

MOIRAI: Towards Optimal Placement for Distributed Inference on Heterogeneous Devices

Beibei Zhang[†], Hongwei Zhu[†], Feng Gao[†], Zhihui Yang[†], Sean Xiaoyang Wang[‡]

[†]Zhejiang Lab, Hangzhou, China

[‡]Fudan University, Shanghai, China

Abstract—The escalating size of Deep Neural Networks (DNNs) has spurred a growing research interest in hosting and serving DNN models across multiple devices. A number of studies have been reported to partition a DNN model across devices, providing device placement solutions. The methods appeared in the literature, however, either suffer from poor placement performance due to the exponential search space or miss an optimal placement as a consequence of the reduced search space with limited heuristics. Moreover, these methods have ignored the runtime inter-operator optimization of a computation graph when coarsening the graph, which degrades the end-to-end inference performance. This paper presents MOIRAI that better exploits runtime inter-operator fusion in a model to render a coarsened computation graph, reducing the search space while maintaining the inter-operator optimization provided by inference backends. MOIRAI also generalizes the device placement algorithm from multiple perspectives by considering inference constraints and device heterogeneity. Extensive experimental evaluation with 11 large DNNs demonstrates that MOIRAI outperforms the state-of-the-art counterparts, i.e., Placeto, m-SCT, and GETF, up to $4.28\times$ in reduction of the end-to-end inference latency. MOIRAI code is anonymously released at <https://github.com/moirai-placement/moirai>.

Index Terms—Model parallelism, device placement, operator fusion, mixed integer linear programming.

I. INTRODUCTION

Recent years have witnessed the prevalence of Deep Neural Networks (DNNs) in diverse spectrum of scenarios ranging from object detection [1] to text generation [2]. In these applications, researchers increase the DNN model capacity, measured by the number of trainable parameters, to achieve better model inference accuracy and generalization performance. As an example, the state-of-the-art language model Megatron-Turing NLG with 530 billion parameters achieves an accuracy score of 87.15% in the task of LAMBADA next word prediction [3]. However, DNN inference expects considerable memory space to store the parameters and intermediate activations, which arouses natural concerns about exceeding the memory capacity of a computing device [4]. For instance, GPT-3 [2] with 175 billion parameters requires 350 GB of GPU memory. This is far beyond the memory sizes of any commercial off-the-shelf GPUs.

The growing size of DNNs necessitates the adoption of heterogeneous resource-constrained computing devices to host a large-scale DNN model. To accommodate this need, we split a DNN into several sub-models, each of which usually consists of continuous operators of a DNN, and distribute them across multiple devices. The devices execute the disjoint

parts of the DNN collectively, which is referred to as *model parallelism* [5]. Fundamentally, model parallelism seeks to map DNN operators to computing devices, commonly termed as *device placement* [6]. The objective of the device placement is to minimize the *makespan*, which identifies the time interval between a single input and its response. In this paper, we focus on providing a device placement solution for model parallelism over heterogeneous devices.

Device placement is challenging for two reasons. Firstly, allocating operators with precedence constraints on separate devices incurs communication overhead between the devices due to the *data flow* between devices. The overhead varies under distinct device placement schemes. Secondly, devices possess heterogeneous computing resources. Executing an operator on different devices results in different processing time. Owing to the discrepancies of model and device configurations, device placement poses a huge solution search space when the number of operators or devices increases. As an example, placing an Inception-v4 [7] with 490 operators on 2 devices has 2^{490} different possible placement solutions.

A direct and intuitive approach to tackle the device placement problem is to manually find the DNN partition plan by machine learning experts. Such an approach might not be favorable for yielding an optimal and scalable results. Consequently, solutions have been proposed to leverage learning-based methods. Particularly, recent studies exploit reinforcement learning techniques, which leverage the execution track of similar operators to learn where DNN operators should be placed in a computer cluster [8]–[10]. Unfortunately, learning-based approaches requires a training process that takes several hours or even several days to deliver a placement solution for a single DNN [9]. Additionally, learning-based methods are unable to generalize to a different collection of devices. The placement solution has to be searched all over again, if the DNN is deployed to a different cluster [11]. The high training overhead and the low generalizability severely hinder the widespread application of the learning-based methods. To avoid these shortcomings, another line of work resorts to algorithmic approaches [4], [11]–[13]. The state-of-the-art algorithmic methods establish cost models that reflect end-to-end inference latency and propose near-optimal placement solutions through combinatorial optimization. According to the combinatorial optimization techniques applied to solving the device placement problem, we further categorize the status-quo algorithmic approaches into heuristic-based solutions [11],

[13], [14] and exact algorithm-based methods [4], [12].

Upon analyzing and experimenting the released implementations of existing algorithmic approaches, we observed that the above methods inherently suffer from the following drawbacks: (1) **Solution optimality**: Given model and device configurations, heuristic-based algorithms quickly yield the placement plan. However, the placement result is sub-optimal, leaving ample room for reducing the makespan. (2) **Graph optimization**: Machine learning compilers, such as TensorFlow XLA [15], TVM [16], and PyTorch Glow [17], represent a DNN as a computation graph with backend optimization to rewrite the graph for reducing the inference makespan. Nevertheless, current algorithmic solutions fail to leverage runtime optimization of a computation graph when coarsening the graph to reduce the solution search space. (3) **Device heterogeneity & Constraints**: Studies revolving around exact algorithms either attempt to produce near-optimal device placement decisions on a small number of homogeneous devices [4], [12] or do not sufficiently consider the device computation and communication constraints.

To address the aforementioned limitations, we propose MOIRAI, an algorithmic solution that considers DNN graph optimization and caters an optimal solution over heterogeneous devices. We embrace two key design considerations to tackle the challenges. Technically, (1) We leverage a runtime graph optimization, *operator fusion*, to merge multiple operators into one fused operator to reduce the search space for device placement. (2) We formalize the problem as a **Mixed-Integer Linear Programming (MILP)** model. Unlike the aforesaid approaches, in which the device heterogeneity is ignored, we craft the MILP to outline computational differences, memory constraints, and communication capability of the devices. An optimal placement result will then be produced by using the optimization solver Gurobi [18].

We conduct MOIRAI in PyTorch and empirically evaluate its efficacy with a total of 11 large models over two-device settings. We serve the trained Swin-Transformer [19], GPT-3 [2], and AlphaFold2 [20] with the placement plan of MOIRAI to two cluster of devices, each of which consists of 4 GPUs. MOIRAI outperforms the heuristic-based solution up to $1.87\times$ in placement optimality in terms of the reduction of the makespan. Compared to the learning-based solution counterparts, MOIRAI reduces the solution generation time from hours to minutes while reducing the makespan up to $4.28\times$. Our method is applicable to a range of DNNs and may be extended to other situations.

The remainder of the paper is organized as follows. First, we introduce background knowledge and provide motivation in section II. Then, we describe the technical details of MOIRAI in section III. Extensive comparisons and evaluations between MOIRAI and its counterparts follow in section IV. Finally, we conclude in section V.

