DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency

Shuai Zhang, Sheng Zhang, *Member, IEEE*, Zhuzhong Qian, *Member, IEEE*, Jie Wu, *Fellow, IEEE*, Yibo Jin, *Student Member, IEEE*, and Sanglu Lu, *Member, IEEE*

Abstract—The booming of Convolutional Neural Networks (CNNs) has empowered lots of computer-vision applications. Due to its stringent requirement for computing resources, substantial research has been conducted on how to optimize its deployment and execution on resource-constrained devices. However, previous works have several weaknesses, including limited support for various CNN structures, fixed scheduling strategies, overlapped computations, high synchronization overheads, etc. In this article, we present DeepSlicing, a collaborative and adaptive inference system that adapts to various CNNs and supports customized flexible fine-grained scheduling. As a built-in functionality, DeepSlicing has supported typical CNNs including GoogLeNet, ResNet, etc. By partitioning both model and data, we also design an efficient scheduler, Proportional Synchronized Scheduler (PSS), which achieves the trade-off between computation and synchronization. Based on PyTorch, we have implemented DeepSlicing on the testbed with real-world edge settings that consists of 8 heterogeneous Raspberry Pi's. The results indicate that DeepSlicing with PSS outperforms the existing systems dramatically, e.g., the inference latency and memory footprint are reduced up to 5.79× and 14.72×, respectively.

Index Terms—CNN inference, edge computing, scheduling, synchronization

1 Introduction

The last decade has witnessed the emergence of deep learning. As a representative, convolutional neural networks (CNNs) are ubiquitously utilized in a variety of applications, e.g., image classification [1], [2], [3], object detection [4], [5], [6] and video analytics [7], [8], [9]. Equipped with dedicated CNNs, these applications can detect and classify objects from images/videos accurately.

Despite these advantages, it is worth noting that CNN inference has a large demand in terms of the computing resources. For instance, VGG-16 requires 15.5G MACs (multiply-add computations) to classify an image with 224 × 224 resolution [10]. On this account, conventional solutions perform inference on computationally powerful clouds to reduce the latency. However, it is the network edge that the input data are generated at. The long distance transmission suffers from delay and jitter and it is hard to meet the requirements of real-time applications.

Recently, the proliferation of Internet of Things (IoT) brings about the growth of computing capability at network edge, hastening the birth of edge computing [11]. User data can be fed into CNNs locally to avoid remote transmissions. In order to alleviate the discrepancy between the limited capability of edge devices regarding the computation and huge resource demands of CNNs, many approaches have been explored, such as model compression [12], [13], [14], model early-exit [15], [16], [17], model partitioning [18], [19], [20], [21], [22], [23], [24], [25], data partitioning [26], [27], [28], [29], [30] and domain specific hardware/tools [31], [32]. More specifically, model compression performs revisions on the target model for a compacted one. Even though the CNN has been compressed, large input data may overwhelm an IoT device if its RAM is limited. Model early-exit tries to bypass some layers to accelerate the inference, which brings about extra training cost. Model partitioning splits CNN models between edge and cloud so as to take advantage of both the network bandwidth of edge and the computing power of cloud. Nevertheless, wide area network (WAN) transmission still exists and the sequential partition execution does not fully utilize the parallel nature of edge devices.

Different from these methods, data partitioning splits the data among edge devices and performs the inference in a parallel manner, fully utilizing the computing resources of each edge device. Besides, data partitioning is edge native, since the connections between devices at network edge are much faster and more stable than that over WAN. With this feature, communications will be more time-efficient, leading to a small inference latency.

While there are several works [26], [27], [28], [29], [30] already focusing on the data partitioning, they have the following weaknesses that exactly motivate our work:

Limited Support for CNN Structures. The structure of commonly used CNNs has become more and more complex. Previous works [26], [27] only focus on chain-like CNNs, e.g., YOLOv2 [6], VGG-16 [1]. In fact, there exist inception and residual blocks in GoogLeNet [2] and ResNet [3], which

- *Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Yibo Jin, and Sanglu Lu are with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China. E-mail: {zhangshuai.cs, yibo.jin}@smail.nju.edu.cn, {sheng, qzz, sanglu}@nju.edu.cn.*
- *Jie Wu is with the Center for Networked Computing, Temple University, Philadelphia, PA 19122 USA. E-mail: jiewu@temple.edu.*

Manuscript received 8 Oct. 2020; revised 3 Jan. 2021; accepted 2 Feb. 2021. Date of publication 11 Feb. 2021; date of current version 16 Mar. 2021. (Corresponding author: Sheng Zhang.)
Recommended for acceptance by R. Prodan.
Digital Object Identifier no. 10.1109/TPDS.2021.3058532

are not the chain structures. As a consequence, supporting general CNN structures is important and required.

Overlapped Computation and High Synchronization Cost. One type of previous works [27], [28], [29], [30] treats each computing part of the output layer as an individual task. As a result, overlaps occur between these tasks, which lead to redundant computation. The other type [26] treats each computing part of each single layer as an individual task, which leads to high synchronization cost among workers. DeepSlicing combines both data partitioning and model partitioning together, leading to a flexible fine-grained partitioning method that finally translates into low latency.

Fixed Scheduling Strategy. Given heterogeneous CNNs and edge environments, it is difficult for a fixed scheduling strategy to perform well in all cases. In other words, the optimal scheduling strategy might change with the specific situation. For example, a given strategy may prefer to distribute the data evenly among devices, which always results in the best performance when the computing capabilities of the devices are the same; however, the performance would be worse if the capabilities of the devices differ greatly from each other. Hence, if we can customize the scheduling strategy based on the knowledge of the hardware and environment, a better performance could be achieved. Albeit important, this is seldom mentioned in previous researches.

