Fasor: A Fast Tensor Program Optimization Framework for Efficient DNN Deployment

Hanxian Huang
University of California San Diego
San Diego, USA
hah008@ucsd.edu

Xin Chen
Intel Corporation
Santa Clara, USA
xin.chen@intel.com

Jishen Zhao
University of California San Diego
San Diego, USA
jzhao@ucsd.edu

ABSTRACT

With the growing importance of deploying deep neural networks (DNNs), there are increasing demands to improve both the efficiency and quality of tensor program optimization (TPO). TPO involves searching for possible program transformations for a given tensor program on target hardware to optimize its execution. TPO is challenging and expensive due to the exponential combinations of transformations and time-consuming on-device measurement of transformations. While prior research has primarily focused on the quality of TPO, i.e., generating high-performance tensor programs, there has been less emphasis on the efficiency of TPO, i.e., optimizing tensor programs with low optimization time overhead.

In this paper, we address the primary inefficiencies in current TPO approaches, especially the extensive time required for on-device measurement and the inefficiency in the search process, and aim to reduce the optimization time for DNNs. To this end, we propose a machine learning-based, end-to-end TPO framework named Fasor. Fasor includes three key design components: 1): a transferable cost model with high transferring efficiency to reduce the on-device measurement time significantly, 2): a search space shrinking module to prune program transformations with low optimization potential, and 3): a two-stage fast exploration module to enhance searching efficiency substantially. Experimental results show that Fasor achieves the best of both worlds in TPO quality and efficiency compared to state-of-the-art TPO frameworks for CPUs and GPUs, contributing to efficient and scalable DNN deployment.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; **Source code generation**; • **Computing methodologies** → **Machine learning**.

KEYWORDS

tensor compilers, auto-tuning, TVM, tensor program optimization, compute schedules, performance models, deep neural networks

ACM Reference Format:
Hanxian Huang, Xin Chen, and Jishen Zhao. 2024. Fasor: A Fast Tensor Program Optimization Framework for Efficient DNN Deployment. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656631

This work is licensed under a Creative Commons Attribution International 4.0 License.

ICS '24, June 04–07, 2024, Kyoto, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0610-3/24/06
https://doi.org/10.1145/3650200.3656631

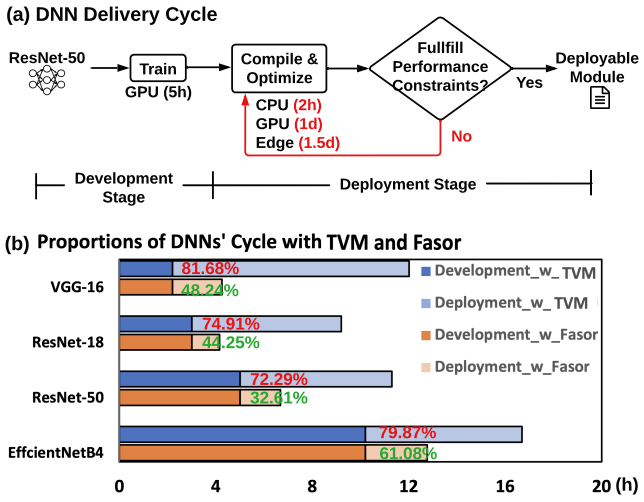


Figure 1: (a) DNN life-cycle. The deployment stage is becoming a critical bottleneck of DNN delivery. (b) Comparison between Fasor and the TVM on a breakdown of the DNN cycle across development (the training time on Nvidia V100 GPUs) and deployment (the optimization time on Intel i9-9900K CPU). The proportion of deployment is marked in red for TVM and green for Fasor. In our evaluation, Fasor speedups the optimization time up to 10.24× and decreases the proportion of the deployment phase significantly.

1 INTRODUCTION

Deploying high-performance deep learning (DL) models on a wide range of platforms (e.g., CPU, GPU, ASIC, FPGA) has become an increasingly crucial research topic [9, 15, 34, 39]. The general life-cycle of a DNN model consists of two stages: *development* (model designing and training) and *deployment* (model compilation and inference). Existing works have significantly shortened the development duration by automating the design of DNNs [28, 29, 45] and accelerating the training process [32, 44, 47, 57] to even a few seconds. However, the deployment stage is becoming a critical bottleneck in DNN delivery, as shown in Fig. 1.

In this paper, we study the trade-off between DNN compilation time efficiency and runtime performance of the compiled DNN. We aim to reduce the time of compiling DNN models while improving output code runtime performance. First, we analyze TVM [15], the state-of-the-art (SOTA) DNN compiler stack on various optimization tasks and hardware backends, and identify the bottleneck of tensor program optimization (TPO) (Sec. 2.3). We observe that the key bottleneck (over 60%) of DNN compilation is the on-device measurement for cost model training (Tab. 2). The long TPO time

is caused by cost model training/transfer learning inefficiency and search sampling inefficiency. Based on our study, we provide two principles to solve the bottleneck and improve TPO efficiency:

(a) **Transferring efficiency:** This focuses on enhancing the cost model's ability to learn the general knowledge of evaluating tensor programs, enabling it to swiftly transfer to new hardware backends with minimal need for costly on-device measurement.
(b) **Sampling efficiency:** This involves strategically navigating the search space to minimize sampling of potential schedules, particularly those with low optimization potential.

Following these two design principles, we propose a **FA**st ten**SOR** program optimization framework, named **Fasor**, which can automatically achieve the best optimization quality and efficiency trade-off. First, to improve transferring efficiency, we propose a transferable cost model (Sec. 4) that effectively captures both tensor program features and hardware features. Different from previous online cost models [6, 7, 16, 63, 65], which only use the costly measurement data generated during the tuning process, our model can be transferred efficiently via a lightweight transfer-learning to learn hardware-specific knowledge, thus significantly reduces the need for costly measurement on the unseen hardware platform during optimization. Different from previous offline cost models [12, 52, 60, 62], which assume the target platform is accessible before compilation, we aim to address a more practical and challenging scenario where the target platform is unknown beforehand, and the cost model tuning must be performed on-the-fly with minimal new data measured on the new target platform for fine-tuning, necessitating higher model transferability. Moreover, accurate cost modeling also helps precisely identify the most efficient configurations, leading to better search results and faster search convergence.

Then, to enhance sampling efficiency, Fasor incorporates a search space shrinking module (Sec. 5) to prune the search space by eliminating candidates with low potential for program improvement, thereby retaining a condensed yet vital subset of the search space. Furthermore, Fasor leverages a two-stage fast exploration module (Sec. 6): it first utilizes optimization task features as a key to index the optimal schedule for the most similar task from the offline collection; it then leverages the pre-tuned schedule as a start point to efficiently search over the shrunk search space by deep reinforcement learning (DRL), with a novel roofline model-based reward. Fasor demonstrates that the pre-tuned optimal schedule is a much more effective search start point to converge in much fewer steps, which prior studies overlook.

We summarize our three major contributions as follows:

- We identify the key bottleneck of TPO, and the proposed Fasor significantly shortens the DNN deployment duration. Experimental results show that Fasor outperforms the SOTA manual libraries and TPO frameworks for both standard deep learning benchmarks and novel NAS models. Fasor improves the compilation efficiency on the Intel CPU and NVIDIA GPU by up to 10.24× and 8.17×, respectively, while delivering better or equal output code latency performance with 1.22× average speedup.
- To improve transferring efficiency, we develop a novel transferable cost model that can quickly adapt to new devices in Fasor. Our cost model significantly reduces the on-device measurement and profiling time required for online compilation by an average of 78.4% compared to TVM.
- To improve the sampling efficiency, we propose a search space shrinking module to generate a compact but vital sub-space. We also propose a novel DRL exploration module that reuses the optimal schedule for a similar tensor program as a more effective search start point to search efficiently. Fasor reduces the search time by 72.2% on average compared to TVM.

2 BACKGROUND AND MOTIVATION

Taking TVM as an example, a deep learning compiler compiles deep learning models into minimum deployable modules on diverse hardware backends, as shown in Fig. 3 (a). TVM takes a deep neural network model as input, typically represented by deep learning frameworks like PyTorch [38] and TensorFlow [5], and analyzes the computational graph and performs graph-level optimizations such as operator fusion, layout transformation, and memory management. TVM then performs tensor-level optimizations, which involve optimizing the computations within each operator, and finally generates optimized code for the target hardware platform. In this paper, we focus on the tensor-level optimization (TPO).

