BIG DATA MINING AND ANALYTICS
ISSN 1007-0214
DOI: 10.26599/BDMA.2026.9020025
Volume x, Number x, xxxxxxx 20xx

FusionAdvisor: Decision Guidance for Efficient Operator Fusion in Deep Models

Chang Jiang, Yanan Yang, Shen Gao, Jianjiang Li*, and Jie Wu

Abstract: As deep learning models grow in size and complexity, optimizing execution efficiency becomes critical. Operator fusion merges multiple operators to reduce memory overhead and enhance computational efficiency. However, the combinatorial explosion of possible fusion combinations in modern models creates a search space that challenges traditional manual and heuristic methods. We propose FusionAdvisor, a neural network–based framework that predicts Top-k fusion strategies from computation graphs. Trained on our constructed fusion strategy dataset, it abstracts fusion selection as a service, enabling developers to choose high-performance strategies and accelerate inference efficiently. Experimental results demonstrate that FusionAdvisor's neural network predicts post-fusion speedups with low error, enabling promising strategy recommendations while maintaining search efficiency comparable to existing methods. This work advances automated operator fusion selection, reduces optimization costs, and improves resource utilization in complex scenarios, including large language models (LLMs) and other large-scale models.

Key words: deep learning; operator fusion; neural network prediction; fusion strategy recommendation

1 Introduction

With the widespread application of deep learning models in natural language processing, computer vision, and recommendation systems, both the size of the model and the architectural complexity have increased dramatically, placing greater demands on computational resources and execution efficiency[1]. This trend is particularly pronounced in the training and deployment of large language models (LLMs). For example, TeleChat[2], a large language model with a hundred billion parameters developed by China Telecom, supports the recognition of over 30 Chinese dialects, including Cantonese, Shanghainese, and Sichuanese. It also demonstrates strong performance in long-text comprehension and multi-turn dialog tasks.

As the complexity of the model increases, the number of operators executed during a single inference can exceed hundreds or even thousands. Consequently, the resulting overhead from operator invocations and memory transfers significantly increases latency and resource consumption. Optimus[3] demonstrated that inter-operator data transfers, primarily driven by frequent off-chip memory accesses, consume more than 80% of the total energy consumption in deep neural network (DNN) accelerators, making them the dominant contributor to overall power consumption.

To accelerate model inference, various optimization techniques have been proposed. Among these, operator fusion has emerged as a critical graph-level strategy to improve inference efficiency. Merging related operators into composite ones reduces memory transfers and

• Chang Jiang and Jianjiang Li are with the Department of Computer Science and Technology, University of Science and Technology Beijing, Haidian, Beijing 100083, China. E-mail: jc@xs.ustb.edu.cn; lijianjiang@ustb.edu.cn.
• Yanan Yang and Shen Gao are with the China Telecom Cloud Computing Research Institute, Beijing 100080, China. E-mail: yangyn11@chinatelecom.cn; gaos13@chinatelecom.cn.
• Jie Wu is with the China Telecom Cloud Computing Research Institute, Beijing 100080, China, and also with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu.
Chang Jiang and Yanan Yang contributed equally to this work.
* To whom correspondence should be addressed.
Manuscript received: 2025-08-28; accepted: 2026-04-30

kernel launches. For example, TVM[4] achieves a speedup of up to 1.2×–2× through fewer memory accesses, while DNNFusion[5] outperforms state-of-the-art frameworks by up to 9.3× across various DNNs. Despite these advantages, existing operator fusion methods often rely on predefined patterns or heuristic search strategies, which struggle to scale efficiently as the complexity of modern DNN architectures increases.

The limitations of current fusion methods arise from the inherent complexity of the fusion search space. The vast number of potential fusion combinations makes it impractical to enumerate all viable options. Selecting effective operator fusion strategies involves a large search space[6, 7] and has been proven to be NP-hard[8]. For operator developers, manually applying fusion strategies without a comprehensive understanding of their effectiveness often requires extensive trial and error. This process can lead to invalid or performance-degrading fusion schemes, resulting in a significant waste of time and effort. For application developers, constructing inference models for real-world deployment involves selecting appropriate individual or fused operators from the operator library to meet performance and resource constraints. This task also presents the challenge of choosing suitable fusion strategies. In this work, we aim to develop an efficient method for predicting the impact of fusion strategies on inference performance, thereby reducing the costs associated with exhaustive exploration.

Optimization of directed acyclic graphs (DAGs) or computational workflows using artificial intelligence or machine learning techniques has been validated in previous studies, motivating us to explore similar approaches for operator fusion. Neural networks can learn expressive representations of computation graphs and capture complex interactions among operators. In this paper, we propose a neural network-based method to predict the performance impact of different fusion strategies by using features extracted from the computation graph. By learning to estimate the fusion results offline, our method facilitates faster and more scalable fusion optimization. However, this approach faces three key challenges: (i) the lack of public datasets linking fusion strategies to their measured performance, which requires systematic profiling of operators across diverse models; (ii) designing an effective prediction model structure and selecting appropriate features for training; (iii) efficiently navigating the combinatorial fusion search space to quickly surface the Top-k most promising strategies without exhaustive exploration.

To address these challenges, we introduce FusionAdvisor, a novel framework that leverages empirical datasets and neural network predictions to support fusion selection. FusionAdvisor is not a general compiler or scheduler and does not create new fused kernels. It serves as a fusion-strategy evaluator and recommender by scoring candidate fusion paths and returning a ranked Top-k list to guide subsequent selection and validation. First, we develop a data collection pipeline that profiles a wide range of operator fusion strategies across multiple deep learning models, resulting in a dataset that links fusion strategies to their measured runtime impacts. Second, we extract and engineer features from each fusion strategy, including the operator combination structure and tensor shape characteristics, and train a lightweight Multilayer Perceptron (MLP) to estimate the speedup benefits of different fusion configurations. Third, we implement a ranking engine that surfaces the Top-k highest-scoring strategies. Together, these components lower the cost of operator fusion decision making by providing predictive guidance to prioritize candidate strategies.

Our contributions can be summarized as follows:

• We propose a neural network–based operator fusion strategy evaluation model that uses post-fusion speedup and structural operator-combination features to predict the performance of fusion combinations across various deep learning architectures.

• We construct a fusion strategy dataset to train and validate our evaluation model. This dataset systematically captures the input and output tensor shapes of each fusion path, execution sequences, measured post-fusion speedups, and resulting performance gains.

• We develop a decision-making framework for fusion strategy recommendation that leverages predictions from our evaluation model to dynamically maintain a Top-k pool of candidate strategies. Experiments demonstrate that the framework can prioritize promising fusion strategies.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents an overview of the framework. Section 4 describes dataset construction and feature engineering. Section 5 describes the core methods of FusionAdvisor. Section 6 presents the implementation details. Section 7 presents

and analyzes the experimental results. Finally, Section 8 concludes the paper.

2 Background and Related Work

Operator fusion is a fundamental optimization technique in deep learning that aims to enhance execution performance by improving the locality of intermediate tensor data and reducing the frequency of memory access. This section reviews existing research on operator fusion, including manual or rule-based methods, heuristic-based approaches, learning-based techniques, and performance prediction strategies, along with brief comparisons to our approach.

2.1 Manual/rule-based fusion methods

Early operator fusion techniques primarily rely on manually defined patterns or compiler rules. XLA[9], an acceleration library integrated with TensorFlow[10], performs pattern matching during compilation to recognize common subgraph structures, such as Conv2D + BiasAdd + ReLU, and generates the corresponding fused operators. TensorRT[11] employs NVIDIA-specific rules for graphics processing unit (GPU) optimization. TVM[4] adopts general pattern-based fusion rules, yet their limited coverage of complex operator combinations may restrict optimization in large-scale or intricate workloads. Although rule-based fusion is efficient and robust for common operator patterns, it remains limited in its ability to explore the full combinatorial fusion space of modern deep learning models.

2.2 Heuristic-based fusion methods

These methods use hand-tuned heuristics to explore fusion spaces, offering improvements over rule-based approaches while still relying on manual search logic. DNNFusion[5] classifies operators through element-wise mapping and applies heuristic rules to fuse cross-type operators. Similarly, Optimus[3] balances fusion granularity with a heuristic memory model. MetaFlow[12] and TASO[13] adopt cost-based graph rewriting with automatically generated substitutions and backtracking-style search to explore equivalent computation graphs. Since the transformation space expands with graph size, the search process may incur higher overheads in very large models. In graph optimization, OCGGS[8] uses graph replacement techniques to automatically identify and substitute inefficient subgraphs, thereby reducing redundancy. It employs pruning-based dynamic programming for the fusion search and formalizes graph optimization through replacements, thereby expanding the fusion space explored by rule-based methods. Apollo[14] performs the fusion of adjacent layer operators through a layer-by-layer process, which reduces computational overhead. Its optimization process incorporates feedback from intermediate performance metrics, making it more flexible than purely rule-based frameworks. Yang et al.[15] propose an evolutionary search method that employs binary vector encoding to efficiently explore operator fusion groupings, but must contend with the exponential search space of possible fusion configurations. Although these methods improve upon purely rule-based fusion, their decisions are largely driven by predefined heuristics or costly compilation and profiling. As model graphs grow, evaluating numerous fusion alternatives becomes increasingly expensive. In contrast, FusionAdvisor uses a learned speedup predictor to efficiently score fusion candidates, providing data-driven guidance for exploring large fusion spaces while reducing reliance on per-candidate compile-and-profile feedback.

