EOP: Efficient Operator Partition for Deep Learning Inference over Edge Servers

Yuanjia Xu*
xuyuanjia2017@otcaix.iscas.ac.cn
University of Chinese Academy of Sciences
Institute of Software, Chinese Academy of Sciences
Beijing, China

Heng Wu*
wuheng@iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
Beijing, China
Nanjing institute of software technology
Nanjing, China

Wenbo Zhang†
zhangwenbo@otcaix.iscas.ac.cn
Institute of Software, Chinese Academy of Sciences
State Key Laboratory of Computer Sciences, Institute of Software, Chinese Academy of Sciences
Beijing, China

Yi Hu
huyi19@otcaix.iscas.ac.cn
University of Chinese Academy of Sciences
Institute of Software, Chinese Academy of Sciences
Beijing, China

Abstract

Recently, Deep Learning (DL) models have demonstrated great success for its attractive ability of high accuracy used in artificial intelligence Internet of Things applications. A common deployment solution is to run such DL inference tasks on edge servers. In a DL inference, each operator takes tensors as input and run in a tensor virtual machine, which isolates resource usage among operators. Nevertheless, existing edge-based DL inference approaches can not efficiently use heterogeneous resources (e.g., CPU and low-end GPU) on edge servers and result in sub-optimal DL inference performance, since they can only partition operators in a DL inference with equal or fixed ratios. It is still a big challenge to support partition optimizations over edge servers for a wide range of DL models, such as Convolution Neural Network (CNN), Recurrent Neural Network (RNN) and Transformers.

In this paper, we present EOP, an Efficient Operator Partition approach to optimize DL inferences over edge servers, to address this challenge. Firstly, we carry out a large-scale performance evaluation on operators running on heterogeneous resources, and reveal that many operators do not follow similar performance variation when input tensors change. Secondly, we employ three categorized patterns to estimate the performance of operators, and then efficiently partition key operators and tune partition ratios. Finally, we implement EOP on TVM, and experiments over a typical edge server show that EOP improves the inference performance by up to $1.25-1.97\times$ for various DL models compared to state-of-the-art approaches.

CCS Concepts: • **General and reference** → **Performance**; • **Software and its engineering** → **Runtime environments**; *Virtual machines*; • **Computer systems organization** → **Data flow architectures**.

Keywords: deep learning inferences, operator partition, heterogeneous resources, edge servers, input tensors

ACM Reference Format:
Yuanjia Xu, Heng Wu, Wenbo Zhang, and Yi Hu. 2022. EOP: Efficient Operator Partition for Deep Learning Inference over Edge Servers. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '22), March 1, 2022, Virtual, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3516807.3516820

*Both authors contributed equally to this research.
†Corresponding author

VEE '22, March 1, 2022, Virtual, Switzerland

ACM ISBN 978-1-4503-9251-8/22/03…$15.00
https://doi.org/10.1145/3516807.3516820

1 Introduction

Running various Deep Learning (DL) models, such as Convolutional Neural Network (CNN) [22], Recurrent Neural Network (RNN) [20], Yolo [21] and Transformers [25] over edge servers can empower artificial intelligence Internet of Things applications. It has been proofed as an important way to achieve significant performance improvement of DL models [15, 23, 30]. In this paper, we focus on **partition optimization** to run partitioned operators in parallel to use heterogeneous resource within an edge server. And we use

execution time of an operator or subgraph [1] to demonstrate its **performance**.

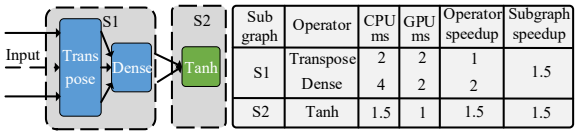
Unlike cloud servers have multiple powerful CPUs and GPUs [24], an edge server is always a desktop or laptop computer with limited power of CPUs (e.g., Intel i7-7700) and GPUs (e.g., NVIDIA GTX2060) which have varying computation throughput [8, 11, 29]. During DL inference, the edge server receives a trained DL model from the cloud server, and collects data from the end devices as input tensors. Then it continues the inference computation and returns result to end devices. After partitioning, operators in DL interfere can run parallelly with parts of its input tensors on heterogeneous resources, and get improved performance.

However, existing edge server based operator partition approaches using static partition ratios. They still suffer from a set of limitations: the assumption of operator partition on CPU and GPU with comparable computational throughput [15], or only on homogeneous devices [27] results in equal partition ratios (each CPU and GPU have 1/2 tensors) and may make CPU/GPU idle. Manually setting fixed operator partition ratios for different operators or subgraphs [23, 29, 30] can only work for heterogeneous resources with same computation architecture (e.g., TPUv2/v3), which cannot be used in CPU/GPU heterogeneous computation (LLVM in CPU and CUDA in GPU).

To address the above limitations, we set the dynamic partitions ratios for operators to gain the full computation potential from heterogeneous resources. As the example shown in Fig. 1(a), it includes three operators $\{Transpose, Dense, Tanh\}$ of two subgraphs $\{S1, S2\}$. These operators are common used in Transformers models [25]. In addition, the speedup of $S1$ is 1.5, and the speedups of *Transpose* and *Dense* are 1 and 2, respectively. Total operator execution time can be reduced by partitioning on CPU and GPU. Existing approaches may get different partition results.

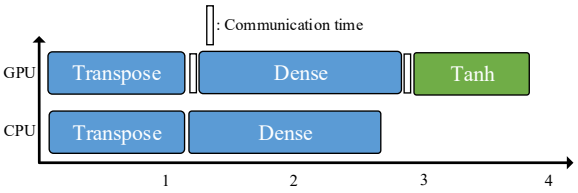
1) **Equal partition ratio**. As shown in Fig. 1(b), this approach focuses on selecting an optimal number of resources that can balance communication overhead and execution time. It tries to split an operator to different number of sub-operators, and then runs them using an empirical partition (e.g., an equal proportions algorithm). After that, it searches for an optimal partition with a smaller operator execution time. The total time can be reduced from 5s to nearly 4s. Obviously, this approach can hardly use heterogeneous resources efficiently. It would cause a large time difference between two sub-operators of *Dense*, and operator *Tanh* has to wait until CPU has been released.

2) **Fixed partition ratio**. As shown in Fig. 1(c), this approach sets partition ratios based on subgraph speedup, both *Transpose* and *Dense* in $S1$ put $1.5/(1+1.5) = 0.6s$ tensor on GPU Unfortunately and $1/(1+1.5) = 0.4s$ on CPU. The total time can be further reduced to 3.3s. However, both sub-operators of *Transpose* and *Dense* need to wait due to subgraph speedup are different from operators'. This may cause CPU/GPU idle and hinder DL inference performance improvement.

