2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application

Machine Learning-enabled Performance Model for DNN Applications and AI Accelerator

Ruohan Wu[1], Mingfan Li[1], Hanxi Li[1], Tianxiang Chen[2], Xinghui Tian[2], Xiaoxin Xu[2], Bin Zhou[2], Junshi Chen[1*], Hong An[1*]

[1] University of Science and Technology of China, Hefei, Anhui, China
{ruohanwu, mingfan, lihanxi}@mail.ustc.edu.cn, {cjuns, han}@ustc.edu.cn

[2] Huawei, Hangzhou, China
{chentianxiang2, tianxinghui, xuxiaoxin, zhoubin63}@huawei.com

***Abstract*—As innovations in deep learning systems and deep neural network (DNN) models continue to grow, accurate performance analysis acts as a promising tool for understanding and navigating the complex software-hardware interplay, especially for the today's heterogeneous AI architecture. However, the actual execution of DNNs on the dedicated accelerators involves challenges from nontrivial dataflow graph analysis, tensor compiler optimizations, and operator performance prediction. In this work, we propose a two-stage performance model framework that combines graph-level analysis and operator-based hotspot modeling to bridge the gap between high-level application performance and its software-hardware systems. By the employ of machine learning (ML) solution, our performance model further captures the low-level hardware-dependent information, including operator fusion and data layout transformation. Our graph analysis for mainstream models from computer vision (CV), natural language processing (NLP) and recommendation domains selects total 26 kinds of operators and builds a dataset on the Huawei Ascend 910. With the well-trained model, our open source[1] performance model finally achieves 15.4% average error for predicting the execution time of DNN models, and our modeling for memory access and performance bottleneck supports efficient running of DNN models for future systems.**

***Index Terms*—deep learning, performance model, accelerator, operator, computation graph**

I. Introduction

Recent years have witnessed the proliferation of deep learning (DL) systems and researchers have developed many kinds of application-specific DNN models, including CV models used in image recognition, NLP models used in search engines and recommendation models used in social networks. To train these models efficiently, accelerators with varying compute units and memory hierarchy such as the TPU [1] and GPU are used to deliver high compute throughput and low memory access latency. Meanwhile, these accelerators are supported by a lot of software-level optimizations [2]–[4]. Due to the complexity of the hardware/software, it is hard to explore the best design trade-offs and configurations for DL systems.

A well-designed performance model [5]–[11] for fast and accurate performance (e.g. iteration training time) prediction without actually executing DNN models on the dedicated hardware can be used to help solve these hardware/software co-design problems. First, as for the hardware, the accelerator offering the highest training performance is not always the same as the most cost-efficient one. When we need accelerators to train certain DNN models, we can use the performance model to help us select the most suitable hardware for training DNNs [5]. Second, as for the software, performance model can help us quickly predict how changing batch sizes and/or feature map sizes affect the execution time of basic blocks in DNN models, which can be used in network architecture search (NAS) to explore best DNN model architectures or be used by compilers to choose tile-size for kernels [6].

Current DL frameworks, such as Tensorflow and PyTorch [12], [13], rely on a computation graph intermediate representation to implement compute and memory management, which simplifies the difficulties of performance modeling. However, accurate performance prediction for real-world DNN applications still faces many constraints.

First, performance model heavily relies on a comprehensive analysis for recognizing hotspot operators to work across a variety of workloads. There are many kinds of DNN applications that each has unique models and operators. For example, while ResNet [14] and ViT [15] are both CV models for image recognition tasks, the former one's operators are mainly Conv2D, ReLU and BatchNorm, and the latter one's operators are mainly MatMul, GeLU and LayerNorm. Meanwhile, a DNN model may include dozens of different operators with varying percentages of execution time. Some performance models [8], [9] mainly focus on compute-intensive operators (e.g. convolution, full-connected layer) and ignore other functional operators. Finally, even operators of the same kind could be different. For example, Layernorm's tensor dimensions in ViT and BERT [16] are 3D and 2D respectively, which makes it more difficult to build a general performance model.

Second, the actual execution of DNN models may be optimized by the tensor compilers such as XLA [2] and TVM [3]. These optimizations include operator fusion that replaces many primitive operators with a functionally equivalent one and data

*Corresponding authors.
[1]Source code available at https://github.com/Huawei-Performance-Model/Ascend-910B

layout transformation that changes the format of input tensors. These operators and tensors are automatically transformed by the compiler, and this process is transparent to the end-users. Therefore, it is hard for the high-level code-based performance model to accurately predict the performance of DNN models optimized by the compilers.

In this work, we address the aforementioned issues by proposing a new ML-enabled performance prediction framework. First, to ensure that our selected hotspot operators are representative enough to characterize the performance of the target workloads, we analyze about 20 kinds of mainstream DNN models in MindSpore's model zoo [17]. We focus on the operators whose sum of execution time is greater than 80% in each DNN model and further select about 30 kinds of hotspot operators from them to build an operator subset that includes many non-compute intensive operators. Second, to overcome the shortcomings that high-level code-based performance models are not able to consider compiler optimizations, we choose to use the kernels and tensors in the optimized computation graph as our simulation input. This graph-level analysis enables us to figure out how tensors and operators are transformed by the compiler. Then we can treat fused operators the same way as normal operators and use the optimized tensors as features for simulation.

So far, using a heuristic method to predict the performance of DNN operators can be difficult due to the complex software libraries such as cuDNN [4] that greatly change the operator performance. Here we choose to use a data-driven method to capture these low-level optimizations by running the selected operators with different configurations on the target accelerator to collect a dataset. Due to the operators' kernel-varying tensor dimensions, it is hard to build a unified ML model for all selected operators. Therefore, we then further divide operators in this dataset into different groups and build ML models for each group. Finally, we can use the performance of the selected operators to make an end-to-end performance prediction for the whole DNN model.

We evaluated our prediction framework on an accelerator called Huawei Ascend 910 [18] and a DL framework called MindSpore [19]. We build a dataset for 26 kinds of operators that include 8 kinds of fused operators to train ML models for operator performance. Then we validate the end-to-end execution time prediction accuracy on 6 mainstream DNN models that have applications in CV, NLP and recommendation with an average error of 15.4%. Besides the iteration execution time, we show the prediction of other metrics such as the performance bottleneck and the memory access.