2.3 Learning-based fusion methods

AutoTVM[16] and Ansor[17] are representative learning-based optimization frameworks; however, they primarily focus on low-level schedule generation rather than on graph-level fusion strategy selection. AutoTVM mainly performs learning-guided searches for operator-level schedules (e.g., tiling, vectorization, and parallelization), while Ansor optimizes tensor program generation through hierarchical search spaces and evolutionary strategies. Their learning components are designed to discover efficient low-level schedules for a given compute definition, rather than explicitly exploring and ranking alternative graph fusion partitions. In contrast, FusionAdvisor focuses on evaluating and recommending fusion strategies. We construct a fusion-specific dataset and train a lightweight predictor to estimate the speedup impact of candidate fusion paths, enabling fast scoring and Top-k recommendations. This naturally connects fusion strategy evaluation with performance prediction, which provides fast runtime estimates for scoring candidates.

2.4 Performance prediction for DNN execution

Accurate prediction of performance for DNN execution is crucial for optimizing fusion strategies and improving

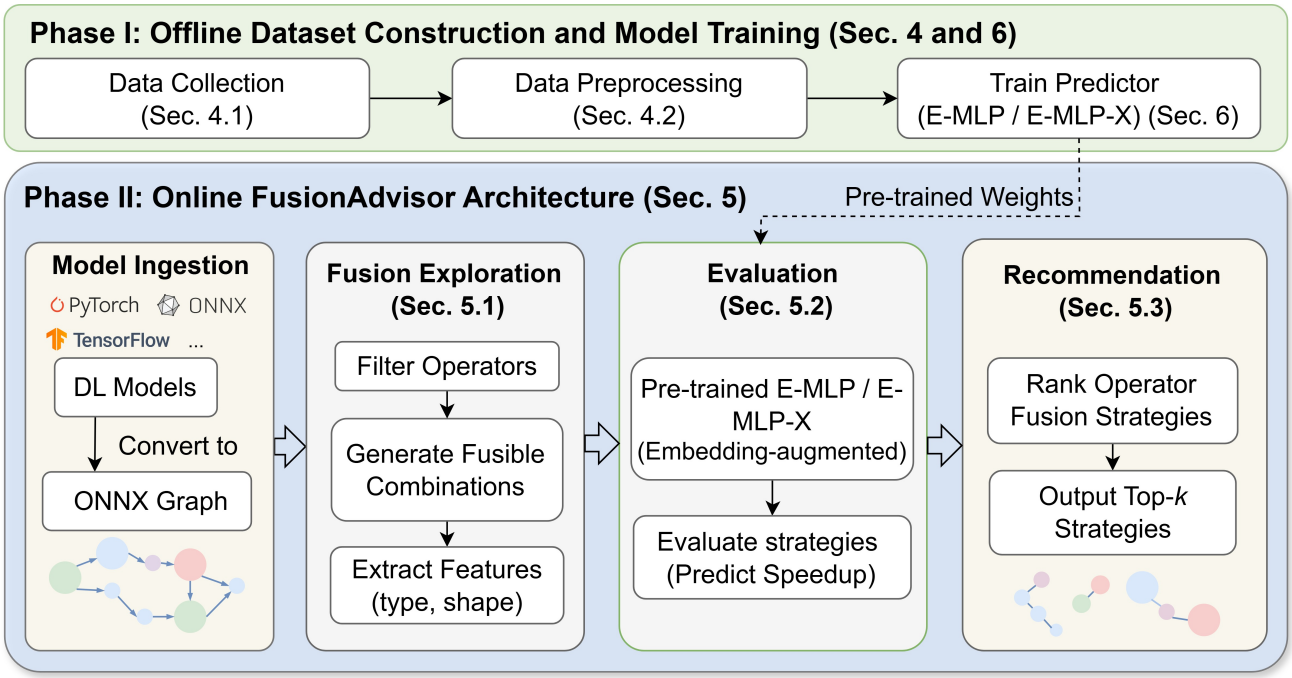


Fig. 1 FusionAdvisor architecture.

the overall efficiency of deep learning models. Several studies have explored techniques for predicting the execution performance of DNNs. In NeuSight[18], the authors propose a GPU performance prediction framework that integrates hardware-level behaviors with library implementation characteristics to estimate the computational and memory efficiency of DNN workloads. Similarly, Disco[19] discusses optimizing DNN compilation for distributed environments, combining operator and tensor fusion to reduce communication overhead and improve distributed training performance. Although these approaches provide valuable global performance estimates, they are not explicitly designed to explore alternative fusion strategies.

3 Framework Overview

FusionAdvisor is a data-driven framework designed to optimize operator fusion in deep learning models; it systematically explores fusion strategies within computational graphs and recommends Top-k candidate fusion schemes through predictive analytics. As illustrated in Fig. 1, the framework follows an online pipeline of model ingestion, fusion exploration, evaluation, and recommendation.

First, FusionAdvisor adopts a framework-agnostic approach by accepting deep learning models from major frameworks, including PyTorch[20] and TensorFlow[10], which are converted into the Open Neural Network Exchange (ONNX) format[21]. This conversion abstracts models into computational graphs that preserve operator semantics, data dependencies, and structural constraints, facilitating consistent processing across diverse architectures. By using ONNX as a unified intermediate representation, FusionAdvisor can apply the same analysis and feature-extraction pipeline across frameworks that support ONNX export.

Second, the fusion exploration module constructs a search space of valid candidates for fusion. Guided by domain-specific rules for fusion legality (adapted from previous work[5]), it utilizes dynamic programming to identify maximal sequences of fusible operators within the computational graph. This approach aims to cover common fusion opportunities while reducing redundant combinations.

Third, the evaluation module employs our Embedding-augmented Multilayer Perceptron (E-MLP), a lightweight neural predictor tailored to estimate the speedup benefits of operator fusion strategies. It represents operator sequences using learnable embeddings of operator identifiers combined with a fixed sinusoidal positional encoding, enabling order-aware scoring of fusion candidates. Finally, the recommendation module maintains a Top-k ranking of

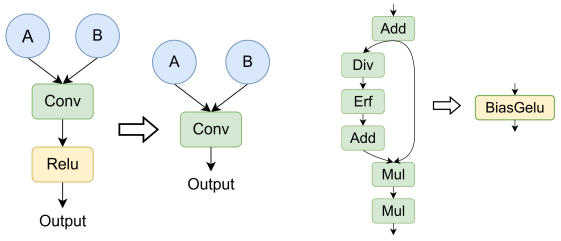
candidate strategies using a priority queue, enabling efficient selection of promising fusion policies.

4 Dataset Construction

To construct a dataset for training our operator fusion prediction model, we followed a two-stage pipeline. First, we executed deep learning models to collect pre-fusion execution graphs, individual and fusion group execution times. We then applied various fusion strategies to collect post-fusion data. In the second stage, the collected data were preprocessed for model training.

4.1 Data collection

Existing resources do not provide per-strategy post-fusion speedup labels in a form suitable for training our predictor, so we construct our dataset using ONNX Runtime[22] and ONNX Model Zoo[23]. We obtain models from the ONNX Model Zoo and execute inference with ONNX Runtime. The process involves three main steps: first, configuring the graph optimization level (e.g., ORT_ENABLE_EXTENDED) to trigger automatic operator fusion; then, initializing an InferenceSession with both Compute Unified Device Architecture (CUDA) and Central Processing Unit (CPU) execution providers; and finally, exporting the optimized graph for further analysis.



(a) Simple operator fusion example (b) Changes in the DistilBERT model

Fig. 2 Illustrations of operator fusion effects.

After inference optimization, the fusion of certain operators combines multiple operators into a single new operator, reducing the number of operators in the computation graph and accelerating computation. As illustrated in Fig. 2(a), a simple case shows Conv and Relu fused into a single Conv operator. Moreover, Fig. 2(b) shows a real case of operator fusion in the DistilBERT model after optimization, where the fusion of Add, Div, Erf, and Mul results in a new operator, BiasGelu.

To identify fused operators, we compare the computation graphs before and after optimization, focusing on the reduction in operator count and the emergence of fused node clusters. By analyzing these changes, we can accurately determine which operators were fused during the optimization process. Based on our analysis of the ONNX Model Zoo models, we identified 11 common operator fusion strategies, their corresponding operator combinations, and representative models that exhibit these strategies after optimization (Table 1).

Enabling ONNX Runtime's profiling feature generates a JavaScript Object Notation (JSON) file that contains detailed performance metrics, such as operator names and individual execution times, as illustrated in Table 2. Based on the analyzed fusion strategies, we extract the execution times before and after the fusion of the corresponding operator groups from the profile data. Following common practice in operator fusion evaluation[5], we quantify the performance improvement using the speedup:

$$\text{Speedup} = \frac{T_{\text{before}}}{T_{\text{after}}} \tag{1}$$

where T_{before} and T_{after} denote the execution times before and after operator fusion, respectively. In addition, we record the maximum input shape for each group and the output shape of the final operator. To ensure reliability, the model is executed multiple times. After discarding data from the warm-up phase, we average the remaining execution times to derive speedup metrics across various fusion strategies and input and output configurations.

A table is constructed where each row represents a sample, with columns including: Pattern, Fused_Operators, Input_Shape, Output_Shape, Time_before, Time_after, and Speedup. Each row corresponds to a specific operator fusion strategy, as illustrated in Table 3. Specifically, the Pattern column represents the operator fusion strategy. The Fused_Operators column denotes the set of operators that are fused. For example, when Conv and Sigmoid are fused, Fused_Operators = [Conv, Sigmoid], and when Sigmoid and Mul are fused, Fused_Operators = [Sigmoid, Mul]. The Input_Shape column indicates the largest input shape among the operators in the fusion group. This is computed by iterating over the nodes in the fusion path and determining the maximum input shape across all nodes that can be fused. Specifically, for a given node i, if its input shape is not $None$, we initialize Input_Shape to the input shape of the current

Table 1 Operator fusion strategies and their corresponding combinations in ONNX models.