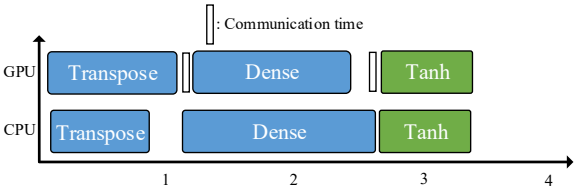


Sub graph	Operator	CPU ms	GPU ms	Operator speedup	Subgraph speedup
S1	Transpose	2	2	1	1.5
	Dense	4	2	2	
S2	Tanh	1.5	1	1.5	1.5

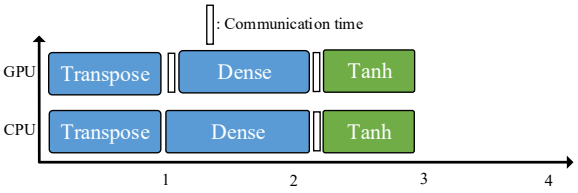
(a) Operators/subgraphs have different speedups without partition



(b) Equal partition ratio: $(0.5, 0.5)$ for *Transpose* and *Dense*



(c) Fixed partition ratio: $(0.6, 0.4)$ for *Transpose* and *Dense*



(d) Dynamic partition ratio: $(0.7, 0.3)$ for *Transpose*, $(0.4, 0.6)$ for *Dense*

Figure 1. Partition results when using different approaches

Actually, different input tensors (decided by partition ratios) may affect operator speedups on heterogeneous resources. CPU has smaller number but faster cores, operators with smaller input tensors can run faster on it. GPU has many but slower cores, operators with large input tensors may get better performance [32]. That means *Transpose* may have speedup < 1 and *Dense*'s > 2 when partitioning, respectively. As the example shown in in Fig. 1(d), if we can accurately estimate each operator's speedup with dynamic partition ratios, we can select the ratios with least execution time.

In this paper, we present EOP, an Efficient Operator Partition method to optimize DL inferences over edge servers,

[1] A DL model can usually be partitioned into several subgraphs, and each subgraph contains a set of operators.

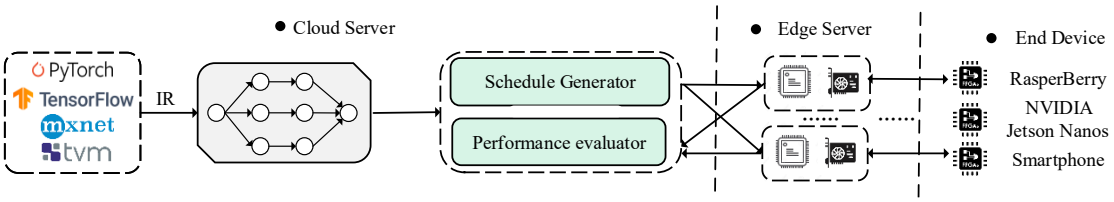


Figure 2. A typical deployment pipeline of DL inference on edge-cloud environment.

to address this challenge. We believe our work makes the following advancements.

- We carry out a large-scale performance evaluation on operators from ten typical DL models, and reveal that the performance of many operators on edge servers is non-linear when input tensor change.
- We summarize three categorized performance variation patterns that can be used to accurately estimate operator performance. We then efficiently partition key operators and tune partition ratios to improve DL inference performance.
- We implement EOP on TVM [6]. Our experiments on Transformers [25] and RNNs [1] show that EOP can improve inference performance by 1.25-1.97×, and can also obtain comparable improvements for traditional CNNs compared to state-of-the-art approaches.

2 Background, Problem and Challenge Analysis

This section first introduce the deployment pipeline of DL inference on edge-cloud environment, and reveals that DL inference performance are affected by many operators. In addition, operator speedups are dynamic with different input tensors, so as to their optimized partition ratios. Then, we propose several challenges to guide the design of EOP.

2.1 Background

As shown in Fig. 2, the deployment pipeline of DL inference comprises several different stages:

1) mainstream deep learning frameworks like Tensorflow [4], Pytorch [19] can be used on cloud servers to train a DL model by a DL developer, with the goal of maximized accuracy.

2) the trained model can be exported to a computation graph described by an Intermediate Representation (IR) language (Halide in TVM [6]), some graph level optimizations like operator fusion can be applied.

3) DL compile optimization tools like TVM can generate many execution schedules for each operator with certain input tensors, while its performance evaluator can put each operator with different schedules on edge servers and collects execution time. Only the operator's schedule with the least execution time can be finally deployed on edge server after many tries.

4) with each optimized operator in the DL model, the edge server can run DL inference by the input tensors from a wide range of end devices.

Note that current TVM [6] does not support operator partition on heterogeneous resources, and we modify IR by repeating target operators and adding tensor *split* and *concat* operators (detailed in Section 3) to overcome this limitation.

2.2 Problem Analysis

Equal partition ratio can hardly get the optimal DL inference performance. As the operator speedup analysis results on an edge server, operators may have 4× varied speedups in a RNN model shown in Fig. 3(a). Equally partitioning all operators will loss a lot of performance improvement potential, since partitioned operators may still have long execution time on CPU or GPU.

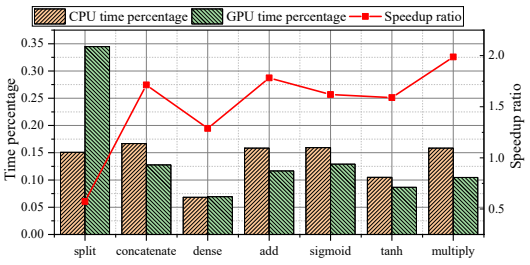
Fixed partition ratio can be hardly used for operators in DL models other than CNNs. Fig. 3(b) shows the variation of execution time and speedup in CNN subgraphs (ResNet-18). We can see that operator *Covn2D* dominates nearly all inference time (nearly 99%) in CNNs. Since *Conv2D* has linear performance when the input changes [23], it is lucky that partition ratios can be set by subgraph speedups. But for RNNs, dozens of operators consume 5%-35% of total inference time, and most of them have dynamic speedups when input tensors change (Section 2.3). Therefore, such distributions make fixed partition ratio inefficient.

2.3 Challenge Analysis

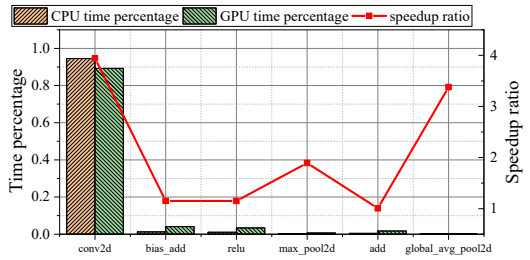
How to estimate the performance of each operator accurately? To understand this challenge, we perform an in-depth analysis by carrying out a large-scale evaluation on operators from different DL models as follows.

1. CNN and Yolo models [3, 21] include of ResNet-50, Inception V3, MobileNet, Yolo v2 and Yolo v3.
2. RNN models [1] include LSTM and GRU.
3. Transformer models [25] include BERT.

For each DL model, we consider three dimensions to cover most operator performance variation when changing inputs:



(a) RNN (LSTM)



(b) CNN (ResNet-18)

Figure 3. Operator execution time percentage and speedup variation in CNN and RNN.

1. Batch size (*batchsize*). It ranges from 2^0 to 2^{10} with a span 16.
2. Height-weight (*HW*): It ranges from (32,32) to (299,299) with a span 16 in CNNs and Yolo, and ranges from (1,40) to (1,200) with a span 10 in RNNs and Transformers.
3. Height-weight-weight (HW_1W_2): It ranges from (32,32,32) to (4096,4096,4096) with a span 32 in RNNs and Transformers.

As shown in Fig. 4, it is clear that the operator *Conv2D* performance shows linear growth when changing *batchsize*. The other operators may have very different patterns on CPU and GPU. Obviously, different performance variation patterns on CPU or GPU makes operators have dynamic speedups (*cputime*/*gputime*) with different input tensors, and operator partition should consider these patterns to improve more DL inference performance.