II. BACKGROUND & MOTIVATION

Before delving into the details of MOIRAI, we provide related works of device placement research. We first give a

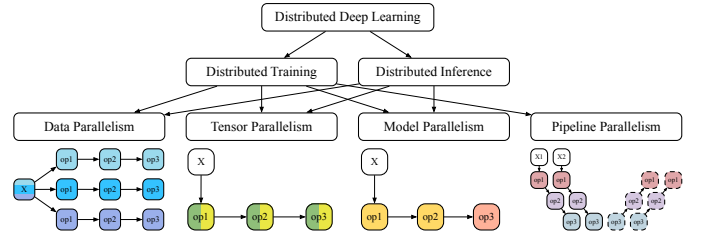


Fig. 1. We demonstrate the taxonomy of state-of-the-art distributed deep learning solutions. Under each method, we briefly visualize the layout of the solution silhouettes, where the computation graph includes three operators and each color indicates a distinct device.

brief overview of distributed deep learning. We then present hitherto device placement approaches, which are categorized into learning-based solutions and algorithmic methods.

A. Distributed Deep Learning

In this subsection, we briefly outline popular distributed deep learning solutions. A DNN life cycle consists of DNN training and DNN inference. In the training phase, we split the dataset into multiple mini-batches. A mini-batch passes through the operators of a DNN in a *forward propagation* phase. The output of the forward phase is compared against *ground truth* to generate a *loss*, which is leveraged to *update* the weights of operators in a *backward propagation* phase. This process is repeated several times to tune the weights until the desired result accuracy is achieved. DNN inference performs *forward calculations* using the trained weights. Distributed deep learning methods execute the above procedure cooperatively on multiple devices aiming to fit the large-scale model or obtain computation speedup.

Fig. 1 depicts the taxonomy of distributed deep learning concepts. According to the DNN life cycle, we categorize distributed deep learning methods into distributed training and distributed inference. We introduce four major types of distributed machine learning mechanisms, namely *data parallelism*, *tensor parallelism*, *model parallelism*, and *pipeline parallelism*. Data parallelism splits a mini-batch into multiple shards and executes multiple instances of the same model on decentralized devices with the shards [21], [22]. Tensor parallelism, also known as *intra-operator model parallelism*, partitions an operator along one of its dimension, enabling parallel processing for an operator [23], [24]. Model parallelism, commonly referred to as *inter-operator model parallelism*, divides a DNN into several consecutive sub-models that are placed and computed across multiple devices. *Pipeline parallelism* meliorates the model parallelism in the distributed training cycle, incorporating the forward phases and the backward stages of several mini-batches to better utilize the idled computation resources [25]–[27]. Pipeline parallelism can be deemed as a well-scheduled pipelined model parallelism, which can overlap the computation of different batches. A new trend focuses on developing a system that contemplates the mixture of the strategies to achieve the DNN training speedup [5], [28]. Under each distributed training method, a series of techniques, such as *gradient accumulation*, *checkpointing* [29], *Param-*

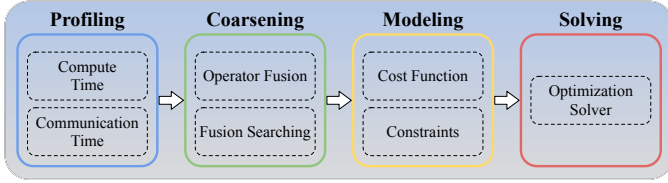


Fig. 2. We solve the device placement problem with four steps.

eter Server [30], *Ring-AllReduce* [31], are raised to fulfill or optimize its implementation. Unlikely, we concentrate on inter-operator model parallelism in the distributed inference, in pursuit of its optimality, hoping to inspire others. Other parallelism strategies and technics are orthogonal to our study.

B. Device Placement

Learning-based solutions. Earlier device placement works generally fall within a reinforcement learning paradigm. Mirhoseini et al. [6] propose to search the placement policy for a DNN with reinforcement learning method. Precisely, the method carries out a repeated series of Monte-Carlo trails with the feedback from real-world servers. It adjusts the scheduling strategy towards the objective of reducing the per-step training time of a DNN until convergence, which requires 17-27 hours of 80 to 160 4-GPU servers to produce the near-optimal placement. To expedite the exorbitant learning process, Post [32] represents device placement as a high-dimensional softmax probability distribution. Post leverages both proximal policy optimization and cross-entropy minimization to achieve a fast convergence. In the face of the cues that all previous methods can only generate a near-optimal placement for a DNN over a set of computing devices, Placeto [9] encodes computation graphs with graph embeddings and offers a policy network to disentangle the placement of a family of computation graphs with reinforcement learning.

Algorithmic methods. A variety of heuristics play an indispensable role for device placement problem. Inspired by the traditional parallel job scheduling, Baechi [11] utilizes three heuristics to portray the task precedence, communication overhead, and memory constraints of the placement policy, which generally requires less than 3 minutes to find a placement strategy. To better serve real-world scenarios dominated by heterogeneous devices, Hare [14] schedules computation graphs with a simple greedy heuristic that always provides higher GPU memory for the next task and keeps the latest completed task. Directly applying the heuristic methods to the device placement produces unsatisfactory end-to-end latency, primarily due to the failure to optimize over the practical computation and communication constraints. Accordingly, Pesto [12] establish an objective function that captures memory and non-overlapping constraints. However, Pesto merely considers device placement on two types of devices. The proposed method fail to generalize to multiple heterogeneous devices. GETF [33] represents the DNN as a DAG and extends the conventional Earliest Time First (ETF) algorithm to incorporate related machines. GETF establishes a Mixed Integer Linear Programming (MILP) model to address the

device placement problem. Nevertheless, the model neglects to integrate machine-dependent data flow communication time as a constraint in the MILP formulation.

Graph Coarsening. Given the substantial number of DNN operators, device placement algorithms encounter a considerable search space forfeiting its ability to generate placement solutions efficiently and effectively. Previous approaches have endeavored to reduce the search space by coarsening the computation graph, merging operators within the graph with tailored heuristics. A pervasive solution is to merge adjacent operators that, when combined, do not create cycles [11], [12]. Recent work considers communication-to-computing ratio of the computation graph as a metric to guide the grouping of operators [34]. However, existing approaches fail to leverage the runtime optimization of a computation graph.

III. PROPOSED METHOD

Broadly, we layout the overall procedures of MOIRAI in Fig. 2, namely, input profiling, graph coarsening, problem modeling, and problem solving. We would like to emphasize that our approach excels in handling heterogeneous devices while maintaining optimal inference latency. In what follows, we elaborate the comprehensive technical details of MOIRAI.

A. Assumptions & Settings

We enumerate a few practical settings and assumptions upon which our algorithm is designed. In contrast to the existing approaches, we attach substantial importance to the heterogeneity of devices, mainly in terms of computation, memory, and communication differences.

