predict performance precisely for high-performing candidates. We also expect the samples on the trajectory are likely to be those that are explored during the knob search in compilation time. As a result, the EBS accelerates the convergence of the knob search, but more importantly, it helps to improve the compiled model performance (details in Section 6.3).

Along with these sampling methods, we also use a data oversampling technique to further increase the quality of data samples by sampling infrequent but faster samples more often based on the inversed frequency.

4.1.3 Feature Generation.

From the sampled codes in the previous steps, we generate training input data for the predictor model. We rely on the template-based code generation scheme in TVM, so we reuse TVM features for template code representation. Unique knob values of each sample determine how a code template is lowered to the loop-level AST, from which TVM generates curve features that represent loop characteristics [13]. Then we append raw knob values and task-identifying information, e.g., input sizes and shapes, batch size, the number of channels, to the TVM curve features as shown in Figure 7. The loop-level AST only implicitly includes these values, e.g., as a node for loop iteration count. We found that with such features the model cannot properly learn correlations between samples and the search space they belong to. Explicit task-specific features at the same level as TVM features help the predictor model to differentiate distinct search spaces, as supported by the ablation study result in Section 6.3.

4.1.4 Predictor Model Architecture.

We conducted an extensive sensitivity study with DL model architectures and hyperparameters to find a model that can learn (task, knob, performance) correlations with high accuracy (details in Section 6.3). Multi-head self-attention mechanism [54] is effective in understanding multi-dimensional correlations in structured data, and among the tested networks provided the best performance with stable convergence. Thus, we design One-Shot Tuner with a model architecture as shown in Figure 7: three stacks of Transformer encoders followed by a linear layer, with 1,024 hidden feature dimensions and four self-attention heads. For the loss function, One-Shot Tuner using mean square error (MSE) produced models with faster inference time than versions that used L1 and ranking [10], so the predictor is trained using MSE loss between real and estimated performance.

4.2 Optimal Code Generation

The optimal code generation phase in Figure 5 replaces the existing iterative auto-tuning framework in TVM. By design, we keep this phase straightforward compared to the model construction phase and reuse many existing TVM components.

During compilation, the auto-tuning back-end first performs a single iteration of space exploration guided by the pre-trained One-Shot Tuner predictor model ④. For the knob search, we use black-box optimizations as implemented in TVM, e.g., simulated annealing and genetic algorithm, but without cost model update. Then, the compiler performs ⑤ validation for the top-K candidates found during the search, in a sense that predicted performance for them is "validated" against actual measurements on hardware. The validation result is used to determine the best-performing candidate. For subsequent optimization and code generation with the identified knob values, we rely on the TVM back-end.

Unlike typical auto-tuning solutions, neither knob search nor hardware measurement is repeated, so this phase completes in tens of seconds per task, significantly reducing the end-to-end compilation time (details in Section 6.1).

Table 2. Hyper-parameters used in **One-Shot Tuner** during model training (values used for results in bold) and code generation.

Model Training	
Hyperparameter	**Values considered**
Batch size	{1, 16, 64, 256, **512**, 1024}
Epoch #	{100,**200**}
Learning rate	{1e-2,1e-3,5e-4,**1e-4**,5e-5,1e-5}
Learning rate decay	{1e-2,1e-3,5e-4,1e-4,**5e-5**,1e-5}
Dropout rate	{0,**0.25**,0.5,0.75}
Attention head #	{**4**,12,16}
Layer #	{1,**3**,5,7}
Hidden state #	{128,256,512,**1024**,2048}
Code Generation	
Hyperparameter	**Value**
Hardware measurements for validation	64
Maximum steps (simulated annealing)	160
Batch size (simulated annealing)	512
Early stop steps (simulated annealing)	50
Batch size (genetic algorithm)	512
Maximum steps (genetic algorithm)	120
Mutation probability (genetic algorithm)	0.1
Elite # (genetic algorithm)	96

5 Methodology

We implemented One-Shot Tuner cost models using a built-in Transformer (`torch.nn.TransformerEncoder`) in PyTorch 1.8.1 [41]. After hyperparameter tuning, we used a four-head Transformer with three feed-forward layers of width 1024. We used PyTorch `WeightedRandomSampler` to implement oversampling in Section 4.1.2. We assume that the cost models are distinct for each tensor operation (e.g., `conv2d`) but are shared by code templates for the operation (e.g., `naïve`, `deformable`, `bitserial`, `group`, `depthwise`, `transpose`, `Winograd`, `tensorcore`). Thus, One-Shot Tuner technically requires a total of five cost models to complete all TVM auto-tuning jobs. We built and evaluated two predictor models for `conv2d` and `matmul`, because our PGS guides us to sample tasks of "actual templates chosen by Relay (TVM front-end) [46] for auto-tuning" for models in model zoos. FC layers are often skipped as an auto-tuning target since TVM finds the search space too small, whereas conv1d and conv3d did not appear in any of the models in the zoos.

For testing, we modified TVM version 0.8dev [12] to run a single iteration of template-based auto-tuning using a pre-trained One-Shot Tuner in place of the original gradient-boosted tree (XGB) model. For search algorithms, we experimented with simulated annealing and genetic algorithm as implemented in TVM. Detailed hyperparameters for model architecture, training process, and search algorithms can be found in table 2. We sampled data and compiled models on NVIDIA 2080 Ti GPU's for GPU results and AWS Intel Xeon CPU's for CPU results.