How to control partition overhead and get optimal DL inference performance? After estimating each operator, our goal is to minimize the execution time of all operators. However, as shown in TABLE 1, the number of operators is very large (hundreds of operators in a DL model). To make matters worse, mainstream DL frameworks [4, 19] now support more than 2, 000 kinds of operators. Partitioning them will introduce a significant overhead (communication between CPU and GPU), which is unacceptable for most inference scenarios. Thus, it would be costly to consider all operators and it is also a challenge to minimize the number of partitioned operators.

Table 1. Number of subgraphs and operators in different DL models.

Kind	Model	Number of subgraphs	Number of operators
CNN	InceptionV3	47	416
	ResNet-18	18	91
	MobileNet	28	113
	SqueezeNet	15	102
Yolo	Yolo v3	106	354
RNN	LSTM	2	347
	GRU	2	337
Transformers	BERT	12	903

3 Key Technologies

In DL inference scenarios, each operator before partitioning needs an exclusive resource (supported by TVM), and all operators run based on common default settings of mainstream DL frameworks [15, 23, 27, 29, 30]. After partitioning, some operators can be partitioned into two sub-operators on edge servers that can improve DL inference performance.

3.1 Problem Statement

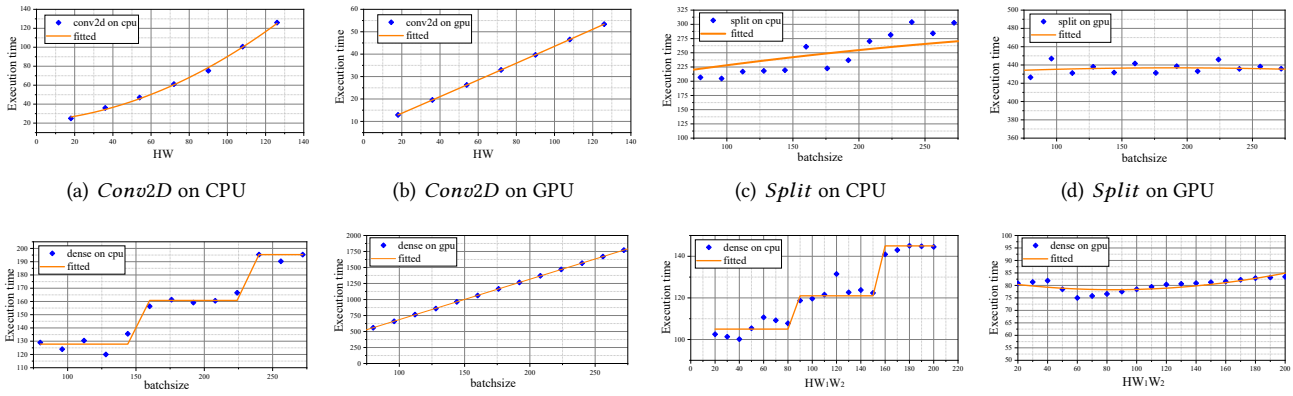
We use G to represent a DL's directed acyclic computation graph, each vertex $op_i \in G$ is an operator, and each edge $(op_i, op_j) \in G(i \neq j, 1 \leq i, j \leq N)$ denotes a dependency between operator op_i's output and operator op_j's input. In addition, each operator before partitioning can run either on CPU or GPU, which is defined as $R = \{CPU, GPU\}$.

As shown in Eq. 1, for an input tensor dimension d, an operator partition $p(op_i, d)$ means op_i can be partitioned into two sub-operators op_i^{CPU} and op_i^{GPU}. Thus op_i may aggregate computation power from both CPU and GPU simultaneously. Here, $op_i^{CPU}(\alpha d)$ and $op_i^{GPU}(\beta d)$ denote two sub-operators receive inputs αd and βd, respectively. The partition ratio is defined as (α, β):

$$\begin{aligned} &p(op_i, d) = \{op_i^{CPU}(\alpha d), op_i^{GPU}(\beta d)\} \\ &subject\ to:\ \alpha + \beta = 1, \alpha, \beta \in [0, 1] \end{aligned} \tag{1}$$

Further, $cm(op_i^{CPU}, op_i^{GPU})$ denotes the introduced communication overhead after partitioning op_i. It can be treated as the execution time of three specific operators. These are *Split*, *Device_copy* and *Concatenate* in TVM [2]. In practice, these operators always fit a constant pattern and we use this pattern to estimate communication overhead.

Based on $p(op_i, d)$ and $cm(op_i^{CPU}, op_i^{GPU})$, the minimized execution time for sub-operators op_i^{CPU} and op_i^{GPU} running on *CPU* and *GPU* is denoted by $T(p(op_i, d), R)$ (Eq. 2). Note that $cm(op_i^{CPU}, op_i^{GPU})$ can occur on both CPU and GPU decided by tensor locations, and operator execution time are decided by the maximum sub-operator time since the two



(a) *Conv2D* on CPU (b) *Conv2D* on GPU (c) *Split* on CPU (d) *Split* on GPU

(e) *Dense* on CPU (f) *Dense* on GPU (g) *Dense* on CPU (h) *Dense* on GPU

Figure 4. Different performance patterns: we collect different operator execution time (μs) on an edge server by changing their inputs from *batchsize*, HW, HW_1W_2 dimensions. Other time-consuming operators have similar performance patterns.

operators can run parallelly.

$$T(p(op_i, d), R) = Max\{t(op_i^{GPU}(\beta d), GPU), \\ t(op_i^{CPU}(\alpha d), CPU)\} + cm(op_i^{CPU}, op_i^{GPU}) \quad (2)$$

In this context, an optimized partition $p(op_i, d)$ can be described by Eq. 3. It means the operator partition $p(op_i, d)$ can shorten op_i's execution time.

$$t(op_i, r_k) > T(p(op_i, d), R) \quad (3)$$

In this paper, we use $AT(G, R)$ to denote all operator execution time, and our goal is to minimize $AT(G, R)$, as shown in Eq. 4:

$$AT(G, R) = \sum_{i=1}^{N} T(p(op_i, d), R), \ s.t.: \ (1 \sim 3) \quad (4)$$

3.2 Estimating Operator Performance

In this paper, we conclude three different performance variation patterns as shown in Fig. 4.

Linear variation pattern. Execution time of some operators like *Conv2D* and *Dense* are linearly increased when inputs grow. As shown in Fig. 4(b) and Fig. 4(f), such linearity can mostly be found on GPU, which benefits from GPU's parallel computation architectures.

Staircase variation pattern. Execution time of some operators like *Dense*, *Multiply* always tend to staircase increase when inputs grow. As shown in Fig. 4(e) and Fig. 4(g), such staircase variation can mostly be found on CPU environments.

Constant pattern. For some operators like *Split* and *Dense*, the execution time can be treated as a constant in some partition dimensions. As shown in Fig. 4(c), Fig. 4(d) and Fig. 4(h), changing *batchsize* or HW does not change $t(op_i, r_k)$.