In summary, our contributions are as follows:

- We design and implement a data-driven performance model that provides end-to-end performance prediction for a variety of DNN workloads including CV, NLP and recommendation.
- We introduce a graph-level analysis to process the varying operators in different applications and collect a labeled and extensive dataset of 26 kinds of operators that enables us to process compiler optimizations.
- Our modeling for memory access and performance bottleneck enables us to provide a more general model-system co-design than previous methods.

II. Background

A. Ascend Architecture

Fig. 1 shows an overview of the architecture of the Ascend 910, which is a kind of device to accelerate machine learning computation. To simplify the representation, we just show compute units such as CUBE for multiply accumulate (MAC), Vector for vector calculation, Scalar for scalar calculation and on-chip memory hierarchy that includes L1 Buffer, L0A/L0B Buffer and Unified Buffer. The communication between on-chip memory and high bandwidth memory (HBM) is managed by the component called the memory transfer engine (MTE).

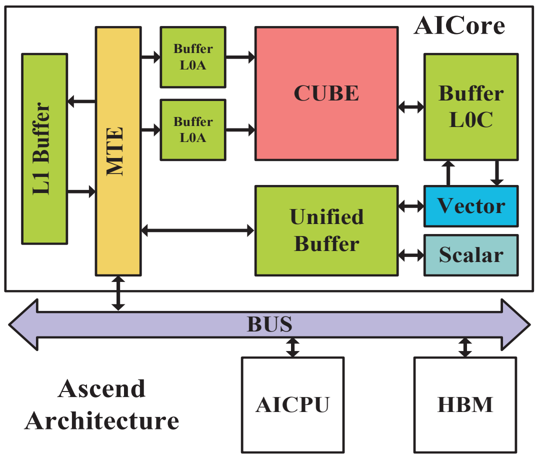

Fig. 1: Heterogeneous compute units and on-chip memory hierarchy of Ascend 910

Here we use MacRatio, VecRatio, MteRatio and Memorybound to quantify DNN model's memory access and utilization of heterogeneous compute units. These metrics as well as execution time are the prediction targets of our performance model. An operator's utilization of MAC unit such as CUBE is called MacRatio, which can be calculated by:

$$MacRatio = \frac{Cycle_{CUBE}}{Cycle_{ALL}} \tag{1}$$

where $Cycle_{CUBE}$ is the cycles of CUBE-related instructions and $Cycle_{ALL}$ is the cycles of all instructions when running an operator. VecRatio and MteRatio are the same as MacRatio, they are calculated by cycles of Vector-related and MTE-related instructions divided by cycles of all instructions. Memorybound is a metric of performance bottleneck. It can be calculated by:

$$Memorybound = \frac{MteRatio}{max(VecRatio, MacRatio)} \tag{2}$$

It is identified that if memorybound is larger than 1, there exists performance bottleneck caused by limited memory access, and the larger it is, the more serious performance bottleneck is.

B. *Optimizations for high-level code*

Data Layout Transformation Accelerators have specialized units to provide intense compute throughput for DNN training. TPU uses a systolic array, GPU uses tensor cores and Ascend uses CUBE. To efficiently perform tensor calculation on these hardware platforms, the shapes of tensors predefined in high-level code have to be transformed. For example, 4×4 tensor core in GPU requires data to be tiled into 4×4 chunks to optimize for access locality. Such optimization is called data layout transformation. In Ascend, a CUBE consists of $16 \times 16 \times 16$ multipliers and accumulators. As shown in Fig. 2, defaulted tensor shape is (N, C, H, W), where N is the batch size, C is the number of input channels, H and W are the size of the input feature map. Then this tensor is transformed into shape $(N, C1, H, W, C0)$, where $C0$ is equal to 16. Similarly, 2D-dimension tensor shape $(C1, C2)$ predefined in high-level code is transformed into $(C3, C4, C0, C0)$.

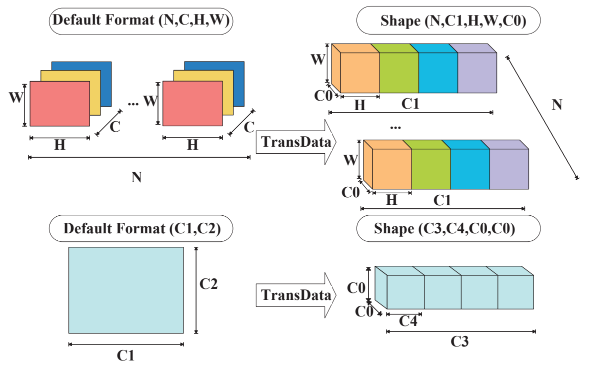

Fig. 2: Data layout transformation on Ascend 910. Tensors are reshaped to be efficiently executed on CUBE. Such kind of transformation on Ascend is called TransData, which is a kind of operator that can be found in computation graph.

Operator Fusion Computation graph is a directed acyclic graph where the nodes correspond to operators and edges correspond to data dependencies between these operators. As shown in Fig. 3, by replacing multiple nodes with a functionally equivalent one, operator fusion reduces data transferring from one node to another and brings speedup for DNN models. Compared to normal operators, fused operators possess a larger number of input and output tensors, and they exist in both forward pass and backward pass. As there is a growing tendency that more kinds of operators are supported by hardware and more complex fused operators are generated by the compiler, it is difficult for high-level operator-based performance models to accurately predict the performance of DNN models with fused operators [5].

C. *Existing performance model*

Performance models for DNN applications are based on two facts. First, the training process of DNNs can be seen as the repetition of many training iterations that have similar execution time. By analyzing a single iteration, we can predict the performance of the whole training process. Second, DNN operators are highly predictable. OPTCNN [20] and Astra [21] have shown that the execution time of each DNN operator is largely independent of the contents of the input data, and it mainly relies on the shapes of the tensors.