Task and data sampling methods. For PGS, tasks were sampled from TorchVision [38] and MXNet [11] model zoos. Synthetic tasks were then generated using the layer frequency information extracted from the tasks in the model zoos, as described in Section 4.1.1. We used 800 tasks for model training. We used a strictly disjoint set of models to to guide generation of the training data set from evaluated models in the test data set. However, as the PGS is designed to statistically sample probable tasks more frequently, so the resulting training and test data sets inadvertently have 9-12% common layers (tasks). We did not post-process them to remove overlapped tasks, which will distort the actual layer distribution.

For EBS, we used TVM built-in search algorithm (simulated annealing) and cost model (XGB cost model) with default configurations. We collected 1,000 samples per task and excluded samples more than 20x slower than the highest-performing sample.

Evaluated models. We tested seven convolutional neural networks (*AlexNet* [32], *VGG-16* [50], *SqueezeNet-v1.0* [28], *ResNet-18* [26], *DenseNet-121* [27], *MobileNet-v2* [49] and *EfficientNet-B0* [53]) and an NLP model (*BERT-base* [20]). We used model implementation provided by MXNet (*EfficientNet, MobileNetV2*) and TVM (the rest) with no modifications.

Evaluated auto-tuning solutions. We compared One-Shot Tuner with several prior auto-tuning solutions in terms of compilation time and model inference time. We evaluated three AutoTVM-based prior solutions (*autotvm* [13] (baseline), *chameleon* [3] and *adatune* [36]), two variants of One-Shot Tuner (*one-shot-sa* with simulated annealing and *one-shot-ga* with genetic algorithm for candidate search), and *ansor*, which is a template-free auto-scheduler in TVM [61]. To evaluate previous work, we used authors' implementation with no modifications.

6 Evaluation

For the evaluation of One-Shot Tuner, we structured our experiments in two dimensions. First, we focused on showing how One-Shot Tuner, as a pre-trained predictor model, can improve an existing auto-tuning solution in terms of compilation and inference performance. For fair and isolated comparison, we made sure evaluated solutions for this part are all based on AutoTVM [13]. Second, we compared two end-to-end auto-tuning solutions for TVM, an AutoTVM-based auto-tuner augmented with One-Shot Tuner and the template-free auto-scheduler which is a more recent addition in TVM based on [61]. These methods significantly differ in the overall structure, search space definition, and exploration mechanisms, so this evaluation focuses rather on exploring the potential that template-based One-Shot Tuner can advance the current state-of-the-art.

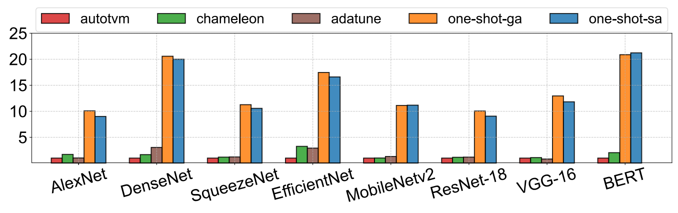
We also provide detailed ablation analysis for different task and sampling methods, input features, and search algorithm parameters for One-Shot Tuner.

6.1 End-to-End Compilation Time

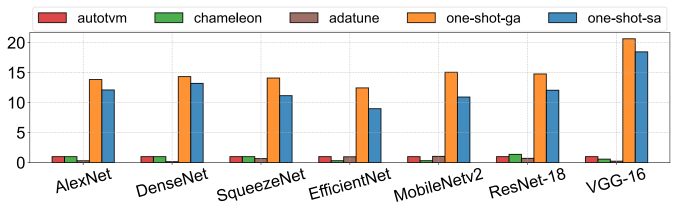
As shown in Figure 8, One-Shot Tuner dominates all prior work on both CPU and GPU platforms in terms of the compilation speed by practically eliminating the repeated search and measurement overheads during auto-tuning. As a result, One-Shot Tuner reduces the end-to-end compilation time dramatically by 13.84x on average against the baseline (*autotvm*), and also against more recent work (8.7x against *chameleon* and 10.55x against *adatune*) on GPUs. For example with *DenseNet*, compared to *chameleon* and *adatune* that compile faster than than *autotvm* by 3x mainly by improving the search algorithm, the speedup by One-Shot Tuner is up to 20x. Even for the models for which *chameleon* or *adatune* performs comparable to or worse than the baseline (*VGG-16* and *BERT*), One-Shot Tuner consistently provides an order of magnitude reduction in compilation time. In terms of wall-clock time, One-Shot Tuner finished compilation under two hours for all the models (under an hour for four models) for which prior work needed several hours to a whole day (Table 3).

Figure 10 shows that One-Shot Tuner compiles *ResNet-18* by orders of magnitude faster than the baseline. [1] The auto-tuning process consists of three steps, (1) candidate search and selection, (2) hardware measurements of candidates, and (3) cost model update. AutoTVM performs numerous iterations of the above steps to fine-tune the cost model and to improve the search results. By default, it goes through 16 iterations to perform 1,000 hardware measurements as a batch of 64 candidates. In contrast, One-Shot Tuner performs only one iteration of steps (1) and (2) (no green sub-bars in the figure). The auto-tuning time in the baseline varies from layer to layer because the auto-tuning process can stop early once a search space is exhausted or the result converges. This explains why many layers in the figure show much less than 16x reduction for One-Shot Tuner. Without early stopping,

[1]Non auto-tuning compilation overheads (front-end parsing, back-end code generation, etc.) are excluded from the graphs as they comprise less than 1% of the total compilation time.