Given performance patterns, the next step is to find the performance pattern and partition ratio (α, β) for each operator op_i. All performance data are collected in an offline manner, and we believe is general enough to support new DL models since time-consuming operators are highly the same (e.g., *Conv2D* in CNNs, *Multiply* in RNNs)

Step 1: sampling a specified operator execution time on CPU or GPU. An operator op_i can receive different shapes of tensors (Table 2), and we sample the specified operator's execution time with each certain shape. For example, when *Conv2D* takes a 244×244 image with 3 input channels, we can sample its execution time by setting its *batchsize* from 1 to 1024. After that, we can get performance data of *Conv2D* with different tensors.

Step 2: selecting a performance pattern with least errors. We construct three categorized performance patterns as described above, and then use performance data (Step 1) to calculate estimation errors of three patterns for an operator. Here, the performance pattern with least error is the target we used for the operator.

Step 3: estimating an operator's performance when partitioning. For an operator op_i with a certain input tensor, we try any possible partition ratio $ratio = (\alpha, \beta)$, where (α, β) can vary from $(1, 0)$ to $(0, 1)$ with a certain span (0.01 in practice) on different partition dimension d, separately (Table 2). Then each sub-operator will get a tensor αd or βd, which its performance have been widely evaluated in the selected pattern. We can use that pattern to accurately estimate each sub-operator's performance on heterogeneous resource, and variables in Eqs. 1 to 3 can be calculated and they can be used for further optimization.

Algorithm 1 shows our performance pattern matching process. For an operator op_i with input tensors *input* on resource r_k, we first evaluate how well our three performance patterns match operator performance (Lines 1-5), then find

a partition dimension d that has least execution time (Lines 6-10).

Algorithm 1 Performance pattern matching.

```
1: function PatternMatch(op_i, inputs, R)
2:     Select partition dimensions {d} from inputs
3:     for d in {d} do
4:         Sample op_i execution time on d.
5:         Build three categorized patterns for op_i.
6:         For r_k ∈ R, get a matched pattern pt(op_i, d, r_k)
7:     end for
8:     Select a d, get pt(op_i, d, r_k) with least execution time
9:     Return pt(op_i, d, r_k)
10: end function
```

3.3 Minimizing Overall Execution Time

We now show how to calculate the optimal $ratio = (\alpha, \beta)$ for each operator op_i separately. we need to find (α, β) that makes:

$$Min\ \varepsilon,\ s.t.:$$
$$|t(op_i^{GPU}(\beta d), GPU) - t(op_i^{CPU}(\alpha d), CPU)| < \varepsilon \quad (5)$$
$$\varepsilon > 0$$

Here, a minimized ε means almost the same execution time of two sub-operators with the partition ratio (α, β) on dimension d. It implies that inputs have been well partitioned and heterogeneous resources can be efficiently used. The calculation process can be selected from performance data estimated from **Step3** (Section 3.2). However, separately finding the optimal partition ratios for all operators does not mean we can meet the goal in Eq. 4 due to the two reasons:

1. Some operators may run very fast, communication overhead introduced by partitioning these operators may slow down operator execution.
2. Separate partition optimization only gets the local optimal for one operator, it may not achieve global optimal results for all operators.

To efficiently partition operators, we use two mechanisms as follows:

Partitioning key operators to reduce communication overhead. As shown in Eq. 4, our goal is to minimize overall operator execution time. As shown in TABLE 2, we find some operators consume 95% of the DL's inference time after our offline performance evaluation, while they only take 5-35% of operator numbers in a DL model. In this paper, if one or some operators consume > 95% of DL inference time, we call it or them as **key operators**.

Collaboratively Tuning partition ratios to achieve global optimization. For each key operator op_i in G, finding partition ratios with nearly equal execution time on CPU and GPU does not mean we can get the most performance improvement. As the example shown in Fig. 5, G has two operators op_1 and op_2, op_1 and op_2 can be partitioned with equal execution time on CPU and GPU, respectively. However, if we collaboratively consider them both and partition less input tensors on GPU for op_1, the total performance can be further improved. With this tuning, it will cause a longer execution time for op_1, but op_2 can use more GPU with higher speedups.

Table 2. Partitioning key operators to reduce communication overhead.

DL kind	Operator name	Partition dimension	Value space
CNN,Yolo	Conv2D, Relu BatchNorm, Transpose	*batchsize*	(1,1024)
		HW	(32,32)-(299,299)
RNN	Sigmoid, Tanh, Add, Concat, Split	*batchsize*	(1,512)
		HW	(1,4)-(1,256)
	Multiple Dense	HW_1W_2	(1,1,1)-(4096,4096,4096)
Transformers	Transpose	HW	(1,1)-(4096,4096)
	Dense	HW_1W_2	(1,1,1)-(4096,4096,4096)

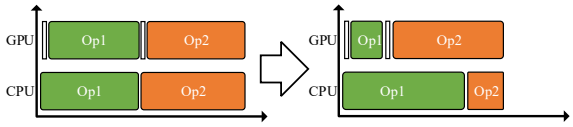


Figure 5. Tuning partition ratios to improve performance.

To tune partition ratios, we inject graph executor in TVM with a tuner, which can collaboratively change partition ratios of two adjacent operators. The tuner will try to adjust partition ratios of two operators to find if it can bring more performance improvement. When we partition the ith operator in G, $\sum_i t(p(op_i, d), R)$ of the first i operators can be calculated by the following three conditions:

C1) $i = 1 : T(p(op_i, d), R)$. In G, it means there is only one operator partitioned currently, and we do not consider collaboratively tuning.

C2) $i = 2 : TuneT(op_{i-1}, op_i, R)$. If collaboratively tuning partition ratios of op_{i-1} and op_i can reduce more execution time, we use this result as the current execution time in G. Otherwise we just partition op_{i-1} and op_i separately, and current execution time is $T(p(op_{i-1}, d), R) + T(p(op_i, d), R)$. Here, the key problem is how to get collaboratively partitioned execution time. We implement $TuneT(op_{i-1}, op_i, R)$ in Algorithm 2. Based on performance patterns, we can get

all operator performance with any partition ratios separately (Line 1-9), then we simultaneous tune one operator's α and the other one's β to see if it can reduce more execution time (Line 10-13).

C3) $i > 2$: *Recursively using C1 and C2*. For more operators, we just recursively use the first two conditions to find the least current execution time when partitioning last one (op_i) or two operators (op_i, op_{i-1}) with collaboratively tuning.

Algorithm 2 Collaboratively tuning partition ratios.

1: **function** TuneT(op_{i-1}, op_i, R)
2: $(\alpha_{i-1}, \beta_{i-1}) = (0, 1)$, $(\alpha_i, \beta_i) = (0, 1)$, $span = 0.01$
3: Get optimal partition dimensions d_{i-1}, d_i from *patternMatch*
4: **for** α_{i-1}, α_i in $[0, 1]$ **do**
5: $\beta_{i-1} = 1 - \alpha_{i-1}$, $\beta_i = 1 - \alpha_i$
6: Estimate op_{i-1}, op_i execution time on d with different partition ratios $(\alpha_{i-1}, \beta_{i-1})$, (α_i, β_i).
7: **end for**
8: Separately find optimal partition ratios $(\alpha_{i-1}, \beta_{i-1})$, (α_i, β_i)
9: $tuningTime = T(p(op_{i-1}, d), R) + T(p(op_i, d), R)$
10: Increase α_{i-1} and β_i, then update $tuningTime$ with least execution time.
11: Increase β_{i-1} and α_i, then update $tuningTime$ with least execution time.
12: Return $tuningTime$
13: **end function**

4 Implementation And Discussion

4.1 Implementation

Algorithm 3 provides the details of how EOP works. It can calculate all time variables of our problem formula in Eq. 1 and Eq. 3, then our goal in Eq. 4 can be resolved. EOP estimates each operator's execution time after partition (Line 1-5), and uses collaboratively tuning mechanisms to get global optimization (Line 6-9), a new computation graph can be exported and deployed on edge servers (Line 10-11).

