Cacomp: A Cloud-Assisted Collaborative Deep Learning Compiler Framework for DNN Tasks on Edge

Weiwei Lin, *Senior Member, IEEE*, Jinhui Lin, Haotong Zhang, Wentai Wu, *Member, IEEE*, Weizheng Wu, Zhetao Li, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—With the development of edge computing, DNN services have been widely deployed on edge devices. The deployment efficiency of deep learning models relies on the optimization of inference and scheduling policy. However, traditional optimization methods on edge devices still suffer from prohibitively long tuning time due to devices' low computational power. Meanwhile, the widely used scheduling algorithm, the dominant resource fairness algorithm (DRF algorithm), struggles to maximize the efficiency of model execution on edge devices and inevitably increases average waiting time as it is not applicable in the real-time distributed computing environment. In this paper, we propose Cacomp, a distributed cloud-assisted deep learning compiler framework that features accelerating the optimization on edge devices with assistance from the cloud and a novel inference task scheduling algorithm. Our framework utilizes the tuning records from the cloud devices and proposes a two-step distillation strategy to obtain the best tuning record set for the edge device. For the scheduling process, we propose an RD-DRF algorithm to allocate inference tasks to edge devices based on dominant resource matching in real time. Extensive results show that our framework can achieve up to 2.19x improvement in the optimization time compared with other methods on edge devices. Our proposed scheduling algorithm significantly shortens the average waiting time of inference tasks by 30% and improves resource utilization by 20% on edge devices.**

***Index Terms*—Deep learning compiler, deep neural networks, edge computing, resource allocation.**

Received 13 August 2024; revised 14 April 2025; accepted 3 May 2025. Date of publication 12 May 2025; date of current version 11 July 2025. This work was supported in part by Guangdong Provincial Natural Science Foundation Project under Grant 2025A1515010113, in part by Guangxi Key Research and Development Project under Grant 2024AB02018, and in part by the Major Key Project of PCL, China under Grant PCL2023A09. Recommended for acceptance by M. Kandemir. *(Corresponding author: Wentai Wu.)*

Weiwei Lin is with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510641, China, and also with Pengcheng Laboratory, Shenzhen 518066, China (e-mail: linww@scut.edu.cn).

Jinhui Lin is with the School of Computer Science and Engineering, South China University of Technology, Guangzhou 510641, China (e-mail: 202321044269@mail.scut.edu.cn).

Haotong Zhang and Weizheng Wu are with the School of Software Engineering, South China University of Technology, Guangzhou 510641, China (e-mail: sewuweizheng@mail.scut.edu.cn; hoyt.zhang77@gmail.com).

Wentai Wu is with the Department of Computer Science, College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: wentaiwu@jnu.edu.cn).

Zhetao Li is with the College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: liztchina@hotmail.com).

Keqin Li is with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TC.2025.3569132

I. Introduction

DEEP learning model has catalyzed an increasing number of modern intelligent Internet-of-Things (IoT) applications, such as autonomous driving [1], [2], augmented reality [3] and surveillance video analysis [4]. These applications typically require abundant computing resources and low latency, whereas IoT devices are resource-limited. Transferring these computations to the cloud will generate unbearable delay and fail to fulfill their QoS requirements. To alleviate this issue, edge computing [5], [6], [7], [8] is proposed, which leverages the nearby servers and infrastructures to complete the computational tasks.

In order to serve the deep learning model inference tasks from IoT devices, edge devices must be capable of executing tasks efficiently. Meanwhile, as edge devices are resource-constrained compared to cloud devices, it is of great significance to fully utilize their computational resources. The inference efficiency of deep learning models on edge devices heavily relies on two aspects of deployment, the model inference optimization and the inference task scheduling strategy.

For the optimization of inference, researchers have developed some deep learning compiler frameworks, such as TVM [9] and TensorFlow-XLA [10]. They can generate high-performance code for various hardware and various deep learning frameworks by utilizing hardware-specific optimization [11] and optimized kernel libraries [12], [13]. However, these methods require enormous engineering effort to tune for each hardware platform and operator manually. This has facilitated the rise of auto-tuning: an automated optimization process aiming at speeding up the inference of deep learning models.

As Fig. 1 shows, auto-tuning methods, such as autoTVM [9] and Ansor [14], decompose deep learning workloads into a series of subtasks and continuously search for transformation steps in the optimization space for each subtask. They use cost model to identify potential transformation steps, execute the modified subtask on the actual machine and record the actual inference time for better prediction performance. In order to achieve a better inference performance, auto-tuning methods require thousands of actual measurements on real hardware even for a small model. The prolonged tuning time has become an obstacle to swiftly deploying the optimized deep learning model. To address this problem, some researchers propose a few methods as Fig. 1 depicts, including parallelling the measurement

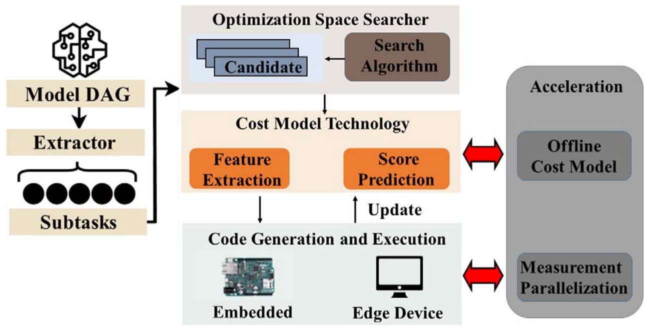


Fig. 1. Process of auto-tuning methods. Offline cost model and measurement parallelization are two methods to accelerate the tuning process.

and training an offline cost model. However, for edge devices, these methods fail to effectively speed up the optimization time as edge devices lack enough computational resources for parallelization or performing millions of measurements to train the offline cost model.

One enlightening approach to reduce the prolonged tuning time on edge devices is to introduce the cloud-edge collaboration paradigm. This appears to be an appealing solution, as it has been successfully applied to the scenario of training models, where the training process is migrated to cloud devices to accelerate the training procedure. However, there exists an obstacle to transferring this paradigm to deep learning model optimization. For instance, tuning the Inception-v3 network on the edge device takes 44 hours and the final inference time is about 670 ms. Tuning on the cloud device only takes 10 hours but the best tuning record set found by it can only reach an inference latency of 820 ms on the edge device. However, the tuning process on edge is such a prolonged one that transferring this process to cloud devices is quite appealing. Therefore, how to leverage the cloud-edge collaboration paradigm to accelerate the auto-tuning process on edge devices while maintaining the speed-up of inference time is a challenge.

Besides from the model optimization, inference task scheduling strategy also plays a crucial role in inference service deployment [15], [16], [17]. In the process of inference task scheduling, different types of models, such as VGG [18], ResNet [19] and MobileNet [20], vary greatly in terms of resource requirements. This brings about the second challenge that without a reasonable resource allocation scheme, the execution of models may be sub-optimal in resource usage. Designing a reasonable scheduling policy takes many factors into consideration and a sub-optimal scheduling policy may lead to a bad user experience for waiting too long or low efficiency in using resources.

Among the traditional resource scheduling algorithms, the dominant resource fairness algorithm [21], [22], [23], known as the DRF algorithm, is the most famous and widely deployed one. However, in our scenario, this algorithm loses its competitiveness as it is not equipped with a resource real-time update mechanism and it does not take the distributed computing scenario into consideration. How to modify this algorithm to make it suitable for deep learning inference task scheduling on edge computing is another challenge.

To address these two challenges, we propose Cacomp, a Cloud-Assisted Distributed Deep Learning Compiler Framework for deep learning model deployment on edge devices. The proposed compiler framework consists of a cloud-assisted model optimization sub-system and a inference task scheduling subsystem. The main task of the first sub-system is to fully leverage cloud devices' abundant computing resources to obtain the optimization tuning records during the optimization process. These records are transferred to edge devices and are further utilized to decide the best tuning record set for model inference on edge devices, which greatly shortens the time for optimization. The inference task scheduling module is responsible for allocating the inference tasks to the edge devices with the best resource match. We design a Realtime Distributed-DRF (RD-DRF) algorithm to generate scheduling decisions, which is developed based on the monitor in real time the resource usage of the devices. Our main contributions are as follows:

1) For optimization acceleration, we propose a cloud-assisted framework that utilizes the records from cloud devices and greatly shortens the time spent on optimization on edge devices. We propose a two-step strategy and utilize the memory access feature of the records to distill out the excellent records.
2) For inference task scheduling problem, we propose the RD-DRF algorithm. It updates the resource allocation promptly and considers the distributed computing environment to overcome the weakness of the DRF algorithm.
3) Experiments on Huawei TaiShan 200 server and Raspberry 4Bs prove that our proposed method achieves 2.19x to 4.02x optimization time improvement over other optimization methods on all testing deep learning models while maintaining the speed-up of inference time. Extensive experiments on the real edge computing environment show that, compared with the DRF algorithm, our RD-DRF algorithm effectively shortens the waiting time of inference task by up to 1.32x and task completion time by up to 1.20x.