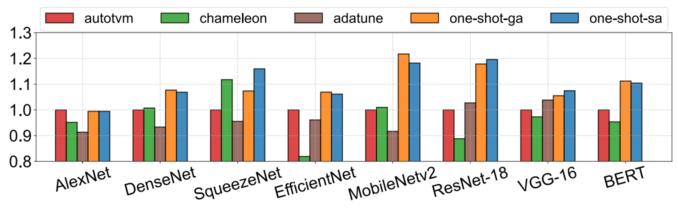


(a) Nvidia GPU

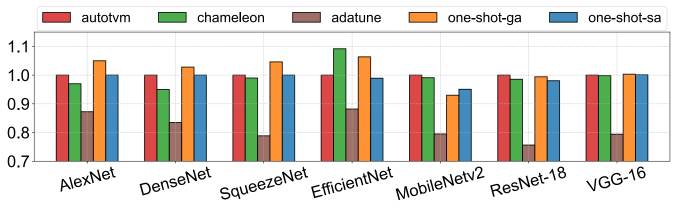


(b) Intel CPU

Figure 8. End-to-end compilation time (normalized to AutoTVM)



(a) Nvidia GPU



(b) Intel CPU

Figure 9. Model inference time (normalized to AutoTVM)

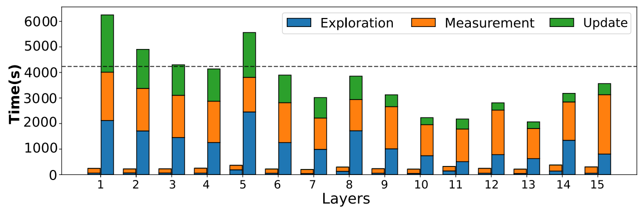


Figure 10. Layer-wise breakdown of compilation time for *ResNet-18* on GPU, compiled by One-Shot Tuner (left bar in each pair) and AutoTVM (right bar in each pair).

Table 3. End-to-end compilation time (hours) on GPU.

	autotvm	chameleon	adatune	one-shot-ga	one-shot-sa
DenseNet	24.01	14.40	7.86	**1.16**	**1.20**
ResNet-18	9.73	8.43	8.20	**0.97**	**1.07**
BERT	4.36	2.13	13.89	**0.21**	**0.21**

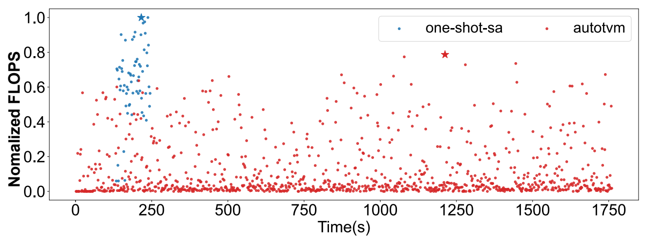


Figure 11. The time-series visualization of the inference time measured on GPU by One-Shot Tuner and AutoTVM (their best samples as stars).

One-Shot Tuner achieves higher than 16x (e.g., 25.2x for layer 1) by getting rid of the cost model update overhead.

6.2 Inference Time

As shown in Figure 9, One-Shot Tuner outperforms all the existing template-based auto-tuning solutions by a significant margin (8.9% on average, up to 21%) for every model in terms of optimized model performance on GPU. The result implies that the pre-trained One-Shot Tuner model can guide the search more effectively towards optimal codes than the fine-tuned cost model after numerous iterations. On CPU, One-Shot Tuner provides comparable model performance to prior work (2% better on average). The knob search spaces are usually only 1/1.1K to 1/97K as large on CPUs as on GPUs; thus the cost model does not make as much difference.

Figure 11 isolates the effectiveness of the cost model in predicting candidate performance and locating high-performing candidates. *autotvm* repeatedly performs hardware measurements and cost model updates during the auto-tuning process (red dots in the figure), but most of the search points, even after many iterations, have low FLOPS. This result shows that most of the sampled data does not make real contributions to performance improvement. In contrast, One-Shot Tuner performs a one-shot validation on target hardware after a single iteration of search, and the entire process finishes in 250s. The top-K candidates found by the search (blue dots in the figure) are high-quality predictions that outperform most of the samples identified by the baseline. We observed that the predictor model functions as a very precise filter that narrows down the search space, but that validation is also essential to compensate for predictor errors and inefficiencies in search algorithms.

Layer-wise decomposition of the model inference time for *MobileNet-v2* (Figure 12) illustrates that the performance gain by One-Shot Tuner over prior work is considerably high in the early or intermediate layers of the network, but relatively

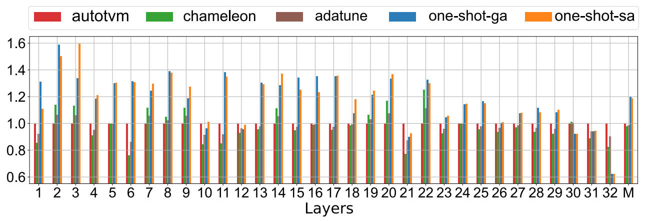


Figure 12. Layer-wise breakdown of model performance for *MobileNet-v2* on GPU.

low for the layers close to the output layer. The later layers are 1x1 convolution layers with many channels; locating the best optimization option for these layers is particularly challenging, because the search space quickly becomes huge as the channel size increases. Overall, One-Shot Tuner consistently outperforms prior work except for a few layers.