EOP is implemented on TVM 0.8.0dev [6] (1800 LoC) with its code is written in Python 3.8.1, and is open-sourced in the community. By using TVM compatible interfaces of mainstream DL frameworks and NetworkX [9], EOP uses four main components to accurately estimate operator partition performance and reduce communication overhead in Fig. 6.

EOP can generate computation graphs from mainstream DL frameworks. The **Pattern Modeler** estimates key operator performance and analyzes patterns with minimal error. Based on these patterns, the **Performance Estimator** can accurately estimate the execution time of key operators. Then, the **Key Operator Partitioner** and **Partition Ratio Tuner** can be applied to these operators to maximize their

Algorithm 3 The algorithm of partitioning key operators and tuning partition ratios.

Input: A computation Graph $G = \{op_i\}$.
A set of resources $R = \{CPU, GPU\}$.
Output: A computation after operator partition: G'
Overall operator execution time of G': $AT(G', R)$
1: $G' = \{\}$, $MT(G', R) = 0$.
2: **for** op_i in G **do**
3: $t(op_i, r_k) \leftarrow PatternMatch(op_i, input, R)$.
4: $cm(op_i^{CPU}, op_i^{GPU}) \leftarrow Split$, $Device\_copy$ and $Concatenate$
5: $T(p(op_i, d), R) \leftarrow p(op_i, d), cm(op_i^{CPU}, op_i^{GPU})$
6: $T(p(op_i, d), R) \leftarrow tuneT(op_i, op_{i-1})$
7: Update $G' \leftarrow p(op_i, d)$
8: Update $AT(G', R) \leftarrow T(p(op_i, d), R)$
9: **end for**
10: For non-key operators op_j in G, update G' and $AT(G', R)$ according to op_j and $t(op_j, R)$, respectively.
11: Return G', $AT(G', R)$

performance. After that, the optimized computation graph for all key operators can be deployed and DL models can be run on heterogeneous resources.

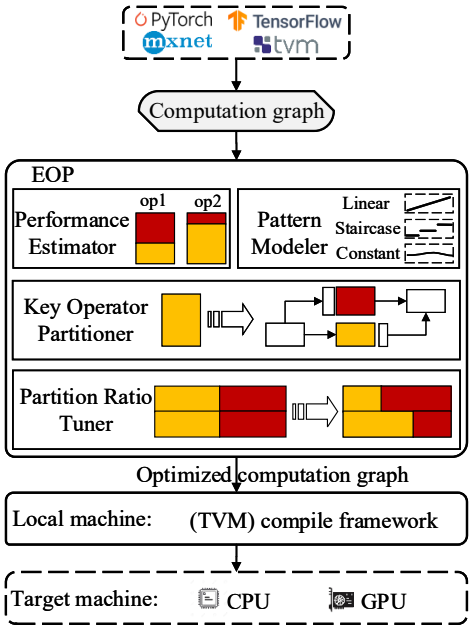


Figure 6. The EOP architecture.

4.2 Discussion

Stable inputs for DL inferences. A typical scenario emerging from real workloads is that a DL inference may run many times with only a few adjustments. So we can assume that the

input is stable and reuse the partitioned results to improve performance.

Coverage of three categorized patterns. Our three categorized patterns described in Section 3 can estimate all key operator performance in modern CNN, RNN and Transformers DL models. By using TVM compatible interfaces [6], we can also support new operators by more performance evaluation in an offline manner. Sample and pattern based performance estimation has proven to have a high accuracy [28, 31].

General partition dimensions. Partitioning *Conv2D*s in CNN can be done by many dimensions, such as input/output channels [14, 29, 31]. In this paper, we focus on three dimensions supported by majority of operators, and we think our approach can be easily extended to adjust more dimensions.

Fine-grained DL data parallelism. Partitioning operators on edge servers is indeed a kind of fine-grained DL data parallelism [27], and it can coexist with others like model parallelism [13, 16] and pipeline parallelism [18].

5 Evaluation

In this section, we evaluate the effectiveness of EOP and try to answer the following questions:

1. Can EOP estimate the performance of different operators accurately with different partition dimensions ($batchsize$, HW and HW_1W_2) in a heterogeneous environment?
2. Can EOP efficiently reduce the execution time for all operators of various DL models?
3. Can EOP still reduce the execution time when varying the settings of DL models?
4. How effective are the two mechanisms of EOP (partitioning key operator and tuning partition ratios)?

5.1 Experiment Setup

Environment. All operator performance data and EOP's efficiency have been evaluated on a typical edge server with one CPU (Intel(R) i7-7700 3.60GHz 4core, 32GB RAM) and one GPU (NVidia RTX 2060, 1920 cuda cores, 6GB RAM). CPU and GPU are connected by PCIe 3.0 ×16, and the operating system is Ubuntu 18.04.

Workloads. To evaluate the performance improvement of EOP, our experiments compare three kinds of DL models:

1) RNNs [1]: with a lot of open-source DL models implemented on TorchText [1], we select LSTM and GRU as RNN workloads.

2) Transformers [25]: since a lot of pre-trained DL models in Transformers are based on BERT, we use BERT as Transformers workload.

3) CNNs: To prove if EOP can get comparable performance improvement on traditional CNNs, we select Inception v3, MobileNet from TVM testing [3], and Yolo v3 from Yolo official [21].

Table 3. Parameters of different workloads: RNN, Transformers and CNN.

Workload kind	Parameters

RNN	LSTM: seq_len=1; input_size=80;
	num_layers=2; hidden_size=64
	num_dimensions=1;
	GRU: seq_len=1; input_size=80;
	num_dimensions=1; hidden_size=64
	num_layers=2;

Transformers	BERT: vocab_size=32000;
	hidden_size=768;
	num_hidden_layers=12;
	num_attention_heads=12;
	intermediate_size=3072

CNN	Inception v3: input_shape=(299,299) MoblieNet: input_shape=(224,224) Yolo v3: (c,h,w)=(3,640,640)

TABLE 3 shows the workload parameters used in our experiments. We choose $batchsize = 16$ as a common setting in DL inference. Such parameters are set by original implementations in TVM [6] and Pytorch [19]. We may change some parameters (e.g., input shape) in Section 5.4 to see if EOP can still get performance improvement. To keep the experiment data stable, we run each operator hundreds of times for each parameter, and record its average execution time by using TVM time evaluator [2]. Note that, the loading time of operators are not included when estimating operator performance. The communication overhead $cm(op_i^{CPU}, op_i^{GPU})$ are from the estimations of operators *Concatenate*, *Split* and *Device_copy*, and we use it to calculate $AT(G, R)$.