In this paper, we propose DeepSlicing, a holistic, collaborative and adaptive CNN inference system with low latency. DeepSlicing models user-specified CNNs as directed acyclic graphs (DAGs). It automatically slices the data and distributes related tasks to edge devices. Low latency is achieved through a balanced trade-off between computation and synchronization. In terms of the system design, DeepSlicing has the following features:

- *Support for a majority of CNNs.* DeepSlicing is able to support all the DAG-structured CNNs, latest state-of-the-art CNNs like ResNeXT101, RegNet included. By providing the corresponding parameters, the user is able to accelerate a customized CNN. Typical CNNs (including AlexNet, VGG, GoogLeNet and ResNet) have been built in DeepSlicing.
- *Support for customized scheduling strategies.* DeepSlicing provides a set of APIs for users to get real-time task status and data locations, conduct fine-grained scheduling and timely memory reclamation. It offers users the capability to customize their own scheduling strategy, enabling them to design the optimal strategy based on their prior knowledge. Besides, DeepSlicing optimizes the communication between workers to avoid the redundant transmissions.

We have implemented DeepSlicing and deployed it over 8 Raspberry Pi's. Empirical measurements show that, with the optimization of the memory reclamation and communication of DeepSlicing, state-of-the-art scheduling schemes are improved significantly, e.g., with the help of DeepSlicing, MoDNN [26] has 7.58× lower memory footprint and 2× smaller communication size than its original version.

On top of DeepSlicing, we also have designed the default scheduler, which is called Proportional Synchronized Scheduler (PSS). PSS uses several synchronized points (SPs) to split the CNN into blocks and assigns block-related tasks to workers in a periodic and synchronous manner. In this way, PSS trades off the cost of synchronization and computation, and senses the real-time available computing resources of each worker, resulting in a load-balanced scheduling. Experimental results show that it achieves up to 5.79× lower latency compared to the state-of-the-art scheduling scheme.

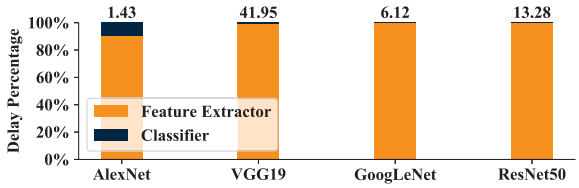


Fig. 1. Feature extractor accounts for almost the entire inference delay.

The remainder of this paper is organized as follows. Section 2 motivates our work. Section 3 presents the overview of DeepSlicing. Section 4 illustrates the Layer Range Deduction (LRD) mechanism in DeepSlicing. Section 5 proposes an efficient default scheduler, PSS. Section 6 evaluates DeepSlicing and PSS. Section 7 reviews the related work. Section 8 discusses the limitations and future work. Conclusions are given in Section 9.

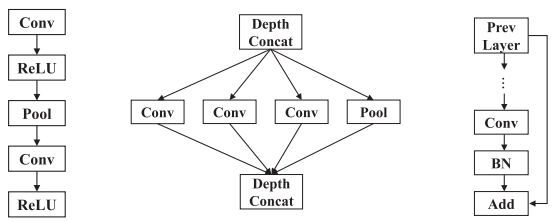
2 Background & Motivation

We aim to build a collaborative adaptive CNN inference accelerating system, in which the feature maps are partitioned and distributed to resource-limited devices, so that both memory footprint and latency are reduced. Thus, we have made an investigation in the following aspects.

2.1 Characteristics of CNN Inference

Generally speaking, a CNN consists of two parts: feature extractor and classifier. The original input is first processed by feature extractor layers and the resultant features are then classified by classifier layers. We compare the inference delay of these two parts in different CNNs in Fig. 1. The annotations on the bars are the inference delays of feature extractors in seconds. Results show that the feature extractor accounts for almost the entire inference delay and hence is exactly the bottleneck of CNN inference. This part mainly includes convolution layers (Conv), pooling layers (Pool), batch normalization layers (BN), activation layers (e.g., ReLU), etc. For each layer in feature extractor, *computing a part of its output only requires a subset of its input*. This peculiarity is beneficial to the acceleration of the feature extractor in a parallel way, which will be detailed later.

Different CNNs may have different structures, as shown in Fig. 2. VGG is a chain, GoogLeNet has multiple branches, and ResNet has two unbalanced branches with different



(a) VGG (b) GoogLeNet (c) ResNet

Fig. 2. Structures of typical CNNs.

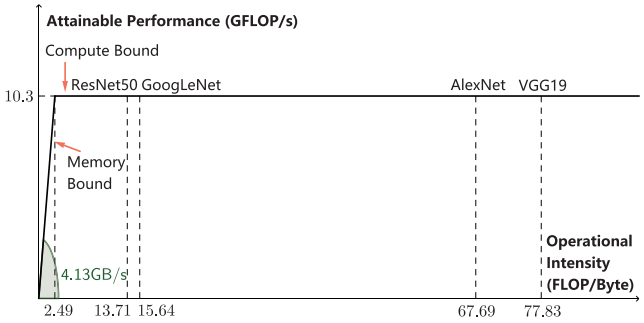


Fig. 3. Roofline model for Raspberry Pi4B and typical CNNs.