Fusion Strategy	Operator Combinations	Models
Conv	Conv + Mul; Conv + Mul + Add + Relu; Conv + Add; Conv + BatchNormalization (BN); Conv + BN + Relu; Conv + BN + Add + Relu	MNIST, ResNet-18, YOLOv4
FusedConv	Conv + Activation; Conv + Add + Activation (Relu, Sigmoid, Tanh, LeakyRelu, Clip)	SSD
Gelu	Div + Erf + Add + Mul + Mul	SqueezeBERT
BiasGelu	Add + Div + Erf + Add + Mul + Mul	DistilBERT
FastGelu	Pow + Mul + Add + Tanh + Add + Mul	OpenAI-GPT
QuickGelu	Sigmoid + Mul	EfficientNet-B0
FusedGemm	Gemm + Activation (LeakyRelu, Relu, Sigmoid, Tanh)	SqueezeBERT
FusedMatMul	MatMul + Div; MatMul + Mul; MatMul + Transpose	CAiT, SqueezeBERT
LayerNormalization	ReduceMean + Sub + Pow + ReduceMean + Add + Sqrt + Div + Mul + Add	DistilBERT
SkipLayerNormalization	Add + LayerNormalization	BERT, DistilBERT
SimpLayerNormalization	Pow + ReduceMean + Add + Sqrt + Div + Mul	T5-Encoder

Table 2 Main JSON fields in ONNX Runtime profiling output.

Field	Description	Example
cat	Event category	”Node”
dur	Duration of the event (microseconds)	5
name	Event name (operator name)	”/conv_stem/Conv_fence_before”
args	Additional arguments	{”op_name”: ”Conv”}
input_type_shape	List of input types and shapes	[”float”:[1,3,224,224]]
output_type_shape	List of output types and shapes	[”float”:[1,32,112,112]]

Table 3 Examples of dataset samples.

Pattern	Fused_Operators	Input_Shape	Output_Shape	Time_before	Time_after	Speedup
ConvRelu	Conv, Relu	1, 196, 13, 13	1, 384, 13, 13	1777	1425	1.25
ConvRelu	Conv, Relu	1, 256, 13, 13	1, 256, 13, 13	1760	1432	1.23
ConvRelu	Conv, Relu	1, 3, 224, 224	1, 64, 55, 55	1497	827	1.81
QuickGelu	Sigmoid, Mul	1, 10, 1, 1	1, 10, 1, 1	35	17	2.02
QuickGelu	Sigmoid, Mul	1, 1152, 7, 7	1, 1152, 7, 7	60	36	1.67

node:

$$S_i = \begin{cases} \text{input\_shape}[i], & \text{if input\_shape}[i] \neq \text{None} \\ \text{None}, & \text{otherwise} \end{cases} \tag{2}$$

For each subsequent node j that can be fused with node i, Input_Shape is updated by taking the dimension-wise maximum between the current Input_Shape and the input shape of node j. If node j has a valid input shape, the update is as follows:

$$S_j[k] = \max\left(S_i[k], \text{input\_shape}[j][k]\right), \forall k \in [1, \ldots, n] \tag{3}$$

where S_i denotes the selected input shape for node i, $S_j[k]$ denotes the k-th dimension of the updated input shape for node j, i and j are node indices in the fusion path, k indexes the tensor dimension, and n is the number of dimensions in the input shape. This process ensures that the maximum input shape reflects the largest possible size for each dimension across all nodes involved in the fusion. Output_Shape corresponds to the output shape of the final operator in the combination. The Time_before column represents the execution time before optimization, Time_after refers to the execution time after optimization, and Speedup denotes the acceleration factor achieved by fusion.

4.2 Data preprocessing

The feature engineering pipeline encodes operator sequences using a two-stage approach that combines integer encoding and embeddings. First, each ONNX operator[24] is assigned a unique integer code based on

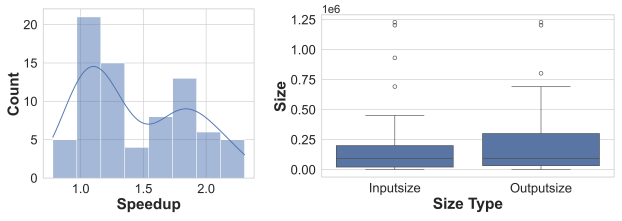
its first appearance in the dataset, starting from 0. For example, the convolution operator (Conv) is assigned the code 0, the sigmoid operator (Sigmoid) is assigned 1, and the multiplication operator (Mul) is assigned 2. Thus, an operator combination such as (Conv, Sigmoid, Mul) is initially encoded as [0, 1, 2]. This integer encoding method offers significant advantages in memory usage and storage space compared to one-hot encoding[25], particularly for large-scale operator sets, since new operators can be assigned new integer codes with ease.

Subsequently, these integer codes are used as indices for an embedding layer, which converts them into dense, low-dimensional vectors. Embedding-based representations[26] provide compact encodings of categorical features and allow the model to learn task-specific representations of operators from data. In this way, the combined integer-to-embedding approach provides a more compact, efficient, and expressive representation of operator features, facilitating improved performance prediction for operator fusion strategies.

For both Input_Shape and Output_Shape, we compute the size by multiplying all dimensions of each shape. Specifically, for Input_Shape, represented as $[b, c, h, w]$, where b denotes the batch size, c denotes the number of channels, and h and w denote the height and width, respectively. Similarly, for Output_Shape, let the output shape of the last operator be represented as $[b, k]$, where k is the number of output channels or features. The size for both input and output is calculated as:

$$\text{size} = \prod_{i=1}^{n} \text{shape}[i] \tag{4}$$

where n is the number of dimensions (e.g., 4 for the input tensor and 2 for the output tensor). These computed values are recorded as Inputsize and Outputsize, reflecting the total number of elements in the respective tensors.



(a) Speedup (b) Input and output size

Fig. 3 Data distribution.

Considering that Inputsize, Outputsize, and Speedup features differ in units and scales, their range variations can affect model performance. Therefore, standardization is essential. We first analyzed the data distribution using histograms and box plots (e.g., Fig. 3) to guide preprocessing decisions. The Speedup feature (Fig. 3(a)) shows a concentrated distribution with values ranging approximately from 0.8 to 2.2 and some outliers. Due to its narrow range, we directly applied the z-score normalization. In contrast, the box plot for Inputsize and Outputsize (Fig. 3(b)) shows right-skewed distributions with long-tail effects, as indicated by the smaller gap between the median and the upper quartile compared to the gap between the median and the maximum value. Although logarithmic transformation $f(x) = \log(1+x)$ is commonly used to mitigate such long-tail effects by compressing the range and enhancing symmetry, we evaluated its impact on model performance through comparative experiments. Our results showed that logarithmic transformation did not improve prediction accuracy (e.g., increased MAE on the test set). This suggests that the original distribution characteristics of these features contain valuable information for the prediction task. Thus, we directly applied the standard z-score normalization[27] to the original features, defined as:

$$z = \frac{x - \mu}{\sigma} \tag{5}$$

where x denotes the original feature value, z denotes the normalized value, and μ and σ represent the mean and standard deviation of the training data, respectively. This process centers data around zero and scales it according to the distribution's spread, improving model performance by making the features comparable across different scales.

We applied stratified sampling to divide the dataset, ensuring balanced class distribution across training and testing sets. First, we isolated and stored Pattern values that appeared only once to avoid distorting the stratification. The remaining patterns were stratified using StratifiedShuffleSplit from scikit-learn[27], preserving the distribution of pattern types. Finally, the resulting training and test sets were saved separately.

4.3 Hardware-aware and semi-automated data extension

To facilitate transfer across devices, we augmented the dataset with hardware-aware features that characterize the resources and performance capabilities of

each platform. These features were empirically measured using standard benchmarking tools and integrated into the dataset to facilitate cross-platform modeling. As shown in Table 4, the newly introduced attributes include is_gpu, hw_id, total_mem, mem_bw, and peak_fp32, where hw_id routes samples to corresponding hardware modules, while is_gpu indicates the general platform category (CPU or GPU). In addition, to further improve the efficiency of data preparation, the collection and preprocessing procedures described in Sections 4.1 and 4.2 were consolidated into a semi-automated data acquisition pipeline. This pipeline can automatically execute most profiling and preprocessing routines, requiring only minimal manual configuration and verification. The semi-automated process reduces the effort needed to extend the dataset to new devices and helps maintain consistent data preparation across different hardware platforms.

Table 4 Hardware-aware features added to the dataset.

Hardware Feature	Description
is_gpu	Binary indicator specifying whether the device is a GPU (1) or CPU (0).
hw_id	Integer domain identifier for routing (e.g., 0 denotes CPU, 1 denotes target GPU; extendable to multiple GPUs).
total_mem (GB)	Total memory capacity of the device, including video random-access memory (VRAM) for GPUs or dynamic random-access memory (DRAM) for CPUs, measured in gigabytes.
mem_bw (GB/s)	Sustained device memory bandwidth, measured using the *STREAM Triad* benchmark[28] for CPUs and the *CUDA bandwidthTest* (Device-to-Device mode) for GPUs.
peak_fp32 (GFLOP/s)	Peak single-precision (FP32) computational throughput, obtained via a single-precision general matrix multiplication (SGEMM) micro-benchmark[29].

5 FusionAdvisor: Methodology

As described in Section 3, the FusionAdvisor methodology comprises three interconnected modules: fusion exploration, performance evaluation, and recommendation strategy. This section introduces the design and implementation of each module in turn.