2.1 Problem Formalization

As a key function of DNN compilers, the TPO problem can be formulated as the following optimization problem:

$$argmin_{s \in S_e} f(s), \quad s.t. T <= B \tag{1}$$

Given a tensor program expression e, the search space S_e denotes the space of possible code implementations that are logically equivalent to the expression e. Let f be the runtime cost, for example, the latency on the hardware, T be the optimization cost, and B be the optimization time budget.

As shown in Eq. 1, there are three key parameters to solve the optimization problem: (1) The *search space* S_e is composed of different combinations of schedule primitives and corresponding parameters as shown in Tab. 1. As the number of parameters to be decided increases, and each parameter has more possible values, the search space becomes larger, resulting in a more complex search problem. (2) The *cost* f, e.g., latency. Measuring all the combinations on the device during searching costs a large time overhead, especially on CPUs and edge devices. Thus, existing TPO frameworks introduce a cost model [6, 15, 63] to estimate the cost instead of measuring it. However, they still require collecting measurement data to train the cost model. (3) The *time used for optimization* T and the *optimization time budget* B: our goal is either to attain superior optimization quality (e.g., a lower latency) with the same budget B, or to achieve the same level of optimization quality within a shorter T.

A TPO Example. Here, we use the general matrix multiplication operator as an example, which is often regarded as the core component of deep learning:

$$C_{i,j} = \sum_k A_{i,k} \times B_{k,j}, \quad 0 \le i, j, k < n \tag{2}$$

The above expression provides a unified mathematical specification, and it can be transformed to multiple logically equivalent variants of low-level code $s \in S_e$ and be optimized to different machine codes on different devices.

Table 1: Configurable parameters in search space.

Primitives	Parameters
Split	split factor
Fuse	the adjacent loops to be fused
Vectorization	vectorization factor
Reorder	the new order index of *for* statements
Unrolling	unrolling depth
Parallel	which loop to parallel
Compute At	computation axis to compute at
Compute Inline	the statement to be inlined

For instance, as shown in Fig. 2, the low-level code s_0, s_1 and s_2 (where s_0 is a default low-level code) specifies various loop split, loop order, vectorization, unrolling, and parallelization details that may lead to different execution performances. Given a user-defined optimization budget B, Fasor aims to achieve the optimal output machine code by solving the Eq. 1. Or given an expected output code performance, Fasor aims to achieve it with minimal T, to solve the tradeoff problem of latency time and compilation time on various devices.

```
1. for i in range(512):                                          s0
2.     for j in range(512):
3.       for k in range(512):
4.         C[i][j] += A[i][k] * B[k][j]
```

```
1. parallel i.0 in range(512):                                   s1
2.     for j.0 in range(4):
3.         unroll j.1 in range(128):
4.             unroll k.0 in range(32):
5.                 vectorize k.1 in range(16):
6.                     C[i.0][j.0*128+j.1] +=
  A[i.0][k.0*16+k.1] * B[k.0*16+k.1][j.0*128+j.1]
```

```
1. parallel i.0 in range(16):                                    s2
2.     for j.1 in range(128):
3.         for k.0 in range(512):
4.             for i.1 in range(32):
5.                 vectorize j.0 in range(4):
6.                     C[i.0*32+i.1][j.1*4+j.0] +=
  A[i.0*32+i.1][k.0] * B[k.0][j.1*4+j.0]
```

Figure 2: Three sample pseudo implementations of expression $C_{i,j} = \sum_k A_{i,k} \times B_{k,j}, 0 \le i, j, k < 512$: s_0, s_1 and s_2. All of them are logically equivalent programs with different performances since they specify various low-level code implementations. Therefore, although the same math equation, the run-times of s_0, s_1 and s_2 are different.

2.2 Importance of TPO Efficiency

Most prior works focused on optimizing the quality of TPO (output code latency), while overlooking the efficiency of TPO (compilation time). Existing compilation frameworks still spend several hours, even days to generate a decent schedule for DNNs. For example, AutoTVM [16] costs 10 hours on an Intel i9-9900K CPU and 7 days on a NVIDIA GeForce RTX 2080Ti GPU, and 10 days on Raspberry Pi 4 to tune all workloads in the ResNet-50 model. The compilation time overhead is scaling up when optimizing different workloads for today's DNNs with more complicated neural architectures [49, 56] on various target platforms. Yet, TPO efficiency is important in modern applications because a long optimization time (1) prolongs the delivery duration of AI products and (2) hurts users' experience while waiting for DNN to be optimized.

As shown in Fig. 1, the long deployment time takes over 70% of the delivery cycle even with the SOTA compilation framework. It delays the overall DNN delivery and goes against the AI companies that are required to deliver AI in a timely manner. It also, in turn, slows down the development of DNN models due to the optimized runtime performance not giving feedback to the DNN designers and developers on time. As the DNN models are scaling and becoming more complicated, efficient TPO is highly demanding.

2.3 Inefficiency in Previous Methods

The SOTA DNN compilation stack leverages the *compute-schedule* paradigm to decouple the high-level description of an algorithm (*compute*) from the description of how it is optimized and executed (*schedule*) on a specific hardware platform, e.g., AutoTVM [16], Ansor [63], Flextensor [65], Chameleon [7], Halide [6]. Based on this paradigm, existing works follow a similar philosophy to manage the TPO problem as a schedule search problem with the following steps: considering a piece of tensor program to be optimized on a given hardware backend, (1) parameterize and define the schedule search space (primitives and parameters) manually (AutoTVM) or automatically (Ansor, Flextensor, Halide); (2) define a code template (AutoTVM, Flextensor, Chameleon, Halide) or sample code sketches (Ansor), and the default schedule settings as the search starting point; (3) at the beginning of searching, the cost (e.g., latency) is measured on device and the cost is collected to train a cost model, e.g., XGBoost-based model (AutoTVM, Ansor, Chameleon), analytical model (Flextensor) or linear layers (Halide); (4) guided by the estimated cost from the cost model, search over the search space using a certain search engine until finish the optimization time budget, e.g., simulated annealing (AutoTVM), evolutionary search (Ansor), beam search (Halide), policy gradient (Chameleon), or heuristic method combining DRL (Flextensor).

It is difficult to achieve both high *quality* (in terms of output code latency) and *efficiency* (in terms of optimization time) of TPO. Previous DNN compilation stacks can search high-performance tensor program schedules, but they can take several hours and even days of optimization overhead, especially on CPUs and edge devices. We study the above-mentioned frameworks and break down the time overhead to demonstrate the bottleneck of DNN compilation is the on-device measurement for cost model updating, which consumes over 60% of compilation time. We show the portion of on-device measurement time in Tab. 2. This observation aligns with previous study [7, 11, 59], emphasizing the need to reduce the costly on-device measurement.

Table 2: The portion of on-device measurement time over the overall optimization time.

CPU / GPU	Ansor	AutoTVM	Chameleon
ResNet-18	87.0% / 77.1%	91.8% / 79.0%	89.7% / 76.8%
VGG-16	70.1% / 62.3%	71.8% / 67.5%	70.3% / 66.2%
AlexNet	78.7% / 72.5%	80.3% / 72.9%	75.6% / 72.4%

There are two aspects of inefficiency that result in the long TPO time in the existing works. **(1) Cost model training or transfer learning inefficiency.** Machine Learning (ML)-based model (AutoTVM, Ansor, Chameleon) is widely adopted to provide accurate estimations of schedule candidates, compared to the traditional analytical model (Flextensor). However, for the ML cost model to be effective, it needs to be well-trained to learn the general knowledge

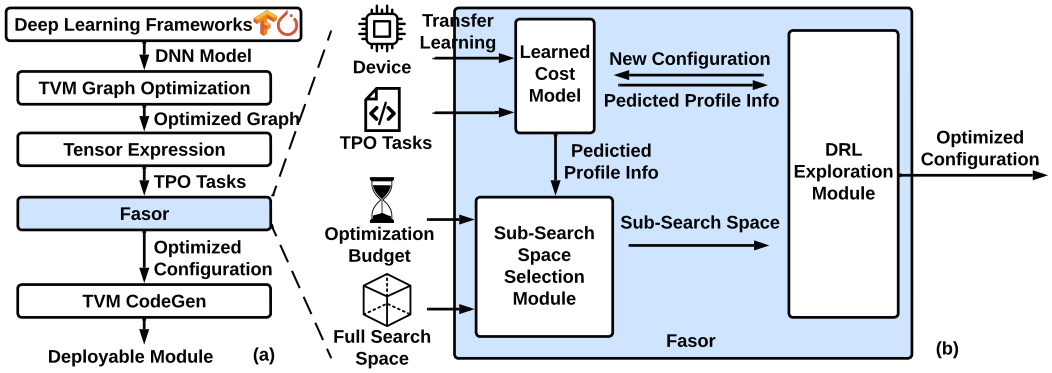


Figure 3: (a) DNN compilation stack. Fasor focuses on the tensor-level optimization. (b) The overview of Fasor design.

required for evaluating tensor program schedules of all kinds – *achieving generalization* – and adapted to capture the characteristics of various hardware platforms – *ensuring transferability*. The emerging novel DNN operators and fast-developing new hardware make training a generalizable and transferable cost model even more difficult. It requires collecting enough training datasets and cannot avoid measuring a large number of scheduled candidates on the device.