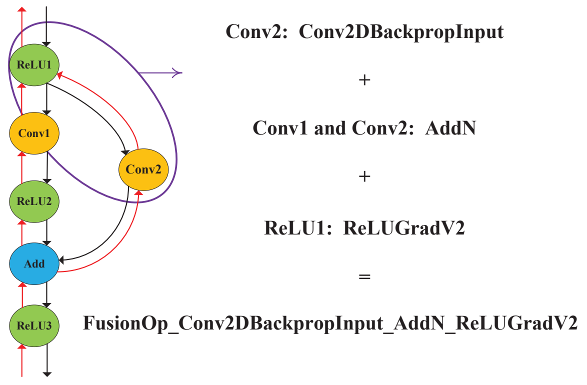
ReLU1
Conv1
Conv2
ReLU2
Add
ReLU3
Conv2: Conv2DBackpropInput
+
Conv1 and Conv2: AddN
+
ReLU1: ReLUGradV2
=
FusionOp_Conv2DBackpropInput_AddN_ReLUGradV2

Fig. 3: Operator fusion on computation graph. This is a classical residual network consists of back propagation (red), forward propagation (black), and many operators. The fusion mechanism of this fused operator is as follows: First, this fused operator takes tensor from Add operator as input and computes gradient (Conv2DBackpropInput) in Conv2. Second, it adds back propagations of Conv1 and Conv2 (AddN) and transfers the outcome to ReLU1. Finally, this fused operator computes gradient in ReLU1 (ReLUGradV2) to generate the outcome tensor.

Current performance models use either analytical methods or ML-based methods. Analytical performance models [10], [11] usually give an abstraction of the workloads and hardware and then use equations to deduce the performance of the workloads on hardware. This abstraction simplifies the difficulties of simulation and in turn reduces the prediction accuracy. ML-based performance models firstly extract features (e.g. the number of channels in the input and output tensors, kernel sizes and strides of convolution operators) from operators and generate a dataset. Then, combined with the dataset, they build ML models to predict the performance of DNN models. It has been proven that a trivial multilayer perceptron (MLP) is able to predict the execution time of DNN operators such as convolution and full-connected layer.

However, the huge search space of this data-driven method incurs a significant amount of off-line dataset collecting time and the trained ML model is sensitive to unknown applications. Meanwhile, some performance models only focus on compute-intensive operators. For example, Perfnet [8] only focus on convolution, pooling and dense layer, and Perfnet is only able to analyze simple DNNs that primarily depend on convolutions and matrix multiplications such as LeNet [22], AlexNet [23] and VGG16 [24]. Habitat [5] processes a broader range of operators, but it does not consider optimization such as operator fusion.

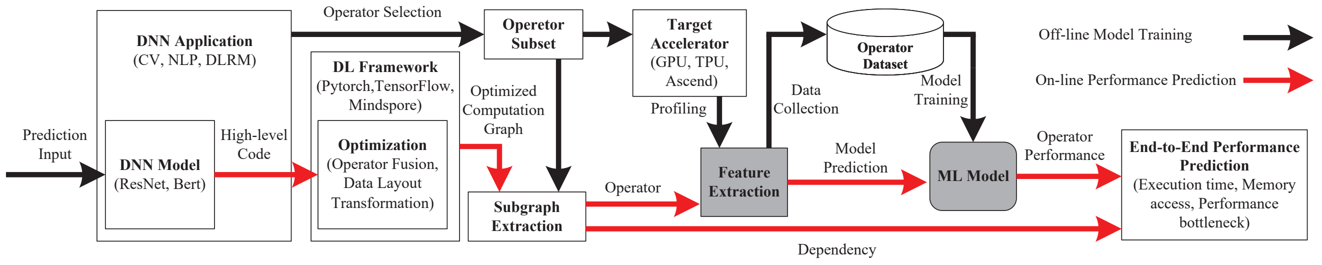

Fig. 4: An overview of performance model

III. Methodology

A. Framework overview

The overall framework of our performance model is shown in Fig. 4, which is composed of off-line model training module and on-line performance prediction module. By analyzing the execution time of operators in DNN applications, we select dominating operators to form an operator subset and neglect unimportant ones. Then, we run the selected operators with different configurations on target accelerator to generate a dataset. The features extracted from the dataset enable us to build ML models for selected operators.

The on-line prediction begins with high-level code of DNN models taken as inputs. We only need to run the high-level code until the optimized computation graph was generated, and the DNN model does not need to be trained on the target accelerator, which means that the prediction stage is hardware independent. Based on the operator subset, we use graph algorithms to extract subgraph from the computation graph. Finally, we leverage the ML model generated in off-line stage to estimate the performance of operators in the subgraph. Combining data dependencies and the performance of operators, we can make an end-to-end prediction for runtime information about DNN models.

B. Optimized computation graph for modeling

Performance modeling based on high-level code mainly faces the following two problems. First, many operators are merged into fused operators, which changes both execution time and memory access of DNN models. Second, tensors predefined in high-level code are reshaped for execution on the target hardware. Therefore, performance prediction based on the high-level code features leads to a gap between the simulation and the DNN model's actual execution.

To address these problems, we choose to use the optimized computation graph rather than high-level code as the input of our performance model. This graph-level simulation allows us to observe that many operators predefined in high-level code are fused, including operator fusion for 2 operators and even 3 operators. When operators meet the requirements for fusion, the compiler then automatically changes them into a fused operator. Therefore, we need a reverse engineering method to construct these fused kernels. As shown in Fig. 3, we need a basic block that consists of Conv1, Conv2 and ReLU1 to construct FusionOp_Conv2DBackpropInput_AddN_ReLUGradV2. Finally, we can process these fused kernels together with other normal operators such as convolution operators and activation operators.

This method also enables us to extract features directly from the computation graph to solve the problem caused by data layout transformation. As shown in Table I, we run 4 ReLU operators and 4 MatMul operators with different input tensors to explain why we choose to extract features from tensors in the computation graph rather than high-level code. For ReLU_OP1, the shape of the tensor before transformation is $(32, 3, 224, 224)$. ReLU_OP2 changes the second dimension of ReLU_OP1's tensor and its tensor shape is $(32, 16, 224, 224)$. However, these two operators predefined in high-level code with different input tensor shapes finally have equal execution time, because these tensors will finally be transformed into the same shape $(32, 1, 224, 224, 16)$. Therefore, performance prediction based on features extracted from computation graph will be more accurate.