DNNs. The primitive computation unit in a DNN is a mathematical operator such as matrix multiplication (`matmul`) or convolution (`conv`). The data flow between operators establishes the dependency constraints among the operators. A group of operators with precedence constraints constitutes a computation graph that describes the DNN inference process. The topology of the computation graph is a Directed Acyclic Graph (DAG) where vertices represent operators and edges depict precedence constraints.

Devices. We focus on discussing the device placement problem on heterogeneous devices. Device heterogeneity is manifested in three folds. Firstly, the computation capability of a device is different resulting in different processing time of a DNN operator. Secondly, the memory capacity of each device is non-uniform, suggesting that the number of DNN operators that can be hosted by each device varies. Thirdly, there are differences in connectivity and bandwidth between devices due to their reliance on various network interfaces and protocols.

Communication. We view heterogeneous computers as a collection of connected devices with one or more addressable network attachments. Network attachments may reside at or above the data link layer and can have various types of interfaces, such as WiFi, Bluetooth, or even application defined interfaces. The communication channel between two devices may be provided by a point-to-point link, a shared broadcast

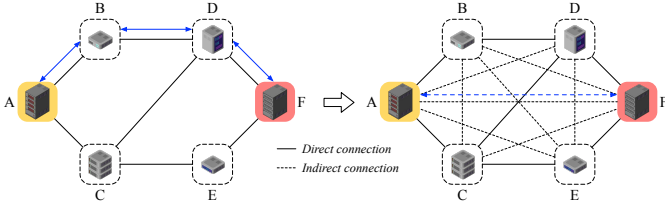


Fig. 3. Device A can communicate with device F via the channel $A \rightarrow B \rightarrow D \rightarrow F$, which can be viewed as indirect connection $A-F$. A device cluster (left) can be viewed as a full-mesh (right).

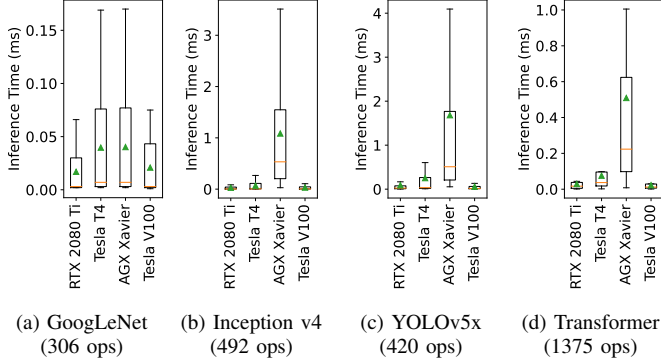


Fig. 4. Inference time distribution of operators in four models on four devices. ‘-’ indicates median and ‘ Δ ’ presents mean.

link, or a switched network. As illustrated in Fig. 3, if two devices in the connected cluster cannot directly communicate with each other, they may establish a multi-hop tunnel via internetworking protocols [35]. We intensify the connectivity in a connected device cluster with *direct* and *indirect* communication channels, enabling all-to-all communications. Therefore, we model the network topology of a connected heterogeneous device cluster as a full-mesh. We consider a bidirectional communication network where the availability and the bandwidth of the uplink and downlink are stable.

B. Graph Coarsening

We profile operator processing time of four widely employed DNNs on four devices and show their processing time distribution in Fig. 4. We note that modern DNNs typically consist of a number of operators with short computation time, increasing the difficulty to address the device placement problem. To reduce the solution search space, we coarsen the computation graph by grouping the closely coupled operators to promote the device placement algorithm.

Operator fusion. An inference backend, such as Eigen [36] and NNPack [37], combines operators that satisfy certain operator types and connections into a single fused operator, referred to as *operator fusion*. Operator fusion avoids storing intermediate results in memory, reducing frequent memory accesses. The memory access latency is often orders of magnitude greater than the computation time, and thus operator fusion extensively speeds up the DNN inference. We show an example in Fig. 5, where operator $op1$, $op2$, and $op3$ are fused into one operator to produce the final result res . Fusing $op1$, $op2$, and $op3$ avoids the necessity to store and access the temporary results of calculations against $op1$ and $op2$.

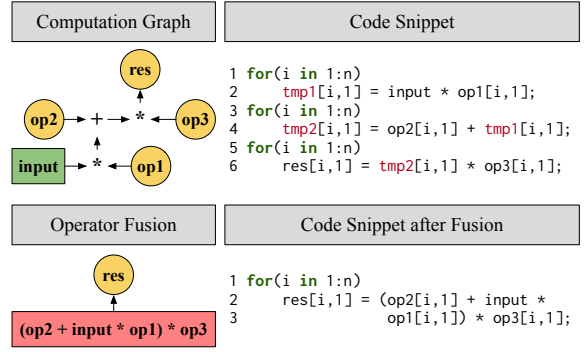


Fig. 5. Operator $op1$, $op2$, and $op3$ are fused by the backend compiler to avoid temporary results.

TABLE I
WE TAKE SEVERAL OPERATOR FUSION RULES PROVIDED BY EIGEN ON A GPU KERNEL AS EXAMPLES.

ID	Operators	Fused Operator
1	conv, bn	conv o bn
2	conv, bn, relu	conv o bn o relu
3	conv, bn, add, relu	conv o bn o add o relu

Upon the above observations, we intrinsically believe that the fused operators should be placed together when we effectuate graph coarsening.

An inference backend uses, for example, the *fusion rules* shown in TABLE I, to define which operators in a DNN should be fused [38]. A fusion rule contains a sequence of ordered operator types, each of which is a string. Fusion rules can be obtained from the design specifications of inference backends. Typically, the connections of DNN operators can be categorized into three types in a computation graph, which are illustrated in Fig. 6, namely *direct connection*, *multi-outputs*, and *multi-inputs*. As pointed out by [39], given a fusion rule, only fusing the operators with direct connection or multi-inputs connection can optimize the inference speed.

Fusion searching. Given a DNN, MOIRAI coarsens its computation graph by grouping operators based on fusion rules. We depict the operators as a set of vertices $\{v_1, v_2, \dots, v_n\}$ in a graph, where n is the number of operators in the DNN and v_i is the i -th DNN operator. For any two vertices v_i and v_j in the graph, we introduce a directed edge (i, j) in the graph if and only if there is a data transfer from operator i to j . With the above settings, we untangle the given DNN by the following DAG

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}), \quad (1)$$

where $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. We theoretically define fusion rules as a set $\mathcal{R} = \{r_1, r_2, \dots, r_m\}$, where m is the ID of a rule and the i -th rule $r_i \in \mathcal{R}$ is an ordered list with operator types as elements. The element order in r_i suggests the precedence constraints of the fused operators. Mathematically, MOIRAI engages in validating and grouping vertices in \mathcal{G} pursuant to a set of ordered list \mathcal{R} , which is described in Algorithm 1.

We elaborate the algorithm with an example in Fig. 7 where we follow the fusion rules provided by TABLE I. GCOF()

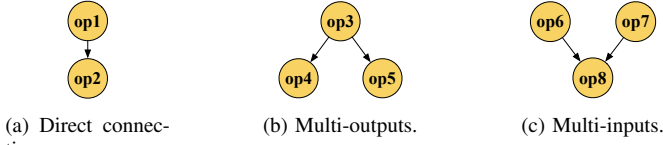


Fig. 6. Three types of operator connections.