6.3 Ablation Analysis

6.3.1 Design Components. The impact of each design component in the paper, i.e., task features (TF), prior-guided task sampling (PGS), and exploration-based data sampling (EBS), on model performance was evaluated by eliminating one at a time from One-Shot Tuner. Figure 13a shows that no one or two combinations of the design components dominate, and their performance contributions vary across models. For *ResNet-18* and *MobileNetv2*, the effect of EBS is prominent. On the other hand, the combination of task features and PGS by itself achieves 90% of the total speedup for *EfficientNet*. *EfficientNet* has NAS-generated non-typical tasks, and requires effective sampling of a broad range of kernels.

6.3.2 Sampling Methods. To examine the effectiveness of PGS and EBS in more detail, we experimented with combinations of different sampling methods as shown in Table 4.

EBS outperforms random sampling for all evaluated models in all combinations, regardless of the task sampling method. For example, with PGS, EBS produces models 22x faster on average (up to 77.7x) than random sampling. We observed that models such as *EfficientNet* and *MobileNetV2* are affected more than the others when they include layers with distinctively larger sample spaces. EBS is more sensitive to the type of task sampling methods than random sampling, showing that the potential for EBS can be maximized when sampled tasks have more probable distributions.

To evaluate PGS, we created two variants of One-Shot Tuner trained with randomly sampled tasks (`random`) and tasks directly sampled from models in the Model Zoo (`model zoo`) respectively. Table 4 shows that One-Shot Tuner (PGS+EBS) outperforms `random`+EBS and `modelzoo`+EBS for four models except for *ResNet-18*. Tasks in *ResNet-18* happen to appear very often in other models resulting in 93% of its tasks in the training set. PGS addresses the danger of overfitting the cost model in `modelzoo` by gently guiding random sampling with the distributions of known tasks. Task sampling methods with random data sampling are consistently slower by up to 77.7x than the counterpart with EBS.

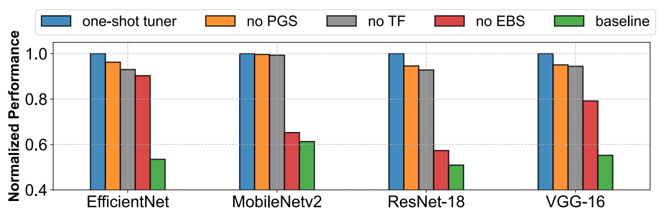
6.3.3 Task Size. We evaluated variant versions of One-Shot Tuner to see how sensitive the predictor model accuracy is to the number of sampled tasks. Figure 13b shows that One-Shot Tuner consistently produces better-performing codes as the number of training task samples increases. The performance improvement saturates around 600 task samples for *ResNet-18* and *VGG-16*, by when prior-guided sampling have collected enough data for common tasks for performance modeling. Models such as *EfficientNet* and *MobileNetv2* that have more diverse and unconventional tasks tend to require more task samples than other models.

6.3.4 Model Architecture. We experimented with different ML and DL model architectures for the pre-trained predictor model. We trained an XGB model (baseline), an MLP-based and an LSTM-based cost model with the same training samples, and compiled *ResNet-18, VGG-16, EfficientNet*, and *MobileNetv2* (One-Shot Tuner, LSTM, and MLP models are similar in model size). One-Shot Tuner outperforms XGB by 10% on average for all evaluated models in optimized inference time. The LSTM model showed comparable performance to One-Shot Tuner except for *EfficientNet* for which it is 2.1x worse than One-Shot Tuner, while the MLP model showed very poor performance with *EfficientNet* and *MobileNetv2* (more than 30x slower inference time). Both *EfficientNet* and *MobileNetv2* include layers of more diverse and unusual shapes and filter sizes than the others, which should be challenging for a simple network without explicit structural correlation to generalize for unseen tasks (and with fewer relevant samples from prior-guided sampling).

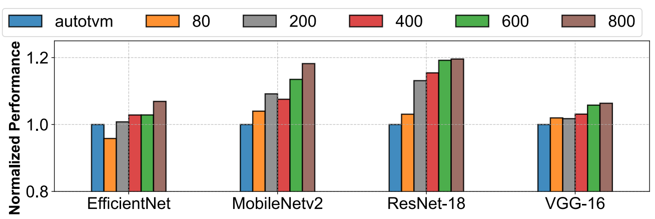
6.4 Comparison with TVM Auto-scheduler

One-Shot Tuner reduced compilation time by 10.64x compared to TVM auto-scheduler for the same reason as it is faster than other auto-tuning solutions (Table 5). Auto-scheduler updates its cost model and profiles candidates for 20,000 iterations for an entire model by default, whereas One-Shot Tuner does not repeat exploration and measurements.

Table 5 shows inference times of CNN models compiled by One-Shot Tuner and Ansor (TVM auto-scheduler) [61]. With a single batch, One-Shot Tuner provides comparable performance to Ansor for *DenseNet* and *ResNet-18* but it is outperformed by 85.7% and 80.3% for *MobileNetV2* and *SqueezeNet*. In contrast, with a batch size of eight, One-Shot Tuner produces 40.2% faster models on average than Ansor for all evaluated models. Different approaches to candidate code generation, i.e., template-based or template-free, explains the result. One-Shot Tuner relies on AutoTVM code templates for high-level code structures, while template-free



(a) Design components



(b) Task size

Figure 13. Ablation analysis on the performance impact by sampled task sizes and proposed methods (TF=task-specific features, EBS=exploration-based sampling, and PGS=prior-guided sampling).

Table 4. Model inference time (ms) on GPU with different combinations of task sampling and data sampling methods.