TABLE I: Data layout transformation and execution time of operators on Ascend 910. For operators of the same kind we only change one dimension of their input tensors. For ReLU, input tensor shape (N, C, H, W) is transformed into $(N, C1, H, W, C0)$, while input tensor shape $(C1, C2)$ of MatMul is transformed into $(C3, C4, C0, C0)$.

Opname	Default tensor	Reshaped tensor	Extime(ms)
ReLU_OP1	[32,3,224,224]	[32,1,224,224,16]	0.20967
ReLU_OP2	[32,16,224,224]	[32,1,224,224,16]	0.20755
ReLU_OP3	[32,18,224,224]	[32,2,224,224,16]	0.41262
ReLU_OP4	[32,32,224,224]	[32,2,224,224,16]	0.41033
MatMul_OP1	[5120,768],[768,3]	[48,320,16,16],[48,1,16,16]	0.01057
MatMul_OP2	[5120,768],[768,16]	[48,320,16,16],[48,1,16,16]	0.01007
MatMul_OP3	[5120,768],[768,18]	[48,320,16,16],[48,2,16,16]	0.02188
MatMul_OP4	[5120,768],[768,32]	[48,320,16,16],[48,2,16,16]	0.02162

C. Operator Selection

To figure out hotspot operators, we analyzed about 20 kinds of DNN models in MindSpore's model zoo and show part of the execution time breakdown in Fig. 5, from which we can get 3 key observations to help us build an operator subset. First, although ResNet50 has overall 30 kinds of operators, BERT has overall 54 operators, and DeepFM [25] has overall 26 operators, it is feasible to build a performance model using a relatively small set of hotspot operators. The sum of the time ratios of top 8 operators is greater than 80% for all three DNN models, and we do not need to consider operators at the bottom of the ranking that lack both compute and memory significance. Second, besides compute-intensive operators, activation, normalization and other functional operators also form a non-negligible proportion of the total execution time. We define operators that use compute units such as CUBE to accelerate matrix multiplication as compute-intensive (e.g. convolution-based and matmul-based operators). Third, many operators are fused in the real execution of the DNNs.

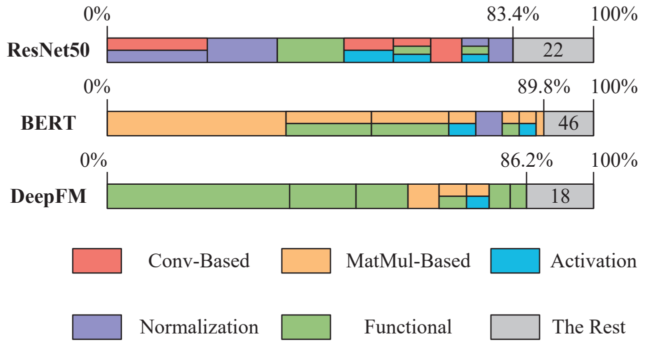

Fig. 5: Iteration execution time breakdown of three DNN models in different applications, including CV (ResNet50), NLP (BERT), and recommendation (DeepFM). We show operators whose proportions rank top 8 and mark out the number of the rest operators. These operators can be divided into conv-based (e.g. Conv2D, Conv2DBackpropInput, Conv2DBackpropFilter), matmul-based (e.g. MatMul), activation (e.g. ReLU, LayerNorm), normalization (e.g. BatchNorm, LayerNorm) and other functional operators such as Add, Mul and Transdata. For fused operators, each row represents one of its constituent operators.

In this work, we select total 26 kinds of operators in Table II. By selecting these representative operators to form an operator subset, our performance model ensures generalizability across different application domains. Another advantage of our operator subset is extensibility. It is open to add more kinds of operators to improve end-to-end prediction accuracy or remove some operators to reduce ML model training time, and users can use their own operator subset to build a unique performance model according to their actual demands.

D. Subgraph Extraction

When it comes to leveraging the performance of operators to estimate the performance of the whole DNN model, we found that the iteration training time of DNNs is calculated by summing up the execution time of all operators and data dependencies between operators are not considered in most cases. As indicated in prior work [6], ASICs such as TPU do not support multi-threading, and the execution of one kernel has no overlap with each other. Performance models for GPU [5], [9], [11] also calculate DNNs' iteration training time by summing up the execution time of all layers. In Ascend, operators are also executed one after another, and only one operator is executed at a time.

To provide a more realistic simulation of the DNN training process, we adopt Algorithm 1 that extracts subgraph from the computation graph. By traversing selected nodes in the computation graph, we use $AppendNode$ to record information about hotspot operators and use $AppendDependency$ to preserve data dependencies between operators in the original computation graph. Using $Visit$ to avoid duplicated graph Depth-First-Search, the time complexity of our algorithm is $O(n^2)$, where n is the number of nodes in the computation graph.

Algorithm 1 Subgraph Extraction

```
Input: Operator subset $S$, computation graph $G$.
Output: Subgraph of computation graph $SG$.
1:  for each node $V_0 \in G$ do
2:      if $V_0 \in S$ then
3:          APPENDNODE($SG, V_0$), DFS($V_0$)
4:      end if
5:  end for
6:  procedure DFS($V$)
7:      for each parent node $U$ of $V$ do
8:          if VISIT($U, G$) = $false$ then
9:              if $U \in S$ then
10:                 APPENDDEPENDENCY($SG, U, V_0$)
11:             else
12:                 VISIT($U, G$) = $true$, DFS($U$)
13:             end if
14:         end if
15:     end for
```

The advantages of our graph-level simulation are as follows. First, the computation graph acts as a normalized interface for performance prediction. If only the DNN model could generate the graph, then our method would be able to predict its performance. Second, data dependencies enable us to explore fine-grained parallelism of DNN training. For example, OPTCNN proposed a cost model [20] that is based on the execution time of operators and data dependencies to evaluate the performance of parallel DNN training. In that case, we could not simply add time together, and data dependency should be taken into consideration. Therefore, we see the potential that combining our performance model and data dependencies to estimate the performance of parallel DNN training with different operator scheduling policies and parallelization strategies on heterogeneous devices.