5.1 Fusion exploration

FusionAdvisor automatically explores sequences of fusible operators based on their types and computational characteristics. The fusion exploration module follows the operator-type classification and fusion rules introduced by DNNFusion[5], starting with the definition of complexity levels for different operator categories. Specifically, operators are divided into five complexity levels, increasing from the simplest *One-to-One* (Level 1, e.g., ReLU), through *Reorganize* (Level 2, e.g., Reshape), *Shuffle* (Level 3, e.g., Transpose), and *One-to-Many* (Level 4, e.g., Gather), to the most complex *Many-to-Many* (Level 5, e.g., Conv). Operators that do not meet the fusion criteria are labeled as *Fusion-Ineligible* with complexity Level 0. A classification function assigns each operator to its category while recording its name, category, and original position, to preserve fusion history and execution order.

Based on this classification, a specifically designed fusion decision function evaluates the compatibility between each pair of consecutive operators. Fusion is rejected if either operator is *Fusion-Ineligible*. In contrast, fusion is allowed when both operators fall into common fusion categories, such as *One-to-One*, *Reorganize*, or *Shuffle*. However, fusion is prohibited if a *One-to-Many* operator is followed by a *Many-to-Many* operator, as well as for two consecutive *Many-to-Many* operators. Conversely, a *Many-to-Many* operator can be fused with a subsequent *One-to-Many* operator. When fusion occurs, the resulting fused operator node inherits the more complex category from the two original categories, thereby preserving a higher-level abstraction of computational complexity. Additionally, fused operators remain eligible for further fusion, provided that they meet compatibility constraints.

Building on this classification system, this paper proposes an operator fusion path search algorithm, as detailed in Algorithm 1, which implements the core logic of sequence exploration. The algorithm begins by loading the specified ONNX model and initializing a set of allowable operators (line 1). Then it scans the entire model graph to classify the operators and add them to the operator list (lines 4–9). Next, the algorithm iterates over this list, a potential starting point for a fusion path (line 10). For each starting node, it attempts to extend the fusion path sequentially by considering subsequent nodes with higher indices ($j > i$). If the candidate

node's category is compatible with the current fusion category, the node is merged into the fusion path and the fusion category is updated accordingly (lines 13–14). Otherwise, the fusion attempt for the current starting node terminates (lines 15–16), and the longest fusion path found so far is recorded (line 19). After processing all starting nodes, a filtering step removes fusion paths that are fully contained within other longer paths, ensuring that unique maximal fusion paths remain (line 21). Finally, the algorithm outputs these unique longest fusion paths (line 22).

Algorithm 1 Operator fusion path search

Require: ONNX model
Ensure: Unique longest fusion paths
1: Load ONNX model and initialize allowed operators
2: Filter nodes by allowed ops and classify each operator into categories
3: Initialize empty list of OpNodes
4: **for all** node in model graph **do**
5: **if** node.op_type is allowed **then**
6: Create OpNode with initial category and index
7: Add to OpNode list
8: **end if**
9: **end for**
10: **for all** starting node i in OpNode list **do**
11: Initialize current fusion path and current category with node i
12: **for all** next node $j > i$ **do**
13: **if** can_fuse(current category, j.category) **then**
14: Merge nodes and update current category
15: **else**
16: Break fusion loop for current starting node
17: **end if**
18: **end for**
19: Record longest path for starting index i
20: **end for**
21: Filter paths to remove subsumed sequences
22: **return** unique longest fusion paths

Ultimately, the algorithm generates the longest fusion path for each starting node and uses a filtering mechanism to eliminate redundant overlaps among candidate paths. The final output comprises fusion paths that detail the sequence of fused operators and their respective positions in the original ONNX model. This structured approach not only reorganizes the operator graph in a fusion-friendly manner but also preserves the inherent dependencies and computational hierarchy of the model.

5.2 Evaluation

5.2.1 E-MLP model architecture

This paper proposes a neural network–based performance prediction model for operator fusion strategies, aiming to quantitatively evaluate the performance impact of various fusion strategies in deep neural networks. The model adopts the time-based speedup as the regression target, with input features comprising user-specified operator sequences and their corresponding structural and statistical attributes. By modeling the feature representations of fused subgraphs, the model can predict potential performance gains, guiding the selection and optimization of operator fusion strategies.

The proposed performance prediction model is initially constructed using a Multilayer Perceptron (MLP)[30]. The MLP is characterized by its simple architecture, ease of implementation and training, and strong generalization capability on small-scale datasets. Since it assumes a fixed-dimensional input, the MLP is well-suited for handling tabular data. Given the limited number of available fusion samples in this study, the initial model encodes and predicts performance directly from the operator feature vectors using a conventional MLP.

However, traditional MLPs exhibit limitations in handling variable-length sequences and capturing inter-operator dependencies. Although Recurrent Neural Networks (RNNs)[31] and their variants, such as Long Short-Term Memory networks (LSTMs)[32], excel in sequence modeling, their recursive computations lower training efficiency. Similarly, Transformer models[33] utilize self-attention mechanisms to capture global dependencies, but their high computational complexity and parameter redundancy make them less suitable for small- to medium-scale datasets.

To enhance the model's ability to represent operator sequence structures, we adopt an embedding-augmented Multilayer Perceptron (E-MLP), as illustrated in Fig. 4. Specifically, the model first uses a learnable embedding layer to map discrete operator identifiers into low-dimensional continuous vectors, providing a compact representation of discrete operator types. It then adds fixed non-learnable sinusoidal positional encodings[33] to preserve operator-sequence order information and facilitate learning of order-dependent patterns from padded operator sequences. The resulting features are flattened and passed through

two fully connected layers with ReLU activations for nonlinear feature extraction, followed by a final linear layer for regression. Compared to a vanilla MLP that ignores operator order, the E-MLP maintains high computational efficiency while explicitly incorporating discrete sequence information, thus improving predictive performance when estimating the speedup of operator fusion strategies.

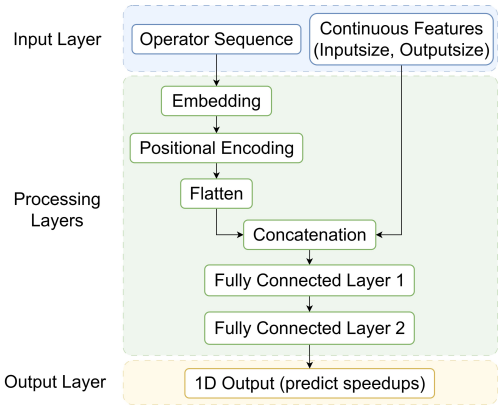


Fig. 4 E-MLP model architecture.

5.2.2 Cross-platform extension (E-MLP-X)

To enable performance prediction across different hardware platforms, we extend the proposed E-MLP into a cross-platform variant called E-MLP-X. Specifically, we design the model with a shared backbone and domain-specific heads. The shared backbone captures the general relationships between operator combinations and tensor shapes, while each hardware platform is assigned an independent prediction head. This design allows the model to disentangle transferable knowledge from platform-dependent characteristics within a single framework.

E-MLP-X incorporates hardware-aware inputs introduced in Section 4.3. A lightweight adapter network first processes these descriptors to produce a semantic embedding. This embedding is then fused with the representation learned from operator combinations and tensor shapes to form a joint feature space. Based on this shared representation, multiple lightweight output heads are defined, each corresponding to a specific hardware platform (e.g., CPU or GPU). This architectural idea is conceptually related to multi-head designs used in multi-domain learning[34]. During inference, the hardware identifier activates the corresponding head, enabling platform-aware predictions while maintaining a shared, hardware-agnostic representation learned by the backbone. By design, all representation layers are shared across platforms, with platform-specific characteristics captured solely by the hardware-aware modules and their respective output heads.

5.3 Recommendation

Algorithm 2 describes FusionAdvisor's method for efficiently maintaining Top-k results while processing candidates. The algorithm utilizes a min-heap data structure, dynamically maintaining a fixed-size min-heap with a capacity of k to store the Top-k elements as candidates are processed. Initially, an empty heap is created (line 1). When a new element arrives and the heap contains fewer than k elements, the element is inserted directly (lines 4–5). When the heap is full, the algorithm compares the new element with the root. The root is the k-th largest value in the window. If the new element is larger, it replaces the root. The heap is then adjusted to restore the heap property (lines 6–7). Finally, a sorted list is produced using a heap sort (line 10). This approach ensures a time complexity of $O(n \log k)$ while providing a favorable balance between memory consumption and computational efficiency. In dynamic candidate processing scenarios, it can improve throughput and responsiveness.

Algorithm 2 Maintain Top-k elements

Require: Integer array $Stream[1 \ldots n]$, integer k
Ensure: A descending list of the top k elements from the array
1: Initialize an empty min-heap $H \leftarrow \emptyset$
2: **for** $i \leftarrow 1$ **to** n **do**
3: $x \leftarrow Stream[i]$
4: **if** $|H| < k$ **then**
5: Push x into heap H ▷ Insert element when heap is not full
6: **else if** $x >$ TOP(H) **then**
7: Pop the root of H and push x into H ▷ Replace root when heap is full and new element is larger
8: **end if**
9: **end for**
10: **return** The elements in H, sorted in descending order

6 Implementation

6.1 Implementation of E-MLP

We implement our prediction model in PyTorch 2.0. The model takes three inputs: padded operator ID sequences, the maximum sequence input size, and the output size of the last operator. The operator sequences are mapped to dense vectors and augmented with

a fixed sinusoidal positional encoding to incorporate order information. The resulting tensor is then flattened and concatenated with the two scalar values to form the input to the feedforward network. The feedforward architecture consists of two hidden layers with 64 and 32 neurons, respectively, each followed by a ReLU activation. The final layer is a linear projection that outputs a single scalar value that represents the predicted speedup of the operator combination. We use the Mean Squared Error (MSE) loss as the training objective and optimize the model using the Adam optimizer with an initial learning rate of 0.001. During training, a batch size of 16 is used, and the model is trained for a maximum of 250 epochs with early stopping (patience = 10).