(2) Search sampling inefficiency. Searching for an optimal scheduling scheme is an NP-hard problem, while the search space is growing exponentially with the number of primitives and the corresponding parameters to be tuned. Searching for effective schedule configurations from a search space with millions of possible configuration combinations is non-trivial. The above frameworks follow a latency-guided search and deploy an extensive exploration over the optimization landscape, coupled with aggressive pruning of the solution space. Their search engines usually rely on the stochastic guarantees of a random walk process, fall short of inadequate search guides, and often end up generating sub-optimal code. Moreover, previous works overlook the potential of reusing pre-tuned schedules for similar optimization tasks.

Previous studies such as Moses [62], BALTO [12], TLP [60], and Verma et al. [52] attempt to improve the capability of transferring cost models to solve the transferring inefficiency. However, they target a scenario where the target platform is accessible in advance, allowing for offline data measurement and cost model training or fine-tuning for the target platforms offline. In contrast, Fasor is for addressing a more practical and challenging setting where the target platform is unknown beforehand, and new data measurement and transfer learning occur online once the device becomes accessible. In such a scenario, fewer samples should be measured online to update the cost model to maintain optimization efficiency, while ensuring that the optimization quality is not compromised.

3 FASOR DESIGN

3.1 Fasor Framework Overview

As depicted in Fig. 3 (b), Fasor consists of three key components:

(1) The cost model is pre-trained offline and takes a given tensor program as input to predict the profile information, e.g., memory bound score, core bound score, and latency rapidly and accurately online. In the online optimization stage, only a few samples will be measured on the device to fine-tune the cost model at the beginning. After that, the cost model only performs inference. The predicted information is forwarded to the sub-search space selection and DRL exploration modules.

(2) The sub-search space selection module captures the critical memory and computation-sensitive characteristics of various operators and primitives, classifies the primitives into memory-sensitive and computation-sensitive subspaces, and prunes the less sensitive primitive subspace.

(3) The DRL exploration module takes a pre-tuned schedule as the search initial state and searches for a new configuration output guided by the feedback from the learned cost model. The designed DRL can converge and find the optimal solution in only one (few) step(s), significantly improving the search sampling efficiency.

4 A LEARNED COST MODEL

The primary goal of the learned cost model is to precisely evaluate the cost of a schedule for any given tensor program, serving as an accurate guide for the search engine. Another goal of the cost model is good transferability so that it does not require too many samples to be measured on a new device to update the cost model. Previous work [40] proposes a zero-shot model, which does not update the cost model during online compilation. However, it does not consider any hardware features in the cost model, making it in-precise to evaluate the costs of new hardware. In this paper, we propose to train a cost model to learn both (1) general knowledge of evaluating all kinds of tensor program schedules and (2) characteristics of various hardware platforms.

4.1 Model and Feature Design

Model Selection: Given the abundance of existing deep learning models adept at modeling the cost function (a regression task) [6, 16, 24], and considering that devising a new model architecture is not our primary goal, we opted for an empirical approach. We evaluated XGBoost, LSTM, and a lightweight Transformer model, all of which demonstrated online inference latencies under 200ms on a GPU, which is negligible compared to the compilation time. Based on performance metric (Sec. 8.5), we selected the lightweight Transformer as the backbone architecture of our cost model. Transformer [18] has shown success on many applications [23, 30, 55], and demonstrated its ability to capture the general features and transfer to new tasks. Fasor adopts a light-weight Transformer encoder to capture the schedule features and learn a general and transferable cost

model. Our cost model incorporates 4 multi-head self-attention layers followed by a linear layer to predict the execution time, memory bound score, core bound score concurrently.

Feature Selection: (1) For the task feature, since we rely on the template-based code generation scheme in TVM, similar to [16], we include arithmetic features and memory access features. Arithmetic features include (i) kernel / input / output shapes and (ii) serialized schedule configurations, such as vectorization, parallelism, and tilling, as shown in Tab. 1. The memory access features include access type, reuse type, and total accessed bytes. (2) For the hardware feature, an initial solution to include hardware features is to utilize essential hardware specifications. For CPUs, we include CPU family types, core frequency, number of cores, main memory size, cache line size, vector unit width, etc. For GPUs, we include GPU family types, memory size and number of threads per block, maximum number of extent virtual threads, thread numbers of a wrap, vector unit width, etc. However, we found that simply including these specifications cannot help the cost model transfer to new hardware efficiently and effectively during compilation time, especially for new hardware with very different modeling. Thus, we propose to implicitly learn hardware features from a small signature set of tensor programs to enable few-shot transfer learning to new hardware.

4.2 Few-shot transfer learning

Given the diversity and complexity of the hardware [10], the runtime latency of a specific low-level tensor program implementation varies on different hardware, which makes the cost model difficult to reuse over different hardware. Existing machine learning-based cost models in AutoTVM, Ansor, and Chameleon are trained from scratch and have to measure the actual cost of a large number of schedules sampled by the search engine online for new hardware. On-device measurement is a high-cost process involving schedule compilation, data transfer, code generation, and on-device execution. As such, on-device measurement is the bottleneck in the TPO process. With the ResNet-18 optimization as an example, it takes on average 83.5% of the optimization time as shown in Fig. 4.

To solve the above issue, we propose to learn the general knowledge of evaluating tensor programs offline (a once-for-all pre-trained model) and transfer it by learning the hardware-specific knowledge via a lightweight fine-tuning online given a new device. The transfer learning is few-shot and uses only few samples. We select a small set of representative data to perform a light-weight transfer learning. To achieve better transfer learning performance with high prediction accuracy, the selection of the data for transfer learning should prioritize tasks that contribute distinct hardware-specific knowledge, enabling the cost model to effectively differentiate and learn from the diverse information. We evaluate the information contribution by clustering the tasks by Spearman similarity [35] considering the task features. The tasks in the same cluster are similar; selecting two similar tasks from the same cluster contributes little information. Thus, we pick a total of top-k tasks from different clusters so they are dissimilar from each other. The hyper-parameter k determines the number of tasks to be measured; a larger value of k results in more tasks being evaluated, potentially leading to a more accurate cost model and better optimization outcomes, albeit

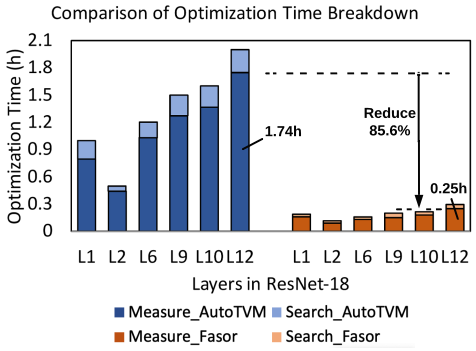


Figure 4: Breakdown comparison of TPO on Intel i9-9900k CPU for ResNet-18 [22]. We identify the on-device measurement time as the bottleneck of TPO. Fasor significantly reduces both the on-device measurement time and search time.

at the expense of increased measurement overhead. We empirically set $k = 10$ to promise model accuracy in practice.

Due to the great performance gaps among various backends, we train different cost models for various backends and only transfer among the platforms with the same backends. For example, We train one cost model for Intel Xeon E5-2666 CPU and transfer it to other target devices with x86 CPU, e.g., Intel i9-9900k CPU. We train another cost model for the Nvidia Tesla V100 GPU and transfer it to the Nvidia GeForce RTX 2080Ti. We do not transfer among desktop CPUs / GPUs to mobile CPUs / GPUs, etc.

5 SUB-SEARCH SPACE SELECTION MODULE

Given the high complexity of the search space, it is infeasible to enumerate all of the search space. Randomly shrinking the search space will lead to heavy output latency performance degradation [63]. Fasor deploys a sub-search space selection module to automatically and wisely select the primitives and parameters that constitute a relatively critical sub-search space. It provides two levels of search space shrinking:

(1) Empirical parameter options pruning. Ideally, the parameters can be chosen from all the feasible options (e.g., any positive integer works as a split factor). However, Fasor leverages observations from our study as well as lessons from previous studies [15, 21] to choose configuration options that are more likely to lead to good optimization performance. For example, when searching for parallelization and vectorization, Fasor gives higher priority to parallelizing the outer-most loop for CPUs and prefers to bind outer loops to thread blocks for GPUs. For vectorization, Fasor only considers powers of two according to the backend support properties. Similarly, for the split factor, Fasor only considers powers of two that are divisible by the loop range to avoid irregular operand sizes.