IV. Performance Model Generation

A. Data Collection

To capture the complex low-level optimizations for DNN operators, here we use a data-driven method to build ML models for operator performance. Therefore, a well-designed dataset collection strategy is needed. As shown in Fig. 6, operators in high-level code can be characterized by a set of features such as shape (N, C, H, W) and precision of input tensor. We execute operators with these configurations on the target accelerator and use MindSpore's profiling tools to record their information. Considering the example in Fig. 6, after data layout transformation and optimization (e.g. img2col) that is used to transform convolution into matrix multiplication, kernel features are stored as a tensor whose shape is $(49, 4, 16, 16)$. Finally, this operator can be characterized by InShape $(32, 1, 224, 224, 16, 49, 4, 16, 16)$, OutShape $(32, 4, 112, 112, 16)$ and its DataType. As for dataset collection for fused operators, we need to collect data by changing the configurations of the basic block such as Conv1, Conv2 and ReLU1 in Fig. 6.

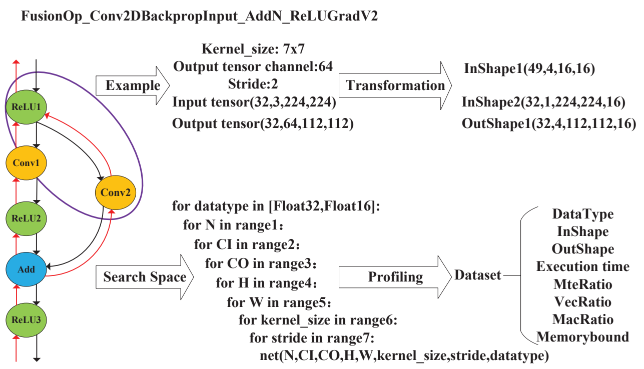

Fig. 6: Operator data collection. InShape and OutShape are shapes of all input tensors and output tensors of an operator in the optimized computation graph. DataType (0 – 1) identifies floating point precision (e.g. Float16 or Float32).

However, it is non-trivial to design a suitable search space for our data-driven method. Due to the operators' multiple features and especially for fused kernels, the search space is rather huge and hardly can we collect a dataset for all the configurations. Here, we first observe the target operator's commonly used configurations in MindSpore's model zoo and then use a uniform distribution approach to reduce the size of the search space. For example, the dataset collection ranges of Conv2D are: kernel_size (1 - 7), stride (1 - 2), image size (1 - 1200), in channel and out channel (1 - 1024) and batch size (1 - 256). Meanwhile, there could be many operators of certain configurations that could not be executed on the existing accelerator because of limited memory, and we should ignore any configurations that may result in out-of-memory.

B. Model Generation

ML-based methods mainly use MLP that comprises several layers and produces a single real number—the predicted performance. However, the variable tensor sizes (the maximum dimension of an operator's input tensors and output tensors) in different applications make it hard to use a unified MLP to predict these complex operators. As shown in Fig. 7, the tensor size in ResNet50 is mainly 5, and the tensor size in DeepFM, BERT and ViT is usually 3 or 4. Even for operators of the

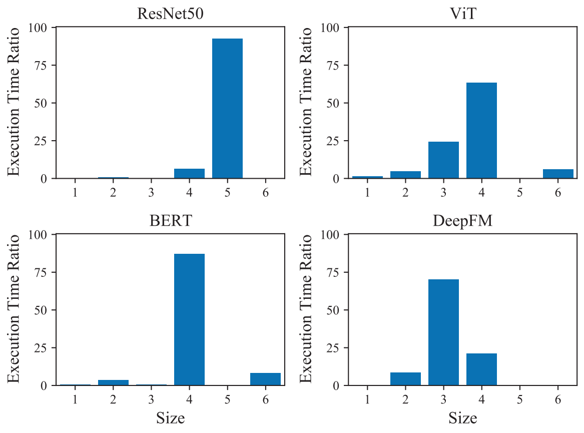

Fig. 7: Operators of different sizes and their execution time ratios in DNNs

same kind, their sizes could be different. For example, one of LayerNorm in ViT has input tensor shape $(256, 50, 768)$ and its size is 3, while one of LayerNorm in BERT has input tensor shape $(32768, 768)$ that is then transformed into $(48, 2048, 16, 16)$ and its size is 4. If we use only one MLP, then the vacancy caused by variable sizes would significantly decrease the prediction accuracy.

One possible solution is to build one MLP for each kind of operator and each kind of size. However, this approach is rather time-consuming for our numerous hotspot operators, operator sizes and prediction targets. As shown in Fig. 8, we build one MLP model for operators of the same size and use the feature OpType to distinguish among different operators. We build 3 kinds of ML models for size 3, 4, 5 and ignore other sizes that hardly appear in mainstream DNNs. When it comes to operator performance prediction, we first identify its tensor size and use the corresponding MLP to predict its performance.