6.2 Implementation of E-MLP-X

E-MLP-X is implemented as an adaptive extension of E-MLP, designed to support cross-platform transfer via few-shot adaptation. The model retains the feedforward backbone while incorporating two hardware-specific components: (i) a numerical adapter that projects four normalized hardware descriptors into an 8-dimensional semantic vector through two fully connected layers with ReLU activations, and (ii) a learnable embedding layer that encodes discrete hardware identifiers. The outputs of these modules are concatenated with the learned representations of operator combinations and tensor shapes, then passed through two fully connected shared layers with 128 and 64 neurons, respectively, incorporating dropout (p = 0.2). Multiple lightweight regression heads are attached, each corresponding to a specific hardware platform (e.g., CPU or GPU). During inference, the head associated with the given hardware identifier is activated.

Training proceeds in two stages. In the first stage, the model is pre-trained on CPU data to learn general operator representations. In the second stage, it is fine-tuned on a limited number of GPU samples, with interleaved replay of CPU data to mitigate catastrophic forgetting. During fine-tuning, a layer-freezing strategy consistent with the design objectives of E-MLP-X is applied: the operator embedding and the first shared fully connected layer (128 neurons) remain frozen to form a cross-platform representation space, and the positional encoding is fixed by design. Meanwhile, the hardware adapter, hardware-identifier embedding, second shared layer (64 neurons), and GPU-specific regression head are updated. The CPU head remains frozen, ensuring that GPU adaptation functions as a lightweight residual correction around the CPU predictor rather than a complete reparameterization.

The total loss combines the GPU regression loss, a CPU replay term weighted by $\alpha = 0.3$, and an L^2-SP regularization term[35] with $\lambda = 10^{-4}$ to constrain deviation from the pre-trained parameters:

$$\mathcal{L} = \mathcal{L}_{\text{GPU}} + \alpha\,\mathcal{L}_{\text{CPU-replay}} + \lambda\|\theta - \theta_0\|_2^2 \qquad (6)$$

where α controls the weight of the CPU replay loss, λ controls the strength of the L^2-SP regularization term, θ denotes the current model parameters, and θ_0 denotes the pre-trained parameters. Both $\mathcal{L}_{\text{GPU}}$ and $\mathcal{L}_{\text{CPU-replay}}$ are instantiated as the Smooth L1 (Huber) loss. The regression target is defined as $\log(1 + \text{speedup})$, which helps stabilize training across different hardware platforms and performance ranges.

7 Experiments and Discussion

7.1 Experimental setup

The experimental setup is outlined as follows:

(1) Deep learning model set. A subset of ONNX models used in this study is listed in Table 1 as examples. Each model was loaded into ONNX Runtime and optimized for inference, employing operator fusion strategies to capture execution times across various schemes for comparison.

(2) Dataset. The dataset comprises 137 entries, including 109 training samples and 28 testing samples, following an 80:20 split. It was generated using the method described in Section 4.

(3) Hardware and software configuration. All experiments were carried out using the same hardware and software configuration, as detailed in Table 5, to ensure consistency in the results.

Table 5 Experimental environment configuration.

Component	Specification
CPU	Intel Core i5-13500H@2.60 GHz
GPU	NVIDIA GeForce RTX 3050 Laptop GPU (6 GB VRAM)
CUDA	12.2
Operating System	Windows 11
Python	3.9
Deep Learning Frameworks	PyTorch 2.0.0, TensorFlow 2.10.0, Keras 2.10.0
Key Tools	TVM 0.18.dev0 (git 751467e98), ONNX Runtime 1.19.2 (git ffceed9d44; ONNX 1.16.0)

7.2 Evaluation metrics

To evaluate the performance of our model in predicting the speedup of operator fusion, we utilize four standard metrics: root mean squared error (RMSE), mean absolute error (MAE), mean absolute percentage error (MAPE), and the coefficient of determination (R^2). RMSE measures the average magnitude of the prediction error, while MAE measures the average absolute error. MAPE provides a normalized percentage error, and R^2 quantifies the goodness of fit. These metrics are widely used in performance evaluation[36, 37], where lower values of RMSE, MAE, and MAPE, along with a higher R^2, indicate improved prediction performance and model generalization.

7.3 Experimental results and analysis

Scalability and runtime overhead of the fusion search algorithm. To evaluate the scalability of Algorithm 1, we measured its search time on 17 ONNX models[23] with varying graph complexities. Table 6 reports, for each model (listed in ascending order of fusible operators), the total operator count (Total Ops), the number of operators eligible for fusion (Fusible Ops), and the number of non-overlapping fusion paths (Paths).

Table 6 Statistics of ONNX models.

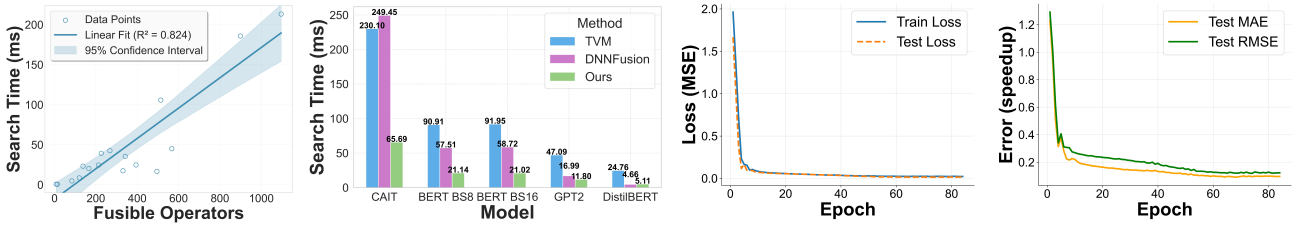
Model	Total Ops	Fusible Ops	Paths
MNIST	12	8	3
AlexNet	20	14	7
EfficientNet-Lite0	160	84	42
GoogLeNet	164	120	57
ResNet-50-v2	174	166	49
MobileNetV3-Small	226	226	54
EfficientNet-B0	239	213	74
DistilBERT	425	268	68
SSD	515	138	40
BERT-Base	533	331	97
YOLOv4	590	394	110
Inception-ResNet	658	567	244
T5-Encoder	689	342	109
SqueezeBERT	742	495	74
GPT-2	1041	514	122
BERT-Large	1527	1096	291
NASNet-A-Large	2615	899	249

Figure 5(a) presents the search time of Algorithm 1 plotted against the number of fusible operators. The linear regression fitted yields an R^2 value of 0.824, indicating near-linear scalability. In particular, even models with large fusible sets, such as BERT-Base (331 fusible operators) and BERT-Large (1096 fusible operators), complete the operator fusion path search in under 220 milliseconds, demonstrating the practical efficiency of the algorithm for large-scale real-world networks. Figure 5(b) compares the search times of different models using TVM, DNNFusion, and our proposed search algorithm. The search times for TVM were obtained from our experiments, whereas the times for DNNFusion were reproduced on the basis of the methods described in the literature. For the models evaluated, our method consistently outperforms both TVM and DNNFusion, achieving faster search times. These results demonstrate the effectiveness of our approach in optimizing search efficiency.

Table 7 Fusion statistics for different models and methods.

Method	Model	Fusion Coverage	Fused Ops	Average Ops/Path	Fusion Strategies
TVM	BERT-Large-bs8	71.84%	1097	4.42	12
	BERT-Large-bs16	71.84%	1097	4.42	12
	DistilBERT	69.88%	297	4.70	10
	GPT-2	59.94%	624	3.69	16
	CAiT	75.82%	2508	4.24	29
	OpenAI-GPT	60.48%	453	4.62	11
DNN Fusion	BERT-Large-bs8	13.04%	160	3.08	4
	BERT-Large-bs16	13.04%	160	3.08	4
	DistilBERT	73.89%	249	3.43	16
	GPT-2	63.13%	464	2.77	18
	CAiT	12.64%	418	3.88	17
	OpenAI-GPT	29.91%	224	2.05	13
Ours	BERT-Large-bs8	71.77%	1096	3.77	12
	BERT-Large-bs16	71.77%	1096	3.77	12
	DistilBERT	63.06%	268	3.94	7
	GPT-2	49.38%	514	4.21	9
	CAiT	74.46%	2463	4.10	21
	OpenAI-GPT	38.85%	291	4.04	7

Table 7 presents the fusion statistics for various models using TVM, DNNFusion, and our proposed search algorithm. It displays the fusion coverage, the average number of operators per fusion path, and the number of distinct fusion strategies across various ONNX models, including BERT, DistilBERT, GPT-2, and others. Here, fusion strategies are defined as the



(a) Search time versus fusible operators across models

(b) Search time comparison across methods for different models

(c) Training and test loss curves

(d) Test MAE and RMSE over epochs

Fig. 5 Search efficiency analysis and training performance. (a) Search time versus fusible operators across models; (b) Search time comparison across methods for different models; (c) Loss curves on training and test sets; (d) Test error metrics over epochs.

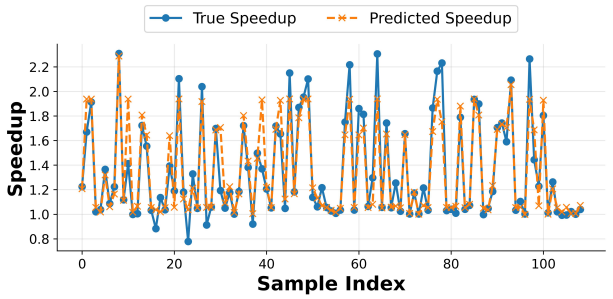
number of distinct fusion paths corresponding to each unique combination of operator sequence, maximum input shape, and last output shape. The Fusion Coverage metric is calculated as the ratio of fused operators to the total number of original operators in the model. Specifically, for each model, Fusion Coverage is computed as:

$$\text{Fusion Coverage} = \frac{\text{Fused Ops}}{\text{Original Ops}} \times 100\% \quad (7)$$

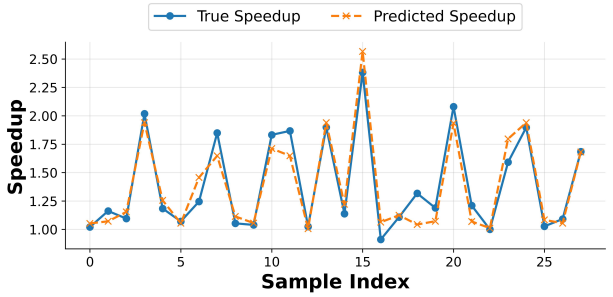
Fused Ops refers to the number of operators that the optimization algorithm can fuse, while Original Ops denotes the total number of operators before optimization. In general, our method achieves a balance between fusion coverage, operator efficiency, and the exploration of multiple fusion strategies, contributing to efficient search and practical fusion-strategy selection.