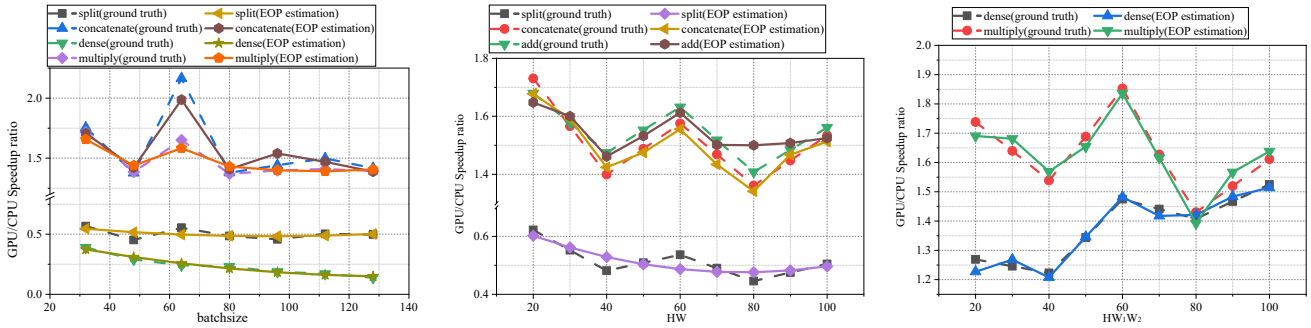
Existing approaches. We compare EOP with the following four approaches:

1. **CPU baseline**: it runs workloads only on CPUs without operator partitioning.
2. **GPU baseline**: it runs workloads only on GPUs without operator partitioning.
3. **Equal** [10, 27]: it implements equal partition ratios for all operators on heterogeneous resources.
4. **Fixed** [23, 29]: it implements fixed partition ratios for operators in a sub-graph.

Comparing with the above approaches, EOP uses three categorized patterns to estimate the operator execution time, and finds key operators with collaboratively tuning partition ratios.

Mechanisms in EOP. Two mechanisms (partitioning key operators and tuning partition ratios) are used in EOP to reduce execution time, and we use three conditions to evaluate them:

1. **Condition 1**: it finds key operators by balancing the execution time of sub-operators on CPU and GPU.



(a) Speedups with increased *batchsize* (b) Speedups with increased *HW* (c) Speedups with increased HW_1W_2

Figure 7. Operator speedups based on their execution time estimation.

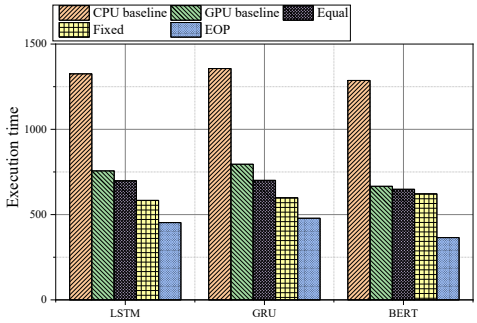


Figure 8. Overall execution time reduction (μs) on LSTM, GRU and BERT.

2. **Condition 2**: it tunes two adjacent operators without partitioning according to their GPU to CPU speedups.
3. **Condition 3**: it combines the above two mechanisms.

5.2 Comparison of Estimated Operator Execution Time for some key operators.

Metrics. We use **operator speedup of GPU to CPU** $sp(op_i) = t(op_i, CPU)/t(op_i, GPU)$ to evaluate whether EOP can accurately estimate operator execution time. When varying *batchsize*, *HW* and HW_1W_2, if speedup values from **EOP estimation** have almost the same as the **ground truth** speedup values, EOP can accurately estimate operator execution time.

Based on estimated operator execution time and categorized performance patterns, EOP can easily find key operators. Fig. 7 shows the key operator speedups when *batchsize*, *HW* and HW_1W_2 increase. Here, dotted lines denote the ground truth speedup values and solid lines denote our estimation results. A lot of key operators have dynamic speedups, which means the fixed speedup used by existing approach is inaccurate. Our observation as follows.

1) Only *split* has almost fixed speedups in Fig. 7(a) when increasing *batchsize*. Other operators like *dense* have large speedup variations.

2) When increasing HW_1W_2 and *batchsize* for *dense* in Fig. 7(a) and Fig. 7(c), *dense* has different speedups, which denotes that partitioning by different dimensions may get diverse performance improvement.

3) *add*, *tanh* and *multiply* in LSTM vary with different *HW*, which are caused by their different performance patterns on heterogeneous resources.

From the above observations, it is clear that EOP can estimate these speedup variations accurately with relatively small errors. Whereas other approaches can only support fixed speedups because they use linear patterns.

5.3 Comparison with Other Approaches

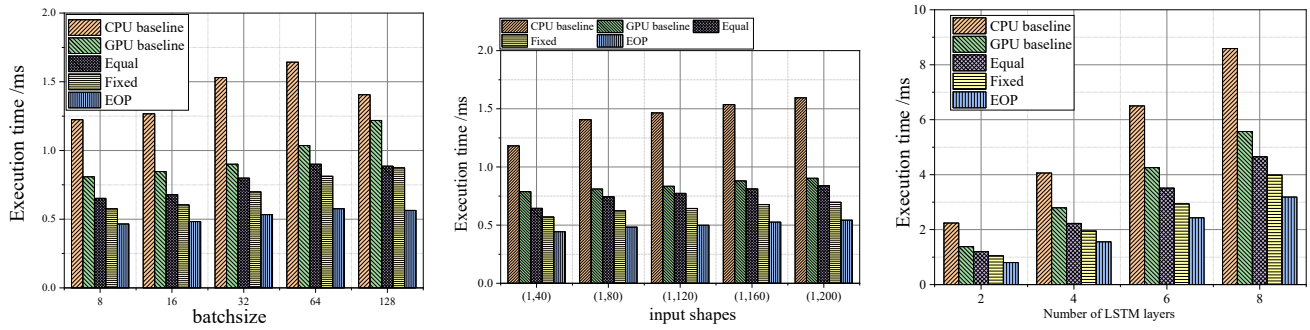
Metrics. We use **overall execution time** $AT(G, R)$ in Eq. 4 to evaluate the performance improvement. Here, a smaller value of $AT(G, R)$ indicates a better performance of DL inference.

Fig. 8 shows the results of overall execution time. Comparing with CPU and GPU baselines, EOP can improve performance of $1.67 - 3.53\times$ and $1.67 - 2.9\times$, respectively. This implies that exploiting the partition mechanism can bring large performance improvements, since key operators can take full advantage of heterogeneous resources. Comparing with existing approaches, EOP can also reduce $1.25 - 1.97\times$ overall execution time. This is because, in combination of various novel techniques, EOP can partition operators with minimized communication overhead.

Table 4. Comparable overall execution time reduction for Inception v3, MobileNet and Yolo v3.

DL model/ time(μs)	Equal	Fixed	EOP	Effect
Inception v3	4999.43	2539.66	1498.49	1.69-3.3
MobileNet	164.20	43.29	33.57	1.29-4.96
Yolo v3	827.66	234.02	221.51	1.06-3.74

In addition, TABLE 4 shows the performance improvement of EOP over the traditional *Conv2D* dominated DL models.



(a) Varying batchsize (b) Varying input shape (c) Varying the number of layers

Figure 9. The performance improvement of LSTM and GRU (both of them have almost the same input tensors and operators).

Table 5. Partition ratios and dimensions of key operators in RNNs (LSTM, GRU) and Transformers (BERT).