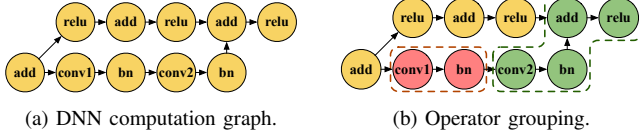


Fig. 7. GCDF() groups the DNN operators according to the TABLE I.

traverses the graph in depth-first order from the root vertex of operator type add. The function first navigates to the upper branch of the graph. Although the first add, relu vertex pair is certified to conforming partial r_3 rule via the function `is_sub_rule()`, the operator connection denoted by the edge between add and relu vertex is essentially part of the multi-output connection of the operator add, which is examined by function `is_valid_conn()`. Therefore, the first pair of add, relu should not be fused. The function then moves forward and binds the other two add, relu vertex pair on the upper branch, generating two new vertices with the bound tag and the $\text{add} \circ \text{relu}$ type. In the lower branch of the DNN, GCDF() fuses conv1 and bn, which comply with rule r_1 and the direct connection. Likewise, conv2 and bn are fused. Next, GCDF() merges the operator $\text{conv2} \circ \text{bn}$ and operator $\text{add} \circ \text{relu}$ at the end of the upper branch in accordance with rule r_3 and multi-inputs connection. Function `unbind()` releases the operators with the bound tag, which is the second add, relu pair on the upper branch in our case. The output of Algorithm 1 is a coarsened graph G of DAG topology. The time complexity of Algorithm 1 is $O(\mathcal{V} + \mathcal{E})$.

C. Input Profiling

Our method takes compute time of each operator, transmission time of data flow, precedence relation among operators, and configurations of devices as its inputs. The operator dependency is manifested by the computation graph and the configurations of devices can be acquired by querying operating system interfaces, whereas the operator processing and the data transmission time requires proper analysis.

Compute time. Scrutinizing the existing research on measuring DNN operator processing time, we notice that there are fundamentally three commonly employed methods to profile the operator processing time, that is *manual testing*, *operational intensity*, and *prediction model*. Though manual testing reveals the actual operator execution time, it is labor intensive to approach the data. Operational intensity [40] theoretically evaluates the task computation latency. However, several factors other than memory-bound and compute-bound affect the actual operator processing time. To balance the operator processing time accuracy and its availability, MOIRAI

Algorithm 1: Graph Coarsening with Operator Fusion: GCDF()

Data: DAG: $\mathcal{G} = (\mathcal{V}, \mathcal{L})$

Fusion rules: \mathcal{R} .

Result: Coarsened graph G by operator fusion.

```

1 function fuse( $v_{pred}, v_{succ}$ ):
2    $v_{new} \leftarrow$  initiate a new operator
3    $v_{new}.in \leftarrow v_{pred}.in \cup v_{succ}.in - v_{pred}$ 
4    $v_{new}.out \leftarrow v_{pred}.out \cup v_{succ}.out - v_{succ}$ 
5    $v_{new}.type \leftarrow v_{pred}.type \circ v_{succ}.type$ 
6    $v_{new}.tag \leftarrow$  fused
7   return  $v_{new}$ 

8 function bind( $v_{pred}, v_{succ}$ ):
9    $v_{new} \leftarrow$  fuse( $v_{pred}, v_{succ}$ )
10   $v_{new}.tag \leftarrow$  bound
11  return  $v_{new}$ 

12 function dfs( $v_{pred}$ ):
13   foreach  $v_{succ}$  in  $v_{pred}.out$  do
14     if is_rule( $v_{pred}, v_{succ}, \mathcal{R}$ ) and
15       is_valid_conn( $v_{pred}, v_{succ}$ ) then
16       |  $v_{next} \leftarrow$  fuse( $v_{pred}, v_{succ}$ )
17       | add  $v_{next}$  in  $\mathcal{G}$ 
18       | remove  $v_{pred}, v_{succ}$  in  $\mathcal{G}$ 
19     else if is_sub_rule( $v_{pred}, v_{succ}, \mathcal{R}$ ) and
20       is_valid_conn( $v_{pred}, v_{succ}$ ) then
21       |  $v_{next} \leftarrow$  bind( $v_{pred}, v_{succ}$ )
22     else
23       |  $v_{next} \leftarrow v_{succ}$ 
24     end
25   end
26   dfs( $v_{next}$ )

27 function unbind( $\mathcal{G}$ ):
28   release all the operators with the bound tag in  $\mathcal{G}$ 

29  $v_{pred} \leftarrow$  initiate traversal with the root vertex of  $\mathcal{G}$ 
30 dfs( $v_{pred}$ )
31  $G \leftarrow$  unbind( $\mathcal{G}$ )

```

chooses to estimate the compute time following the ideas in [41].

Communication time. Communication time between two devices amounts to the ratio of the data flow size and the communication bandwidth. In an indirect communication channel, the bandwidth of a multi-hop path depends on the minimum bandwidth on the path. Take the device cluster in Fig. 3 as an instance, suppose that the bandwidth of link $A - B$ and link $B - D$ is 10MB/s and 5MB/s, transmitting a 100MB data requires 20s.

D. Problem Modeling

Up to this point, we have presented how to obtain a coarsened graph to reduce the solution search space and prepared the necessary inputs for our algorithm. Next, we introduce

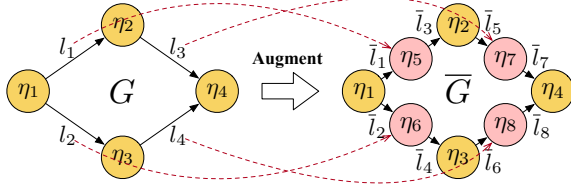


Fig. 8. We convert the links into new nodes that hold the same weight, resulting in a augmented DAG \bar{G} . Additional direct links without weights are inserted among the original nodes and the newly added nodes to maintain the node precedence and the graph topology.

TABLE II
NOMENCLATURE.