	AlexNet	SqueezeNet	EfficientNet	MobileNetV2	ResNet-18	VGG-16
Random + Random	1.27	7.97	42.63	47.38	3.58	15.79
Random + EBS (no PGS)	0.99	1.08	10.35	10.49	1.59	5.10
ModelZoo + Random	1.03	6.42	34.85	33.31	2.01	15.55
ModelZoo + EBS	0.79	4.35	7.72	5.82	**0.80**	4.28
PGS + Random (no EBS)	1.03	1.10	41.47	52.06	3.09	10.87
PGS + EBS	**0.79**	**0.69**	**1.27**	**0.67**	0.93	**3.78**

Table 5. Model inference time (ms) and end-to-end compilation time (h) on GPU, optimized by One-Shot Tuner and Ansor in NHWC format with batch size 1 and 8.

Model inference time

Batch Size	Method	DenseNet	SqueezeNet	MobileNetV2	ResNet-18
1	Ansor	2.45	0.51	0.7	1.3
	One-Shot	2.49	0.92	1.3	1.36
8	Ansor	5.45	1.89	2.6	5.24
	One-Shot	4.66	1.54	1.73	2.89

End-to-end compilation time

Batch Size	Method	DenseNet	SqueezeNet	MobileNetV2	ResNet-18
1	Ansor	8.02	9.93	9.73	9.35
	One-Shot	0.93	0.86	0.96	0.73
8	Ansor	8.40	9.93	10.28	9.70
	One-Shot	0.96	0.93	1.16	0.81

auto-scheduler can more freely transform codes with rewriting rules and possibly generate better candidates. The single-batch result is consistent with results in [61], and thus prove the above point. However, [61] cannot easily opt to use specialized algorithms (e.g., winograd convolution or kernels that use NVIDIA Tensor Core) in its code-generation scheme, while AutoTVM can do so with pre-defined templates for them. With a batch size of eight, One-Shot Tuner can generate optimized codes based on a `conv2D` template for batch execution on Tensor Core. As future work, we will integrate One-Shot Tuner with TVM auto-scheduler so that the resulting solution can reduce the compilation time by pre-training One-Shot Tuner with template-free candidates, and also improve the model performance as well by partially adopting template-based solutions.

7 Discussion

Data sampling and model training cost. It takes about 30 minutes to collect 1,000 data samples for a task using EBS on the GPU. Sampling the 800-task data set we used for evaluation takes roughly 24 hours on our 16-GPU server (400 GPU hours), and training the cost model with the data takes about one hour. These sum up to a one-time fixed cost prior to compilation, which will be easily compensated with less than 23 compilations.

Moving a large part of recurring compilation costs to a one-time pre-compilation cost can be translated into energy savings and carbon footprint reduction. As a preliminary analysis, we estimated how much CO_2 emission could be reduced by using One-Shot Tuner compared to AutoTVM, based on methods in [33]. One-Shot Tuner generates 30kg CO_2 for pre-compilation data sampling and model training and 0.07kg CO_2 for each compilation, whereas AutoTVM generates 1kg CO_2 for each compilation. Thus, after 32 compilation instances, One-Shot Tuner outperforms AutoTVM in terms of total energy impact. In addition, the reduction in model execution time by One-Shot Tuner against prior work saves 0.58kg CO_2 per 100 GPU hours.

Generalization across target hardware platforms. One-Shot Tuner currently does not consider hardware platforms as a variable parameter, i.e., assumes that a separate model is trained and deployed for each hardware. We expect that adding hardware-extracted features will enable One-Shot Tuner to understand both hardware and code characteristics in correlation with performance distributions. As future work, we will explore how we can extend One-Shot Tuner to generalize across platforms.

Integration with existing auto-tuning frameworks. One-Shot Tuner is implemented as a plug-in cost model for the TVM auto-tuning framework, and therefore can be seamlessly integrated into any TVM-based existing auto-tuning solutions. As a learned cost model with high accuracy, it will help reduce the time taken by the search algorithm to locate high-performing codes, and will improve both optimization time and model performance of prior work that reuses the default XGBoost-based cost model in TVM.

8 Related Work

Black-box auto-tuning. Search-driven optimizations have shown the effectiveness in optimizing complex systems [5, 17, 40]. [23, 56] exploit empirical optimizations for generating math library routines such as BLAS and FFT. [4] proposes an auto-tuning framework with a multi-arm bandit meta-technique to select a search mechanism. [9, 16] use auto-tuning to solve the problems of I/O parameter and work-group size selection. More recently, [18] provides a compiler infrastructure in LLVM for testing and evaluating different auto-tuning mechanisms.

Automatic optimization for tensor operations. DL frameworks have increasingly adopted auto-tuning to improve compute-intensive tensor operations in DL models. AutoTVM [13] searches optimal schedules for tensor operations guided by a cost model adapted online during compilation, while [3, 36] reduce the search and measurement overheads by removing unnecessary measurements from [13]. [60] improves the convergence speed for individual kernel auto-tuning by using multi-armed bandit to dynamically learn the cost model for the entire network. FlexTensor [62] and Ansor [61] automatically create schedules to reduce the overheads of defining code templates and find better schedules. [62] auto-tunes platform-independent schedules for tensor operations, and [61] takes a hierarchical approach to high-level code structure generation and parameter auto-tuning. One-Shot Tuner proposes a pre-trained cost model that can be plugged into prior work, while prior work focuses mainly on improving search mechanisms and reducing measurement overheads.