Workload	Operator	(α,β)	dimension
RNNs	split	(0.62,0.38)	HW
	concatenate	(0.23,0.77)	HW
	dense	(0.15,0.85)	batch size
	add	(0.90.0.10)	batch size
	sigmoid	(0.80,0.20)	HW
	tanh	(0.87,0.12)	HW
	multiply	(0.18,0.22)	HW_1W_2
	substract	(0.32,0.68)	batch size
Transformers	transpose	(0.70,0.30)	HW_1W_2
	dense	(0.28,0.72)	HW_1W_2

Since *Conv2D* consumes more than 99% of the inference time, EOP can obtain a 1.06× improvement for Yolo v3. For other CNN models with a smaller proportion of *Conv2D*, EOP can improve $AT(G,R)$ by $1.2-1.6\times$. Such results imply that EOP can also partition operators in CNNs with comparable performance to that of existing approaches.

TABLE 5 shows the partitioned key operators. For operators in the same subgraph, we find that operators *split*, *add*, *sigmoid*, *tanh* run faster on the CPU than on the GPU, so we need to partition more inputs on the CPU ($\alpha > \beta$). In contrast, operators *concatenate*, *dense*, *multiply* and *substract* have more partitions on the GPU ($\alpha < \beta$). All of these operators take more than 5% of the overall execution time. Note that the operator *dense* in Transformers and RNNs have different partitioning ratios and dimensions, which means that it may have dynamic speedups when the partition dimensions change. We believe that this feature has not been studied in existing approaches, and that this is one of the reasons for their low accuracy. In addition, the key operator accounts for only $8-25\%$ of the total operators, which shows that analyzing key operators can effectively reduce communication overhead.

5.4 Comparison with Diverse DL Models

Metrics. DL developers may change the settings of DL models frequently in different scenarios. These settings may affect the performance of EOP. In this case, the following different settings may change the number of inputs and operators in G, which we denote it as G'. If we can still get a minimized $AT(G',R)$ than existing approaches', EOP can work well on diverse DL models.

1. **Varying** *batchsize*: *batchsize* can be changed according to how heavy their workloads are.
2. **Varying input shapes**: Such shapes are different when DL models process images and sentences in inference scenarios.
3. **Varying the number of layers**: Layers can be also different with the requirements of DL inference accuracy. Note that CNNs and Transformers may have different kinds of layers.

Varying *batchsize* **in RNNs**. Fig. 9(a) shows the overall execution time for LSTM and GRU $AT(G',R)$ by varying the value of *batchsize* (8, 16, 32, 64 and 128). EOP improves performance by $1.5-2.9\times$ compared to CPU and GPU baselines. When increasing the *batchsize*, existing approaches only have a small performance improvement. The overall execution time of all approaches shows a linear growth pattern when $batchsize < 64$, while it decreases when $batchsize < 128$. This means that the execution time of RNNs may have a staircase pattern, which is different from the linearity in CNNs. Therefore, the assumption of a linear relationship between operator execution time and input is wrong for RNN models, and it hinders existing approaches [23, 27, 29] from further improving performance.

Varying input shape in RNNs. By setting HW from $(1,40)$ to $(1,200)$ in steps of 40, Fig. 9(b) shows how the change in HW affects performance. When the input shape becomes larger, the execution time of all approaches increase flat, and EOP maintains an improvement of 1.23×. CPU baselines grow faster than GPU's, which means that the speedup of LSTM is not fixed. When HW is between $(1,120)$ and

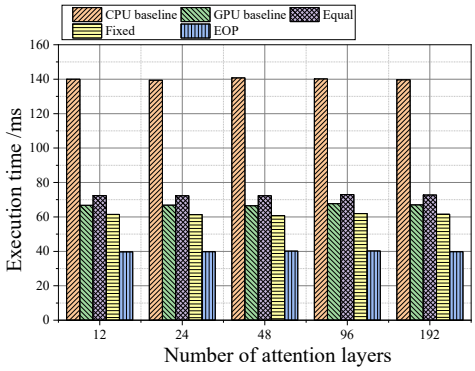


Figure 10. The performance improvement of BERT when varying the number of attention layers.

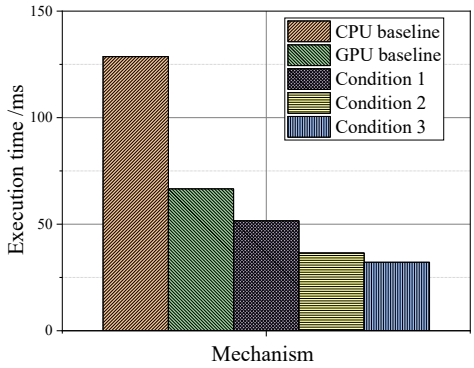


Figure 11. The performance improvement of BERT inference when using different mechanism in EOP.

$(1, 200)$, EOP reduces the execution time almost as much as LSTM, which means that EOP is more likely to maintain a steady improvement when the input data shape changes in a small range, even if its baseline increases.

Varying the number of layers in RNNs and Transformers. As shown in Fig. 9(c), when the number of layers is set to 2, 4, 6 and 8, the execution time of both LSTM and GRU linearly increases. We can see that EOP maintains a performance improvement of $1.1 - 1.25\times$. This is because the communication overhead consumes a small percentage of the overall execution time, and the key operators in the new layers are the same as in the old ones.

As shown in Fig. 10, we change the number of attention layers in BERT from 12 to 192, and the result shows that all execution time change by less than 1% and EOP maintains an improvement of 1.4×. Similar conclusions can also be drawn from changing hidden layers. Such results imply that the operators in the hidden and header layers are not key operators and changing these layers has little impact on inference performance. Moreover, if the DL developer does not change the structure of BERT, EOP can obtain stable and huge improvements without considering too many non-key operators.

5.5 Comparison with Different EOP Mechanisms

Metrics. We now compare **overall execution time reduction** $AT(G, R)$ in Eq. 4 by using different partition mechanisms (**Condition 1 to 3** in Section 5.1), respectively. We find which mechanisms can minimize $AT(G, R)$, and evaluate all mechanisms on BERT.

Fig. 11 shows the comparison results. We observe that any of the mechanisms can reduce execution time, and using them both (**Condition 3**) can obtain an improvement of 1.45×. The reason why only using one mechanism (either Condition 1 or 2) gets suboptimal performance can be explained in two ways.

1) The partitioning key operator mechanism (**Condition 1**) in EOP makes an operator use only the CPU or GPU, so the other resource is idle at the same time and operators cannot use the full computational power.

2) The tuning partition ratios mechanism (**Condition 2**) enables each operator to use heterogeneous resources at the same time, but the resources may be overused by non-key operators, which is not efficient.

6 Related Work

EOP improves DL inference performance on edge servers by: 1) accurately estimating operator execution on heterogeneous resources, and 2) efficiently partitioning key operators. Note that a lot of work are mainly on edge-cloud collaborations [11, 29, 30] and EOP is non-intrusion with them. As such, we discuss the related work from these two research aspects.

Operator estimation. Recent researches can be largely categorized into two kinds. The first deeply studies software and hardware issues that may affect operator execution time, and consider these issues to do estimation [7, 17]. While Talos [26] and DUET [32] mainly concentrate on hardware speedup issues. Most of them aim at the scenario with fixed operator inputs, and they cannot be directly used for operator partition since DL inputs may change.