Notations	Descriptions
<i>Parameters</i>	
$G = (N, L)$	The DAG of the coarsened computation graph.
$\bar{G} = (\bar{N}, \bar{L})$	The augmented DAG of G , where links are converted to nodes.
$\text{Succ}(i)$	A set of direct and indirect successors of node η_i in G , where $\eta_i \in N$.
$\bar{\text{Succ}}(i)$	A set of direct and indirect successors of node η_i in \bar{G} , where $\eta_i \in \bar{N}$.
M^s, M^l, M^r	Three large numbers where $M^s \gg 0$, $M^l \gg 0$, $M^r \gg 0$.
<i>Variables</i>	
k	$k \in K$ is the index of a device, which specifies the device.
p_{ik}	The processing time of i -th operator on the k -th device, where $\eta_i \in N$.
$p_{qk'k''}^{\text{comm}}$	The transmission time of data flow q from device k' to device k'' , where $\eta_q \in \bar{N} - N$ and $k', k'' \in K$.
m_i	The memory footprint of i -th operator.
Mem_k	The memory size of k -th device.
S_i	$S_i \in \mathbb{R}^+$, expresses the start time of task i where $\eta_i \in \bar{N}$.
C_i	$C_i \in \mathbb{R}^+$, expresses the complete time of task i where $\eta_i \in \bar{N}$.
<i>Indicators</i>	
x_{ik}	$x_{ik} \in \{0, 1\}$, $x_{ik} = 1$ indicates placing the task denoted by $\eta_i \in N$ to device k , otherwise $x_{ik} = 0$.
z_q	$z_q \in \{0, 1\}$, $z_q = 1$ indicates a non-zero transmission time of the data flow q , where $\eta_q \in \bar{N} - N$. Otherwise $z_q = 0$.
$u_{qk'k''}$	$u_{qk'k''} \in \{0, 1\}$, $u_{qk'k''} = 1$ indicates the selection of communication channel from device k' to k'' to transmit the data flow q , where $\eta_q \in \bar{N} - N$, $k', k'' \in K$ and $k' \neq k''$.

our MILP model seeking to portray the inter-operator model parallelism.

DAG representation. Given a coarsened computation graph, we refer to the set of α operators in the coarsened graph as a set of nodes $N = \{\eta_i\}_{i=1}^{\alpha}$. Further, we depict the data flow among the operators as a set of links $L = \{l_i\}_{i=1}^{\beta}$. Thus, we model the coarsened computation graph as a DAG

$$G = (N, L), \quad (2)$$

where $|N| = \alpha$, $|L| = \beta$, and $L \subseteq N \times N$. The node weight represents operator processing time on devices and link weight exhibits data transmission time over a communication network. To facilitate our design, we augment the DAG G by altering

the links into a set of new nodes. Subsequently, the weight of the link is directly applied to the new node corresponding to it. We denote the augmented DAG of G by

$$\bar{G} = (\bar{N}, \bar{L}), \quad (3)$$

where $\bar{N} = \{\eta_i\}_{i=1}^{\alpha+\beta}$ and $\bar{L} = \{\bar{l}_i\}_{i=1}^{2\beta}$. $\bar{l}_i \in \bar{L}$ can be indicated by the index of its two end points (e.g., $\bar{l}_1 = (1, 5)$ in Fig. 8). Given the DAG G of a DNN, we conveniently leverage $\eta_i \in \bar{N}$ to outline both the data flow and the operator. To identify data flow and operators respectively, we refer to a data flow as $\eta_i \in \bar{N} - N$ and an operator as $\eta_i \in N$. An example of the relationship between G and \bar{G} is shown in Fig. 8.

MILP model. We summarize key notations of the model in TABLE II. Presented with the DNN computation graph termed as $G = (N, L)$, the device placement for operators in the DNN is achieved by solving the following MILP.

$$\text{minimize} \quad \max_{i \in N} C_i, \quad (4)$$

$$\text{subject to} \quad C_i \leq S_j, \quad \forall \eta_i \in \bar{N}, \forall \eta_j \in \bar{\text{Succ}}(i), \quad (4a)$$

$$C_i = S_i + \sum_{k \in K} p_{ik} x_{ik}, \quad \forall \eta_i \in N, \quad (4b)$$

$$\sum_{k \in K} x_{ik} = 1, \quad \forall \eta_i \in N, \quad (4c)$$

$$\text{Memory constraints}, \quad (4d)$$

$$\text{Non-overlapping constraints}, \quad (4e)$$

$$\text{Communication constraints}, \quad (4f)$$

$$\text{Congestion control}. \quad (4g)$$

Equation (4) represents the completion time of the last operator that ends the computation, which amounts to the end-to-end inference latency of the entire DNN when the inference starts at time 0. The objective function (4) is optimized subject to the constraints from equation (4a) to (4g). The data flow of a DNN, defined by the input and output of its operators, naturally establishes the execution precedence relationships of the operators. That is, the successor of an operator can only be processed after the completion of the current operator. Moreover, the output of the operator can only be transmitted after it is produced. We cast such operator processing and data transmission dependencies with equation (4a). Equation (4b) bridges the relation between the start time and end time of processing an operator. We ensure that each operator is assigned to only one device by equation (4c).

Constraints (4d) to (4g) account for the heterogeneity of devices.

(1) *Memory constraints.* Conventionally, the cumulative memory footprint of the operators allocated to a device should not surpasses the memory size of the device, known as memory constraints. We describe the memory constraints (4d) with

$$\underbrace{\sum_{\eta_i \in N} m_i x_{ik}}_{\text{Total memory of operators on device } k} \leq Mem_k, \forall k \in K, \quad (5)$$

to avoid the out of memory (OOM) error. For each device, we obtain the memory footprint of each operator through the APIs (e.g., `torch.profiler`) and constrain the total memory of the operators placed on a device not to exceed the memory capacity of the device.

(2) *Non-overlapping constraints.* By default, inference frameworks, such as PyTorch and TensorFlow, execute operators placed on the same device sequentially. Therefore, for any two operators assigned to the same device, we ensure that their processing time does not overlap. Equation (4a) maintains the order of execution for the two operators with precedence relationship. Given the two operators i and j without precedence constraints, we mathematically express the non-overlapping condition with

$$\begin{cases} S_i \geq C_j - M^s \delta_{ij} - M^l(2 - x_{ik} - x_{jk}), \\ S_j \geq C_i - M^s(1 - \delta_{ij}) - M^l(2 - x_{ik} - x_{jk}), \\ i \neq j, \\ \forall \eta_i, \eta_j \in N, \\ \eta_i \notin \text{Succ}(j) \text{ and } \eta_j \notin \text{Succ}(i), \\ \forall k \in K, \end{cases} \quad (6)$$

where $\delta_{ij} \in \{0, 1\}$ is a 0-1 indicator variable. If two operators i and j are placed on the device k , which is termed as $x_{ik} = x_{jk} = 1$, the inequalities in constraints (6) are

$$\begin{cases} S_i \geq C_j - M^s \delta_{ij}, \\ S_j \geq C_i - M^s(1 - \delta_{ij}), \end{cases}$$

in which both inequalities hold when δ_{ij} takes different values. For every two operators of a DNN, we apply the constraints in (6).