The rest of this paper is organized as follows. Section I discusses the related work. Section III introduces the overall architecture and the design details of our distributed deep learning compiler framework. Section IV conducts the corresponding experiments and analyzes the experiment results; Finally, the summary of our work is concluded in Section V.

II. Related Work

A. Deep Learning Compiler and Auto-Tuning

Deep Learning Compiler. For the better deployment and performance of deep learning model execution, recent researchers are dedicated to various DL compiler systems, including TVM [9], TensorFlow-XLA [10] and Multi-Level Intermediate Representation (MLIR) [24]. These DL compilers embrace a series of optimization methods, including hardware-specific optimizations, optimized kernel libraries and auto-tuning techniques. Hardware-specific optimization leverages hardware intrinsic mapping, memory allocation strategy and memory latency hiding to generate high-performance codes targeting specific hardware [11]. For the optimized kernel libraries, DL compilers can leverage existing libraries, such as oneDNN [12] and cuDNN [13], by generating function calls during code generation. These two intricate methods both require extremely elaborate design. Once a new hardware platform or operator is involved, researchers have to restart the optimization process and manually tune for them.

Auto-tuning Method. In order to free researchers from significant manual effort, auto-tuning methods for the optimization of deep learning model are proposed. autoTVM [9] and Ansor [14] are two typical auto-tuning methods and a series of works are made based on them. Meta-Schedule [25] comes up with a domain-specific probabilistic programming language abstraction to use domain experts to analyze the program. Chameleon [26] introduces reinforcement learning and develops an adaptive sampling algorithm to explore previously unseen design space for code optimization. Haotuner [27] proposes a hardware-adaptive deep learning operator auto-tuner specifically designed for dynamic shape tensors for GPU devices. Roller [28] takes a different construction-based approach to generate operator kernels for various accelerators. Droplet [29] develop a search method based on the coordinate descent algorithm to find the optimal operator transformation step.

Accelerating Auto-tuning Optimization. Some methods have been proposed to alleviate the prolonged auto-tuning time. Adatune [30] proposes an adaptive evaluation method and reduces the measurement overhead from autoTVM. DOPpler [31] introduces a parallel auto-tuning measurement infrastructure whilst maintaining high-quality tensor program optimization. [32] proposes a new technique to alleviate costly candidate measurements. Moses [33] proposes MLP-based pre-trained cost models for tensor compilers to generate tensor programs much more efficiently. TLP [34] used the schedule primitives to construct an powerful offline cost model based on the Tenset dataset [35]. However, these works do not take into account the prolonged optimization time required for accelerating inference on edge devices with limited computational power. In contrast, we explore the feasibility of utilizing the optimization records from devices with abundant computational power and significantly accelerate the optimization process.

B. DRF Algorithm

Dominant Resource Fairness (DRF) scheduling algorithm is a generalized max-min algorithm for multi-resource systems [23]. DRF has four key properties: 1) Strategy-proofness, 2) Envy-freeness, 3) Pareto Efficiency, 4) Sharing Incentive. A key concept for the DRF algorithm is the dominant resource quota or Dominant Share (*DS*) for resource allocation. The formula for *DS* is as follows:

$$DS_i = \mathbf{max}_{j=1}^{n} \frac{u_{i,j}}{r_j} \tag{1}$$

DS is the dominant resource quota. *DS* is the maximum of all resource type quotas. n is the resource type. $u_{i,j}$ is the value of j class of resource assigned to model task i. r_j is the system j type resource value. In this paper, we calculate the *DS* of model tasks according to the formula (1) and allocate resources to the model based on *DS*. *DS* ensures the fairness of resource allocation. Based on this algorithm a lots of works and frameworks are proposed. Li et al. [36] considered the particularity of cloud server bandwidth resources and developed a mechanism in the cloud–edge collaborative computing system. Zhao et al. [37] propose a new allocation mechanism that generalizes bottleneck-aware allocation under fairness constraints. Jiang et al. [38] considered fairness not only in terms of a user's dominant resource but also in another resource dimension which is secondarily desired by the user. Sadok et al. [39] improved the DRF by looking at past allocations and enforcing fairness in the long run while keeping the fundamental properties of the DRF algorithm. Zhu et al. [40] introduced the concept of soft fairness and proposed QKnober to balance fairness and efficiency. However, none of these improved DRF algorithms track the resource of the device in real time or take the distributed computing environment into consideration at the same time. Our proposed RD-DRF algorithm considers these two problems and makes the original DRF algorithm suitable in our distributed edge computing environment.

III. Distributed Deep Learning Compiler Framework

We propose the cloud-assisted collaborative deep learning compiler framework (CaComp) to accelerate the optimization time for deep learning models on edge devices as well as efficiently scheduling inference tasks on edge devices. CaComp can fully exploit the tuning records from cloud devices during the optimization and swiftly find the best tuning record set for edge devices. Moreover, it improves the traditional DRF algorithm to make it possible to schedule the inference tasks in the distributed computing scenario in the most efficient way. The main structure of CaComp is shown in Fig. 2. CaComp consists of three worker modules: cloud-based model optimization worker, edge-based model optimization worker and edge-based inference task scheduling worker. The cloud-based model optimization worker runs the auto-scheduler part of TVM and sends the tuning records to the edge device. These records contain information about how the deep learning model is transformed and the corresponding performance on cloud devices. The edge-based model optimization worker next utilizes these records to select the best tuning record set for the edge device with a two-step distilling strategy. After that, the edge-based inference task scheduling worker starts its scheduling scheme and utilizes the RD-DRF algorithm to make decisions about which edge device to run the next inference task.

In conclusion, CaComp is built upon a cloud-assisted environment and endeavors to accelerate the process of model optimization on edge devices and increase scheduling efficiency. It aims to minimize the model optimization time and the waiting time of inference tasks on edge devices as well as maximizing the resource utilization rate. The detail of our framework is described in the following sections.

A. Cloud-Based Model Optimization Worker

We here first introduce the first module targeting producing tuning records for further optimization and deployment on edge

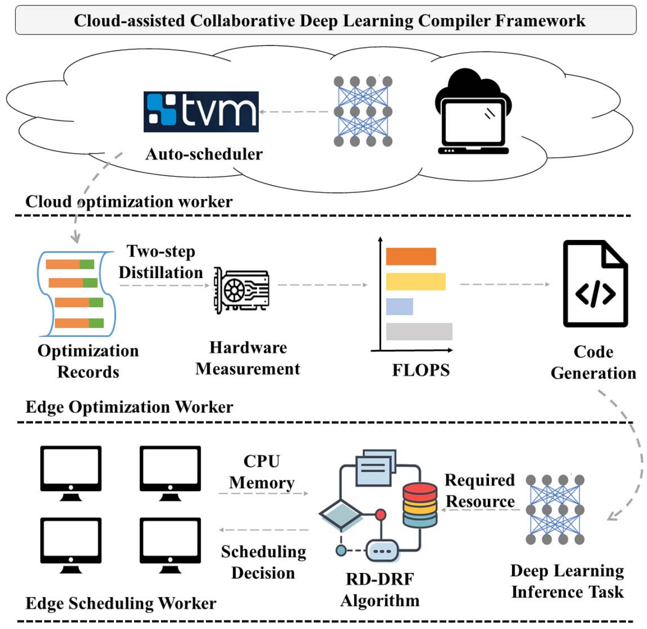
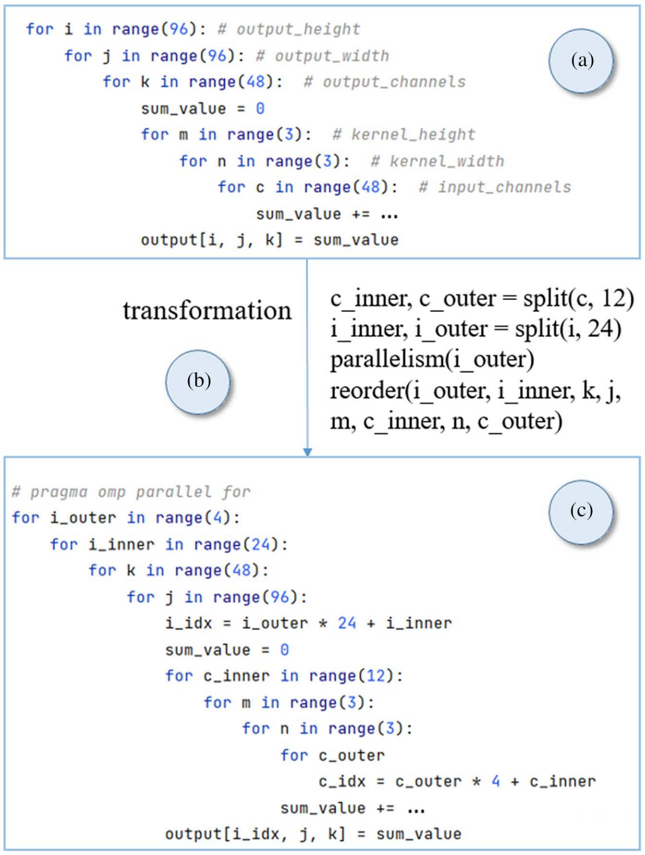