The second collects a lot of performance data of DLs, and builds estimation models: nn-meter [31] uses random sampling to estimate a subgraph, Habitat [28] combines learning and heuristics based estimators to support heterogeneous resources. These data are always for subgraphs or *Conv2D*s, which are much different from operators' in various DL models.

To support various DL inferences, EOP studies three operator performance variation patterns to support operator estimation with inputs change. While it uses operator sequential and repeated execution to get stable performance data.

Operator partition. There have been subgraph-level and operator-level partition approaches when DL inference runs

on heterogeneous resources. Subgraph-level partition approaches [14, 23, 29] consider computation and communication both, and always concentrate on subgraphs in CNNs. They are not general enough to support operator partition in other DL models. Other approaches like Super [12], Eagle [16] and Optimus [5] supports subgraph-level without partition, and EOP can be integrated to these approaches and improve more performance.

Operator-level partition approaches can be implemented by fastT [27], Grnn [10] and μLayer [15]. They do a lot of optimizations like partition for operators running on homogeneous resources. CPU and GPU are considered separately or regarded same computation throughput. Our partitioned operators can use these optimizations (always implemented on TVM [6]) and support heterogeneous resources.

EOP is a new efficient partition approach, it selects a few key operators to partition. Then EOP provides a partition ratio tuning conditions to make more resources consumed by high speedup operators, which can further improve performance.

7 Conclusion

Existing operator partition approaches either only consider homogeneous resources or just focus on optimizations of traditional CNNs, which is inefficient. In addition, they can hardly get accurate operator execution time by linear performance patterns. Thus, we provide EOP, an efficient operator partition approach, which can accurately estimate operator execution with three performance variation patterns, and partition key operators to bring more performance improvement with reduced overhead. In future work, EOP will support more partition scenarios when using heterogeneous edge devices and edge-cloud communication.

Acknowledgments

This work was partially supported by National Key Research and Development Program of China (2018YFB1402803) and Provincial Key Research and Development Program of Shandong, China (2021CXGC010101).

References

[1] [n. d.]. Pytorch text. https://pytorch.org/text/stable/index.html.
[2] [n. d.]. TVM. https://github.com/apache/tvm.
[3] [n. d.]. Tvm relay testing. https://tvm.apache.org/docs/api/python/relay/testing.html.
[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *OSDI*. 265–283. https://doi.org/10.5555/3026877.3026899
[5] Xuyi Cai, Ying Wang, and Lei Zhang. 2021. Optimus: towards optimal layer-fusion on deep learning processors. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 67–79. https://doi.org/10.1145/3461648.3463848
[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *OSDI*. 578–594. https://doi.org/10.5555/3291168.3291211
[7] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati
[8] Peizhen Guo, Bo Hu, and Wenjun Hu. 2021. Mistify: Automating DNN Model Porting for On-Device Inference at the Edge. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 705–719. https://www.usenix.org/conference/nsdi21/presentation/guo
[9] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[10] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. 2019. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16. https://doi.org/10.1145/3302424.3303949
[11] Zhaowu Huang, Fang Dong, Dian Shen, Junxue Zhang, Huitian Wang, Guangxing Cai, and Qiang He. 2021. Enabling Low Latency Edge Intelligence based on Multi-exit DNNs in the Wild. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 729–739. https://doi.org/10.1109/ICDCS51616.2021.00075
[12] Arpan Jain, Tim Moon, Tom Benson, Hari Subramoni, Sam Adé Jacobs, Dhabaleswar K Panda, and Brian Van Essen. 2021. SUPER: SUb-Graph Parallelism for TransformERs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 629–638. https://doi.org/10.1109/IPDPS49936.2021.00071
[13] Beomyeol Jeon, Linda Cai, Pallavi Srivastava, Jintao Jiang, Xiaolan Ke, Yitao Meng, Cong Xie, and Indranil Gupta. 2020. Baechi: fast device placement of machine learning graphs. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 416–430. https://doi.org/10.1145/3419111.3421302
[14] Emre Kilcioglu, Hamed Mirghasemi, Ivan Stupia, and Luc Vandendorpe. 2021. An Energy-Efficient Fine-grained Deep Neural Network Partitioning Scheme for Wireless Collaborative Fog Computing. *IEEE Access* (2021). https://doi.org/10.1109/ACCESS.2021.3084689
[15] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *Eurosys*. 1–15. https://doi.org/10.1145/3302424.3303950
[16] Hao Lan, Li Chen, and Baochun Li. 2021. EAGLE: Expedited Device Placement with Automatic Grouping for Large Models. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 599–608. https://doi.org/10.1109/IPDPS49936.2021.00068
[17] Cheng Li, Abdul Dakkak, Jinjun Xiong, Wei Wei, Lingjie Xu, and Wen-mei Hwu. 2020. Xsp: Across-stack profiling and analysis of machine learning models on gpus. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 326–327. https://doi.org/10.1109/IPDPS47924.2020.00042
[18] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15. https://doi.org/10.1145/3341301.3359646
[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037. https://doi.org/10.5555/3454287.3455008

[20] Mirco Ravanelli, Titouan Parcollet, and Yoshua Bengio. 2019. The pytorch-kaldi speech recognition toolkit. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 6465–6469. https://doi.org/10.1109/ICASSP.2019.8683713

[21] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[22] Edgar Riba, Dmytro Mishkin, Daniel Ponsa, Ethan Rublee, and Gary Bradski. 2020. Kornia: an Open Source Differentiable Computer Vision Library for PyTorch. In *2020 IEEE Winter Conference on Applications of Computer Vision (WACV)*. 3663–3672. https://doi.org/10.1109/WACV45572.2020.9093363

[23] Linghao Song, Fan Chen, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2020. Accpar: Tensor partitioning for heterogeneous deep learning accelerators. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 342–355. https://doi.org/10.1109/HPCA47549.2020.00036

[24] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/nsdi22/presentation/weng

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

[26] Yuanjia Xu, Heng Wu, Wenbo Zhang, Chen Yang, Yuewen Wu, Heran Gao, and Tao Wang. 2021. Talos: A Weighted Speedup-Aware Device Placement of Deep Learning Models. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 101–108. https://doi.org/10.1109/ASAP52443.2021.00023

[27] Xiaodong Yi, Ziyue Luo, Chen Meng, Mengdi Wang, Guoping Long, Chuan Wu, Jun Yang, and Wei Lin. 2020. Fast Training of Deep Learning Models over Multiple GPUs. In *Proceedings of the 21st International Middleware Conference*. 105–118. https://doi.org/10.1145/3423211.3425675

[28] Geoffrey X. Yu, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 503–521. https://www.usenix.org/conference/atc21/presentation/yu

[29] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608. https://doi.org/10.1109/TNET.2020.3042320

[30] Qunsong Zeng, Yuqing Du, Kaibin Huang, and Kin K Leung. 2021. Energy-efficient resource management for federated edge learning with CPU-GPU heterogeneous computing. *IEEE Transactions on Wireless Communications* (2021). https://doi.org/10.1109/TWC.2021.3088910

[31] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93. https://doi.org/10.1145/3458864.3467882

[32] Minjia Zhang, Zehua Hu, and Mingqin Li. 2021. DUET: A Compiler-Runtime Subgraph Scheduling Approach for Tensor Programs on a Coupled CPU-GPU Architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 151–161. https://doi.org/10.1109/IPDPS49936.2021.00024