Performance of the proposed E-MLP model. Here, we first report the convergence behavior of our E-MLP model within the training setup described in Section 6.1. Figure 5(c) illustrates the training and test loss curves of the E-MLP model over 84 epochs. The results show that the regression loss gradually decreases as the number of training epochs increases. Starting with an initial training loss of approximately 1.96 and a test loss of 1.67, both curves drop rapidly to roughly 0.23 and 0.12 by the 4th epoch. They then gradually converge to 0.022 (training) and 0.015 (test) by the 84th epoch. In addition, Fig. 5(d) shows the trajectories of the mean absolute error (MAE) and the root mean square error (RMSE), both of which steadily decrease over time, indicating a consistent improvement in the model's predictive accuracy.

Figure 6 compares the predicted speedup values with the actual measurements for the training and test sets, respectively. These figures illustrate the performance of the E-MLP model by showing how closely its



(a) Comparison of predicted and true speedup values on the training set



(b) Comparison of predicted and true speedup values on the test set

Fig. 6 Speedup prediction performance on (a) the training set and (b) the test set.

predictions align with the true values. In each figure, the x-axis represents the sample indices and the y-axis denotes the speedup values. The blue curve shows the actual speedup measurements, while the orange curve displays the model's predictions. The similar trends observed in both curves indicate that the training process has been effective and that the model is capable of accurately predicting the speedup for operator fusion strategies. Analyzing the result plots, it can be observed that the predicted speedup in the test set has a small error compared to the actual speedup for most samples. However, there are some individual samples with relatively large errors. Despite this, the

Table 8 Latency of FusionAdvisor on CPU and GPU.

Metric	CPU (ms)		GPU (ms)	
	Median	P95	Median	P95
Forward latency	0.098	0.121	0.736	1.590
Scoring latency	0.183	0.231	1.268	2.696

overall trend is still roughly aligned with the actual data.

Beyond predictive accuracy, practical deployment also demands low overhead. We therefore quantify the computational overhead of FusionAdvisor in terms of model size, memory usage, and inference latency. The prediction model is lightweight, consisting of 4,937 trainable parameters with an FP32 static footprint of approximately 0.02 MB. The peak runtime GPU memory allocation measured during a forward pass is 8.15 MB. Inference latency is measured by scoring one fusion candidate per call. Table 8 reports latency measurements on both the CPU and GPU (CUDA). Latency is reported in the steady state, after discarding the first two calls as a warm-up. We report two metrics: forward latency and scoring latency. Forward latency measures the model forward pass only. Scoring latency measures the per-call time required to score a single fusion candidate, including feature construction, normalization, device transfer, and model inference.

Comparison with baseline models. Next, we compare our proposed E-MLP model with three representative baselines: LSTM[32], XGBoost[38], and LightGBM[39]. To ensure a fair and reproducible comparison, we utilize identical data splits, hyperparameter settings, and an early stopping strategy based on RMSE with a patience of 10 epochs. Furthermore, all experiments are conducted with a fixed random seed (set to 42) to control for randomness in weight initialization, data shuffling, and model-specific stochastic processes.

Table 9 and Fig. 7 summarize the performance of all models on both the training and the test sets. As illustrated, the proposed E-MLP achieves favorable results across all metrics. Specifically, it records the lowest RMSE (0.1480 compared to 0.1931-0.2274), MAE (0.0911 compared to 0.1437-0.1652) and MAPE (6.37% compared to 9.82%-11.64%) on the training set, and maintains its advantage on the test set with an RMSE of 0.1226, MAE of 0.0967, MAPE of 6.90% and the highest R^2 score (0.9117 compared to 0.6966-0.8097). This consistency across training and test sets further indicates the model's robustness and

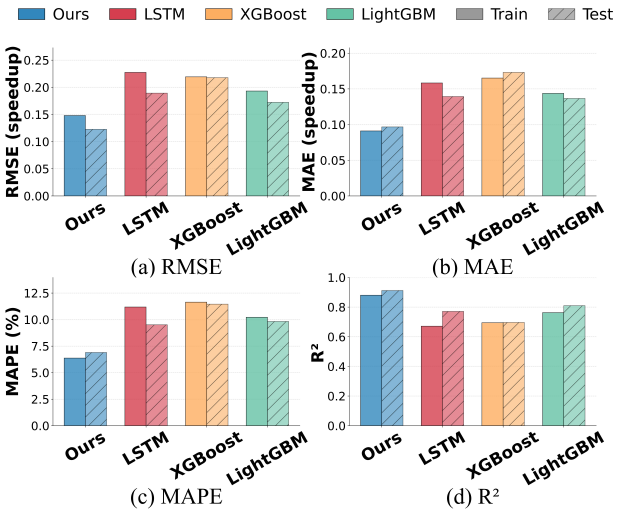


Fig. 7 Comparison of performance metrics on the training set and test set.

generalization capability.

Among the baseline models, XGBoost demonstrates the fastest convergence time at 0.04 seconds; however, it has a limited capacity to model sequence-structured operator input, resulting in higher prediction errors and suboptimal R^2 values. LightGBM achieves moderate precision but exhibits training instability, often requiring excessive boost iterations to attain satisfactory performance. The LSTM model, despite its theoretical advantage in capturing temporal dependencies, presents several practical limitations: (i) it is particularly prone to overfitting on small-scale datasets due to its extensive parameter space; (ii) its performance is highly sensitive to hyperparameter choices, including hidden layer size and sequence length; and (iii) it typically requires significantly more training time and computational resources compared to other methods, while yielding inferior results in our experiments.

Overall, these results demonstrate that the E-MLP not only achieves strong predictive accuracy but also offers a favorable trade-off between convergence speed, robustness, and computational cost, making it well-suited for practical performance modeling tasks.

Coverage-aware generalization and error analysis. We investigate generalization behavior using an expanded dataset on a single CPU. This expanded dataset includes more batch configurations. We evaluate robustness across multiple random seeds using standard training and testing splits. We also evaluate group-wise generalization using GroupKFold splits by model

Table 9 Performance comparison of different models on training and test sets.

Model	Set	RMSE	MAE	MAPE (%)	R^2	Train Time (s)	Early Stop
Ours	Train	**0.1480**	**0.0911**	**6.37**	**0.8800**	1.99	84
	Test	**0.1226**	**0.0967**	**6.90**	**0.9117**		
LSTM	Train	0.2274	0.1583	11.18	0.6712	3.02	92
	Test	0.1892	0.1393	9.51	0.7709		
XGBoost	Train	0.2192	0.1652	11.64	0.6943	0.04	50
	Test	0.2177	0.1731	11.45	0.6966		
LightGBM	Train	0.1931	0.1437	10.22	0.7628	0.06	339
	Test	0.1724	0.1366	9.82	0.8097		

Table 10 Robustness and group-wise generalization of E-MLP.

Dataset	Setting	RMSE	MAE	MAPE (%)	R^2
Original ($N = 137$)	Train–test split (5 seeds; $N_{\text{train}} = 109$, $N_{\text{test}} = 28$)	0.1294±0.0063	0.1004±0.0057	7.2244±0.5208	0.9015±0.0096
	Leave-one-model-out (GroupKFold, 5 folds)	0.5092±0.3763	0.4249±0.3031	30.4134±16.8737	-8.4129±12.1863
	Leave-one-pattern-out (GroupKFold, 5 folds)	0.4888±0.1858	0.4154±0.1504	30.6542±7.3076	-7.9860±11.8717
Expanded ($N = 188$)	Train–test split (5 seeds; $N_{\text{train}} = 150$, $N_{\text{test}} = 38$)	0.1528±0.0082	0.1171±0.0060	7.6205±0.4873	0.9168±0.0089
	Leave-one-model-out (GroupKFold, 5 folds)	0.4665±0.3170	0.3967±0.2982	26.1244±14.2132	-0.8230±3.0871
	Leave-one-pattern-out (GroupKFold, 5 folds)	0.4987±0.0708	0.4293±0.0620	32.1421±5.8652	-8.7714±19.4094

and fusion pattern. Table 10 summarizes the aggregate results.

We also decompose errors based on fusion-pattern coverage (i.e., fusion operator sequences) under a leave-one-model-out protocol. A test sample is considered in-coverage if its fusion pattern is observed in the corresponding training fold; otherwise, it is treated as out-of-distribution (OOD). Table 11 reports the subset share in the test set (%) and the number of non-empty folds for each subset (out of five folds). Shares are computed per fold, with empty subsets contributing 0% in that fold. Importantly, the model remains accurate on in-coverage instances, while errors increase primarily on OOD instances. Overall, Table 11 shows a consistent performance gap between the in-coverage and OOD subsets, indicating that the dominant failure mode is fusion-pattern coverage shift (previously unseen fusion operator sequences) rather than random noise.