(3) *Communication constraints.* Naturally, when two adjacent operators are placed on two distinct devices, a communication overhead is incurred by the data flow between the operators. We formalize the communication overhead with the indicator z_q . Moreover, our model depicts bandwidth difference of the uplink bandwidth and downlink bandwidth between two devices. We capture the selection of channels between two devices with the indicator $u_{qk'k''}$.

$$\begin{cases} \left. \begin{aligned} z_q &\leq 2 - x_{ik} - x_{jk} \\ z_q &\geq x_{ik} - x_{jk} \\ z_q &\geq x_{jk} - x_{ik} \end{aligned} \right\}, \quad \forall q \in \overline{N} - N, \\ \left. \begin{aligned} \sum_{k' \in K} \sum_{k'' \in K} u_{qk'k''} &= z_q \\ u_{qk'k''} &\geq x_{ik'} + x_{jk''} - 1 \\ C_q &= S_q + \underbrace{\sum_{k' \in K} \sum_{k'' \in K} u_{qk'k''} \cdot p_{qk'k''}^{\text{comm}}}_{\text{Transmission time of data flow } q \text{ over the channel } k' \rightarrow k''} \end{aligned} \right\}, \quad \begin{aligned} &\forall q \in \overline{N} - N, \\ &(i, q), (q, j) \in \overline{L}, \\ &\forall k', k'' \in K, \\ &k' \neq k''. \end{aligned} \end{cases} \quad (7)$$

Given two contiguous operators i and j , if only one of them is placed on device k , which implies that there is a communication overhead for data flow q , $z_q = 1$ is enforced by the first three inequalities in constraints (7). Otherwise, $z_q = 0$. The

channel where data flow q is transmitted is indicated by $u_{qk'k''}$. If the transfer of data flow q exists, which implies $z_q = 1$, the fourth equation ensures that the communication task q selects at most one communication channel $k' \rightarrow k''$ for transmission. The fifth constraint indicates that the communication task q selects the channel $k' \rightarrow k''$ for transmission only when task i , task j are deployed on device k' and device k'' respectively (i.e. $x_{ik'} = x_{jk''} = 1$). The last equation in (7) bridges the start and end time of transmitting the data flow q .

(4) *Congestion control.* Lastly, we address the contention of data transmission, when there are multiple outputs waiting to be transferred on the same communication channel. Given a device k and two pairs of adjacent operators a, b and c, d where $(a, q), (q, b), (c, r), (r, d) \in \overline{L}$, the congestion happens when $x_{ak} = 1, x_{bk} = 0$ and $x_{ck} = 1, x_{dk} = 0$. In other words, two communication operations should not be processing simultaneously on the same channel. In other words, either $S_q \geq C_r$ or $S_r \geq C_q$ holds. Formally, the congestion control can be casted as

$$\begin{cases} S_q \geq C_r - M^s \delta_{qr} - M^l(2 - z_q - z_r) \\ \quad + M^r(x_{ak} + x_{ck} - x_{bk} - x_{dk} - 2), \\ S_r \geq C_q - M^s(1 - \delta_{qr}) - M^l(2 - z_q - z_r) \\ \quad + M^r(x_{ak} + x_{ck} - x_{bk} - x_{dk} - 2), \\ S_q \geq C_r - M^s \delta_{qr} - M^l(2 - z_q - z_r) \\ \quad + M^r(x_{bk} + x_{dk} - x_{ak} - x_{ck} - 2), \\ S_r \geq C_q - M^s(1 - \delta_{qr}) - M^l(2 - z_q - z_r) \\ \quad + M^r(x_{bk} + x_{dk} - x_{ak} - x_{ck} - 2), \\ q \neq r, \\ \forall \eta_q, \eta_r \in \overline{N} - N, \\ \eta_q \notin \overline{\text{Succ}}(r) \text{ and } \eta_r \notin \overline{\text{Succ}}(q), \\ (a, q), (q, b), (c, r), (r, d) \in \overline{L}, \\ \forall k \in K, \end{cases} \quad (8)$$

where $\delta_{qr} \in \{0, 1\}$ is a 0-1 indicator variable. For multiple transmission tasks on the same communication channel, we bound every two communication tasks by the constraints (8).

We solve the MILP model described in (4) using the optimization solver Gurobi [18]. Indicator variable x_{ik} implies the placement decision of each operator. After obtaining the placement decision of operators, we employ PyTorch to implement the inter-operator model parallel inference.

IV. EXPERIMENTS

In this section, we conduct extensive experiments to empirically evaluate MOIRAI. Broadly, we intend to answer the following research questions:

- **RQ1:** How does MOIRAI compare against the state-of-the-art approaches?
- **RQ2:** How much does our graph coarsening method contribute to the performance of MOIRAI?
- **RQ3:** What interesting insights and findings can we obtain from the empirical results?

Next, we present our experiment settings, followed by answering the above research questions one by one.

TABLE III
TESTBED CONFIGURATIONS OF TWO EXPERIMENT SCENARIOS.

Scenario	Device	Memory (GB)	Network Interface	Average Network Bandwidth (Gbps)			
				Device A	Device B	Device C	Device D
Inter-server	A: NVIDIA GeForce RTX 2080 Ti	11	InfiniBand	N.A.	44.26	32.92	44.28
	B: NVIDIA Tesla T4	16		42.39	N.A.	35.32	44.51
	C: NVIDIA Tesla P4	8		33.2	35.31	N.A.	32.95
	D: NVIDIA RTX 3060 Ti	8		42.08	43.22	33.28	N.A.
Intra-server	A: NVIDIA Tesla V100	32	NVLink + NVSwitch	N.A.	1170.04	626.10	610.56
	B: NVIDIA Tesla V100	32		1148.16	N.A.	618.98	581.09
	C: NVIDIA Tesla P100	16		630.43	609.82	N.A.	571.96
	D: NVIDIA Tesla P100	16		622.67	575.08	581.35	N.A.

TABLE IV
MODEL ARCHITECTURE WITH INCREASING NUMBER OF PARAMETERS. M: MILLION, B: BILLION.

Model	Parameters	Layer Number	Hidden Size	Head Number	N.O. Operators in Original Graph	N.O. Operators in Coarsened Graph
Swin-Transformer [19]	{1.8B, 6.6B, 13B}	{32, 48, 56}	{512, 768, 1024}	{16, 24, 32}	{6496, 14352, 22120}	{5204, 11512, 17947}
GPT-3 [2]	{330M, 1.3B, 2.7B, 13B}	{24, 32, 32, 40}	{1024, 2048, 2560, 5120}	{16, 32, 32, 40}	{4872, 9480, 12640, 19640}	{3682, 7308, 9825, 15283}
AlphaFold2 [?]	{87M, 930M, 2.4B, 3.2B}	{48, 64, 96, 128}	{256, 512, 1024, 1024}	{8, 16, 32, 32}	{5136, 12992, 37920, 50560}	{3618, 9252, 26824, 35096}

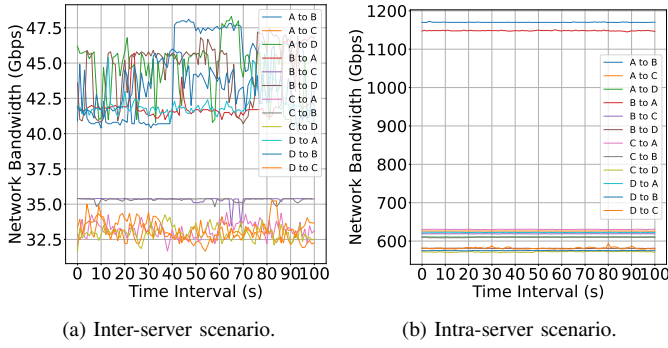


Fig. 9. Network bandwidth between two devices over 100s.