Fig. 2. Overview structure of CaComp. CaComp contains three modules. The cloud-based model optimization worker runs the optimization process and stores the tuning records. Edge-based model optimization worker uses a two-step distillation strategy to get the best tuning record set for the edge device. Edge-based inference task scheduling worker leverages our RD-DRF algorithm to make scheduling decisions.


(a)
transformation
(b)
(c)

```
for i in range(96): # output_height
    for j in range(96): # output_width
        for k in range(48):  # output_channels
            sum_value = 0
            for m in range(3):  # kernel_height
                for n in range(3):  # kernel_width
                    for c in range(48):  # input_channels
                        sum_value += ...
            output[i, j, k] = sum_value
```

c_inner, c_outer = split(c, 12)
i_inner, i_outer = split(i, 24)
parallelism(i_outer)
reorder(i_outer, i_inner, k, j, m, c_inner, n, c_outer)

```
# pragma omp parallel for
for i_outer in range(4):
    for i_inner in range(24):
        for k in range(48):
            for j in range(96):
                i_idx = i_outer * 24 + i_inner
                sum_value = 0
                for c_inner in range(12):
                    for m in range(3):
                        for n in range(3):
                            for c_outer
                                c_idx = c_outer * 4 + c_inner
                            sum_value += ...
                output[i_idx, j, k] = sum_value
```

Fig. 3. Transformation and the corresponding conv layer. (a) Is the original for-loop of the conv layer. (b) Are the transformation steps that are applied to it. (c) Is the transformed for-loop.

devices. The auto-scheduler module requires a large amount of computing resources to profile the transformed model, train the cost model for exploring the solution space and so on. However, the computational capability of edge devices is insufficient, leading to the model optimization being a time-consuming one.

Thus, we develop a cloud-assisted module to accelerate the optimization process by conducting the whole optimization process on cloud devices. Cloud devices are equipped with abundant computational resources, which makes them able to execute the optimization much quicker than edge devices do. For example, cloud devices can leverage GPUs to train the cost model and have many more CPU cores to run multiple profile inference tasks at the same time.

We slightly modify the source code of the auto-scheduler to meet our demand. Each profiling sub-process is only allocated with a few CPU cores so that the profiling results can be as close as possible to those achieved when running on edge devices. Each CPU core is assigned to one sub-processing to ensure these profiling sub-processes do not interfere with each other.

After the optimization is finished, a series of tuning records are stored and will be further exploited in the second module. Each tuning record consists of a transformation item and the result part as Fig. 3 shows. This record is about a frequently occurring Inception-v3 net layer with the kernel size of 3 * 3, feature map size of 49 * 49, input and output channel size of 96. As traditional deep learning model operators, such as batch normalization operator and convolution operator, are fundamentally matrix multiplication, the transformation on them is essentially the transformation on for-loops. Here for simplicity, we only exhibit the transformation steps with reorder, split and parallelism. The record item records how this layer is modified by traditional methods for optimizing for-loops, including tiling, reordering, unrolling and so on. These transformations may speed up the inference process by improving the locality of the code and utilizing the advantage of multi-threading. The result part records the performance after the transformation concerning running time. After the cloud device finishes its optimization process, it will send these tuning records to the edge device for further exploitation.

B. Edge-Based Model Optimization Worker

We propose the second module for swiftly constructing the optimized deep learning model for inference on edge devices. Our current focus is to leverage the tuning records from the cloud to determine the best tuning record set for the model to run on edge devices. As can be seen in Fig. 4, this module leverages a two-step distillation strategy to reduce the final measurement overheads on edge devices. The first step is picking out the relatively excellent ones based on their performance on cloud devices. The next step is using the XGBoost classifier to further distill out the edge-friendly tuning records. The details of this sub-module are as follows.

1) First Step Distillation: It is evident that directly applying the best tuning record for cloud devices will lead to sub-optimal optimization on edge devices. The best tuning record set for each device is unique as different devices possess different

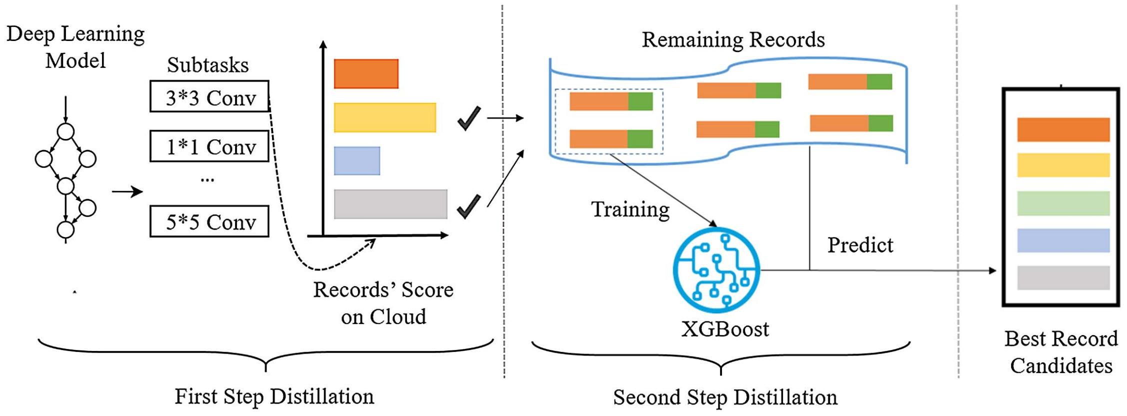


Fig. 4. Two-step distillation strategy of edge-based model optimization worker.

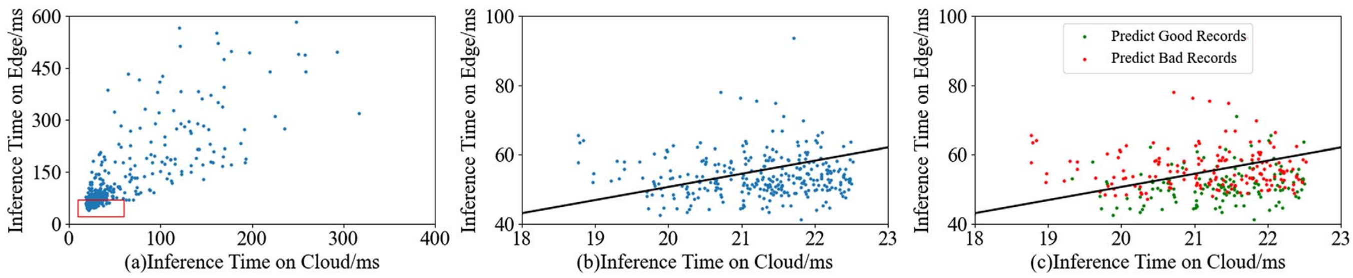


Fig. 5. (a) Is the inference time of all records from the inception-v3 conv layer on cloud and edge, respectively. (b) Is the inference time of all records on cloud and edge after the first step distillation, respectively. (c) Is the classification result of all records after the first step distillation.

hardware configurations. Besides, looping over all the records and measuring their performance on edge devices will still bring enormous overheads. Therefore, it is necessary to narrow down the searching space on edge devices by reducing the number of records.

The performance on cloud devices of these records is an excellent metric. If a tuning record reaches a relatively satisfying performance on a cloud device, we can assert that it can also behave well on an edge device as it exhibits great parallelism and locality. Conversely, the records falling short in inference time on cloud devices are more likely to be degraded ones on edge devices.