(2) Profile-guided critical primitives selection. In practice, we observe that not all primitive tuning significantly contributes to optimization. Tuning only parts of the selected primitives, while fixing other primitives as the default settings can significantly shrink the sub-search space. Fasor selects the most critical primitives based on their sensitivity to memory or computation bound. We first perform a sensitive study on the dataset (Sec. 7.1) offline. Each time we only tune one primitive while fixing all the other primitives and profile the memory and computation-bound score changes. We weigh the

primitives according to the sensitivity of memory or computation score changes. In the compilation time, users can pass a hyper-parameter n_p ranging from 5 to 8 to decide the number of primitives to be tuned. $n_p = 7$ by default. The learned cost model will first predict the memory and computation bound scores to check if the task is more memory or computation-sensitive. Then, the top n_p primitives, ranked by their memory/computation weights, will be selected. The n_p controls the optimization quality and efficiency trade-off by users' requirements. $n_p = 8$ means tuning all the primitives in Tab. 1. $n_p = 5$ is the threshold to avoid too much output performance degradation in our study.

6 DRL EXPLORATION MODULE

Given the critical sub-search space to be explored, Fasor leverages a two-stage fast exploration module to select the optimal configurations of schedule primitives.

6.1 Stage1: Exploiting Pre-tuned Schedule as the Search Start Point.

Pre-tuned schedules are difficult to reuse directly during compilation due to the diversity of tensor programs and differences among hardware platforms. As such, very few previous works [19, 59] have explored reusing pre-tuned schedules. Lorien [59] builds a database to store pre-tuned schedules while performing random sampling for those new schedules not existing in the database. However, building a database to store all pre-tuned schedules is infeasible due to the diverse tensor programs and hardware platforms to be used. Moreover, random sampling cannot guarantee an optimal result. Transfer-tuning [19] directly applies pre-tuned schedules for the same kernel regardless of different data shapes. Directly applying pre-tuned schedules for the same kernel with a different input data shape without adaptation is sub-optimal. Because a different data shape contributes to a different tensor program, leading to a new optimization task and a different search space. Directly reusing pre-tuned schedules is not generalizable to unseen optimization tasks.

In this paper, we investigate the similarity of tensor programs and the relationship between optimal schedules of similar tensor programs. We pose such a question: *"How to leverage the pre-tuned schedules considering the characteristics and similarity of tensor programs and their search spaces?"* In this paper, we empirically show the pre-tuned optimal schedules for the tensor programs in the pre-trained dataset (Sec. 7.1) can be reused as a better search start point, which helps search converge faster and better compared to TVM's default settings. First, starting with a pre-tuned schedule for a task similar to the new tensor program allows the search algorithm to converge more quickly. This is because the starting point is already closer to an optimal solution than a random start point. This aligns with the observations in previous works [19, 59]. Second, combining the DRL searching in stage 2, it achieves a better optimization result with the pre-tuned schedule serving as a better basis than the default settings provided by TVM. The rationale behind this is to enhance the searching efficiency and quality of auto-tuning by utilizing existing knowledge (pre-tuned schedules) in a wiser way, tailoring it to the specific needs of new tensor programs, rather than searching from scratch or relying on generic default settings.

Operationally, we reuse the pre-tuned schedules from the same kernel class – kernels sharing the same sequence of operations, for example, *conv2d-bias-relu*. We build a small database to store only the optimal schedule for each kernel class on each hardware platform. At compile time, the schedule can be retrieved by the key of the kernel using a hash function, with an average time complexity of O(1). The kernel's key is illustrated in Tab. 3, and the corresponding value for the best schedule comprises the serialized schedule parameters. Should such a key-value pair be absent in the database, we resort to the pre-tuned schedule exhibiting the highest Spearman similarity to the current optimization task, taking into account the key attributes. We set a temperature $\tau = 0.6$ to decide whether we want to use such a pre-tuned schedule. If the Spearman similarity falls below τ, the default starting point of TVM is selected instead. The temperature setting controls the randomness of the search's starting point, enhancing the search engine's robustness by preventing it from becoming trapped at a local optimum.

Table 3: Key attributes to store pre-tuned schedules.

Attribute	Description
Target type	The type of target platform for this task, e.g., x86 CPU, ARM CPU, NVIDIA GPU
Task key	A unique task key defined by the auto-tuning framework dialect used when generating tasks.
Kernel class	The sequence of operations, e.g., conv2d-bias-relu
Arguments	The task arguments or attributes.

6.2 Stage2: Fast DRL Search.

Deep Reinforcement Learning (DRL) has shown success in efficiently searching over high-dimensional problems (e.g., go game [42]), which is also suitable for solving configuration search problems. We leverage the efficient DRL model and propose a novel reward function better to adjust it to our schedule parameter searching problem and generate good-quality results in one or very few iterations (episodes) of the DRL search. Fig. 5 shows Fasor's DRL model, which consists of (a) an agent with an action-value network and (b) an environment. The agent with the action-value network responds to a state (a tensor program with certain configurations) and outputs an action (a set of configurations of the schedule primitives). The environment tests the new state by the cost model to return a reward to the agent.

6.2.1 Agent, State, Action, and Reward. As shown in Fig. 5, our DRL model consists of four major components: agent, state, action, and reward.

Agent. An agent is an executor of actions in a state. In our model, a state is a representation of the tensor program with certain schedule configurations, while an action is to assign the configurations of the schedule. The agent repeatedly obtains a new state s_t from the environment, executes an action a_t on the state, and gets a corresponding reward $r_t(a_t, s_t)$ by the environment, until the DRL model converges (does not get improvement in consecutive steps) or finish the optimization time budget. The object is to find good actions under time budget T to maximize $\sum_{t \leq T} r_t(s_t, a_t)$.

Action. An action is to decide all the primitive parameters that maximize the Q-value from the n_p adjacent points in Tab. 1. The

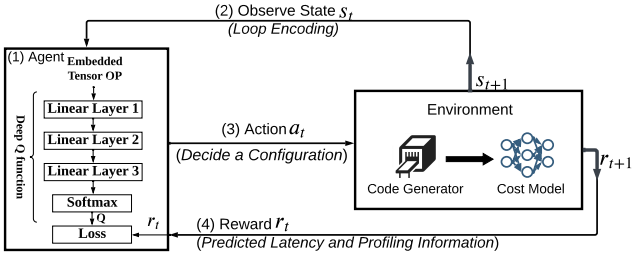


Figure 5: Fasor's DRL model consists of an agent and an environment. During model training, the agent repeatedly makes and sends actions to the environment based on the rewards and states it receives.

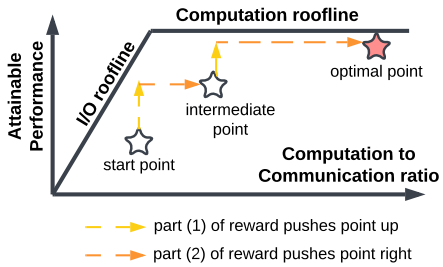


Figure 6: Fasor's reward is inspired by the roofline model, aiming to push the schedule point upwards and towards the right to achieve optimal performance.

adjacent point means only tuning one primitive to its adjacent value. For example, the possible choices for one factor are $[1, 2, 4, 8, 16]$, and the adjacent choices for the current value 4 are 2 and 8.

State. A state is a certain program implementation defined by a combination of schedule primitives and corresponding parameters for a given tensor code description. The states are encoded as the task feature mentioned in Sec. 4 to be the input of the DRL model.

Reward. A reward is used to measure the consequences of the actions and feedback to the agent to generate better actions. During the search process, the agent assigns a set of new configurations, which will get a reward feedback defined as:

$$r_t = \phi_1(T_{t-1} - T_t) + \phi_2 \frac{CS}{MS} + \phi_3 \#Iter \tag{3}$$

It is the weighted sum of (1) the difference between the execution time (T_t) of the new configuration at step t and the time (T_{t-1}) of the previous configuration at step $t-1$, (2) the computation (computation score CS) to communication (memory score MS) ratio $\frac{CS}{MS}$, (3) the number of iterations $\#Iter$ used for searching. The factors ϕ_1, ϕ_2 and ϕ_3 are hyper-parameters to manage different scales of different parts and decide the relative importance of them, controlling TPO quality-efficiency trade-off. The execution time, CS, and MS are outputs of the cost model. The reward designed is hinted by the roofline model [54] as shown in Fig. 6. The first part of the reward encourages the model to learn to generate low-latency schedules, which pushes the point up in the space. The second part of the reward encourages the model to learn to generate low-communication schedules given the computation is relatively stable, and pushes the point right in the space. The third part of the reward encourages the model to fast search for the optimal results in fewer iterations. Guided by the proposed reward function, the DRL model is usually able to search for a comparable result to TVM in one or very few iterations in our evaluation. Thus, we empirically set DRL to search for only one iteration by default. We then use our few-shot fine-tuned cost model to evaluate the candidates and select the optimal result with the smallest cost.