A. Experiment Setup

Testbed configurations. We demonstrate the advancement of MOIRAI through two scenarios. (1) **Inter-server inference:** We investigate an inter-server inter-operator model parallel inference setting, where multiple GPU servers are interconnected with a 100Gbps InfiniBand network. (2) **Intra-server inference:** We scrutinize an intra-server inter-operator model parallel inference setting, where within each server rack, GPUs are connected via NVLink and expanded through NVSwitch to enable all-to-all communication among the GPUs. We measured the bandwidth between every device during a 100-second period and performed calculations using the average bandwidth over this duration. The network bandwidth over the 100 seconds is presented in Fig. 9. We list the machine configurations and the network conditions of the two scenarios in TABLE III. We install NCCL 2.16 and PyTorch v1.12 on all devices.

Models. We empirically evaluate MOIRAI with with three emerging models from diverse domains, including computer vision, natural language processing, and biology analysis. For each model, we apply MOIRAI to its variants in different model size that is shown in TABLE IV. (1) **Swin-Transformer** [19] is a highly accurate vision model designed for image recognition. We set the resolution of input images

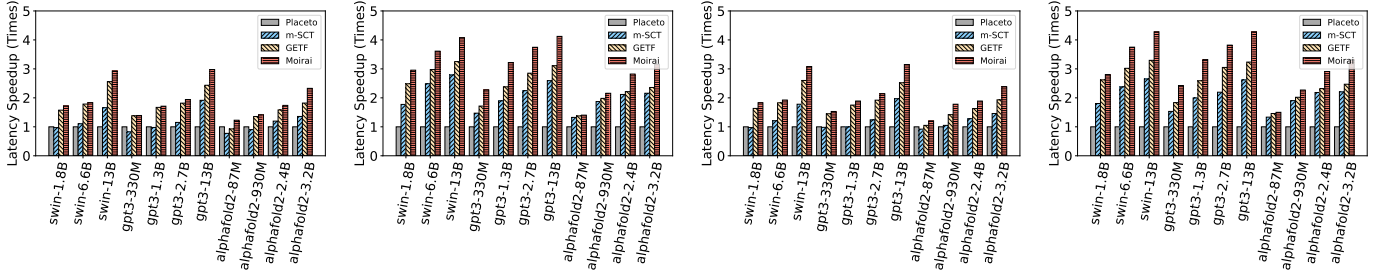
to 1100×1100 . (2) **GPT-3** [2] is a cutting-edge language model based on Transformer. The input of GPT-3 is word tokens. We employ a language sequence of 2048 tokens as its input. (3) **AlphaFold2** [20] is a biological model lies in its ability to accurately predict protein structures. Following the experiment setting in [20], we choose the input sequence batch size of 128.

Methods. We compare MOIRAI with both learning-based and algorithmic methods. (1) **Placeto** [9] is a reinforcement learning approach. We revise its reward function to accommodate only the forward calculations of an input. Placeto serves as the baseline in our experiment. (2) **m-SCT**, which is raised in Baechi [11], is a heuristic-based solution. m-SCT places operators on a device with the earliest start time and leverages an ILP model to find child operators. (3) **GETF** [33] is an exact algorithm-based method. We implement GETF following the guidelines outlined in [33]. We solve the GETF MILP with Gurobi.

Metrics. We employ two performance metrics to evaluate the inter-operator model parallel inference performance of MOIRAI. (1) **End-to-end inference latency** refers to the time required for a DNN to process an input and generate an output. We deploy DNN models based on the placement generated by each algorithm and measure its end-to-end inference latency. To mitigate variances caused by warm-up, we exclude the running time of the first five batches from the measurement, as specified in [8]. (2) **Placement generation time** denotes the duration taken by a device placement algorithm to generate a placement solution. We implement MOIRAI and its counterparts on devices equipped with Intel Core i7-8700 CPU and NVIDIA RTX 2080 Ti GPU, measuring the placement generation time.

B. Comparison with Existing Methods (RQ1)

We compare the end-to-end inference speedup of MOIRAI with three counterparts under two scenarios. To verify the impact of the proposed graph coarsening method, we try ap-



(a) Inter-server scenario with original computation graphs. (b) Intra-server scenario with original computation graphs. (c) Inter-server scenario with coarsened computation graphs. (d) Intra-server scenario with coarsened computation graphs.

Fig. 10. Inference latency speedup comparison among four algorithms.

TABLE V
PLACEMENT GENERATION TIME.

Model	Type	Original Computation Graph				Coarsened Computation Graph			
		HRL	m-SCT	GETF	MOIRAI	HRL	m-SCT	GETF	MOIRAI
Swin-Transformer	1.8B	4hrs	17.63s	2min	2.15min	4hrs	15.24s	1.58min	1.64min
	6.6B	5hrs	64.32s	12.4min	13.28min	4.8hrs	58.57s	9.72min	10.51min
	13B	5hrs	148.25s	19.75min	22.54min	5hrs	142.9s	14.95min	15.46min
GPT-3	330M	3.5hrs	15.92s	1.25min	1.38min	3hrs	14.4s	58.04s	1.04min
	1.3B	4hrs	39.82s	8.65min	9.02min	3.85hrs	37.39s	5.29min	6.88min
	2.7B	5hrs	54.17s	10.82min	11.48min	4.75hrs	51.52s	7.84min	8.51min
	13B	5.5hrs	125.53s	15.7min	16.5min	5hrs	117.48s	10.92min	11.07min
	87M	3.5hrs	20.18s	1.78min	1.9min	3.3hrs	18.62s	1.24min	1.38min
AlphaFold2	930M	5hrs	56.62s	11.05min	12.87min	4.82hrs	53.71s	7.95min	8.75min
	2.4B	5.75hrs	226.34s	22.5min	23.91min	5hrs	215.48s	17.92min	18.02min
	3.2B	7hrs	507.65s	35.75min	38.42min	6.5hrs	485.35s	21.5min	22.18min

plying the MILP model of MOIRAI on both the original DNN computation graph and the coarsened computation graph.

End-to-end inference latency. Fig. 10(a) demonstrates the end-to-end inference latency speedup of MOIRAI under inter-server scenario on original computation graphs. The result shows that MOIRAI reduces the end-to-end inference latency up to $2.98\times$, $1.77\times$, $1.33\times$ compared to Placeto, m-SCT, and GETF respectively. Fig. 10(b) provides inference acceleration details of MOIRAI under intra-server scenario on original computation graphs. We observe that MOIRAI provides the latency speedup up to $4.12\times$, $1.7\times$, $1.35\times$ compared to Placeto, m-SCT, and GETF respectively.

Next, we are interested in evaluating the impact of the graph coarsening method of MOIRAI on reducing the end-to-end inference latency. Fig. 10(c) and 10(d) exhibits the inference latency speedup of MOIRAI after coarsening the DNN computation graphs with Algorithm 1. In the inter-server setting, MOIRAI outperforms Placeto, m-SCT, and GETF up to $3.15\times$, $1.9\times$, and $1.25\times$ respectively in reduction of the end-to-end latency. In the intra-server setting, MOIRAI surpasses Placeto, m-SCT, and GETF up to $4.28\times$, $1.74\times$, $1.34\times$ respectively in reduction of the end-to-end latency.