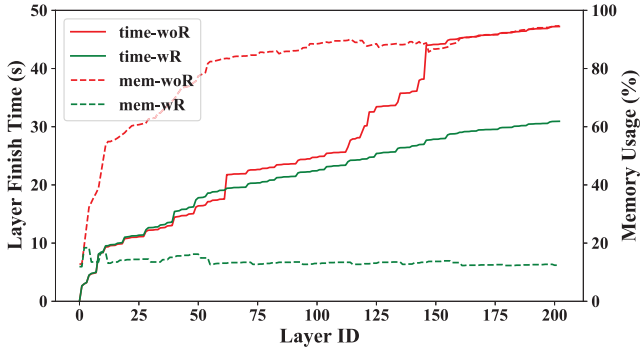


Fig. 5. Layer finish time (time) and memory usage (mem) with and without memory reclamation (R).

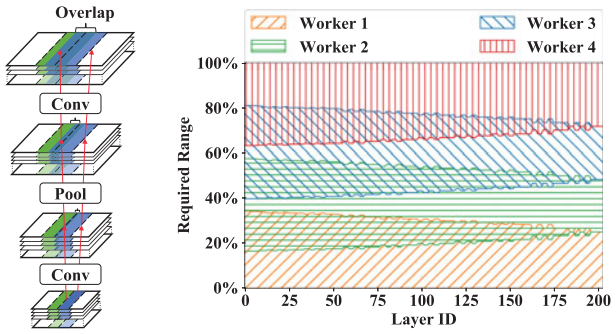
layer numbers. An adaptive system is supposed to handle various structures sensibly. However, to the best of our knowledge, no existing works [26], [27], [28], [29], [30] have mentioned supporting various CNN structures.

Roofline Analysis. We collect the performance of Raspberry Pi4B and the requirement of CNN feature extractors, calculate their operational intensities, and draw the roofline model [33] in Fig. 3. When the operational intensity is less than 2.49, the program running on Pi4B is memory-bound, otherwise it is compute-bound. It is obvious that all the typical CNNs in our experiments on Pi4B are compute-bound, which proves that the collaboration of devices is an effective way to accelerate the inference.

2.2 Overlaps in Data Partitioning

The feature map of CNN usually has two spatial dimensions (width and height) and one depth dimension. Among them, the spatial dimensions have the peculiarity mentioned in Section 2.1. Specifically, for the feature extractor layers and on the spatial dimensions, the computing of an element in the output only requires a small part of the input, not the whole. Mao *et al.* have illustrated that splitting *the longer one* among the width and height dimensions of an input feature map is more beneficial than splitting the feature map into 2D-grids because of the decrease of the number of neighbors [26]. Therefore, in this paper, we always split feature maps along the longer dimension of height and width. A natural question is how to split it.

On the one hand, some of the previous data partitioning-based work [27] treats each computing part of the output layer as an individual task. This could lead to overlapped computation and redundant tasks. We use Fig. 4a to better explain this. There are two non-overlapped regions in the bottom feature map; to compute them, the previous outputs are required. The required regions of each bottom region are marked in the same color. As we move backward, the size of the overlapped feature map continues to increase. When a CNN is deep, there is no doubt that the overlapped computation leads to a longer inference latency. Fig. 4b shows a concrete example using the split strategy in DeepThings [27]: four workers are responsible for computing four non-overlapped regions in the output layer of GoogLeNet. Layers are numbered according to the computing sequence. Different colors indicate different required input ranges of the 4 workers. The overlapped area indicates the redundant computation. It can be seen from the figure that the closer the layer is to the first layer, the more redundant computation it has.



(a) Stacked layers (b) GoogLeNet

Fig. 4. Overlapped use of feature maps in CNNs.

On the other hand, it is also not advisable to treat each computing part of each single layer as an individual task, which is the Biased One-Dimensional Partition (BODP) method used in MoDNN [26]. This will lead to extremely high synchronization cost among workers, and the mutual waiting would also greatly prolong the inference latency.

DeepSlicing combines both data partitioning and model partitioning, enabling flexible fine-grained scheduling in CNN. While traditional DAG task scheduling on heterogeneous multi-processors considers tasks as independent black boxes [34], [35], tasks in DeepSlicing can be further split and the resultant sub-tasks are correlated.

2.3 In-Time Memory Reclamation

Limited memory resources hinder the execution of complex CNNs and large input data. During the execution of a CNN, most of the intermediate feature maps will be obsolete and related memory should be reclaimed in time. Otherwise, the lack of the memory easily leads to a dramatic decrease of performance. We execute GoogLeNet on Raspberry Pi 4 Model B in two ways - with and without memory reclamation - and record the time cost and device memory usage when each layer finishes. Results are shown in Fig. 5. Without memory reclamation, memory usage soars even in the first few layers and later the time costs of some layers suddenly increase due to the lack of memory. In contrast, when memory reclamation is enabled, the memory footprint is reduced from 94.7 to 12.4 percent, 7.64× less and the inference time is reduced from 47.22 s to 30.93 s, 1.53× less.

TABLE 1
Comparision of Different Frameworks

	DeepSlicing	DeepThings [27]	MoDNN [26]
Partition method	One-dim.	2D-grid	One-dim.
Dynamic scheduling	✓	✓	×
Scheduling granularity	Arbitrary layers	CNN	Layer
General CNN	✓	×	×
Custom strategy	✓	×	×
Memory reclamation	✓	×	×

2.4 Category

Table 1 summarizes the key characteristics of the state-of-the-art distributed CNN inference frameworks. DeepThings partitions data in the 2D-grid style, shares data via a coordinator device, performs dynamic scheduling in the way of work stealing, and refers to all the layers of CNN as a single task. MoDNN partitions data in one dimension, and assigns tasks to devices according to preconfigured computing capabilities, so dynamic scheduling is not supported. Neither DeepThings nor MoDNN considers the generality of CNN, custom scheduling strategy, and memory reclamation.