In this way, Fasor is progressively shrinking the optimized search space. Starting from the pre-tuned schedule, it shrinks the space and enables the DRL model to explore the space that is concentrated to the optimal result. Guided by the proposed reward function, the DRL effectively explores and generates a batch of 64 candidates. The cost model evaluates the candidates and finally picks the top-1 schedule. During this searching process, Fasor does not require any cost model updating or any on-device measurement, which greatly reduces the need for costly on-device measurement. We use this efficient searching mode as *high-efficiency* mode by default. We also provide another *high-quality* mode, which can be specified by users. Under the *high-quality* mode, DRL is able to explore the search space until convergent or until consuming all the optimization budget, instead of exploring for only one iteration. We evaluate both modes in our evaluation.

6.2.2 Action-value Network. The agent decides which action to take generated by an action-value neural network. Fasor models the search problem as a deterministic Markov Decision Process process and deploys DQN [33] to assign the Q-value, which calculates an expected reward given states and actions to evaluate an action. We choose DQN because it learns and reuses information by memory replay during searching iteration (Algo. 1) and considers the contribution of future rewards in Q-value to search the configuration space and achieve optimal schedules efficiently.

Algorithm 1 Q function Algorithm.

```
Initialize replay memory D and initialize the trainable parameters θ in DQN models
for epoch from 1 to optimization budget B do
    Initialize sequence s_1 = x_1, φ_1 = φ(s_1)
    for t from 1 to T do
        if random generated probability < ε then
            Select a random action a_t
        else
            Select the action with the maximum reward
            a_t = max_a Q(φ(s_t), a; θ)
        end if
        Take the action a_t and get reward r_t and next state x_{t+1}.
        Update s_{t+1} = s_t, a_t, x_{t+1}.
        D stores the transition φ_t, a_t, r_t, φ_{t+1} = φ(s_{t+1}).
        Inquire D for dataset and get y_j = r_j + γ max_{a'} Q(φ_{j+1}, a'; θ).
        Minimize Loss (y_j − Q(φ_j, a_j; θ))^2.
    end for
end for
```

We show the neural architecture of the action-value network in Fig. 5. The encoded state is passed to three fully connected layers. The *Softmax* function will output a Q-value, and the action will be selected either by maximizing the reward or with a small probability of playing a random action, as shown in Algo. 1. Then the loss function will be calculated by the real reward r and Q-value and the trainable parameters in Q function will be updated.

7 IMPLEMENTATION

We implement Fasor based on TVM [15] (version 0.8.dev) for code generation. We implement the cost model and the DRL model with PyTorch [38] using two Nvidia GeForce GTX 2080 Ti GPUs with 11 GB of memory. For the cost model, we apply 4 Transformer layers, with 128 hidden states and 8 attention heads, initialized with Xavier [20]. We use the Adam optimizer [25] with a learning rate of 0.0005, $\beta_1 = 0.9$, $\beta_2 = 0.999$, a weight decay rate of 0.01, and linear decay of the learning rate. We set the dropout rate as 0.1 and pre-train the cost model with a batch size of 64 for 20 epochs. We set the decay rate $\alpha = \frac{K}{K+T}$ to achieve temporal decay, with $K = 10$ and T as the epoch number. We set the discount rate γ as 0.95 and train the DRL model for 20 epochs. We set $\phi_1 = 0.35, \phi_2 = 0.25, \phi_3 = 0.4$ for the *high-efficiency* mode and $\phi_1 = 0.45, \phi_2 = 0.35, \phi_3 = 0.2$ for the *high-quality* mode. The cost model and DRL model training take approximately 2 hours in total on GPUs, which is performed once offline and is not counted into the online optimization time overhead. Note that this training time can be further reduced by running on more advanced GPUs, e.g., H100 [37] in parallel.

7.1 Dataset for the Learned Cost Model

A comprehensive dataset is necessary to promise the generalization of the cost model. Since the optimization tasks come from the subgraphs of DNNs, we first include rich types of DNN architectures: (1) models in PyTorch's Model Zoo [38]. (2) models generated by Neural Architecture Search (NAS): unlike the manual-designed models in Model Zoo, NAS generates models by searching for basic block combinations and bringing more derived models. For example, ProxylessNAS [13] changes the widths, depths, and resolutions of convolutions in a supernet to generate various subnets. We include the NAS models from NAS works [13, 49]. (3) Randomly generated models that are unseen in (1) and (2). It consists of (i) linear stacks of various convolution kernels with different configurations. (ii) non-linear models with complicated graph topologies from [56] to make the cost model more general to artificial neural topologies. Finally, we collected 914 different tasks (with various operator types and parameter configurations).

We then generate the schedules based on the above-mentioned DNNs. Given the conclusion that the schedules are non-uniformly distributed and numerous configurations are invalid [7], we do not randomly sample the schedules to generate the dataset. Instead, we adopt our search engine (Sec. 6) to sample the valid schedules that likely appear in the real compilation time. For each optimization task, we sample 1000 different schedules and remove redundant/invalid schedules, resulting in $\sim 0.9M$ samples in the dataset. We then measure the execution time and profile the memory and computation scores, which count the percent of memory or computational bottleneck [58] by TVM PAPI tool [4] and Intel PMU Profiling Tool [2] on the source devices for the schedule samples. The collection of training data takes less than two weeks and is done only once offline. We separate the dataset into training and testing sets by randomly sampling 80% of it as the training set and the rest 20% as the testing set. We exclude all the samples of the tested operators (operator type with specific parameter combinations) from the training set to avoid data contamination in testing. Notice that due to the limited types of operators in today's DNNs, it is possible that the operators in training and testing are similar, e.g., with a similar type of operator but with different operator parameters. It is important to note that similar operators do not necessarily indicate similar optimization tasks. Through the extensive evaluation, we observe similar operators can have significantly different schedule searching traces. We design and train the cost model to effectively capture the underlying relationship between optimization tasks and profile information, allowing it to generalize well to unseen optimization tasks rather than relying on observing similar operators/tasks during offline training.

For the pre-tuned schedule database, we select the best schedules for the tasks in the training dataset and store the task attributes and the best schedules in the built-in dictionary (hash-table) in Python. We query the schedule by task key or the key with the most similar key attributes.

7.2 Dataset for the DRL Model

We perform offline training to better initialize the weight parameters in the DRL model. The training dataset used for the DRL model is similar to the dataset for the learned cost model. To show the generalization ability of the pre-trained DRL model, we test it with optimization tasks that are never seen (new operator types with new data shapes on new hardware) by the DRL model and measure how well the DRL model can generate the optimization results in the inference process.

8 EVALUATION

8.1 Evaluation Settings

We evaluate Fasor for various well-known DNNs on different hardware platforms. We compare Fasor with two representative methods: the hand-tuned libraries in PyTorch (version 1.13.0, with MKL for Intel CPU, CuDNN for NVIDIA GPU) and the SOTA automated compilation framework TVM with Ansor. Ansor is integrated into TVM, which samples programs from a hierarchical representation of the search space, and fine-tunes the sampled programs using evolutionary search with a learned cost model. We run Ansor for 1000 measurement trials for each testing case, which is the number for search convergence reported by TVM. Since our goal is to achieve effective and efficient TPO, we compare both output performance in terms of inference latency and search time. For each benchmark, we use float32 precision with batch size 1 for inference. We pre-train the cost model for CPU on Intel Xeon Ice Lake 8375C and the cost model for GPU on V100. To evaluate the online compilation results on unseen devices, we use Intel i9-9900k, Intel Xeon E5-2686 v4, and AMD EPYC 9R14 CPUs, and Nvidia GeForce GTX 2080 Ti, Nvidia A100, and Nvidia T4 GPUs. This simulates a practical scenario in which the compilation task is typically performed on new devices. For the optimization search results, we report the median of 5 runs. For the latency results, we report the average results on 1000 measurements. The reported results are normalized compared to the results of Ansor.