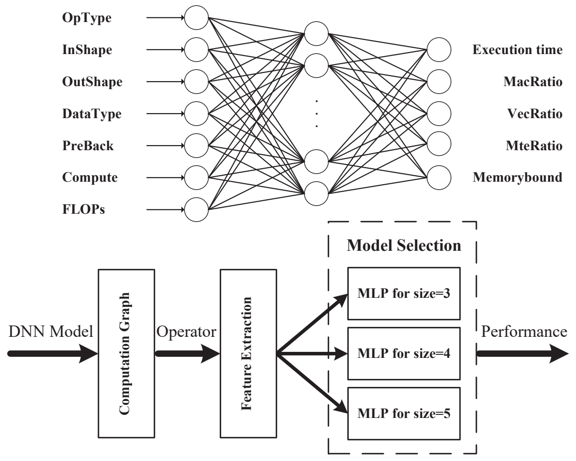

Fig. 8: Runtime procedure of performance model

C. Feature Extraction

To achieve higher prediction accuracy, we use feature engineering to provide more features besides InShape, OutShape and DataType. As shown in Fig. 8, PreBack (0 – 1) identifies whether an operator is a forward pass operator or a back propagation operator. Compute (0 – 1) identifies whether an operator is compute-intensive or not. Paleo [10] uses floating point operations (FLOPs) to analyze the execution time of operators, from which we can learn that there is a strong positive correlation between FLOPs and execution time. For compute-intensive operators and other operators, we can use equations:

$$FLOPs_{compute-intensive} = \prod_{i=0}^{N} MUL\_All(Tensor_i) \quad (3)$$

$$FLOPs_{others} = \sum_{i=0}^{N} MUL\_All(Tensor_i) \quad (4)$$

where N is the number of operator's input tensors and $MUL\_All(Tensor_i)$ is the product of the integers over all dimensions of input tensor i. The FLOPs of a fused operator can be calculated by the sum of the FLOPs of all its primitive operators.

Meanwhile, some features need to be further processed. We use the one-hot encoding of OpType, the logarithm of FLOPs and the standardization of InShape and OutShape. Finally, all these features are arranged in the form of a one-dimensional vector to be fed into the MLP model. Some operators such as fused operators may have a larger number of input and output tensors and we set the value of all vacancies of InShape and OutShape to 0.

V. EVALUATION

A. Model Training

As shown in Table II, by analyzing DNN models in MindSpore's model zoo, we selected 26 kinds of operators to form our operator subset and run these operators with different configurations on Ascend 910B to collect a dataset. Then this dataset was split into a training set (90%) and a test set (10%). The feasibility of this ML-based method has been proven by many prior works [5], [9], and here we do not focus on how to build ML models. Our code for training performance models is written in Pytorch and executed on GPU using CUDA 11.3. We use the Adam optimizer [26] and the loss function is mean absolute percentage error (MAPE).

As shown in Fig. 9, we trained 12 target-specific ML models for different sizes and prediction targets. In most cases, it costs about 300 epochs to converge to a stable loss value. The size 3 model tends to have a lower loss value and converge earlier than the size 4 model and size 5 model. This is probably because operators whose sizes are equal to 3 have simpler calculation rules compared with MatMul of size 4 and Conv2D of size 5. These results confirm that not only execution time of operators is predictable, but also other information such as MteRatio, VecRatio and Memorybound are predictable. We do not show the MacRatio because there are some of our selected operators that do not use CUBE, and the values of their MacRatio are always equal to zero. Therefore, we could not use MAPE as loss function to train ML models for MacRatio. After the training was completed, every trained model then became a part of our performance model.

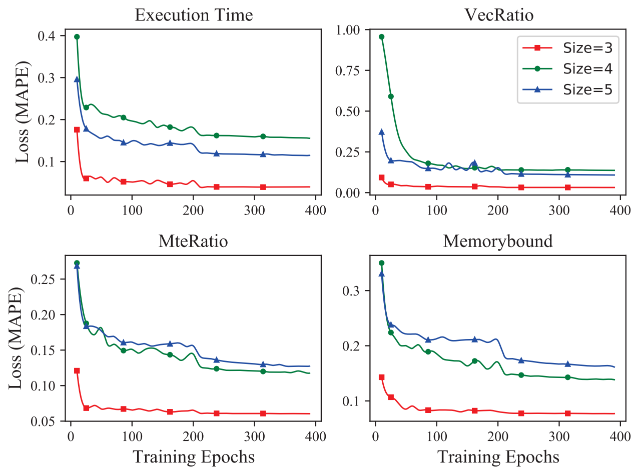

Fig. 9: The training process of ML models

B. Operator Performance

The evaluation index of our method is the prediction accuracy of all the prediction targets that we have mentioned in Section II-A, and Table II shows the prediction accuracy of our selected operators. The column named Size is used to indicate the sizes to which this operator belongs. We tested individual operator performance using our operator dataset. For an operator that belongs to more than one kind of size, we need to use the ML model of the corresponding size to get its prediction error and then use the arithmetic mean value of all errors to represent the accuracy of this operator. The result shows that the errors of most operators are less than 20%, and the average prediction errors of execution time, MteRatio, VecRatio and Memorybound are 8.88%, 8.93%, 6.93% and 13.8% respectively.

These selected operators can be further divided into conv-based, matmul-based, normalization, pooling, activation and other kinds of functional operators. We noticed that the prediction errors of operators that have simpler calculation rules tend to lower. For example, the execution time errors of compute-intensive MaxPool, Conv2D and MatMul are 16.2%, 7.64% and 15.9% respectively, while the errors of Mul, AddN and GeLU are 4.60%, 4.19% and 0.51% respectively. These selected operators also include 8 kinds of fused operators that are of high importance. For example, in ResNet50, all Conv2D operators execute in the form of fused operator FusionOp_Conv2D_BNTrainingReduce in the computation graph and all ReLU operators execute in the form of FusionOp_BNTrainingUpdate_Add_ReLUV2 and FusionOp_BNTrainingUpdate_ReLUV2. Finally, the prediction