3 DeepSlicing Overview

The goal of DeepSlicing is to *accelerate the collaborative inference of CNN* on the resource-constrained devices by flexible fine-grained scheduling and in-time memory reclamation. Furthermore, DeepSlicing supports various CNNs and customized scheduling strategies. Typical CNNs have been built in DeepSlicing, including AlexNet, VGG, GoogLeNet, and ResNet. Users can also define new CNNs by APIs provided in DeepSlicing. The scheduling strategy is abstracted as the Scheduler in the master and users can develop new scheduling strategies according to their own understanding of CNNs and devices. Besides, we provide an efficient scheduler PSS (Section 5) as the default scheduling strategy.

Fig. 6 overviews the architecture of DeepSlicing. There are two roles for devices: master and worker. The master monitors the global status and coordinates workers while workers perform the concrete computation tasks and transmit data to each other. DeepSlicing slices all the feature maps on only one dimension and thus we use the term *range* to represent the interval on the sliced dimension. Both the master and workers maintain the structural information of a CNN and adopt LRD (Section 4) to calculate the dependency between two arbitrary feature maps.

Master. The left part of Fig. 6 shows the architecture of the master. It maintains the Runtime Information for all layers, updates the real-time status by the Runtime Monitor, and uses the obtained information to support the decisions. The structure of the Runtime Information is the same as that of target CNN, and each vertex in this component corresponds to the metadata of each layer. Each vertex contains multiple tasks assigned by the Scheduler. The status of each task is recorded in this component too. For example, in the master part of Fig. 6, the filled slices represent those finished tasks and the blank slices represent those unfinished ones. Furthermore, each layer has 3 finished tasks, whose layer IDs are ranging from 0 to 4.

The Scheduler is the key component in master. When assigning a new job to a worker, the Scheduler first selects several layers pending for execution, and then for each layer, the Scheduler chooses a slice from unfinished output range as a task. These tasks of the selected layers form a job for a worker. For example, in the master of Fig. 6, the last job of worker 1 contains 5 tasks and the related layers are layers 0 to 4; for each layer, worker 1 only computes 1/3 of the entire output range. At the beginning of the inference, the Scheduler generates the initial job for each worker. Every time a task is finished, host worker notifies the Runtime Monitor and the Scheduler. The Runtime Monitor then tracks the related vertex in the Runtime Information and updates its status. The Scheduler further checks the status of the layers, marks those layers no longer in use as garbage and returns garbage marks to the worker. The memory occupied will be reclaimed by the worker soon. When a worker finishes its job, it requests the Scheduler for a new one.

Worker. The architecture of the worker is shown in the right part of Fig. 6. It stores the sliced feature maps in the Data Storage, executes assigned tasks in the Computation Thread, and communicates with the master by the Updater Thread. It provides and fetches remote data using the Data Server Thread and Data Fetcher Thread, respectively. Similar to the Runtime

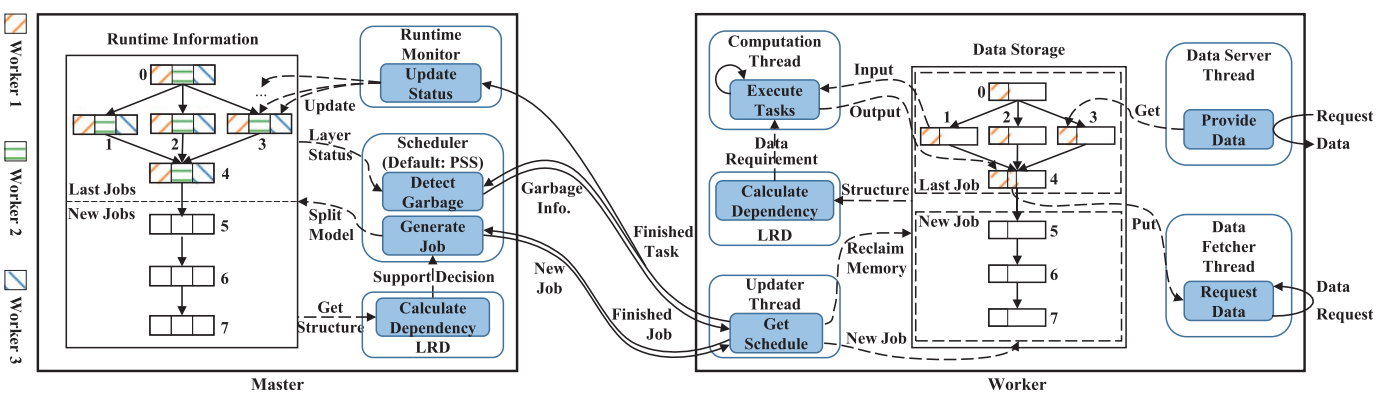
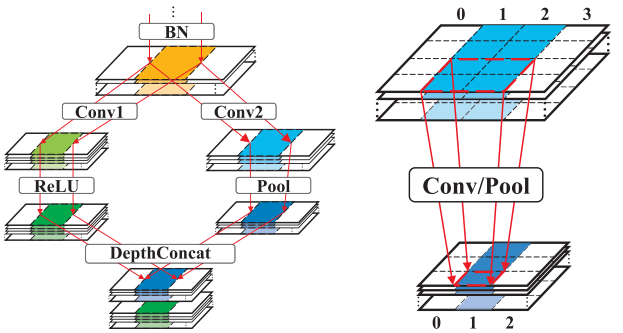


Fig. 6. The architecture of designed DeepSlicing.