In Fig. 5(a), we select out the tuning records about the convolution layer we mentioned before. We run the transformed subtask on the edge device and record the running time to study the relationship between cloud and edge in terms of inference time. As shown in the figure, the actual inference time on the edge device almost linearly matches the one on the cloud device with a correlation coefficient of 0.84. Furthermore, there only exist few records that perform badly on cloud devices but achieve a great score on edge devices. Most of the excellent records on cloud devices still show superior performance on edge devices. This can be recognized from Fig. 5(a) where there merely exist few records on the lower right side. These results show that the best tuning record set for edge devices must exist in the collection of better records of cloud devices. Therefore, the first step we take is to retain the relatively good ones and clear out the rest. Using this simple rule we can filter out those who fail to leverage the hardware configuration of edge devices.

2) Second Step Distillation: After we filter out the badly performing records, there still exists quite a few tuning records for each subtask. Therefore, we need to further strategically distill out the potential superior records. To further gain an insight into the next step, we profile the records extracted by the first step on the edge device and depict the inference time on cloud and edge in Fig. 5(b), respectively. We can observe that some records exhibit exceptional performance on cloud devices. However, their performance degrades rapidly on edge devices, which even rank in the bottom 10% and are located at the upper left of the map. In contrast, some records also exist on the lower right of the map implying that they outperform others on edge devices although they are ordinary and even worse on cloud devices.

These two types of vividly contrary records possess different memory access features. The former one may read too much data at a time and can fully utilize the cache system on the cloud device. However, this in turn hurt the locality when running on edge as more cache misses occur which result in their performance loss. The latter may read fewer data at a time and can not completely leverage the cloud device's cache system. Nevertheless, this makes them succeed in fully exploiting the locality of edge devices and thus significantly excel over others. Therefore, we can utilize their memory access features to classify these two types of records, the edge-friendly one and the edge-unfriendly one.

To construct the classifier a training dataset is indispensable. We randomly select a portion of the filled records and the size is controlled by the size parameter α. We directly reuse the feature extraction module from the auto-scheduler and use these features to construct the input feature of the dataset, including the max step of unrolling, the production of the outer loop and so on.

After the features are obtained, we need to get the real inference time on the edge device to label these records. Only the selected records will be run and thus this procedure will not consume too much time. These sampled records come from different sub-tasks and their inference time differ in terms of time scale. We use the following formula to normalize the inference time on cloud and edge, respectively.

$$time_i = minTime(Records, i.task)/time_i \tag{2}$$

$minTime(i.task)$ is the minimum inference time among all records whose tasks are the same as record i. After this we calculate the final calibrated score by

$$time_{final} = time_{edge}/time_{cloud} \tag{3}$$

In this way, we can both consider the impact of inference time on cloud and edge. For example, if two records reach the same inference time on edge and one of them is marked with a worse score on the cloud, then it is more likely that it leverages the hardware configuration of the edge device better than the other one. Therefore, it should be given much more attention by given a higher score.

The last step is to divide these records into two categories. We set up a threshold β and those records whose scores are higher than it will be classified into the edge-friendly kind and the rest automatically go to the edge-unfriendly kind.

Finally, before training the model, these data need to be discriminated based on their score. The samples that are located at the lower right of the Fig. 5(b) are the most important ones as they fully utilize the hardware configuration of the edge device and the best tuning record exists among the ones that possess similar memory access features. The classifier's priority is to learn and extract these records' features. When it enters its prediction process, it can distinguish the relatively excellent ones from others with higher confidence. This is true for the point at the upper left of the Fig. 5(b). The rest of the records exist near the threshold line, which indicates that they perform ordinarily on edge devices, not way too excellent or awful. These records contribute nothing to our goal as they do not succeed in exploring the best transformation step in the optimization space. Meanwhile, the misclassification of them does not harm to the final result. These records should be assigned with less weight and the classifier does not need to pay too much attention to them. In our practice, we directly weigh these records by the distance between them and the threshold line. By doing so, our classifier can better serve our aim of finding the best record set.

With all data given reasonable weight, the next step is to choose and train the classifier. We use the eXtreme gradient boosting(XGBoost) model as our classifier. XGBoost outperforms other algorithms in terms of prediction accuracy by using the information of the first and second derivatives in its optimization. It also allows us to assign weight to each record.

After the XGBoost model finishes its training process, it can be utilized to further predict the possible edge-friendly records. As one record generates a few feature sets and the XGBoost model gives each feature set its classification result, we use the voting method to decide whether a tuning record is an edge-friendly one. Fig. 5(c) shows the final classification result for the aforementioned layer. As it can be seen, the records that are far away from the threshold line are almost all correctly classified. These records, especially the edge-friendly ones, are exactly what we intend to search for. The records that are adjacent to the threshold line are somewhat misclassified. This is absolutely acceptable as they have no possibility of achieving the best inference performance and the number of records that are classified as edge-friendly is still nearly the half number of total records.

Algorithm 1: Cloud-assisted Edge Inference Optimization Algorithm.

```
Input: machine learning workload $T$, cloud device $C$, edge device
    $E$, max record count $M_c$, score threshold $\beta$, sample rate $\alpha$
Output: best tuning record set $R_{best}$ for $E$
 1: Initialize subtask set $S_{task}$, tuning record list $L$, distilled tuning record list $L_d$, final tuning record list $L_f$, training dataset $D$ and XGBoost model $Xgb$.
 2: subtask set $S_{task} \leftarrow splitTask(T)$
 3: while $len(L) < M_c$ do
 4:     $L \leftarrow L \cap tuneOnCloud(C)$
 5: end while
 6: for subtask $S_i$ in $S_{task}$ do
 7:     $Score_{best} \leftarrow maxScore(L, S_i)$
 8:     $L_d \leftarrow L_d \cap distill(L, S_i, Score_{best})$
 9: end for
10: $D \leftarrow sample(L_d, \alpha)$
11: $labelTransform(D)$
12: $Xgb.train(D)$
13: $L_f \leftarrow Xgb.predictEdgeFriendly(L_d)$
14: $R_f = measureOnEdge(L_f)$
15: for subtask $S_i$ in $S_{task}$ do
16:     $R_{best}.append(bestRecord(L_f, S_i))$
17: end for
18: $return\ R_{best}$
```

Next we gather these edge-friendly records and run them on edge devices. The tuning records with the least inference time among them are the most suitable records for edge devices. Therefore, the best optimization for the deep learning model on edge is at hand. Algorithm 1 describes the entire procedure.

Our algorithm requires the deep learning model workload, the max record count, the score threshold and the sample rate as input. In lines 2-5, we first run the optimization process on a cloud device and stop when the number of records is greater than the max record count. Then in lines 6-8, we first filter out some of the records according to their score on the cloud device and the score threshold parameter controls how many records will be discarded. After that in lines 9-12, we randomly choose records from the remaining records, run them on edge device, preprocess and weigh them to train the XGboost classifier. In line 13, we use the trained model to pick out the potential records. Finally, in lines 14-17, we run the remaining records and return the best ones for the edge device.

C. Edge-Based Inference Task Scheduling Worker

We propose the last module for the scheduling of the deep learning inference tasks. All the models have already been optimized in terms of inference time and are ready for deployment on edge devices. During their inference, an amount of hardware resources, such as memory and CPU cores, are required and need to be managed by the scheduling policy.

DRF algorithm is a fairness-aware scheduling algorithm for multiple resource types, which makes it an excellent strategy for the scheduling of deep learning inference tasks. DRF algorithm has the characteristics of sharing incentives, policy verification, Pareto efficiency and being envy-free. It has been proven to be a successful multi-resource allocation scheme. However, in distributed edge computing scenarios where resource changes drastically in real time, the DRF algorithm suffers from the following defects:

1) DRF algorithm does not provide the details of the resource real-time update mechanism. DRF algorithm only records the allocated resource and does not record the resource released when a task is done. In this case, the released resource cannot be reused, resulting in a waste of resources. In addition, the lack of real-time monitoring of resources leads to inaccurate Dominant Share (*DS*) of the model. Inaccurate *DS* cannot faithfully reflect the resource relationship between devices and tasks, resulting in unfair resource allocation and unreasonable inference tasks scheduling.
2) DRF algorithm does not consider distributed scenarios. DRF algorithm considers the resources on different devices as a whole. DRF algorithm may result in the total amount of idle resources meeting the requirements while no nodes actually have enough idle resources.