Placement generation time. TABLE V presents the placement generation time of all the approaches. It is observed that the HRL algorithm requires several hours for training and generating the final results. m-SCT, due to its small algorithm search space, requires the least amount of time. Considering the vast search space and limitations of the Gurobi optimizer, both GETF and our algorithm need minutes to generate placement. However, since this process is offline, and the optimal placement solutions of MOIRAI are superior to those of the m-SCT, we believe that our approach still holds an

advantage in the placement generation. Moreover, by further relaxing MOIRAI MILP model, we can significantly reduce the placement generation time.

C. Contributions of Graph Coarsening (RQ2)

The last two columns of TABLE IV show the number of operators in the original computation graphs and the number of operators in the coarsened computation graphs. According to the results shown in Fig. 10, compared to using the original computation graph to generate placement results, the graph coarsening method has reduced the end-to-end inference latency by up to 5.7% in the inter-server setting and up to 3.8% in the intra-server setting. From TABLE V, we observe that the graph coarsening method has a significant effect on reducing the placement generation time, with an average time reduction to 71.87% of the placement generation time with the original computation graphs.

D. Discussion (RQ3)

The experiments on three types of DNNs suggest that incorporating higher communication bandwidth between devices achieves better inference speedup. Interestingly, a point that attracts our attention is: we observe that as DNN models become larger, although the computation resource demand rises, large models provide greater search space, which increases the parallelism of the model, making full use of computation resources, and achieving a better inference acceleration with MOIRAI.

V. CONCLUSION

In this paper, we proposed an algorithmic solution named MOIRAI for the device placement problem. MOIRAI incorporates graph optimization, device heterogeneity, and inter-operator model parallel inference constraints. We have used a graph coarsening method that considers runtime operator fusion to shrink the solution search space while maintaining optimality. The use of MILP in MOIRAI accommodates various constraints and can be extended to multiple heterogeneous devices. Extensive experiments demonstrate that MOIRAI consistently outperforms the state-of-the-art methods in reducing the end-to-end inference latency while ensuring a reasonable placement generation time. Future work involves proposing a meta-heuristic algorithm to expedite the placement generation

process and providing proofs concerning the approximation ratio of the meta-heuristic algorithm.

REFERENCES

- [1] Z.-Q. Zhao, P. Zheng, S.-t. Xu, and X. Wu, "Object detection with deep learning: A review," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 11, pp. 3212–3232, 2019.
- [2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [3] J. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhmoey, G. Zerveas, V. Korthikanti *et al.*, "Using deepspeed and megatron to train megatron-turing nlG 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.
- [4] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15451–15463, 2020.
- [5] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Jul. 2022, pp. 559–578.
- [6] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 2430–2439.
- [7] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017.
- [8] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean, "A hierarchical model for device placement," in *International Conference on Learning Representations*, 2018.
- [9] R. Addanki, S. B. Venkatakrishnan, S. Gupta, H. Mao, and M. Alizadeh, "Placeto: learning generalizable device placement algorithms for distributed machine learning," in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, 2019, pp. 3981–3991.
- [10] H. Lan, L. Chen, and B. Li, "Accelerated device placement optimization with contrastive learning," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.
- [11] B. Jeon, L. Cai, P. Srivastava, J. Jiang, X. Ke, Y. Meng, C. Xie, and I. Gupta, "Baechi: fast device placement of machine learning graphs," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 416–430.
- [12] U. U. Hafeez, X. Sun, A. Gandhi, and Z. Liu, "Towards optimal placement and scheduling of dnn operations with pesto," in *Proceedings of the 22nd International Middleware Conference*, 2021, pp. 39–51.
- [13] B. Zhang, T. Xiang, H. Zhang, T. Li, S. Zhu, and J. Gu, "Dynamic dnn decomposition for lossless synergistic inference," in *2021 IEEE 41st International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2021, pp. 13–20.
- [14] F. Chen, P. Li, C. Wu, and S. Guo, "Hare: Exploiting inter-job and intra-job parallelism of distributed machine learning on heterogeneous gpus," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 253–264.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] Gurobi. (2021) Gurobi optimizer. [Online]. Available: <https://www.gurobi.com>
- [19] Z. Liu, H. Hu, Y. Lin, Z. Yao, Z. Xie, Y. Wei, J. Ning, Y. Cao, Z. Zhang, L. Dong *et al.*, "Swin transformer v2: Scaling up capacity and resolution," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12 009–12 019.
- [20] P. Cramer, "Alphafold2 and the future of structural biology," *Nature structural & molecular biology*, vol. 28, no. 9, pp. 704–705, 2021.
- [21] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.
- [22] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo, "A unified architecture for accelerating distributed {DNN} training in heterogeneous {GPU/CPU} clusters," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 463–479.
- [23] Z. Bian, H. Liu, B. Wang, H. Huang, Y. Li, C. Wang, F. Cui, and Y. You, "Colossal-ai: A unified deep learning system for large-scale parallel training," *arXiv preprint arXiv:2110.14883*, 2021.
- [24] C. Hu and B. Li, "Distributed inference with deep learning models across heterogeneous edge devices," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022.
- [25] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.
- [26] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 1–15.
- [27] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.
- [28] Z. Lin, Y. Miao, G. Liu, X. Shi, Q. Zhang, F. Yang, S. Maleki, Y. Zhu, X. Cao, C. Li *et al.*, "Superscaler: Supporting flexible dnn parallelization via a unified abstraction," *arXiv preprint arXiv:2301.08984*, 2023.
- [29] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *arXiv preprint arXiv:1604.06174*, 2016.
- [30] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 583–598.
- [31] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.
- [32] Y. Gao, L. Chen, and B. Li, "Post: Device placement with cross-entropy minimization and proximal policy optimization," *Advances in Neural Information Processing Systems*, vol. 31, 2018.
- [33] Y. Su, X. Ren, S. Vardi, and A. Wierman, "Communication-aware scheduling of precedence-constrained tasks on related machines," *arXiv preprint arXiv:2004.14639*, 2020.
- [34] H. Xu, Y. Liao, H. Xie, and P. Zhou, "Celeritas: Fast optimizer for large dataflow graphs," *arXiv preprint arXiv:2208.00184*, 2022.
- [35] C. Perkins, E. Belding-Royer, and S. Das, "Rfc3561: Ad hoc on-demand distance vector (aodv) routing," 2003.
- [36] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," <http://eigen.tuxfamily.org>, 2010.
- [37] M. Dukhan *et al.*, "Nnpack," <https://github.com/Maratyszcza/NNPACK>, 2016.
- [38] M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare, "On optimizing operator fusion plans for large-scale machine learning in systemml," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, 2018.
- [39] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, "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*, 2021, pp. 81–93.
- [40] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.
- [41] 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.