(a) Arbitrary feature maps (b) Conv/Pool (K=2, S=1)

Fig. 7. Feature map dependency in a CNN.

Information in the master, each vertex in the Data Storage corresponds to the output of each layer in a CNN. The difference is that the Runtime Information only records the output range of each layer, i.e., the meta information, while the Data Storage saves the raw output data. When a worker receives a job, it first uses LRD to check whether it has the required data. If not, the Data Fetcher Thread requests the data and saves it in the Data Storage. Accordingly, the Data Server Thread provides data that other workers need.

When the data is ready, the Computation Thread gets the data from the Data Storage and executes all the tasks in the job. The output of these tasks will be saved in the Data Storage for future use. Since it stores a large amount of data, the Data Storage tends to occupy lots of device memory. To avoid this, every time a task is finished, the Updater Thread communicates with the Runtime Monitor and asks the Scheduler for garbage. According to the marked garbage layers, the Updater Thread deletes the related data to reclaim the memory. When the current job is finished, the Updater Thread updates the results to the Scheduler.

Customization. The user can easily use the customized Scheduler to accelerate a given CNN. The customization mechanism details are in the supplemental material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2021.3058532. Here is a brief introduction:

- CNN: To accelerate a customized CNN, the user needs to provide the corresponding load function. When starting, DeepSlicing will call this load function to generate the DAG of CNN. This load function receives no arguments and returns three parameters which contain structure and LRD information.
- Scheduler: A user-customized scheduler is able to access the Runtime Information and should implement four abstract methods, including: initializing the Scheduler, generating the initial task assignment, scheduling the tasks and marking the garbage data.

4 Layer Range Deduction

In order to support various CNN structures and facilitate customized scheduling strategy design, DeepSlicing provides Layer Range Deduction to query the dependency between parts of feature maps from one certain layer to another layer in a DAG-structured CNN. For example, in Fig. 7a, a worker already holds a part (the orange region) of the feature map outputted by layer BN. Prior to the following inference, LRD can calculate the parts that this worker can compute at the descendant layers. In this example, the corresponding parts are the colored regions in the outputs of Conv1, Conv2, ReLU, Pool, and DepthConcat. Conversely, if a worker is planned to compute a certain output part of some specified layer (e.g., DepthConcat), LRD can calculate the minimal input part for the previous layer (e.g., Conv1). Because only one dimension is sliced here, a *range* refers to a closed interval at the sliced dimension of the feature maps. Elements in a range are counted from 0.

4.1 Adjacent Feature Maps

In this subsection, we introduce the range deduction for two adjacent feature maps, which are only separated by one layer, e.g., the input and output of Conv1 in Fig. 7a.

In DAG-structured CNNs, there are two types of layer location that need attention: fork point and merge point. A layer at the fork point has multiple descendants and its output feature map is directly fed into the descendants, like BN in Fig. 7a. A layer at the merge point has multiple ancients and receives all their outputs, for example, DepthConcat in Fig. 7a. For such layers, outputs from different ancients have different depths but the same widths and heights, e.g., Inception in GoogLeNet and Bottleneck in ResNet. As a consequence, their input ranges are bijective to the corresponding output ranges. Activation (e.g., ReLU) and BN perform element-wise operations to the input feature maps, so that their input ranges are also bijective to the corresponding output ranges. However, the Conv and Pool layers are more complicated. They use a kernel to slide on the feature map with a certain stride and compute a pixel of output using the region that the kernel covers.

Let the input range, kernel size, and stride size be $[x, y]$, K, and S, respectively. Then the corresponding output range of layer L will be

$$out\_range_L(x,y) = \begin{cases} [\lceil \frac{x}{S} \rceil, \lfloor \frac{y-K+1}{S} \rfloor] & L = \text{Conv } or \text{ Pool} \\ [x,y] & o.w. \end{cases} \tag{1}$$

Similarly, if we need to know the required input range of layer L for the output range $[x, y]$, then it is

$$in\_range_L(x,y) = \begin{cases} [Sx, Sy+K-1] & L = \text{Conv } or \text{ Pool} \\ [x,y] & o.w. \end{cases} \tag{2}$$

Fig. 7b shows an example in which $K = 2$ and $S = 1$. Given an input range [1,2] (marked in light blue on the top), the corresponding output range (marked in dark blue on the bottom) is [1,1].

The aforementioned calculation of the expected output range and the required input range is simple, though, in real-world scenarios, there are two situations that might bring extra complexity: padding and ceil mode. Padding refers to padding zeros around the input on both width and height dimensions. Ceil mode means the kernel can cover the region on the border of an input, which does not in fact exist. For layers that enable padding, LRD pads the input of

Eq. (1) or the output of Eq. (2). For layers whose ceil mode is true, LRD complements the missing part on the border of the input when using Eq. (1) and removes the non-existent part on the border of the output when using Eq. (2).

4.2 Arbitrary Feature Maps

For two arbitrary feature maps, we develop the Arbitrary Input Range (AIR) algorithm to calculate the required range of an input or output. Take Fig. 7a for example: If the scheduler plans to make a worker responsible for computing a certain output range (marked in blue) of DepthConcat, AIR can give the minimal input range of BN that this worker requires for its responsibility. Since this worker might not have the entire range of the required feature map and needs to request related data from others, calculating the minimal range can reduce the communication size. Besides, minimal input size leads to the minimal computation.