We further quantify the benefit of additional measurements through a controlled pattern-drop k-shot recovery experiment. Specifically, we remove several frequent fusion patterns (ConvRelu, BiasGelu, ConvClip) from the training set and add back k samples per dropped pattern from the held-out set. Fig. 8 reports the OOD RMSE on the remaining held-out OOD samples after adding back k samples per dropped pattern ($n = 15$ in total), shown as mean ± std over five random resamplings. The OOD RMSE decreases from 0.5706 ($k = 0$) to 0.4099±0.0451 ($k = 1$), 0.3712±0.0749 ($k = 2$), and 0.2565±0.0725 ($k = 4$). These results show that even a small number of additional measurements can substantially reduce error under coverage shift.

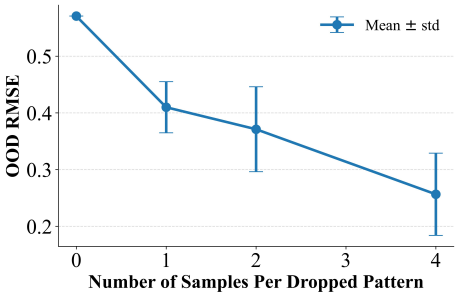


Fig. 8 K-shot recovery under pattern-coverage shift (single CPU).

Table 11 Coverage-aware error decomposition under leave-one-model-out.

Dataset	Subset	Non-empty folds	Test-set share (%)	RMSE mean (std)	MAPE (%) mean (std)
Original dataset ($N = 137$)	In-coverage	4/5	26.7	0.230 (0.124)	14.0 (6.0)
	OOD	5/5	73.3	0.613 (0.536)	40.2 (33.0)
Expanded dataset ($N = 188$)	In-coverage	5/5	53.1	0.346 (0.055)	17.3 (5.1)
	OOD	4/5	46.9	0.531 (0.458)	34.1 (19.1)

Comparison with previous methods. We compare our method with two representative previous methods: NeuSight[18] and Disco[19]. For consistency, we define estimation error using the following relative error formula:

$$\text{Error} = \frac{|\text{Predicted} - \text{Actual}|}{\text{Actual}} \times 100\% \tag{8}$$

A key distinction is that our method predicts speedup ratios, while NeuSight and Disco predict latency values. Although the hardware platforms differ, with NeuSight evaluated on L4, A100-40GB, and H100 GPUs, Disco tested on a GTX 1080 Ti GPU, and our method assessed using the experimental setup detailed in Section 7, this comparison provides a useful reference for understanding the performance of each approach.

Table 12 Comparison of estimation errors for different models and methods.

Model	NeuSight[18]	Disco[19]	Ours
BERT-Large-bs8	14.5%-24.6%	N/A	**12.7%**
BERT-Large-bs16	13.4%-20.9%	N/A	**8.6%**
ResNet-50	N/A	11.1%	**5.5%**
VGG-19	N/A	12.4%	**5.9%**

Note: N/A indicates that the corresponding method did not report results for that model.

The results shown in Table 12 highlight the performance of each method in terms of estimation error. N/A entries indicate that the corresponding method did not report results for that model, because NeuSight and Disco used distinct benchmark sets with no overlap in the listed models. For NeuSight, we report the error as a range (e.g., 14.5% to 24.6% for BERT-Large-bs8) because the results are reported for different hardware configurations. For our method, the models listed in Table 12 are excluded from the training dataset, and the reported errors are therefore evaluated on held-out models. In contrast, our method provides a single estimation error value for each model. Despite hardware differences, our method demonstrates competitive performance, achieving lower estimation errors for most models, including BERT-Large, ResNet-50, and VGG-19. It is important to note that hardware differences may influence the results. However, we argue that the comparison remains valuable, as it allows us to evaluate the relative performance of these methods across different platforms.

Validation of Top-k recommendation accuracy. We validate the effectiveness of FusionAdvisor's Top-k recommendation module by comparing the predicted and actual rankings of fusion strategies on three models: VGG-19, BERT-Large-bs16, and BERT-Large-bs8. For each model, FusionAdvisor ranks strategies based on predicted speedup to generate Top-k recommendations. These predicted rankings are then directly compared with the ground-truth rankings derived from actual speedup measurements. Table 13 presents a comparison that includes the input and output shapes for each strategy. The predicted rankings generally align with the actual rankings, suggesting that the model can help identify effective fusion strategies, especially in the context of shape-specific optimizations.

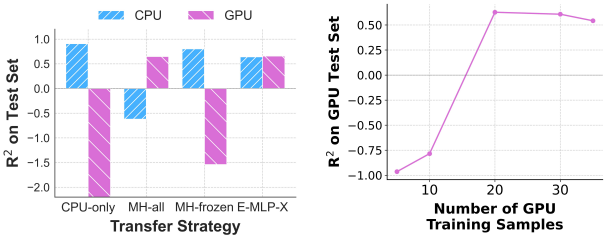
Cross-platform transferability of E-MLP-X. To assess cross-platform adaptability, we evaluate layer-freezing strategies for fine-tuning E-MLP-X from CPU to GPU. Operator-level performance is inherently platform-dependent. A single-head CPU model trained on CPU data performs well on the CPU test set ($R^2 \approx 0.91$, MAPE $\approx 7\%$) but shows poor generalization on the GPU test set ($R^2 \approx -2.5$, MAPE $\approx 60\%$), even when explicit hardware descriptors are included. This indicates that zero-shot cross-device prediction is unreliable without targeted adaptation on the target platform.

A straightforward single-head fine-tuning approach can achieve an R^2 value of approximately 0.73 on the GPU test set but results in significant forgetting on the CPU side. To address this issue, we introduce multi-head prediction, where each hardware platform is paired with a lightweight output head that shares a common representation backbone. Additionally, we

Table 13 Alignment between predicted and actual Top-k rankings of fusion strategies.

Model	Strategy	Input Shape	Output Shape	Predicted / Actual Rank
VGG-19	ConvRelu	(1, 3, 224, 224)	(1, 64, 224, 224)	1 / 1
	ConvRelu	(1, 64, 224, 224)	(1, 64, 224, 224)	2 / 2
	ConvRelu	(1, 64, 112, 112)	(1, 128, 112, 112)	3 / 3
	ConvRelu	(1, 128, 112, 112)	(1, 128, 112, 112)	4 / 4
	ConvRelu	(1, 256, 56, 56)	(1, 256, 56, 56)	5 / 5
	ConvRelu	(1, 512, 28, 28)	(1, 512, 28, 28)	6 / 6
	ConvRelu	(1, 512, 14, 14)	(1, 512, 14, 14)	7 / 7
BERT-Large-bs8	BiasGelu	(8, 128, 4096)	(8, 128, 4096)	1 / 1
	SkipLayerNormalization	(8, 128, 1024)	(8, 128, 1024)	2 / 2
	MatMulDiv	(8, 16, 128, 64)	(8, 16, 128, 128)	3 / 3
	GemmTanh	(8, 1024)	(8, 1024)	4 / 4
BERT-Large-bs16	BiasGelu	(16, 128, 4096)	(16, 128, 4096)	1 / 1
	SkipLayerNormalization	(16, 128, 1024)	(16, 128, 1024)	2 / 2
	MatMulDiv	(16, 12, 128, 64)	(16, 12, 128, 128)	3 / 3
	GemmTanh	(16, 1024)	(16, 1024)	4 / 4

explore several fine-tuning strategies for E-MLP-X using 35 GPU samples, as summarized in Fig. 9(a). When the entire backbone is trainable (MH-all), the GPU-side performance improves substantially ($R^2 \approx 0.65$), but the model still forgets the CPU knowledge. In contrast, a fully frozen backbone (MH-frozen) preserves accuracy on the CPU side but fails to adapt to the GPU, resulting in a negative R^2.



(a) Test R^2 across different layer-freezing strategies (b) Data efficiency of GPU few-shot fine-tuning

Fig. 9 Model transfer performance and data efficiency analysis.

The proposed E-MLP-X with partial backbone freezing achieves a favorable trade-off. We freeze the operator embedding and the first shared fully connected layer to establish a stable, platform-agnostic representation, while the positional encoding remains fixed by design. The hardware adapter, hardware-identifier embedding, the second shared layer, and the GPU-specific head are then fine-tuned for platform-specific adaptation. Under the same few-shot condition (35 GPU samples), this strategy improves GPU-side performance from a strongly negative R^2 before adaptation to approximately 0.66 (MAPE $\approx$ 12%), while retaining reasonable CPU-side performance ($R^2 \approx 0.64$, with MAPE comparable to the CPU-only model). These results indicate that partial backbone freezing enables effective cross-platform transfer while retaining reasonable source-domain performance.

Table 14 Multi-GPU transfer of E-MLP-X.

GPU	GPU test		CPU test	
	R^2	MAPE (%)	R^2	MAPE (%)
RTX 2080 Ti	0.774 ± 0.059	16.2 ± 3.5	0.732 ± 0.065	12.9 ± 1.5
RTX 4090	0.855 ± 0.027	16.6 ± 3.6	0.739 ± 0.091	12.2 ± 1.5

We evaluate the data efficiency of E-MLP-X by fine-tuning the model with varying amounts of GPU data (5, 10, 20, 30, and 35 samples). As shown in Fig. 9(b), the model's performance on the GPU test set improves rapidly with increasing fine-tuning sample size and stabilizes around 30 to 35 samples. This analysis highlights the model's practical data efficiency and consistent learning behavior when trained with limited data. Minor fluctuations observed with increasing k are likely due to the stochastic nature of small-sample optimization and do not affect the overall upward trend.

We further validate E-MLP-X on two additional GPUs from different generations (RTX 2080 Ti and RTX 4090). We use the same training protocol and fine-tune using $k = 35$ GPU samples. Table 14 reports the

Table 15 Prediction error by speedup range.