8.2 End-to-end DNN Evaluation

Workloads. We benchmark the end-to-end inference execution time of several well-known DNNs, including models in the computer vision domain: AlexNet [26], ResNet18 and ResNet-50 [22],

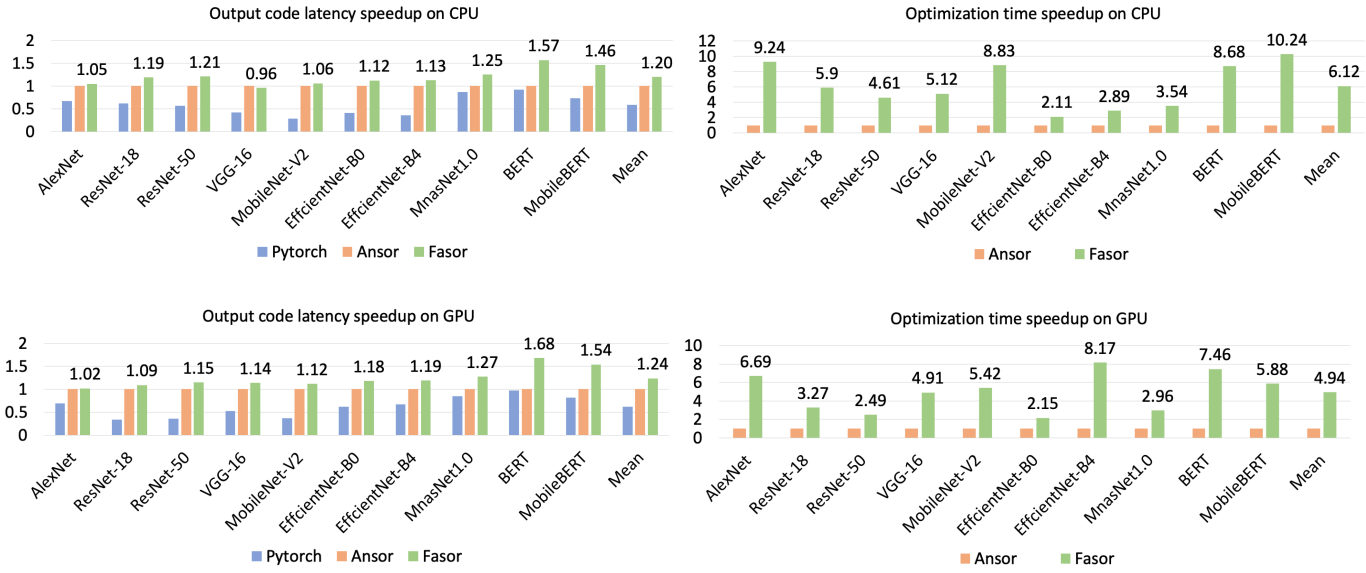


Figure 7: End-to-end evaluation on Intel i9-9900k CPU and Nvidia GeForce GTX 2080 Ti GPU (speedup normalized to Ansor).

Table 4: Output code latency (CL) and optimization time (OT) speedup compared to Ansor.

CL / OT	Intel Xeon E5-2686	AMD EPYC 9R14	Nvidia A100	Nvidia T4
ResNet-50	1.08× / 3.60×	1.22× / 4.85×	1.12× / 3.75×	1.15× / 2.98×
EfficientNet-B4	1.22× / 3.71×	1.18× / 2.98×	1.28× / 7.38×	1.30× / 5.84×
MnasNet1.0	1.18× / 3.98×	1.31× / 3.67×	1.38× / 3.93×	1.30× / 3.21×
BERT-base	1.62× / 7.79×	1.53× / 6.12×	1.35× / 9.05×	1.43× / 6.88×
MobileBERT	1.37× / 9.11×	1.45× / 6.82×	1.40× / 7.80×	1.54× / 7.25×

Table 5: Optimization time speedup compared to Tenset.

	ResNet-50	MobileNet-V2	EfficientNet-B4	BERT-base	MobileBERT
CPU	2.35×	3.67×	0.95×	4.85×	5.12×
GPU	0.83×	1.68×	1.83×	2.10×	2.33×

VGG16 [43], MobileNet-V2 [41]; NAS models: EffcientNet-B0, EfficientNet-B4 [49], and MnasNet1.0 [48]; and models in natural language processing (NLP) domain: BERT [18] and MobileBERT [46]. The results are summarized in Fig. 7 and Tab. 4.

Results. For the output code latency, Fasor achieves up to 1.57 × /3.45× speedup and 1.20 × /1.71× arithmetic mean speedup for CPU compared to Ansor / Pytorch (Fig. 7 (a)); Fasor achieves up to 1.68 × /2.94× speedup and 1.24 × /1.60× arithmetic mean speedup for GPU (Fig. 7 (c)) compared to Ansor / Pytorch. For the optimization time, Fasor achieves up to 10.24× speedup and mean 6.12× speedup for CPU (Fig. 7 (b)) compared to Ansor; Fasor achieves 8.17× speedup and mean 4.94× speedup for GPU compared to Ansor (Fig. 7 (d)). Fasor also outperforms Ansor on all DNN models and all other devices, as shown in Tab. 4. Overall, Fasor achieves comparable or better output code with significantly less optimization time compared to both hand-tuned library and tensor compiler.

Search-based metric. To evaluate the TPO efficiency improvement, we further compare Fasor with TenSet [64], which trains a cost model on $\sim 8.6M$ measurement records and fine-tunes the model on 40 online measurement records. Our cost model is trained with only 0.9M records, which is $\sim \frac{1}{10}$ of the data size in TenSet. During the online TPO stage, Fasor only requires $k = 10$ online measurement records to fine-tune the cost model. Following the settings (search-based metric) in TenSet, we fix a converged latency (the latency achieved by Ansor with 1000 trials), and compare the search time used to reach it on the Intel i9-9900k CPU / Nvidia GeForce GTX 2080 Ti GPU. We show the results in Tab. 5. Fasor achieves better results than TenSet on most settings with much higher training and fine-tuning data efficiency.

Discussions. Based on the evaluation results, we discover that (1) Fasor performs almost the best or equally the best in these optimization tasks in terms of output code latency, with a great speedup in terms of optimization time. According to our observations and analysis, there are three reasons as followings: i) the learned cost model is quickly adapted to the new device and avoids many on-device measurements and profiling but still performs accurate performance prediction thanks to the few-shot transfer learning; ii) the sub-search space selection can both significantly shrink the size of search space and remain the critical configurations without sacrificing the optimization performance. iii) our proposed DRL exploration module converges fast and searches better results thanks to the roofline model-guided reward design and the effective search start point selection. The evaluation of the effectiveness of each designed module will be detailed in Sec. 8.5.

(2) The average optimization time speedup is greater for CPU than GPU. The measurement time for the same operator or DNN is usually longer on a CPU than on a GPU. Consequently, the portion of on-device measurement time is higher for optimization on a CPU, showing more potential for improvement. Fasor significantly enhances the transfer learning efficiency and reduces the physical measurement overhead, thereby gaining more benefit from optimization for CPU. Furthermore, it demonstrates the potential for Fasor to achieve more substantial measurement time reduction on other devices that also have a significant portion of measurement time, such as mobile CPUs and FPGAs.

(3) The improvement in code latency for well-known models (e.g.,

Table 6: The time reduction (↓) breakdown into physical measurement and the search time compared to Ansor.

	CPU		GPU	
Model	measure (↓)	search (↓)	measure (↓)	search (↓)
AlexNet	90.4%	85.7%	85.6%	83.6%
ResNet-18	82.7%	85.6%	63.5%	89.4%
ResNet-50	80.2%	72.7%	56.3%	65.8%
VGG-16	84.4%	71.2%	84.4%	67.5%
MobileNet-V2	88.5%	89.2%	85.4%	67.6%
EffcientNet-B0	57.6%	46.6%	55.7%	45.3%
EffcientNet-B4	69.3%	53.5%	87.6%	88.0%
MnasNet1.0	75.4%	58.4%	66.9%	50.5%
BERT	89.7%	83.6%	87.5%	81.0%
MobileBERT	92.5%	82.3%	84.3%	75.2%
Mean	**81.1%**	**72.9%**	**75.7%**	**71.4%**

AlexNet, VGG16, ResNet-18) is relatively small, as existing frameworks (Pytorch, TVM) have already performed comprehensive optimization for these models. However, the speedup for NAS models and NLP models is relatively larger, indicating that Fasor has the potential to achieve better optimization quality on relatively new DNNs.