Algorithm 1 shows the details of AIR. To compute the output range $[x_{out}, y_{out}]$ of layer O, Algorithm 1 gives the minimal required input range $[x_{in}, y_{in}]$ of layer I. Beginning from layer O, it recursively traverses backward all the layers between I and O in the DAG, calculates the minimal output ranges required by these layers, and finally gives the minimal required input range of layer I. We use the notation $\mathbb{S}$ to represent the minimal required output ranges of the intermediate layers between I and O.

Initially, the output range $\mathbb{S}_L$ of each intermediate layer L is set to $\varnothing$ to indicate that it has not been calculated. First, AIR first checks whether $\mathbb{S}_L$ is exactly $\varnothing$ to avoid potential unnecessary recursion. If not, the current procedure must be called by a parent procedure who has calculated the output range of O. At this point, AIR compares the current range and the calculated range $\mathbb{S}_O$. If $[x_{out}, y_{out}]$ is already contained in $\mathbb{S}_O$, then for all the layers in front of O, their ranges that are deduced by $\mathbb{S}_O$ will certainly cover those deduced using $[x_{out}, y_{out}]$. In this situation, the recursion of $[x_{out}, y_{out}]$ is redundant and AIR directly returns $\varnothing$ to mark this situation. If $\mathbb{S}_O$ does not contain $[x_{out}, y_{out}]$, then the output ranges of layers in front of O need to be updated. To calculate the ranges that cover both $[x_{out}, y_{out}]$ and $\mathbb{S}_O$, $[x_{out}, y_{out}]$ is updated to their union.

Then, AIR checks whether I is exactly O. If so, the result will be calculated directly by in_range defined as Eq. (2). If not, AIR checks whether layer O has any ancient. If layer O has no ancient, then this layer is the first layer of the whole CNN, and there can be no more backward recursion, which means that there is no path between I and O, and hence no dependency between the two given feature maps. Otherwise, if layer O has ancients, the required input range of layer I will be calculated recursively. The required input range $[x', y']$ of layer O is calculated by Eq. (2) and $[x', y']$ is the output range of ancient layers of layer O. AIR calls a sub-procedure for each ancient layer and each sub-procedure will return the minimal input range $[x_A, y_A]$ of layer I and the corresponding $\mathbb{S}'$, which is used to update $\mathbb{S}$. As mentioned before, $\varnothing$ is ignored. Because all the range requirements are supposed to be satisfied, the union of valid results of sub-procedures is returned as the final result.

The Arbitrary Output Range (AOR) algorithm is similar to AIR, so its details are omitted due to space limitation.

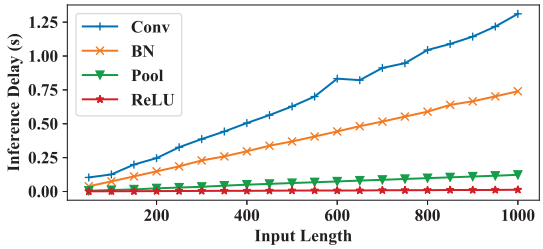


Fig. 8. The inference delay of different layers in GoogLeNet.

Algorithm 1. Arbitrary Input Range (AIR) Alg

Input: input layer I, output layer O, expected output range $[x_{out}, y_{out}]$
Output: minimal input range $[x_{in}, y_{in}]$ of layer I, minimal output ranges $\mathbb{S}$ of intermediate layers

Initialization: $\mathbb{S} \leftarrow \{\mathbb{S}_L | \mathbb{S}_L = \varnothing, \forall L\}$
if $\mathbb{S}_O \neq \varnothing$ **then**
  if $[x_{out}, y_{out}]$ *in* $\mathbb{S}_O$ **then**
    return $\varnothing, \mathbb{S}$
  else
    $[x_{out}, y_{out}] \leftarrow [x_{out}, y_{out}] \bigcup \mathbb{S}_O$
$\mathbb{S}_O \leftarrow [x_{out}, y_{out}]$
if $I = O$ **then**
  return $in\_range_O(x_{out}, y_{out}), \mathbb{S}$
else if $ancients(O) = \varnothing$ **then**
  No dependency between I and O
$[x', y'] \leftarrow in\_range_O(x_{out}, y_{out})$
$[x_{in}, y_{in}] \leftarrow \varnothing$
foreach $A \in ancients(O)$ **do**
  $[x_A, y_A], \mathbb{S}' \leftarrow$AIR($I$, A, $[x', y']$)
  Update $\mathbb{S}$ using $\mathbb{S}'$
  if $[x_A, y_A] \neq \varnothing$ **then**
    $[x_{in}, y_{in}] \leftarrow [x_{in}, y_{in}] \bigcup [x_A, y_A]$
return $[x_{in}, y_{in}], \mathbb{S}$

5 Proportional Synchronized Scheduler

The Scheduler distributes tasks to workers and also marks the garbage layers after the executions.

5.1 Estimation on Inference Delay

During the parallel collaboration, mutual waiting can severely affect the parallelism and prolong the overall latency. To avoid this, accurate prediction on the inference delay of layers is vital to the task assignment. Some previous works have studied the offline version of this problem [23], [28], [36]. Most of them try to build a regression model for each type of layer, in which the parameters of a layer are treated as the input. However, in an online environment, the historical delay can be easily obtained, which can be exploited to make predictions. We claim that *the inference delay of a layer is roughly proportional to its input size on the sliced dimension*. To verify this, we have tested all the layers in the feature extractor of GoogLeNet. For each layer type, we set the lengths of other dimensions as fixed, vary the length of the input on the width dimension, and record the inference delay. Fig. 8 shows the results, which are consistent with our claim.