Range	2080 Ti n	2080 Ti MAPE (%)	4090 n	4090 MAPE (%)
$[1, 2)$	5	14.6	5	8.2
$[2, 4)$	0	–	0	–
$[4, 8)$	8	14.1	6	13.9
$[8, \infty)$	1	35.1	2	11.1

mean and standard deviation over five random seeds. E-MLP-X achieves GPU test R^2 of 0.774 ± 0.059 on RTX 2080 Ti and 0.855 ± 0.027 on RTX 4090, with MAPE of $16.2 \pm 3.5\%$ and $16.6 \pm 3.6\%$, respectively. The CPU test accuracy remains stable after adaptation, with CPU test R^2 of 0.732 ± 0.065 and 0.739 ± 0.091. Table 15 breaks down the error by speedup range. MAPE is generally more stable in bins with more test samples, while variance increases in rare ranges. For example, the $[8, \infty)$ bin on RTX 2080 Ti has only one test sample and shows a higher MAPE (35.1%). The GPU test sets are small (13–14 samples for each device), so some bins contain few samples and show higher variance.

7.4 Discussion

A practical limitation is that generalization depends on the coverage of fusion patterns in the training set. When evaluation includes previously unseen fusion patterns, prediction errors can increase. We report group-based generalization results and coverage-aware error analyses, and further show in Fig. 8 that adding a small number of targeted measurements can substantially reduce error under coverage shift. Our hardware descriptors summarize macro-level hardware characteristics (memory bandwidth, peak FLOPs, and memory size) and do not explicitly model microarchitectural or software-stack effects (e.g., Tensor Core utilization, cache hierarchy, or kernel implementations). In practice, E-MLP-X can partially account for these residual platform differences through the learned hardware embedding and few-shot target-device adaptation (Table 14).

8 Conclusions and Future Work

This paper presents a framework that utilizes a neural network to evaluate operator fusion performance and recommend Top-k candidate strategies in deep learning models. The framework constructs a dedicated dataset by extracting operator features and calculating fusion-induced speedups using ONNX Runtime. It also employs an improved model design to learn fusion performance patterns. Experimental results demonstrate that our method achieves favorable results in search efficiency and prediction accuracy under the evaluated settings. With a root mean squared error (RMSE) of 0.1226 and a mean absolute error (MAE) of 0.0967, the model exhibits strong performance across a variety of deep learning models. Furthermore, E-MLP-X adapts from CPU to GPU with limited target-domain data. It uses a partially frozen backbone and hardware-aware modules to improve GPU accuracy while retaining reasonable CPU-side performance. In future work, we will broaden hardware coverage and investigate richer hardware and kernel descriptors. This may further reduce the need for target-device measurements and move toward zero-shot transfer, which we leave beyond the scope of this paper.

Acknowledgment

This work was supported by the National Science and Technology Major Project of China (2023ZD0120503). We are also grateful to Anfa Zhang, a colleague at China Telecom Cloud Technology Co., Ltd.

References

[1] G. Menghani, Efficient Deep Learning: A survey on making deep learning models smaller, faster and better, *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–37, 2023.

[2] Z. Wang, X. Liu, S. Liu, Y. Yao, Y. Huang, M. Li, et al., TeleChat: An open-source billingual large language model, in *Proceedings of the 10th SIGHAN Workshop on Chinese Language Processing (SIGHAN-10)*, pp. 10–20, 2024.

[3] X. Cai, Y. Wang, and L. Zhang, Optimus: An operator fusion framework for deep neural networks, *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 1, pp. 1–26, 2022.

[4] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, 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 18)*, pp. 578–594, 2018.

[5] W. Niu, J. Guan, Y. Wang, G. Agrawal, and B. Ren, DNNFusion: accelerating deep neural networks execution with advanced operator fusion, in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 883–898, 2021.

[6] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare, On optimizing operator fusion plans for large-scale machine learning in SystemML, *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1755–1768, 2018.

[7] T. Elgamal, S. Luo, M. Boehm, A. V. Evfimievski, S. Tatikonda, B. Reinwald, and P. Sen, SPOOF: Sum-product optimization and operator fusion for large-scale machine learning, in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2017.

[8] J. Fang, Y. Shen, Y. Wang, and L. Chen, Optimizing DNN computation graph using graph substitutions, *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2734–2746, 2020.

[9] TensorFlow, XLA: Optimizing compiler for machine learning, https://www.tensorflow.org/xla. Accessed: 26-Feb-2026.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., TensorFlow: A system for large-scale machine learning, in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[11] NVIDIA, TensorRT: Programmable Inference Accelerator, https://developer.nvidia.com/tensorrt. Accessed: 26-Feb-2026.

[12] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, Optimizing DNN computation with relaxed graph substitutions, *Proceedings of Machine Learning and Systems*, vol. 1, pp. 27–39, 2019.

[13] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, TASO: Optimizing deep learning computation with automatic generation of graph substitutions, in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

[14] J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng, and X. Jin, Apollo: Automatic partition-based operator fusion through layer by layer optimization, in *Proceedings of Machine Learning and Systems*, vol. 4, pp. 1–19, 2022.

[15] Y. Yang, B. Diao, H. Liu, Q. Chen, Q. Wang, and Y. Xu, An evolutionary search-based operator fusion method with binary representation for deep learning inference acceleration, in *Proceedings of the 27th International Conference on Pattern Recognition (ICPR 2024)*, pp. 32–45, 2024.

[16] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, Learning to optimize tensor programs, in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 3393–3404, 2018.

[17] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, Ansor: Generating high-performance tensor programs for deep learning, in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879, 2020.

[18] S. Lee, A. Phanishayee, and D. Mahajan, Forecasting GPU performance for deep learning training and inference, in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25)*, pp. 493–508, 2025.

[19] X. Yi, S. Zhang, L. Diao, C. Wu, Z. Zheng, S. Fan, S. Wang, J. Yang, and W. Lin, Optimizing DNN compilation for distributed training With joint OP and tensor fusion, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 12, pp. 4694–4706, 2022.

[20] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., PyTorch: An imperative style, high-performance deep learning library, *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, pp. 8024–8035, 2019.

[21] ONNX: Open Neural Network Exchange, https://onnx.ai/. Accessed: 26-Feb-2026.

[22] Microsoft, ONNX Runtime, https://github.com/microsoft/onnxruntime. Accessed: 26-Feb-2026.

[23] ONNX, ONNX Model Zoo, https://onnx.ai/models/. Accessed: 26-Feb-2026.

[24] ONNX, ONNX Operators, https://onnx.ai/onnx/operators/. Accessed: 26-Feb-2026.

[25] C. M. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer, 2006.

[26] J. T. Hancock and T. M. Khoshgoftaar, Survey on categorical data for neural networks, *Journal of Big Data*, vol. 7, no. 1, Art. no. 28, pp. 1–41, 2020.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, et al., Scikit-learn: Machine learning in Python, *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[28] J. D. McCalpin, Memory bandwidth and machine balance in current high performance computers, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[29] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic linear algebra subprograms for Fortran usage, *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[30] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, Learning representations by back-propagating errors, *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[31] J. L. Elman, Finding structure in time, *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[32] S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, Attention is all you need, in *Proc. Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

[34] S. Masaki, T. Hirakawa, T. Yamashita, and H. Fujiyoshi, Multi-domain semantic-segmentation using multi-head model, in *Proceedings of the 2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pp. 2802–2807, 2021.

[35] X. Li, Y. Grandvalet, and F. Davoine, Explicit inductive bias for transfer learning with convolutional networks, in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, PMLR, vol. 80, pp. 2825–2834, 2018.

[36] Y. Gao, X. Gu, H. Zhang, H. Lin, and M. Yang, Runtime performance prediction for deep learning models with graph neural network, in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 368–380, 2023.

[37] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, Habitat: A runtime-based computational performance predictor for deep neural network training, in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 503–521, 2021.

[38] T. Chen and C. Guestrin, XGBoost: A scalable tree boosting system, in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pp. 785–794, 2016.

[39] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, LightGBM: A highly efficient gradient boosting decision tree, in *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 3146–3154, 2017.

Chang Jiang is currently a master's degree candidate at the University of Science and Technology Beijing, China. Her current research interests include deep learning and operator fusion strategies.

Shen Gao is currently employed at the China Telecom Cloud Computing Research Institute, Beijing. He received his Bachelor's degree in computer science and technology from Tianjin University, Tianjin, in 2016, and his Master's degree in computer science from the Illinois Institute of Technology, Illinois, in 2018. His research interests include network function virtualization (NFV), cloud computing, and 5G.

Jie Wu (Fellow, IEEE) is Laura H. Carnell Professor at Temple University and the Director of the Center for Networked Computing (CNC). His current research interests include cloud computing and networks. He serves on several editorial boards, including IEEE/ACM ToN. Dr. Wu is/was general chair/co-chair for IEEE IPDPS'23, ACM MobiHoc'23, and IEEE CCGrid'24, as well as program chair/co-chair for IEEE INFOCOM'11 and CCF CNCC'13. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair of the IEEE Technical Committee on Distributed Processing (TCDP). He is a Member of the Academia Europaea (MAE). Currently, Dr. Wu is on leave working at China Telecom as a Scientist in Cloud Computing.

Yanan Yang is currently a researcher at China Telecom Corp. Ltd. He received his Ph.D. degree from the College of Intelligence and Computing, Tianjin University, China, in 2024. His research focuses on cloud computing and serverless computing. He has published papers in EuroSys, SC, ASPLOS, USENIX ATC, IEEE TPDS, and ACM TOCS, and received the ACM SIGOPS Outstanding Doctoral Dissertation Award in December 2024.

Jianjiang Li is currently a professor at the University of Science and Technology Beijing, China. He received his Ph.D. degree in computer science and technology from Tsinghua University in 2005. His current research interests include parallel computing, high-performance computing, cloud computing, big data, and the fusion of artificial intelligence and supercomputing.