Learning-based cost model. [39] trains an LSTM-based cost model with program control flow graphs, then uses the model to predict basic block throughput in LLVM optimization passes. [6] proposes an LSTM-based cost model that encodes hardware characteristics and loop structures to predict optimal code transformations for Tiramisu [7] compiler. [31] utilizes a graph neural network-based cost model for tile-size selection and operator fusion passes to predict optimized kernel performance. [51] proposes a bidirectional LSTM-based model to predict the performance of loop-based tensor optimizations on a CPU. The model is trained using model-dependent epsilon-greedy beam search with value iterations, whereas One-Shot Tuner focuses on training the cost model with a dataset sampled via model-independent black-box optimization.

9 Conclusion

This paper introduces a NAS-inspired approach to the black-box optimization pass in DL compilers. The approach can drastically reduce the expensive auto-tuning overhead to a one-shot validation with a pre-trained predictor model. Building a general yet powerful cost model that understands distinct instances of huge and sparse search spaces faces challenges in many aspects, but we showed that our proposed data sampling methods and improved input representation can effectively make the search space tractable. The results show that One-Shot Tuner generates comparable or faster models than the state-of-the-art, while significantly reducing the optimization time (up to 67.6x) for a variety of DNN models. We plan to extend One-Shot Tuner to a cross-platform solution that enables more portable auto-tuning and integrating it with template-free auto-tuning solutions to exploit flexible code generation.

Acknowledgments

We thank Yongwon Shin for constructive comments on the manuscript. We also thank the anonymous reviewers for their valuable feedback. This work was supported by Institute for Information & communications Technology Promotion (IITP) grants funded by the Korea government (MSIP) (No. 2019-0-01906, 2021-0-00871, and 2021-0-00310).

Data Availability Statement

The data that support the findings of this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.6337971.

A Artifact Appendix

A.1 Abstract

Our artifact includes an implementation of the One-Shot Tuner predictor model and a variant of the TVM compiler modified to use One-Shot Tuner. We provide a fully trained One-Shot Tuner predictor model, along with model source codes, training data samples obtained using PGS and EBS methods, and scripts to use the data to re-train the model. For the compiler, we provide binaries and source codes for the TVM compiler modified to use the trained One-Shot Tuner predictor model for a single iteration of search and validation in place of its AutoTVM-based auto-tuning process. This will allow evaluation and reproduction of our results on the NVIDIA GPU and Intel CPU systems described in the paper.

We also provide customizable scripts for generating datasets using PGS and EBS methods. The scripts can use different sets of models for prior knowledge extraction and different exploration algorithms for EBS and knob search.

A.2 Description

A.2.1 Artifact Check-List (Meta-information).

- **Algorithm:** Methods for (1) collecting data samples for the One-Shot Tuner predictor model, (2) training the predictor model, and (3) performing one-shot validation during auto-tuning.
- **Program (Model):** *AlexNet, VGG, ResNet, SqueezeNet, DenseNet* as provided by TVM, *EfficientNet* and *MobileNetV2* as provided by *gluoncv2* for MXNET/Gluon (https://pypi.org/project/gluoncv2/), and *Hugging Face transformers* for *BERT-base* (https://github.com/huggingface/transformers).
- **Compilation:** Modified Apache TVM compiler based on version 0.8.dev0 (commit: 772fa6b). Binaries and sources provided.
- **Binary:** Included for Linux (Ubuntu 18.04) with CUDA 10.2 runtime for X86-64. Source code and scripts included to regenerate binaries.
- **Runtime environment:** Provided binaries for Linux (Ubuntu 18.04) X86-64, but source code given.
- **Data set:** Training data samples are included. They can be regenerated by following the procedure in A.4, but due to the probabilistic nature of the sampling process, the distribution of the resulting samples will vary.
- **Hardware:** We recommend systems with NVIDIA GeForce RTX 2080 Ti GPU and Intel Xeon CPU E5-2666 v3 CPU (AWS c4 4x large instances) for verifying GPU and CPU results respectively. Similar systems should give comparable results but we do not claim such cross-platform performance portability.
- **Metrics:** Model inference time (inference queries are randomly generated) and end-to-end compilation time measured by TVM built-in timing interfaces.
- **Output:** Performance results are provided as console output and also stored in a file specified by an evaluation script.
- **Experiments:** The experiment flow is described in detail in Section A.4 and README at our archive locations.
- **How much disk space is required (approximately)?:** 1GB including training data samples
- **How much time is needed to prepare workflow (approximately)?:** Reproducing data samples for 800 tasks used in the paper will take roughly 600 GPU hours on NVIDIA GeForce RTX 2080 Ti GPU. Pre-training the One-Shot Tuner predictor model takes about one hour.
- **How much time is needed to complete experiments (approximately)?:** Up to 18 hours for compiling all eight models using One-Shot Tuner, and four days using the baseline (AutoTVM).
- **Publicly available?:** Yes.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** Yes (https://doi.org/10.5281/zenodo.6337971)

A.2.2 How Delivered.
Our predictor model (pre-trained binary and sources), training data samples, scripts, and a `git` patch file for the TVM modifications are available at the following link: https://zenodo.org/record/6337971. We also maintain a GitHub repository (https://github.com/ryujaehun/one-shot-tuner).

A.2.3 Hardware Dependencies.
We recommend systems with NVIDIA GeForce RTX 2080 Ti GPU and Intel Xeon CPU E5-2666 v3 CPU (AWS c4 4x large instances) for verifying GPU and CPU results respectively. Similar systems should give comparable results but we do not claim such cross-platform performance portability.

A.2.4 Software Dependencies.
Our code is implemented and tested on Ubuntu 18.04 x86-64 system, with CUDA 10.2 and cuDNN 7. Additional software dependencies include minimal pre-requisites on Ubuntu for TVM and deep learning frameworks (PyTorch v1.6.0) for model implementations. We recommend using a docker image, "jaehun/ost:v2", with all software dependencies and pre-requisites included.