Observing that the data size on the sliced dimension decreases linearly, it is also safe to assume that the input

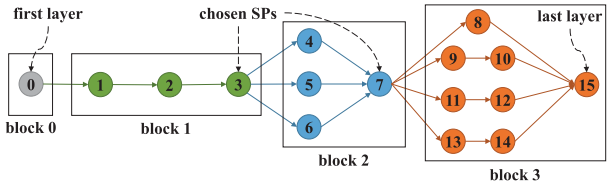


Fig. 9. PSS partitions a DAG into multiple blocks according to a user-specified parameter and its cut points.

size on this dimension is approximately proportional to the inference delay for multiple consecutive layers in a CNN.

5.2 Design of Proportional Synchronized Scheduler

As we mentioned before, a task should not contain all layers or only one layer. The former causes a large amount of redundant computation while the latter suffers from frequent synchronization. Empowered by the fine-grained flexible scheduling mechanism of DeepSlicing, we devise the Proportional Synchronized Scheduler to achieve the trade-off between computation and synchronization.

Model Partitioning. The basic idea of PSS is to partition the DAG into multiple blocks according to a user-specified parameter and corresponding cut points.[1] In each block, PSS distributes the related computation tasks to workers using data partitioning in a load-balancing manner.

PSS partitions the DAG into blocks via a series of Synchronized Points, which are chosen from cut points. Distinguishingly, the first and last vertices are always SPs such that the inference can be initialized and completed normally. For instance, if a user wants to partition the DAG into $n = 4$ blocks (as shown in Fig. 9), apart from vertices 0 and 15, PSS chooses 3 and 7 to make sure each block (except the first block which contains only one layer) has roughly the same number of layers. Workers cannot start to compute a new block before the SP of the current block is finished. Specifically, only when layer 3 is finished can PSS generate new jobs for workers, and each worker would compute a range in the output of layer 7 in block 2.

Data Partitioning. Algorithm 2 gives the details of PSS. When a user wants to partition a given CNN into n blocks, PSS chooses $P_0, P_1, \ldots, P_{n-1}$ as SPs, where P_0 and P_{n-1} are the first and last layers, respectively. As the first job, PSS evenly distributes the output range of layer P_0 to each worker. Afterwards, PSS schedules workers as shown in Algorithm 2 whenever a SP P_c is completely finished. The input parameters of Algorithm 2 are managed by PSS. For each worker w, PSS records the input range length r_w and the time cost t_w of its last job to estimate the computing capability. Besides, PSS stores the expected output length ζ_L for each layer L, which is calculated by AOR in advance. As the scheduling result, PSS returns J_w for each worker w, and J_w^L is the output range of layer L that worker w needs to compute. Correspondingly, J^L is the output range assignment of workers at the layer L. PSS first determines the output range $J^{P_{c+1}}$ of layer P_{c+1} and then generates the minimal output ranges of other layers by AIR.

1. A cut point of a graph is a vertex whose removal increases the number of connected components, e.g., the cut points in Fig. 9 are 1, 2, 3, and 7, respectively.

Based on the analysis in Section 5.1, the inference delay is proportional to the input size, so the ratio of input length of the last job to the time cost of the last job can be an estimation of the computing capability of a worker. In Algorithm 2, PSS first calculates the estimated computing capability s_w for each worker w and then splits the expected output range of layer P_{c+1} such that for each worker, the output range length is proportional to its computing capability. However, in order to make the time cost of each job as close as possible, it is the input length that should be proportional to the computing capability, not the output length.

As a consequence, PSS iteratively optimizes the output range assignment as follows. The new job contains layers between P_c (not included) and P_{c+1} (included) and thus, the input of this job is the output range of layer P_c. For each worker w, PSS uses AIR to calculate the required output range $\mathbb{S}_{P_c}$ for the current output range $J_w^{P_{c+1}}$ and estimates the time cost τ_w by the ratio of the input range length $|\mathbb{S}_{P_c}|$ to the computing capability. The $(\max(\tau) - \min(\tau))$ is the estimated gap between the least and largest time costs and also the metric of load balance of workers. A large value indicates that the load of some workers does not match their computing capabilities. Hence PSS fine-tunes the output range assignment $J^{P_{c+1}}$ to minimize the $(\max(\tau) - \min(\tau))$ by shortening the output range of light-load workers and lengthening that of heavy-load workers. This process will be repeated until $(\max(\tau) - \min(\tau))$ can be reduced no more. At this time, the input ranges will be approximately proportional to the computing capabilities of workers and the corresponding time costs of workers will be close, namely load-balanced. Then PSS uses AIR to generate the corresponding jobs. Taking the current $J_w^{P_{c+1}}$ as input, AIR gives the minimal required output ranges of layers between P_c and P_{c+1} for worker w, and the generated output ranges are the tasks in the new job J_w of worker w.