Considering the above two problems, we propose the RD-DRF algorithm. In our distributed deep learning compiler framework, the RD-DRF scheduler extracts the most efficient model running resource requirements from the inference logs and monitors the devices' resources in real time. We run each model with the number of allocated CPU cores from 1 to max. Here we define the efficiency indicator as follows:

$$Efficiency(i) = InferTime(1)/(InferTime(i) * i) \tag{4}$$

i is the number of allocated cores. We choose the number with the best efficiency indicator for each model. RD-DRF algorithm uses both the real-time resource and the most efficient model running resource requirement as the scheduling inputs.

Algorithm 2 shows the pseudo-code of our RD-DRF algorithm. RD-DRF algorithm firstly obtains the real time resource R available on the devices and the real-time resource U allocated to the inference tasks and calculates the *DS* of the inference tasks. RD-DRF algorithm selects the inference task with the smallest *DS* (the smallest s_j) and obtains the resource requirement D_j of the inference task. RD-DRF algorithm traverses all devices, finds the device i that satisfies the demand value D_j and generates a scheduling decision to schedule the inference task j to device i. If there are multiple devices to meet the requirements, the RD-DRF algorithm will preferentially select the device with more idle resources. After generating the scheduling decision, the RD-DRF algorithm updates the resource value U_j of the inference task j. If no devices meet the requirement D_j, the RD-DRF algorithm waits for resource release before generating a new scheduling decision. RD-DRF algorithm repeats the scheduling process until the inference task waiting queue Q is empty.

Algorithm 2: Realtime Distributed-DRF Algorithm.

```
Input:
    The device i's real-time CPU and memory; $R_i = <r_{i,cpu}, r_{i,memory}>\ (i = 1..n)$
    The multi-inference tasks waiting queue; $Q$
    The inference task j's allocation resources; $U_j = \backslash < u_{j,cpu}, u_{j,memory}>\ (j = 1..m)$
    The inference task j's demand resources; $D_j\ (j = 1..m)$
1:  while $Q$ is not Empty do
2:      $R \leftarrow$ devices' real-time cpu and memory capacities
3:      $U \leftarrow$ inference tasks' real-time resources
4:      for $j = 1; j <= m; j++$ do
5:          $s_j = max(u_{j,cpu}/R_{CPU}, u_{j,memory}/R_{memory})$
6:      end for
7:      pick inference task j with lowest dominant share $s_j$
8:      $D_j \leftarrow$ demand resources of model task j
9:      for $i = 1; i < n; i++$ do
10:         if $D_j < R_i$ then
11:             $U_j = U_j + D_j$
12:             scheduleToDevice(j, i)
13:             break
14:         end if
15:     end for
16:     if $i > n$ then
17:         wait()
18:     end if
19: end while
```

RD-DRF algorithm records the device resource in real time and updates the inference tasks' *DS* in real time, ensuring the fairness of resource allocation and improving resource utilization. In addition, the RD-DRF algorithm considers distributed scenarios and detects whether the devices have sufficient resources before scheduling. Compared with the DRF algorithm, RD-DRF algorithm effectively improves the success rate of scheduling and reduces the average waiting time. Like the DRF algorithm, RD-DRF algorithm uses a binary heap to store the *DS* of the model tasks. However, after each scheduling, RD-DRF algorithm will traverse all model tasks and recalculate the *DS* of the model tasks. As a result, the time complexity of the RD-DRF algorithm is $\mathcal{O}(n)$.

IV. Experiments

Our framework aims at reducing the time spent on optimization for deep learning models on edge devices as well as increasing the throughput and resource utilization during the model inference process. In this section, we provide the final experimental results.

TABLE I
THE CONFIGURATION OF THE CLOUD SERVER AND EDGE DEVICE

	Huawei TaiShan 200 server	Raspberry 4Bs
CPU cores	96	4
Memory size	256 GB	8 GB
Cache size	64 KB of L1 cache capacity, 512 KB of L2 cache capacity and 48MB of L3 cache capacity	16 KB of L1 cache capacity and 128 KB of L2 cache capacity

TABLE II
THE CPU AND MEMORY REQUIREMENTS OF THE FOUR TYPES OF MODELS FOR THE MOST EFFICIENT INFERENCE ON THE RASPBERRY 4BS

Model	CPU Cores	Memory
Inception-V3	2	80 MB
Mobilenet-V2	1	330 MB
Resnet-18	2	1257 MB
VGG-19	3	1658 MB

A. Experiments Settings

Our cloud-assisted environment is supported by a cloud server and four edge devices. We use the Huawei TaiShan 200 server provided by Pengcheng Laboratory as the cloud server and Raspberry 4Bs as edge devices. The configuration of these two servers is shown in Table I. The operating system of each node is Centos7.7. The version of TVM is 0.12.dev0.we tested four classic visual convolutional neural networks (Mobilenet-v2, VGG-19, Resnet-18 and Inception-v3), which are the backbone model of nowadays deep-learning-based IoT applications.

We compared our cloud-assisted optimization methods with several prior works in terms of optimization time and inference time. We evaluated five solutions, including Auto-scheduler [14], TLP [34], Adatune [30], Meta-schedule [25] and One-Shot Tuner [41]. We did not modify any of these work's implementation.

In the scheduling experiments for the edge computing scenario, we use Kubeedge to manage our edge devices and the managing master is one of the edge device. The optimized deep learning models are containerized. At every time of making a scheduling decision, the master node will monitor every edge device's real-time resources and deploy the task to them according to the scheduling algorithm.

Our experiments test RD-DRF, DRF, Round Robin and Random Search algorithms. RD-DRF algorithm is our proposed method of improving scheduling algorithm of the DRF algorithm. Round Robin algorithm is the deep learning compiler default scheduling algorithm. Random Search algorithm is a comparison item. We evaluate the performance of the algorithms by analyzing the sequence diagram of inference tasks, the average waiting time and the average CPU utilization of the four scheduling algorithms on the worker nodes. The resources required by each inference task are shown in Table II. The number of each deep learning model's inference task is 400.

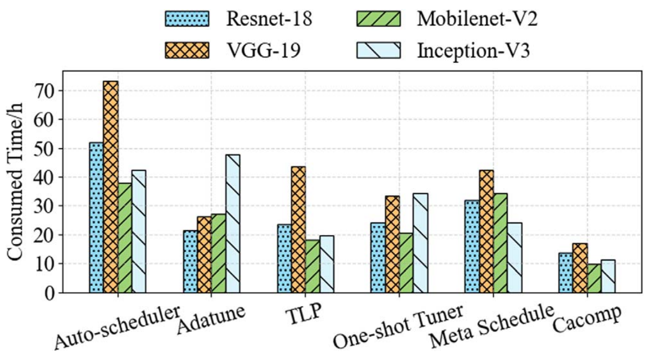


Fig. 6. End-to-end optimization time.

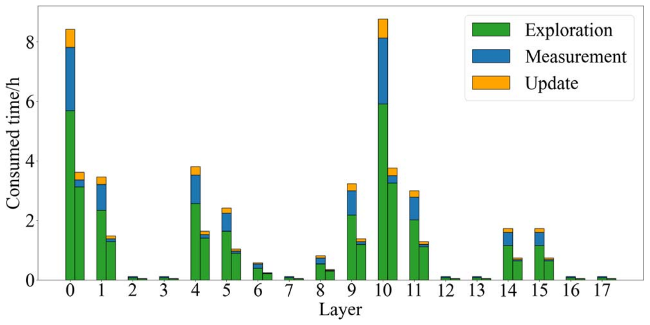


Fig. 7. Layer-wise breakdown of optimization time for VGG-19 by auto-scheduler(left bar in each pair) and cacomp(right bar in each pair).

B. Model Optimization Results

For the evaluation of our method, we use the end-to-end optimization time and inference time as the efficiency indicator of our method and other methods. We also provide ablation analysis for different classifiers and distillation strategies for our method.

1) End-to-End Optimization Time: The results of total optimization time are shown in Fig. 6, Cacomp dominates all prior work on all the convolutional neural networks in terms of consumed time by transferring the optimization process to the cloud device and reusing its tuning records to reduce the measurement overheads on edge device. As a result, our method reduces the end-to-end optimization time by 4.02, 2.39, 2.47 and 2.19 on average against other methods, respectively. The methods we compare do make an effort on how to reduce the optimization process time. However, they underestimate the cost due to the insufficient computational power on edge devices. For example, the One-Shot Tuner and TLP both pretrain a cost model to accelerate the optimization process. To obtain this cost model they need to run multiple tuning tasks on edge devices and the measurement overheads are unbearable. Our method introduces the abundant computational power from cloud devices and swiftly explores the enormous optimization space. Our two-step distillation strategy also reduces the consumed time by only measuring the relatively excellent tuning records on edge devices, which is the reason why we outperform other methods. In terms of wall-clock time, our method finished the optimization process in under 20 hours for all the models on our edge device for which other work needed 1 day to 3 days.