A.3 Installation

A.3.1 Docker Installation.
docker and *nvidia-docker* packages are required to use the One Shot Tuner docker image. The following command will run the image as an isolated container.

```
docker run -it --gpus 1 --rm  jaehun/ost:v2 bash
```

A.3.2 Local Installation (Zenodo).
After installing TVM and its python dependencies (https://tvm.apache.org/docs/install/from_source.html), you can apply a `git` patch file for One-Shot Tuner and bulid TVM using `cmake`:

```
git am one-shot-tuner.patch --ignore-whitespace \
--no-scissors --ignore-space-change
```

Detailed instructions can be found at the archive location.

A.3.3 Local Installation (GitHub).
The GitHub repository includes the modified TVM compiler. You can clone the repository to your local machine, install TVM and its python dependencies, and then build TVM using `cmake`.

```
git clone \
https://github.com/ryujaehun/one-shot-tuner.git
```

Detailed instructions can be found in the GitHub README.

A.4 Experiment Workflow

The overall experiment workflow is composed of three steps: data set generation, predictor model training, and optimal knob search using the trained model. You can jump to Step 2 if you reuse data samples already collected using PGS and EBS, or to Step 3 if you reuse the pre-trained predictor model.

Step 1: To sample data using the PGS and EBS configurations in the paper, execute the following command:

```
python3 dataset_generate/sampling.py -p -e
```

Step 2: To train the One-Shot Tuner predictor model using the pre-sampled dataset, execute the following commands:

```
python3 train_model/train.py --dataset_dir \
<dataset path> --layout NCHW --batch 1
```

Step 3: You can compile all benchmark models using the trained model and reproduce the reported values in the paper with the following command:

```
./main.sh
```

A.5 Evaluation and Expected Result

After each auto-tuning compilation, log files are generated and saved in the numpy format. These log files include performance results such as model inference time (FLOPS) and end-to-end compilation time. The files are stored under directories automatically created in the user-specified script home directory. You can collect information about multiple networks by executing the following script command:

```
python3 get_result.py
```

The expected evaluation result is as follows.

```
network alexnet algorithm ga inference time\
second 0.75655ms end2end 0.62h
network alexnet algorithm sa inference time\
second 0.75658ms end2end 0.70h
network densenet-121 algorithm ga inference time\
second 0.60628ms end2end 0.74h
network densenet-121 algorithm sa inference time\
second 0.62021ms end2end 0.79h
network efficientnet algorithm ga inference time\
second 0.61417ms end2end 1.51h
network efficientnet algorithm sa inference time\
second 0.56854ms end2end 1.61h
network resnet-18 algorithm ga inference time\
second 0.78022ms end2end 0.97h
network resnet-18 algorithm sa inference time\
second 0.76898ms end2end 1.07h
network mobilenetv2 algorithm ga inference time\
second 0.58570ms end2end 1.50h
network mobilenetv2 algorithm sa inference time\
second 0.60312ms end2end 1.49h
network squeezenet_v1.0 algorithm ga inference time\
second 0.61417ms end2end 1.51h
network squeezenet_v1.0 algorithm sa inference time\
second 0.56854ms end2end 1.61h
network bert algorithm ga inference time\
second 2.96953ms end2end 0.21h
network bert algorithm sa inference time\
second 2.99085ms end2end 0.21h
network vgg-16 algorithm ga inference time\
second 2.83573ms end2end 1.49h
network vgg-16 algorithm sa inference time\
second 2.78476ms end2end 1.64h
```

A.6 Experiment Customization

Scripts are all customizable with different sampling, modeling, and search methods. For data set sampling, other than the default configuration that uses the both sampling methods (PGS+EBS), Random+EBS, PGS+Random, and Random+Random (as experimented in the sensitivity analysis) can be tested. For model training, different neural network architectures, e.g., MLP, LSTM, and Transformer, are available for the feature encoder. Training hyperparameters, e.g., learning rate, l2 decay, batch size, and loss function, are also adjustable. For optimal knob search, you can experiment with different search algorithms (e.g., simulated annealing and genetic algorithm) and search hyperparameters (e.g., batch size, mutation probability, population size).

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (jul 2019).

[3] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. 2020. Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation. In *8th International Conference on Learning Representations (ICLR) 2020.*

[4] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *23rd International Conference on Parallel Architecture and Compilation Techniques (PACT) 2014.*

[5] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. 2016. COBAYN: Compiler Autotuning Framework Using Bayesian Networks. *ACM Trans. Archit. Code Optim.* 13, 2, Article 21 (jun 2016).

[6] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman amarasinghe. 2021. A Deep Learning Based Cost Model for Automatic Code Optimization. In *Proceedings of Machine Learning and Systems (MLSys) 2021.*

[7] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2019.*

[8] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2018. Accelerating Neural Architecture Search using Performance Prediction. In *6th International Conference on Learning Representations (ICLR) 2018, Workshop Track Proceedings.*

[9] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. 2013. Taming Parallel I/O Complexity with Auto-Tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2013.*

[10] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to Rank: From Pairwise Approach to Listwise Approach. In *Proceedings of the 24th International Conference on Machine Learning (ICML) 2007*.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015).

[12] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI) 2018*.

[13] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS) 2018*.

[14] Xiangxiang Chu, Tianbao Zhou, Bo Zhang, and Jixiang Li. 2020. Fair DARTS: Eliminating Unfair Advantages in Differentiable Architecture Search. In *16th European Conference on Computer Vision (ECCV) 2020*.