TABLE II: Performance prediction error (MAPE) of our selected operators

operator type	operator name	Size	Execution Time	MteRatio	VecRatio	Memorybound
Conv-Based	Conv2D	5	7.64%	11.0%	16.7%	15.4%
	Conv2DBackpropFilter		11.3%	15.2%	13.0%	24.4%
	Conv2DBackpropInput		10.6%	11.6%	14.5%	16.2%
	FusionOp_Conv2DBackpropInput_AddN_ReLUGradV2		7.55%	10.4%	7.38%	16.0%
	FusionOp_Conv2DBackpropInput_ReLUGradV2		10.2%	11.5%	7.39%	14.4%
	FusionOp_Conv2D_BNTrainingReduce		10.1%	12.8%	4.99%	15.8%
MatMul-Based	MatMul	4	15.9%	8.51%	9.51%	22.9%
	FusionOp_MatMul_GeLU		4.67%	10.2%	2.35%	13.7%
	FusionOp_MatMul_GeLUGrad		2.48%	7.01%	0.96%	13.6%
	FusionOp_MatMul_Mul		10.7%	6.95%	6.64%	23.2%
Normalization	BNInfer	5	8.50%	6.16%	7.02%	11.2%
	BNTrainingReduceGrad		4.91%	5.32%	3.81%	8.90%
	BNTrainingUpdate		7.23%	9.90%	7.01%	11.4%
	BNTrainingUpdateGrad		9.20%	6.75%	4.47%	12.2%
	FusionOp_BNTrainingUpdate_Add_ReLUV2		14.6%	13.3%	9.73%	18.6%
	FusionOp_BNTrainingUpdate_ReLUV2		15.1%	19.5%	11.1%	19.9%
	LayerNorm	3,4	17.9%	12.3%	2.45%	15.2%
	LayerNormXBackpropV2		5.28%	6.13%	2.40%	8.96%
Actication	GeLU	3,4	0.51%	6.77%	0.26%	6.82%
	GeLUGrad		1.91%	2.89%	0.16%	2.97%
	ReLUGradV2	5	7.50%	6.02%	5.15%	12.5%
pooling	MaxPoolGradWithArgmax	5	16.8%	6.48%	7.81%	11.6%
	MaxPoolWithArgmax		16.2%	11.4%	16.0%	16.3%
Others	AddN	3,4,5	4.19%	3.96%	5.37%	6.96%
	Mul		4.60%	4.73%	8.22%	10.5%
	TransData	4,5	5.31%	5.49%	5.70%	10.3%

errors of these fused operators for four prediction targets are 9.43%, 11.5%, 6.32% and 16.9% respectively.

C. End-to-End Prediction

We evaluated the end-to-end execution time and memorybound prediction accuracy of our performance model using a variety of cross-domain DNN models. These models and their datasets listed in Table III are developed under MindSpore's model zoo [17]. We first trained these DNN models on Ascend 910B for several iterations with their corresponding datasets and used Huawei Mindspore's profiling tools to get the measured execution time and memorybound of every single operator as our benchmark for evaluation. Then we used the optimized computation graph produced by the compiler as the input of our performance model to get the predicted operator performance.

TABLE III: Configurations of DNN models for evaluation. CV applications can be further divided into Image Recognition and Object Detection.

Application	DNN Model	Dataset
Image Recog.	ResNet50	CIFAR-100 [27]
Image Recog.	Inception v3 [28]	CIFAR-100
Image Recog.	ViT	ImageNet2012 [29]
Object Detect.	YOLO v5 [30]	COCO2017 [31]
NLP	BERT	zhwiki [32]
Recommendation	DeepFM	Dataset in [25]

As indicated in Section III-D, by traversing selected nodes in the computation graph and adding their execution time together, we can get the DNN model's measured iteration execution time:

$$T_{measured} = \sum_{i=0}^{N} T_{M(i)} \tag{5}$$

where N is the number of selected operators in the computation graph and $T_{M(i)}$ is the measured execution time for operator i. Predicted iteration execution time $T_{predicted}$ can be calculated in the same way using predicted operator execution time $T_{P(i)}$.

Fig. 10 shows prediction errors for these aforementioned models. We tested three different batch sizes for each DNN model. The result shows that the average prediction error across all DNN models is 15.4%, and the average prediction errors across all ResNet-50, Inception v3, ViT, YOLO v5, BERT, and DeepFM with different batch sizes are 14.3%, 19.1%, 16.5%, 15.8%, 12.8%, and 13.8% respectively.

To use operator performance to evaluate the memorybound of the whole DNN model, we can use equations:

$$Memorybound_{measured} = \frac{\sum_{i=0}^{N} MB_{M(i)} * T_{M(i)}}{T_{measured}} \tag{6}$$

$$Memorybound_{predicted} = \frac{\sum_{i=0}^{N} MB_{P(i)} * T_{P(i)}}{T_{predicted}} \tag{7}$$

where $MB_{M(i)}$ is the measured memorybound for operator i and $MB_{P(i)}$ is the predicted memorybound for operator i.

Fig. 11 shows how measured and predicted memorybound change as batch sizes grow. For different batch sizes, the measured memorybound of ResNet50 are 0.74, 0.73 and

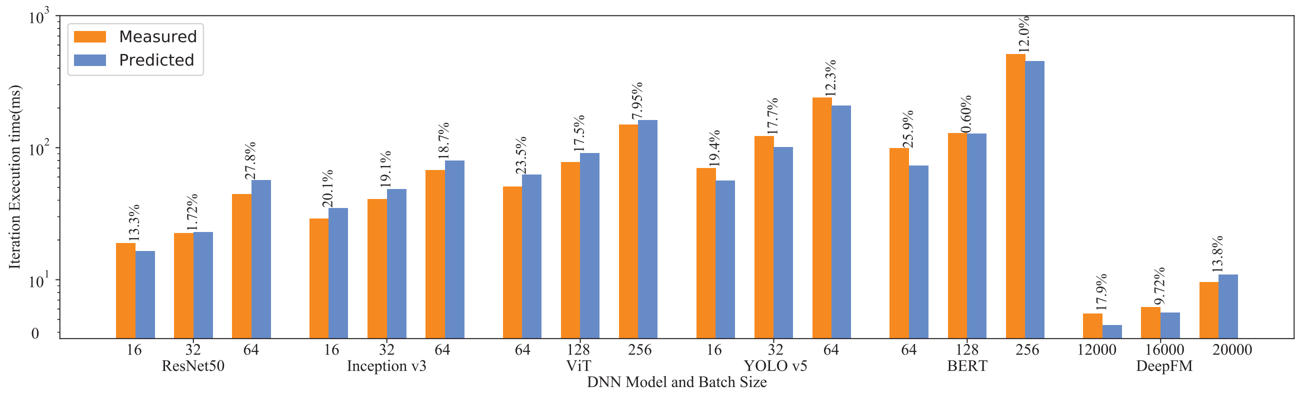

Fig. 10: End-to-End execution time prediction accuracy for cross-domain DNN models