Fig. 7 shows that our method optimizes VGG-19 by orders of magnitude faster than the auto-scheduler. The optimization

TABLE III
THE INFERENCE TIME OF FOUR DEEP LEARNING MODELS AFTER DIFFERENT AUTO-TUNING OPTIMIZATION METHODS (MS)

	Ansor	Adatune	TLP	One-shot Tuner	Meta Schedule	Cacomp
Inception-V3	684.22 ± 6.71	763.61 ± 5.21	686.72 ± 10.04	761.09 ± 9.59	**671.99 ± 8.52**	680.81 ± 7.61
Mobilenet-V2	58.58 ± 1.25	76.42 ± 2.31	57.04 ± 1.61	78.58 ± 2.25	**56.89 ± 0.99**	59.23 ± 0.98
Resnet-18	114.36 ± 4.05	126.56 ± 3.55	**112.05 ± 5.15**	127.14 ± 4.64	117.36 ± 5.95	116.36 ± 5.08
VGG-19	**991.21 ± 20.51**	1191.81 ± 18.75	1002.55 ± 18.78	1156.72 ± 29.11	1042.83 ± 24.73	1026.19 ± 17.01

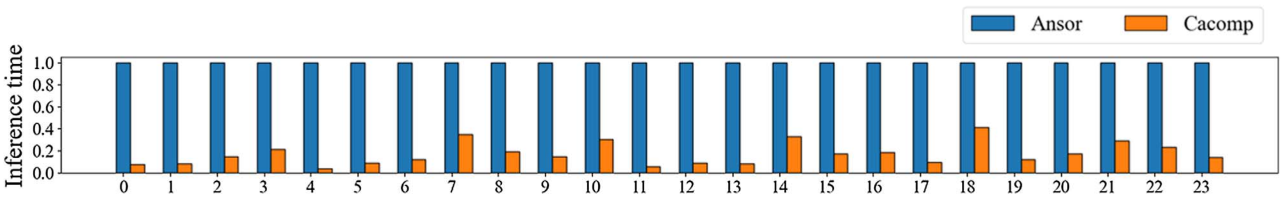


Fig. 8. Layer-wise breakdown of normalized average inference time of all tuning records for resnet-18 by ansor and cacomp.

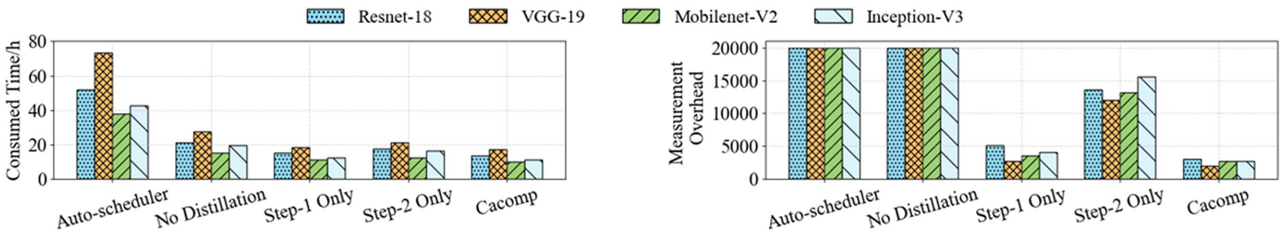


Fig. 9. Ablation analysis on the performance impact by different component of cacomp in terms of optimization time and measurement overhead on edge device.

includes three steps, the search and selection for the transformation candidate, the hardware measurements of the transformed layers and the update of the cost model. The optimization on edge devices alone performs extremely slowly as each of these steps requires abundant computing resources. In contrast, our method accelerates this process and fully utilizes the computing resources from the cloud. It speeds up the first step and the last step of the optimization by up to 4 times and the second step by up to 9 times on average.

2) Inference Time: As shown in Table III, Cacomp performs almost as well as the best inference time found by other auto-tuning methods. Adatune and One-shot Tuner are based on autoTVM, which requires manually written templates to define operators' search space. They are limited by the design of the template and thus fall short in inference time. TLP, meta schedule and Cacomp are extended from Ansor. These four methods can all find the best candidate in the optimization space regardless of their tuning efficiency. The final optimization results implies that our methods can find the best tuning record set much quicker than other auto-tuning methods do without losing any inference speed-up performance.

Fig. 8 isolates the effectiveness of Cacomp in distilling tuning records and locating high-performing candidates. As can be seen, Ansor repeatedly executes hardware measurements on edge devices. However, most of them achieve unacceptable performance and the cost model learns these tuning records' features, which makes the auto-tuning process less efficient to search in the optimization space. In contrast, Cacomp only runs the distilled tuning records from cloud devices. These high-quality records efficiently leverage the hardware configuration of edge devices and outperform most of the candidates found by Ansor. We observed that our two-step distillation strategy functions as a precise filter that efficiently narrows down the optimization space.

C. Ablation Analysis

1) Design Components: The impact of each design component in our method, i.e., the cloud-assisted optimization part, the first step distillation and the second step distillation on model optimization time is evaluated by eliminating one at a time. Fig. 9 shows that all these design components make contributions to the acceleration of optimization time. Fig. 9 also exhibits different distillation strategy's actual exploration searching space, which can be quantified by the real measurement count of tuning records. For all the models the effect of transferring the optimization process to the cloud device makes the greatest contributions as it reduces the tuning time from 51 hours on average to around 12.5 hours. The first step distillation also greatly eases the measurement overheads on edge devices as it effectively gets rid of the bad tuning records. The second step distillation can also decrease the measurement overheads on edge devices as it efficiently picks out the edge-friendly records.

2) Record Classifier: We evaluated different model architectures for the record classifier. We trained an XGBoost model, a random forest model and an LSTM-based model with the same sampled tuning records. We used the following indicator to evaluate the performance of these models:

$$Score = \frac{count(predict(records_{0.9}) == true)}{count(records_{0.9})} \tag{5}$$

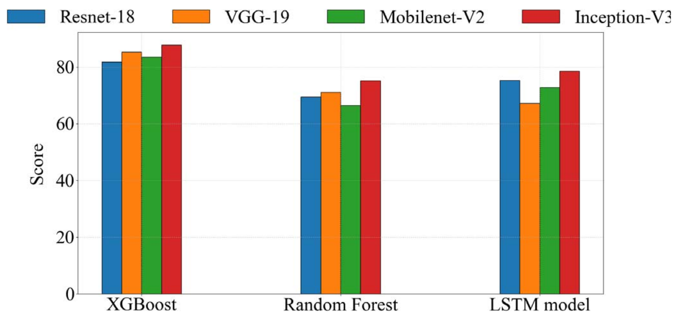


Fig. 10. Score of different classification model on different models.

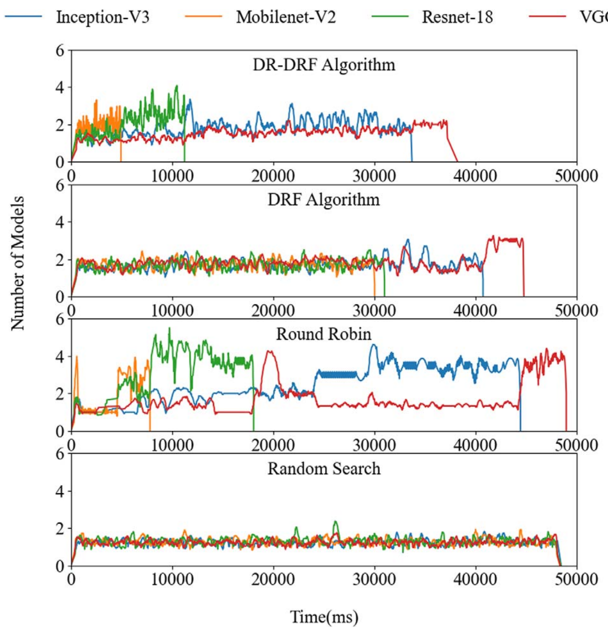


Fig. 11. Averaged number of inference tasks in execution using different algorithms.

As our classifier's target is to obtain the edge-friendly records, this indicator can fully evaluate how many records we need are truly distilled out. Here we define the records whose normalized scores are higher than 0.9 as excellent ones. As Fig. 10 shows, the XGBoost model outperforms other classifier models over all deep learning models (14.06% higher than random forest model and 11.1% higher than LSTM-based model on average).