8.3 Transferring and Sampling Efficiency

As the main design principles, our goal is to improve both the transferring efficiency of the cost model and the sampling efficiency of the searching method. We evaluate the time reduction of on-device measurement and time reduction on searching when achieving the results in Fig. 7 and show the breakdown results in Tab. 6.

Transferring efficiency As mentioned in Sec. 2 and Tab. 2, the on-device measurement is one of the bottlenecks of TPO time, usually taking over 60% of the total compilation time. We compare the time reduction between Fasor's transfer learning and Ansor's cost model in Tab. 6. Through the few-shot transfer learning, Fasor quickly adapts to a new hardware backend with very few on-device measurements (on k samples as introduced in Sec. 4). Fasor gains on average 81.1%/75.7% time reduction on CPU / GPU in physical measurement during the online optimization stage, efficiently solving the bottleneck of TPO.

Sampling efficiency With our designed DRL exploration engine and the wise choice of search start point, Fasor is able to explore the search space in very few iterations and reduce on average 72.9%/71.4% search time on CPU / GPU.

8.4 Results of high-quality mode

As mentioned in Sec. 6, we also provide a *high-quality* mode, which enables the search engine to search the optimal results until convergent or until consuming all the optimization time budget. We also evaluate the high-quality mode on the Intel i9-9900k CPU and Nvidia GeForce GTX 2080 Ti GPU and show the results in Tab. 7. The results show that, with more searching iterations, Fasor is able to search output code that is on average 1.41 × /1.43× better than Ansor, with on average 2.89 × /2.66× faster searching than Ansor, for CPU /GPU. Compared to the default *high-efficiency* mode, the *high-quality* mode improves the output code latency by 1.1×/1.15× on CPU / GPU with relatively moderate optimization time speedup. Both modes of Fasor achieve a good balance between the quality and efficiency of TPO.

Table 7: The output latency and optimization time improvement (↑) of the Fasor 'high-quality' mode compared to Ansor.

	CPU		GPU	
Model	output latency (↑)	optimization time (↑)	output latency (↑)	optimization time (↑)
AlexNet	1.25×	5.31×	1.27×	4.35×
ResNet-18	1.36×	2.15×	1.41×	1.59×
ResNet-50	1.38×	1.62×	1.30×	1.62×
VGG-16	1.25×	2.34×	1.25×	3.01×
MobileNet-V2	1.16×	4.01×	1.34×	2.54×
EffcientNet-B0	1.35×	1.15×	1.29×	1.27×
EffcientNet-B4	1.45×	1.47×	1.52×	4.22×
MnasNet1.0	1.57×	2.19×	1.58×	1.37×
BERT	1.76×	4.63×	1.71×	3.43×
MobileBERT	1.58×	3.75×	1.58×	3.15×
Mean	**1.41**×	**2.89**×	**1.43**×	**2.66**×

8.5 Effectiveness on Each Design Component

We evaluate each module by each time replacing one module with a baseline method in the default Fasor, shown in Fig. 8. The default Fasor consists of three modules: (1) the learned cost model with few-shot transfer learning; (2) the sub-search space module with the number of tunable primitives $n_p = 7$; (3) the DRL exploration module with the designed reward function and pre-tuned schedule as the search start point.

8.5.1 Effectiveness of the Cost Model and Transfer Learning. **Model accuracy:** We evaluate the prediction accuracy of the pre-trained cost model on the testing set with the coefficient of determination (R^2) [36]. The closer the value of R^2 is to 1, the better the linear regression fits the data. We compare our Transformer-based model with the XGBoost-based model in Ansor and a two-layer LSTM model [61]. Results are shown in Tab. 8 that our model outperforms other models.

Transfer learning accuracy: We show the results of transferring knowledge of the pre-trained model trained on one device to three other devices in Tab. 9. The results show that the general knowledge of evaluating tensor programs can be transferred well, and the hardware-specific knowledge can be learned well via the few-shot transfer learning, with tolerable performance drops in terms of R^2.

Transfer learning efficiency: We evaluate the effect of the few-shot transfer learning by replacing it with training from scratch (Fasor_wo_TL in Fig 8). Fasor_wo_TL shows lightly better optimization quality trading optimization time off, which has 3.6× longer optimization time compared to the default Fasor.

The above-shown results show that the few-shot transfer learning adopted by Fasor significantly reduces the online optimization time and adapts the cost model well over various hardware to perform accurate cost prediction.

Table 8: Comparison among different choices of the cost model.

R^2	Fasor	XGBoost [16]	LSTM [8]
CPU	**0.972**	0.923	0.920
GPU	**0.959**	0.914	0.908

8.5.2 Effectiveness of the Sub-search Space Selection Module. We change the number of tunable primitives n_p from 5 to 8 (Fasor (n_p=i) in Fig. 8) to show Fasor can achieve the best of both worlds in the quality and the efficiency of TPO. n_p controls the optimization quality and efficiency trade-off. A larger n_p(=7, 8) leads to more on-device measurements but expects a better optimization quality.

Table 9: Pre-training (PT) and transfer learning (TL) accuracy.

	CPU	R^2	GPU	R^2
PT	Intel Xeon Ice Lake 8375C	0.972	NVIDIA V100	0.959
TL_1	Intel i9-9900K	0.951	NVIDIA GeForce RTX 2080Ti	0.939
TL_2	Intel Xeon E5-2686 v4	0.931	Nvidia A100	0.927
TL_3	AMD EPYC 9R14	0.899	Nvidia T4	0.915

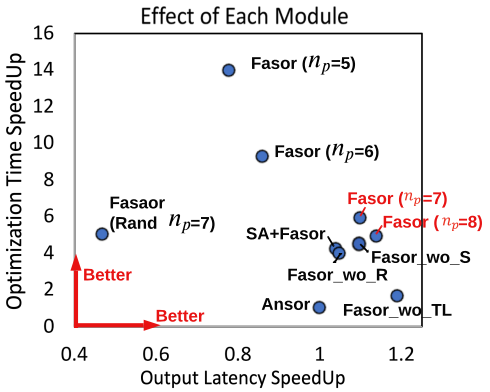


Figure 8: Evaluation results of Fasor variants (ablation study). Fasor ($n_p = 7$ and $n_p = 8$ achieve the best trade-off in terms of optimization quality and efficiency. The results are the speedup in terms of output code latency (optimization quality) and optimization time (optimization efficiency) for the ResNet-18 optimization task on Intel i9-9900k CPU and are normalized to Ansor. Similar conclusions also hold in other optimization tasks.

A smaller n_p(=5, 6) could be considered in the circumstance that requires frequent and fast compilation to achieve high efficiency (over 10× speedup). We evaluate the effect of the selection module by replacing the selection method by randomly selecting n_p=7 primitives to be tuned (Fasor (Rand n_p=7)). We can figure out a random selection suffers a great degradation in output code latency.

8.5.3 Effectiveness of the DRL Exploration Module. **Reusing pre-tuned schedule** We evaluate the effect of using the pre-tuned schedules with similar optimization task configurations. Reusing pre-tuned schedules provides a good search start point, which helps the search converge within fewer trials and improve the sampling efficiency, outperforming using the TVM default start point (Fasor_wo_S in Fig. 8).

DRL search engine We evaluate the effect of the DRL exploration module by replacing the DRL searching by **S**imulated **A**nnealing (SA) search strategy (SA+Fasor). Fasor (n_p=7) and (n_p=8) both outperform SA+Fasor in terms of optimization quality and efficiency. This is because SA relies on the stochastic property to guarantee a reasonable solution after a great number of iterations. Meanwhile, the DRL model is able to capture the correlation between different tasks and configurations and reuse information during search iterations. Thus, DRL improves both optimization quality and efficiency.

Roofline model-based reward We evaluate the effect of roofline model-guided reward by removing the second term in Eq. 3. Similarly, Fasor (n_p=7) and (n_p=8) both outperform (Fasor_wo_R). It shows that the roofline model-guided reward provides more hints on optimization exploration direction than just considering latency.

9 RELATED WORK

Prior studies on TPO can be mainly classified into two folds: (1) hand-optimized libraries and (2) DNN compilers.

In the first class, vendors provide kernels dedicated to their hardware. On CPUs, the high-performance mathematical library MKL [53] is designed to accelerate linear algebra applications. MKL-DNN [3] is designed to optimize CNN on Intel Xeon CPUs. CuBlas [1] is designed to accelerate linear algebra kernels and CuDNN [17] is designed for deep learning applications with state-of-the-art efficient algorithms such as Winograd [27] and FFT[31] for NVIDIA GPUs. Most deep learning frameworks [5, 14, 38] rely on these libraries to achieve high performance. However, all these libraries require the manual design of the high-performance implementation, demanding experience, and expertise in both algorithms and hardware.