[15] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (ICML) 2021*.

[16] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2015. Autotuning OpenCL Workgroup Size for Stencil Patterns. *CoRR* abs/1511.02490 (2015). arXiv:1511.02490

[17] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT) 2017*.

[18] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. (2021). arXiv:2109.08267

[19] Xiaoliang Dai, Alvin Wan, Peizhao Zhang, Bichen Wu, Zijian He, Zhen Wei, Kan Chen, Yuandong Tian, Matthew Yu, Peter Vajda, and Joseph E. Gonzalez. 2020. FBNetV3: Joint Architecture-Recipe Search using Neural Acquisition Function. *CoRR* abs/2006.02049 (2020). arXiv:2006.02049

[20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

[21] Aditya Dhakal, Junguk Cho, Sameer G. Kulkarni, K. K. Ramakrishnan, and Puneet Sharma. 2020. Spatial Sharing of GPU for Autotuning DNN models. arXiv:2008.03602

[22] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2015*.

[23] M. Frigo and S.G. Johnson. 2005. The Design and Implementation of FFTW3. *Proc. IEEE* 93, 2 (2005).

[24] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin V. Bonilla, John Thomson, Christopher K. I. Williams, and Michael F. P. O'Boyle. 2011. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *Int. J. Parallel Program.* 3 (2011).

[25] Jian Guo, He He, Tong He, Leonard Lausen, Mu Li, Haibin Lin, Xingjian Shi, Chenguang Wang, Junyuan Xie, Sheng Zha, Aston Zhang, Hang Zhang, Zhi Zhang, Zhongyue Zhang, Shuai Zheng, and Yi Zhu. 2020. GluonCV and GluonNLP: Deep Learning in Computer Vision and Natural Language Processing. *Journal of Machine Learning Research* 21, 23 (2020).

[26] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2016*.

[27] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2017*.

[28] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. (2016). arXiv:1602.07360

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM) 2014*.

[30] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. 2010. An auto-tuning framework for parallel multicore stencil computations. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS) 2010*.

[31] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems (MLSys) 2021*.

[32] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012).

[33] Alexandre Lacoste, Alexandra Luccioni, Victor Schmidt, and Thomas Dandres. 2019. Quantifying the Carbon Emissions of Machine Learning. arXiv:1910.09700

[34] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[35] Jian Li, Yong Liu, Jiankun Liu, and Weiping Wang. 2020. Neural Architecture Optimization with Graph VAE. *CoRR* abs/2006.10310 (2020). arXiv:2006.10310

[36] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. 2020. AdaTune: Adaptive Tensor Program Compilation Made Efficient. In *34th Conference on Neural Information Processing Systems (NeurIPS) 2020)*.

[37] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *7th International Conference on Learning Representations (ICLR) 2019*.

[38] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia (MM) 2010*.

[39] Charith Mendis, Alex Renda, Saman P. Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML) 2019*.

[40] Eunjung Park, Christos Kartsaklis, and John Cavazos. 2014. HERCULES: Strong Patterns towards More Intelligent Predictive Modeling. In *43rd International Conference on Parallel Processing (ICPP) 2014*.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In

Advances in Neural Information Processing Systems 32.

[42] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. 2018. Efficient Neural Architecture Search via Parameter Sharing. In *Proceedings of the 35th International Conference on Machine Learning (ICML) 2018.*

[43] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. 2005. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"* (2005).

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 2013.*

[45] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI) 2019.*

[46] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A New IR for Machine Learning Frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL) 2018.*

[47] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. 2019. Glow: Graph Lowering Compiler Techniques for Neural Networks. arXiv:1805.00907

[48] Jaehun Ryu and Hyojin Sung. 2021. MetaTune: Meta-Learning Based Cost Model for Fast and Efficient Auto-tuning Frameworks. arXiv:2102.04199

[49] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2018.*

[50] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. (2014). arXiv:1409.1556

[51] Benoit Steiner, Chris Cummins, Horace He, and Hugh Leather. 2021. Value Learning for Throughput Optimization of Deep Learning Workloads. In *Proceedings of Machine Learning and Systems (MLSys) 2021.*

[52] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730

[53] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning (ICML) 2019.*

[54] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, Vol. 30.

[55] Wei Wen, Hanxiao Liu, Yiran Chen, Hai Helen Li, Gabriel Bender, and Pieter-Jan Kindermans. 2020. Neural Predictor for Neural Architecture Search. In *Computer Vision - ECCV 2020 - 16th European Conference on Computer Vision (ECCV) 2020.*

[56] R. Clint Whaley and Jack J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (SC) 1998.*

[57] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. 2019. SNAS: stochastic neural architecture search. In *7th International Conference on Learning Representations, (ICLR) 2019.*

[58] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. 2020. PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search. In *8th International Conference on Learning Representations (ICLR) 2020.*

[59] X. Zeng, T. Zhi, Z. Du, Q. Guo, N. Sun, and Y. Chen. 2020. ALT: Optimizing Tensor Compilation in Deep Learning Compilers with Active Learning. In *IEEE 38th International Conference on Computer Design (ICCD) 2020.*

[60] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. 2021. DynaTune: Dynamic Tensor Program Optimization in Deep Neural Network Compilation. In *The Ninth International Conference on Learning Representations (ICLR) 2021.*

[61] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 2020.*

[62] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. 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) 2020.*

[63] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations (ICLR) 2017.*

[64] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR) 2018.*