D. *Inference Task Scheduling Results*

Fig. 11 are the sequence diagrams of models of RD-DRF algorithm, DRF algorithm, Random Search algorithm and Round Robin algorithm, respectively. From the results of the RD-DRF algorithm in Fig. 11, the inference task completion order is: Mobilenet-V2 $>$ Resnet-18 $>$ Inception-V3 $>$ VGG-19. RD-DRF algorithm preferentially schedules the inference task with the smallest *DS*. RD-DRF algorithm updates the *DS* of the inference tasks in real time through the data in Table II and the real-time resource of the inference tasks. According to the real-time *DS*, the RD-DRF algorithm calculates the priority of the inference tasks and finally generates a scheduling decision. The inference task completion order of the DRF algorithm

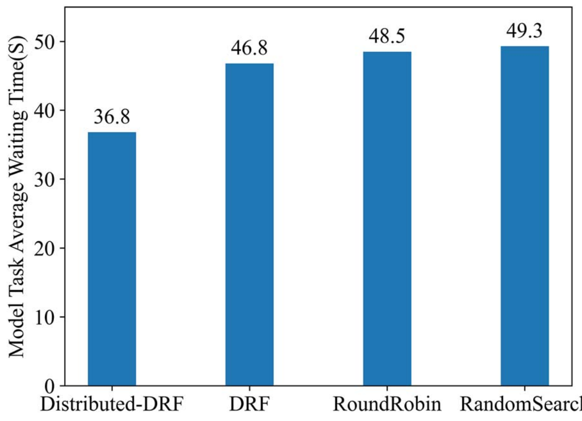


Fig. 12. Averaged inference task waiting time of four scheduling algorithms.

model is the same as that of the RD-DRF algorithm: Mobilenet-V2 $>$ Resnet-18 $>$ Inception-V3 $>$ VGG-19, but the completion time of each kind of inference task is later than RD-DRF algorithm. DRF algorithm does not consider distributed scenarios and does not update the inference tasks' resources in real time, leading to insufficient accuracy and real-time performance of *DS*. When allocating resources to inference tasks, DRF algorithm cannot allocate enough resources to the inference tasks with the smallest *DS*, and does not guarantee the fairness of resource allocation. In Fig. 11, the scheduling results of the RD-DRF algorithm show that in different periods, the number of inference tasks with the smaller *DS* is particularly prominent. However, in the scheduling results of the DRF algorithm in Fig. 11, we cannot find the advantages of the inference tasks with the smaller *DS*. This is one of the reasons why the completion time of the DRF algorithm is longer than the RD-DRF algorithm. In the figure of Round Robin, we can see that although there are occasional glitches, the overall number of inference tasks for each inference task is relatively uniform, which is in line with the characteristics of the Round Robin algorithm. For the random search algorithm, the distribution of inference tasks is chaotic, which is in line with the characteristics of the Random Search algorithm. Round Robin algorithm and Random Search algorithm do not distinguish the priority of inference tasks, resulting in unfair scheduling and a long completion time.

Figs. 12 and 13 are the diagrams of the average waiting time of the inference task and the averaged CPU utilization, respectively. From Fig. 12, the average waiting time of the inference tasks for the RD-DRF algorithm is shortened by 32.5%, 38.8% and 45.2% compared with the DRF algorithm, Round Robin algorithm and Random Search algorithm, respectively. RD-DRF algorithm ensures the efficiency of scheduling inference tasks and monitors the resource of edge devices in real time. The resources of edge devices are limited. RD-DRF algorithm monitors and recycles the resource of edge devices in real time, which is beneficial to improve the resource utilization of edge devices and reduce the average waiting time of inference tasks. In Fig. 13, the average CPU utilization on the worker nodes of the RD-DRF algorithm reaches 93.2%. Compared with the DRF algorithm, Round Robin algorithm and Random Search algorithm, the average CPU utilization of the RD-DRF algorithm is increased by 20.4%,

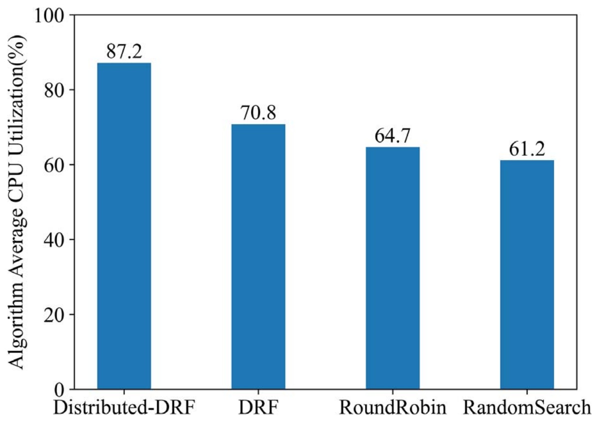


Fig. 13. Averaged CPU utilization of four scheduling algorithms.

27.5% and 29.0%. respectively. In edge scenarios with limited resources, RD-DRF algorithm greatly improves the average CPU utilization. The higher average CPU utilization of the worker nodes indicates that more CPU resources are efficiently used for model inference. The highest average CPU utilization of the RD-DRF algorithm is one of the reasons for its shortest inference tasks' completion time and shortest average waiting time.

V. Conclusion and Future Work

In this paper we present a distributed deep learning compiler framework that greatly shortens the optimization time on edge devices as well as scheduling deep learning inference task in an efficient way. Our model optimization sub-system utilizes the tuning records from cloud devices and distills out the outstanding ones. Our inference task scheduling sub-system monitors the resource usage of edge devices in real time. We design the RD-DRF algorithm to generate scheduling decisions according to the inference tasks' resource requirements and the devices' capacity. Results of the optimization time show the advantage of our two-step cloud-assisted algorithm. Results of the experiments on edge devices demonstrate that our RD-DRF algorithm can effectively shorten the inference task waiting time and improve resource utilization. In the future, we plan to consider the scenario of heterogeneous computing where GPU is further considered. We also plan to further investigate the chance of combining our work with other optimization methods of the deep learning compiler.