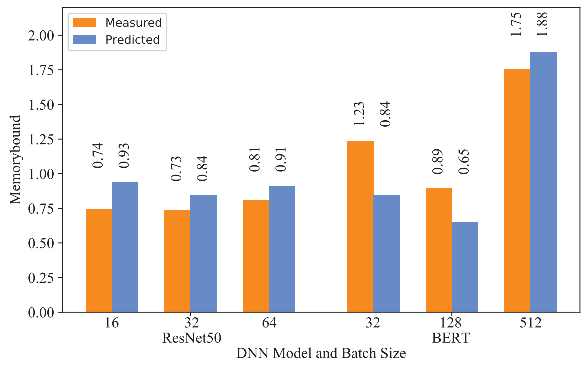

Fig. 11: The measured and predicted memorybound of two DNN models for different batch sizes

0.81, and the measured memorybound of BERT are 1.23, 0.89 and 1.75. This result shows that memorybound is not proportional to batch sizes and varies from model to model. Based on the definition of memorybound, if the target DNN model's memorybound is larger than 1, then there could be a performance bottleneck caused by limited compute power, and we could accelerate this compute-intensive DNN model by adding compute units on the next generation of chips. Otherwise, there could be a performance bottleneck caused by limited memory access, and we could accelerate this memory-intensive DNN model by improving memory access efficiency (e.g. increasing bandwidth). Therefore, our modeling for memorybound quantifies the DNN model's performance bottleneck and enables us to provide important metrics for model-system codesign.

VI. Conclusion and Future Work

In this work, we design and implement a new ML-enabled performance model for cross-domain DNN applications. By performing a comprehensive graph-level analysis of the real-world DNN applications, we select 26 kinds of hotspot operators and collect an extensive and labeled operator dataset on Ascend 910. We then further divide these operators into 3 groups according to the number of their tensor dimensions and use feature engineering to build more accurate ML models. Using the optimized computation graph as the input, our method overcomes the shortcomings that previous works do not consider optimizations such as operator fusion and data layout transformation and achieves 15.4% average end-to-end execution time prediction accuracy for 6 mainstream DNN models.

Although Ascend 910 is the off-the-shelf device to accelerate machine learning computation, our work is still very meaningful. Our modeling for the MteRatio, VecRatio, and Memorybound quantifies the performance bottleneck of DNN models and offers guidance for next-generation deep learning chip design. For example, we may assume that Ascend 910 has 16 cores now, and we would like to know how the performance of the DNN model changes when the number of cores extends to 32 for the next generation of Ascend 910. Using our performance model, we can predict DNN models' performance on the next generation of Ascend 910 using compute units and memory access together instead of simply dividing the execution time by two.

Our future work is focused on three different aspects: 1) targeting other deep learning frameworks such as Tensorflow, Pytorch and accelerators such as GPU, TPU. For example, in Ascend and Mindspore, our work is based on Huawei's official profiling tools. In GPU and other deep learning frameworks, we may need to use other profiling tools such as NVIDIA Nsight Compute [33]. 2) combining our performance model with multi-accelerator DNN training such as data parallelism and model parallelism. And 3) exploring more fine-grained operator scheduling policies and parallelism strategies with our performance model.

Acknowledgment

We would like to thank other people in the Advanced Computer System Architecture (ACSA) Laboratory of University of Science and Technology of China for helping in sampling data. This work is supported by the National Natural Science Foundation of China (Grant No. 62102389).

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[2] C. Leary and T. Wang, "Xla: Tensorflow, compiled," *TensorFlow Dev Summit*, 2017.

[3] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[4] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[5] X. Y. Geoffrey, 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)*, 2021, pp. 503–521.

[6] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, "A learned performance model for tensor processing units," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 387–400, 2021.

[7] Z. Lin, L. Feng, E. K. Ardestani, J. Lee, J. Lundell, C. Kim, A. Kejariwal, and J. D. Owens, "Building a performance model for deep learning recommendation model training on gpus," *arXiv preprint arXiv:2201.07821*, 2022.

[8] Y.-C. Liao, C.-C. Wang, C.-H. Tu, M.-C. Kao, W.-Y. Liang, and S.-H. Hung, "Perfnetrt: Platform-aware performance modeling for optimized deep neural networks," in *2020 International Computer Symposium (ICS)*. IEEE, 2020, pp. 153–158.

[9] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE international conference on big data (Big Data)*. IEEE, 2018, pp. 3873–3882.

[10] E. R. S. Hang Qi and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[11] Z. Pei, C. Li, X. Qin, X. Chen, and G. Wei, "Iteration time prediction for cnn in multi-gpu platform: modeling and analysis," *IEEE Access*, vol. 7, pp. 64 788–64 797, 2019.

[12] 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 *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[13] 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*, vol. 32, 2019.

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

[15] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] Huawei, "Mindspore's model zoo," 2021, https://gitee.com/mindspore/models/.

[18] H. Liao, J. Tu, J. Xia, H. Liu, X. Zhou, H. Yuan, and Y. Hu, "Ascend: a scalable and unified architecture for ubiquitous deep neural network computing: Industry track paper," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 789–801.

[19] Huawei, "Mindspore," 2020, https://www.mindspore.cn/en.

[20] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks." *Proceedings of Machine Learning and Systems*, vol. 1, pp. 1–13, 2019.

[21] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting predictability to optimize deep learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 909–923.

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[24] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[25] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference for Learning Representations (ICLR'15)*, 2015, pp. 909–923.

[27] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[28] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[29] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[30] G. Jocher, A. Stoken, J. Borovec, A. Chaurasia, L. Changyu, A. Laughing, A. Hogan, J. Hajek, L. Diaconu, Y. Marc *et al.*, "ultralytics/yolov5: v5. 0-yolov5-p6 1280 models aws supervise. ly and youtube integrations," *Zenodo*, vol. 11, 2021.

[31] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[32] Wikimedia, "zhwiki," 2021, https://dumps.wikimedia.org/zhwiki/.

[33] NVIDIA, "Nvidia nsight compute profiling tool," 2022, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.