In the second class, to eliminate the issues of long developing time and heavy labor effort, DNN compiler stacks [15, 39] are designed to provide different levels of abstraction and automate the compilation of DNN workloads. Halide auto-scheduler [39] leverages automatic tree searching but it is focused on code for image processing pipelines. Tensor Comprehensions [50] uses polyhedral model [51] to automatically optimize algorithms, but it is limited to a narrow range of hardware and only achieves limited performance speedup. AutoTVM [16] leverages a learned cost model in searching. Ansor [63] proposes a hierarchical representation of the search space. Flextensor [65] aims for heterogeneous systems. However, none of these code generation frameworks focus on improving compilation efficiency. Different from these frameworks, Fasor considers more critical profiling information and achieves the best compilation quality and efficiency trade-off.

10 CONCLUSION

Upon studying the bottleneck of DNN deployment, we have identified two essential principles for developing tensor program optimization (TPO): transferring efficiency and sampling efficiency. To address these principles, we propose Fasor, an end-to-end machine learning-based framework that achieves an optimal trade-off between TPO quality (latency time) and efficiency (compilation time). By leveraging an offline pre-trained cost model, Fasor utilizes few-shot transfer learning to reduce the cost of frequent hardware measurements and profiling and obtains a more efficient cost model that accurately predicts critical profile information online. Furthermore, Fasor selects a sub-search space for efficient searching and uses deep reinforcement learning to output optimal solutions in very few steps. Our experimental results demonstrate that Fasor achieves competitive performance compared to state-of-the-art TPO frameworks for CPUs and GPUs, significantly reducing optimization time for efficient DNN deployment. We leave generalizing Fasor over more target hardware such as mobile and edge devices, and other tensor compiler frameworks as future work.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable feedback and comments. This paper is supported in part by NSF grants 1829524, 1817077, 2011212, and the PRISM center in JUMP 2.0, an SRC program sponsored by DARPA.

REFERENCES

[1] [n. d.]. CuBLAS: Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.

[2] [n. d.]. Intel PMU profiling tools. https://github.com/andikleen/pmu-tools.

[3] [n. d.]. Intel(R) Math Kernel Library for Deep Neural Networks (Intel(R) MKL-DNN). https://github.com/rsdubtso/mkl-dnn.

[4] [n. d.]. The Performance Application Programming Interface (PAPI). https://tvm.apache.org/docs/how_to/profile/papi.html.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[6] 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), 12 pages. https://doi.org/10.1145/3306346.3322967

[7] 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, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=rygG4AVFvH

[8] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. 2021. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems* 3 (2021), 181–193.

[9] 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 *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[10] Hugues Berry, Daniel Gracia Pérez, and Olivier Temam. 2006. Chaos in computer performance. *Chaos: An Interdisciplinary Journal of Nonlinear Science* 16, 1 (Mar 2006), 013110. https://doi.org/10.1063/1.2159147

[11] Jun Bi, Qi Guo, Xiaqing Li, Yongwei Zhao, Yuanbo Wen, Yuxuan Guo, Enshuai Zhou, Xing Hu, Zidong Du, Ling Li, et al. 2023. Heron: Automatically constrained high-performance library generation for deep learning accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 314–328.

[12] Jun Bi, Xiaqing Li, Qi Guo, Rui Zhang, Yuanbo Wen, Xing Hu, Zidong Du, Xinkai Song, Yifan Hao, and Yunji Chen. 2022. BALTO: fast tensor program optimization with diversity-based active learning. In *The Eleventh International Conference on Learning Representations*.

[13] Han Cai, Ligeng Zhu, and Song Han. 2018. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332* (2018).

[14] 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. *arXiv preprint arXiv:1512.01274* (2015).

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[16] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems* 31 (2018).

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

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

[19] Perry Gibson and José Cano. 2022. Transfer-tuning: Reusing auto-schedules for efficient tensor program code generation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 28–39.

[20] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 249–256.

[21] Ameer Haj-Ali, Nesreen K Ahmed, Ted Willke, Yakun Sophia Shao, Krste Asanovic, and Ion Stoica. 2020. Neurovectorizer: End-to-end vectorization with deep reinforcement learning. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 242–255.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[23] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. 2020. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the Association for Computational Linguistics* 8 (2020), 64–77.

[24] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU cost model for XLA tensor programs. In *Proc. Workshop ML Syst. NeurIPS*. 1–6.

[25] Diederik Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*. San Diega, CA, USA.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[27] Andrew Lavin and Scott Gray. 2016. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4013–4021.

[28] Hayeon Lee, Eunyoung Hyung, and Sung Ju Hwang. 2021. Rapid neural architecture search by learning to generate graphs from datasets. *arXiv preprint arXiv:2107.00860* (2021).

[29] Guihong Li, Sumit K Mandal, Umit Y Ogras, and Radu Marculescu. 2021. FLASH: F ast Neura l A rchitecture S earch with H ardware Optimization. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–26.

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[31] Michael Mathieu, Mikael Henaff, and Yann LeCun. 2013. Fast training of convolutional networks through ffts. *arXiv preprint arXiv:1312.5851* (2013).

[32] Hiroaki Mikami, Hisahiro Suganuma, Yoshiki Tanaka, Yuichi Kageyama, et al. 2018. Massively distributed SGD: ImageNet/ResNet-50 training in a flash. *arXiv preprint arXiv:1811.05233* (2018).

[33] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[34] Thierry Moreau, Tianqi Chen, and Luis Ceze. 2018. Leveraging the VTA-TVM Hardware-Software Stack for FPGA Acceleration of 8-Bit ResNet-18 Inference. In *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning* (Williamsburg, VA, USA) *(ReQuEST '18)*. Association for Computing Machinery, New York, NY, USA, Article 5. https://doi.org/10.1145/3229762.3229766

[35] Leann Myers and Maria J Sirois. 2004. Spearman correlation coefficients, differences between. *Encyclopedia of statistical sciences* 12 (2004).

[36] Nico JD Nagelkerke et al. 1991. A note on a general definition of the coefficient of determination. *Biometrika* 78, 3 (1991), 691–692.

[37] Nvidia. 2023. An Order-of-Magnitude Leap for Accelerated Computing. https://www.nvidia.com/en-us/data-center/h100/

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[39] 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. *SIGPLAN Not.* 48, 6 (jun 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[40] 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*. 89–103.

[41] 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*. 4510–4520.

[42] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484–489.

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

[44] Peng Sun, Wansen Feng, Ruobing Han, Shengen Yan, and Yonggang Wen. 2019. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *arXiv preprint arXiv:1902.06855* (2019).

[45] Zihao Sun, Yu Hu, Longxing Yang, Shun Lu, Jilin Mei, Yinhe Han, and Xiaowei Li. 2021. STC-NAS: Fast Neural Architecture Search with Source-Target Consistency.

Neurocomputing (2021).

[46] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984* (2020).

[47] Akihiro Tabuchi, Akihiko Kasagi, Masafumi Yamazaki, Takumi Honda, Masahiro Miwa, Takashi Shiraishi, Motohiro Kosaki, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, et al. [n. d.]. Extremely Accelerated Deep Learning: ResNet-50 Training in 70.4 Seconds. ([n. d.]).

[48] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2820–2828.

[49] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[50] 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 preprint arXiv:1802.04730* (2018).

[51] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–23.

[52] Gaurav Verma, Siddhisanket Raskar, Murali Emani, and Barbara Chapman. 2024. Cross-Feature Transfer Learning for Efficient Tensor Program Generation. *Applied Sciences* 14, 2 (2024), 513.

[53] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[54] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[55] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[56] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. 2019. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 1284–1293.

[57] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. 2019. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650* (2019).

[58] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 35–44.

[59] Cody Hao Yu, Xingjian Shi, Haichen Shen, Zhi Chen, Mu Li, and Yida Wang. 2021. Lorien: Efficient Deep Learning Workloads Delivery. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/3472883.3486973

[60] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 833–845.

[61] Jiachen Zhao, Fang Deng, Yeyun Cai, and Jie Chen. 2019. Long short-term memory-Fully connected (LSTM-FC) neural network for PM2. 5 concentration prediction. *Chemosphere* 220 (2019), 486–492.

[62] Zhihe Zhao, Xian Shuai, Neiwen Ling, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2023. Moses: Exploiting Cross-Device Transferable Features for on-Device Tensor Program Optimization. In *Proceedings of the 24th International Workshop on Mobile Computing Systems and Applications*. 22–28.

[63] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 863–879.

[64] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. 2021. 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)*.

[65] 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*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508