References

[1] Q. Song, E. Engström, and P. Runeson, "Industry practices for challenging autonomous driving systems with critical scenarios," *ACM Trans. Softw. Eng. Method.*, vol. 33, no. 4, pp. 1–35, 2024.
[2] S. Tang et al., "A survey on automated driving system testing: Landscapes and trends," *ACM Trans. Softw. Eng. Method.*, vol. 32, no. 5, pp. 1–62, 2023.
[3] P. A. Rauschnabel, B. J. Babin, M. C. Tom Dieck, N. Krey, and T. Jung, "What is augmented reality marketing? Its definition, complexity, and future," pp. 1140–1150, 2022.
[4] M. Zhang, J. Wang, Q. Qi, Z. Zhuang, H. Sun, and J. Liao, "Cognition guided video anomaly detection framework for surveillance services," *IEEE Trans. Services Comput.*, vol. 17, no. 5, pp. 2109–2123, Sep./Oct. 2024.
[5] X. Wang et al., "Wireless powered mobile edge computing networks: A survey," *ACM Comput. Surv.*, vol. 55, no. 13, pp. 1–37, 2023.
[6] X. Dai, Z. Xiao, H. Jiang, and J. C. Lui, "Uav-assisted task offloading in vehicular edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 23, no. 4, pp. 2520–2534, Apr. 2023.
[7] L. Yin, J. Sun, J. Zhou, Z. Gu, and K. Li, "ECFA: An efficient convergent firefly algorithm for solving task scheduling problems in cloud-edge computing," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3280–3293, Sep./Oct. 2023.
[8] W. Wei, Q. Ke, A. Zielonka, M. Pleszczyński, and M. Woźniak, "Vehicle parking navigation based on edge computing with diffusion model and information potential field," *IEEE Trans. Services Comput.*, vol. 16, no. 5, pp. 3827–3836, Sep./Oct. 2023.
[9] T. Chen et al., "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implement. (OSDI)*, 2018, pp. 578–594.
[10] A. Sabne, "Xla: Compiling machine learning for peak performance," *Google Res*, 2020.
[11] M. Li et al., "The deep learning compiler: A comprehensive survey," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 3, pp. 708–727, Mar. 2020.
[12] oneDNN Contributors, "OneAPI deep neural network library (ONEDNN)." [Online]. Available: https://github.com/oneapi-src/oneDNN
[13] S. Chetlur et al., "CUDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.
[14] L. Zheng et al., "Ansor: Generating {High-Performance} tensor programs for deep learning," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2020, pp. 863–879.
[15] M. Xue, H. Wu, G. Peng, and K. Wolter, "DDPQN: An efficient DNN offloading strategy in local-edge-cloud collaborative environments," *IEEE Trans. Services Comput.*, vol. 15, no. 2, pp. 640–655, Feb. 2021.
[16] M. Xue, H. Wu, R. Li, M. Xu, and P. Jiao, "EOSDNN: An efficient offloading scheme for DNN inference acceleration in local-edge-cloud collaborative environments," *IEEE Trans. Green Commun. Netw.*, vol. 6, no. 1, pp. 248–264, Jan. 2021.
[17] M. Xue, H. Wu, and R. Li, "DNN migration in IOTS: Emerging technologies, current challenges, and open research directions," *IEEE Consum. Electron. Mag.*, vol. 12, no. 3, pp. 28–38, Mar. 2022.
[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.
[19] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[20] A. G. Howard et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*.
[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implement. (NSDI)*, 2011.
[22] X. Li, W. Li, and X. Zhang, "Extended efficiency and soft-fairness multiresource allocation in a cloud computing system," *Computing*, vol. 105, no. 6, pp. 1217–1245, Jun. 2023.
[23] J. H. Sun, S. Choudhury, and K. Salomaa, "An online fair resource allocation solution for fog computing," *Int. J. Parallel Emergent Distrib. Syst.*, vol. 37, no. 4, pp. 456–477, 2022.
[24] N. Vasilache et al., "Composable and modular code generation in MLIR: A structured and retargetable approach to tensor compiler construction," 2022, *arXiv:2202.03293*.
[25] J. Shao et al., "Tensor program optimization with probabilistic programs," *Adv. Neur. Inf. Process. Syst.*, vol. 35, pp. 35783–35796, 2022.
[26] B. H. Ahn, P. Pilligundla, A. Yazdanbakhsh, and H. Esmaeilzadeh, "Chameleon: Adaptive code optimization for expedited deep neural network compilation," 2020, *arXiv:2001.08743*.
[27] P. Mu et al., "Haotuner: A hardware adaptive operator auto-tuner for dynamic shape tensor compilers," *IEEE Trans. Comput.*, vol. 72, no. 11, pp. 3178–3190, Nov. 2023.
[28] H. Zhu et al., "{ROLLER}: Fast and efficient tensor compilation for deep learning," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implement. (OSDI 22)*, 2022, pp. 233–248.
[29] M. Canesche, V. Rosário, E. Borin, and F. Quintão Pereira, "The droplet search algorithm for kernel scheduling," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 2, pp. 1–28, 2024.
[30] M. Li, M. Zhang, C. Wang, and M. Li, "Adatune: Adaptive tensor program compilation made efficient," *Adv. Neural Inf. Process. Syst.*, vol. 33, pp. 14807–14819, 2020.
[31] D. Borowiec, G. Yeung, A. Friday, R. Harper, and P. Garraghan, "Doppler: Parallel measurement infrastructure for auto-tuning deep

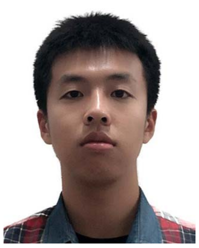
learning tensor programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 7, pp. 2208–2220, Jul. 2023.
[32] B. Steiner, C. Cummins, H. He, and H. Leather, "Value learning for throughput optimization of deep learning workloads," *Proc. Mach. Learn. Syst.*, vol. 3, pp. 323–334, 2021.
[33] Z. Zhao et al., "Moses: Efficient exploitation of cross-device transferable features for tensor program optimization," 2022, *arXiv:2201.05752*.
[34] Y. Zhai et al., "Tlp: A deep learning-based cost model for tensor program tuning," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Oper. Syst., vol. 2*, 2023, pp. 833–845.
[35] L. Zheng et al., "Tenset: A large-scale program performance dataset for learned tensor compilers," in *Proc. 35th Conf. Neural Inf. Process. Syst. Datasets Benchmarks Track (Round 1)*, 2021.
[36] X. Li, W. Li, and X. Zhang, "Multi-resource fair allocation with bandwidth requirement compression in the cloud–edge system," *Comput. Electr. Eng.*, vol. 105, 2023, Art. no. 108510.
[37] L. Zhao, M. Du, and L. Chen, "New multi-resource allocation mechanism: A tradeoff between fairness and efficiency in cloud computing," *China Commun.*, vol. 15, no. 3, pp. 57–77, 2018.
[38] S. Jiang and J. Wu, "Multi-resource allocation in cloud data centers: A trade-off on fairness and efficiency," *Concurrency Comput. Pract. Exp.*, vol. 33, no. 6, 2021, Art. no. e6061.
[39] H. Sadok, M. E. M. Campista, and L. H. M. Costa, "Stateful DRF: Considering the past in a multi-resource allocation," *IEEE Trans. Comput.*, vol. 70, no. 7, pp. 1094–1105, Jul. 2020.
[40] S. Tang, C. Yu, and Y. Li, "Fairness-efficiency scheduling for cloud computing with soft fairness guarantees," *IEEE Trans. Cloud Comput.*, vol. 10, no. 3, pp. 1806–1818, Mar. 2022.
[41] J. Ryu, E. Park, and H. Sung, "One-shot tuner for deep learning compilers," in *Proc. 31st ACM SIGPLAN Int. Conf. Compiler Construct.*, 2022, pp. 89–103.

Weiwei Lin (Senior Member, IEEE) received the B.S. and M.S. degrees from Nanchang University, in 2001 and 2004, respectively, and the Ph.D. degree in computer application from the South China University of Technology, in 2007. Currently, he is a Professor with the School of Computer Science and Engineering, South China University of Technology. His research interests include distributed systems, cloud computing, and AI application technologies. He has published more than 150 papers in refereed journals and conference proceedings. He has been a Reviewer for many international journals, including IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON COMPUTERS, IEEE TRANSACTIONS ON CYBERNETICS etc. He is a Distinguished Member of China Computer Federation.

Jinhui Lin is currently working toward the master's degree with the School of Computer Science and Engineering, South China University of Technology, China. His research interests include cloud computing and cloud-edge collaboration.

Haotong Zhang received the bachelor's and master's degrees from the South China University of Technology, in 2016 and 2019, respectively. He is currently working toward the Ph.D. degree with the South China University of Technology. His research interests include internet of things, edge computing, and cloud edge collaboration.

Wentai Wu (Member, IEEE) received the bachelor's and master's degrees in computer science from the South China University of Technology, in 2015 and 2018, respectively. Sponsored by CSC, he received the Ph.D. degree in computer science from the University of Warwick, U.K. His research interests include parallel and distributed computing, distributed machine learning, and energy-efficient computing.



Weizheng Wu received the master's degrees with the School of Software Engineering from the South China University of Technology, in 2023. His research interests include cloud computing, distributed system, distributed computing, and resource scheduling.

Zhetao Li (Member, IEEE) received the B.Eng. degree in electrical information engineering from Xiangtan University, in 2002, the M.Eng. degree in pattern recognition and intelligent system from Beihang University, in 2005, and the Ph.D. degree in computer application technology from Hunan University, in 2010. From 2013 to 2014, he was a Postdoctoral Researcher in wireless network with Stony Brook University. Currently, he is a Professor with the College of Information Science and Technology/College of Cyber Security, Jinan University. He is a member of CCF.

Keqin Li (Fellow, IEEE) received the B.S. degree in computer science from Tsinghua University, in 1985 and the Ph.D. degree in computer science from the University of Houston, in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University, China. He has authored and co-authored more than 1080 journal articles, book chapters, and refereed conference papers. He holds over 75 patents announced or authorized by the Chinese National Intellectual Property Administration. Since 2020, he has been among the world's top few most influential scientists in parallel and distributed computing regarding single-year impact (ranked #2) and career-long impact (ranked#4) based on a composite indicator of the Scopus citation database. He is listed in Scilit Top Cited Scholars (2023–2024). He was a 2017 recipient of Albert Nelson Marquis Lifetime Achievement Award for being listed in Marquis Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in American Education for over 20 consecutive years. He received the Distinguished Alumnus Award from the Computer Science Department, University of Houston, in 2018. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing, in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing, in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic), in 2023. He was a recipient of the 2022–2023 International Science and Technology Cooperation Award and the 2023 Xiaoxiang Friendship Award of Hunan Province, China. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a member of the European Academy of Sciences and Arts. He is a member of Academia Europaea (Academician of the Academy of Europe).