Algorithm 2. PSS

Input: finished SP P_c, input range r_w and time cost t_w of the last job of each worker w, the output range ζ_L of each layer L, the number of workers W
Output: new job J_w for each worker w

for w *from 0 to* $W-1$ **do**
  $s_w \leftarrow \frac{r_w}{t_w}$
$b \leftarrow 0$
for w *from 0 to* $W-1$ **do**
  $J_w^{P_{c+1}} \leftarrow [b, b + \frac{s_w}{\sum_i s_i}\zeta_{P_{c+1}})$
  $b \leftarrow b + \frac{s_w}{\sum_i s_i}\zeta_{P_{c+1}}$
repeat
  for w *from 0 to* $W-1$ **do**
    $[x_{in}, y_{in}], \mathbb{S} \leftarrow \text{AIR}(P_c, P_{c+1}, J_w^{P_{c+1}})$
    $\tau_w \leftarrow \frac{|\mathbb{S}_{P_c}|}{s_w}$
  Fine tune $J^{P_{c+1}}$ to reduce $\max(\tau) - \min(\tau)$
until *no more reduction on* $(\max(\tau) - \min(\tau))$
for w *from 0 to* $W-1$ **do**
  $[x_{in}, y_{in}], J_w \leftarrow \text{AIR}(P_c, P_{c+1}, J_w^{P_{c+1}})$
return *new job* J_w *for each worker* w

Garbage Detection. Given a job corresponding to SP P_c, a worker only needs to request the data of the precursor SP P_{c-1}. For example, in Fig. 9, a job of block 2 only needs the

output of layer 3. Once the output of layer 3 is ready, this job will be independent to other workers because the output of layer 3 is enough to compute the output ranges of layers in this job (i.e., block 2). Hence, during the execution of a job, except for the output of the previous SP, the output of other layers can be deleted. On the other hand, we can also identify the SPs whose descendant layers have been finished or whose output has been obtained by workers, then DeepSlicing can delete the output of these SPs. PSS detects these garbage data in time and the memory they occupied will be reclaimed shortly, thus keeping the memory footprint at a low level.

Knob for Blocks. The number of blocks as a knob can be customized for various situations. By default, the number of blocks is 4, which is found to be a good choice empirically. When the available computing resources are unknown, an extra "explore step" is provided in DeepSlicing. It adds a virtual SP in front of the original ones, which can make PSS perceive the computing capabilities earlier and partition the range of the output of the next block using that information. In fact, the number of blocks determines the frequency of synchronization and the optimal selection of the number of blocks is related to device load, transmission bandwidth, CNN structure and input size. The load-fluctuating environment needs more synchronization to sense the real-time computing capability. A high transmission bandwidth reduces the synchronization cost, hence the blocks can be more to further lessen the computation. A deep CNN tends to have much overlap data and the blocks should be more. A large size on the sliced dimension of the input barely has impact on the optimal number of blocks. We have discussed these factors in detail in the supplemental material, available online.

6 Evaluation

We implement DeepSlicing in Python, and the deep learning framework used in DeepSlicing is PyTorch, one of the most popular frameworks. In this section, we compare DeepSlicing with PSS against state-of-the-art distributed CNN inference frameworks to validate its performance.

6.1 Methodology

Metrics. Evaluation metrics are listed as follows.

Layer finish time refers to the time from the start of the CNN inference to the completion of that layer. The finish time of the last layer is exactly the total *latency* of the inference. Note that layers are numbered in the order of execution on a single machine; however, in a distributed environment, it is possible that a layer with a large ID finishes before a layer with a small ID. For example, layer 6 may finish before layer 4 in Fig. 9.

Memory footprint is the amount of memory occupied at the completion of each layer in each worker. Considering that multiple devices may be used in an experiment, we use a band-shaped region in the figures to represent the memory footprint of a framework on different workers. The middle curve in the band is the median of workers' memory footprints and the upper and lower bounds of the band are the maximum and minimum of workers' memory footprints, respectively.

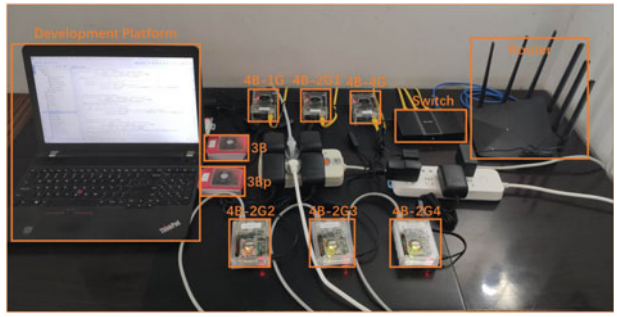


Fig. 10. Heterogeneous collaborative edge testbed.

Total computation time of a worker is the time (including computation time and synchronization time) from the beginning of the inference of a CNN to the its end.

Total communication size of a worker is the size of data exchanged between this worker and other workers.

Baselines. In evaluation, two typical state-of-the-art frameworks are implemented and used as baselines: DeepThings [27] and MoDNN [26]. DeepThings partitions the feature map of each layer into small tiles and fuses them vertically to form an independent task. Thus, the original task is divided into multiple independent tasks. Work stealing is adopted to schedule these tasks adaptively. MoDNN uses prior knowledge of computing capabilities of workers to assign tasks. Using the MapReduce programming model, MoDNN synchronizes workers at each layer and transfers data via a coordinator device.

Evaluation Setup. We implement DeepSlicing and PSS on a real-world collaborative edge computing testbed that consists of 8 Raspberry Pi's, 1 switch, and 1 router. Fig. 10 shows the implemented hardware platform for DeepSlicing. These devices represent heterogeneous hardware capabilities (Table 2). We use a router TP-LINK WDR7660 (2.4/5 GHz WiFi, 1900 Mbps) and a switch TP-LINK SG1008M (2000 Mbps) to represent wireless and wired connections, respectively. The router is used by default. We use an image with the size of 1920×1080 as the default input, which is one of the most popular video resolutions.

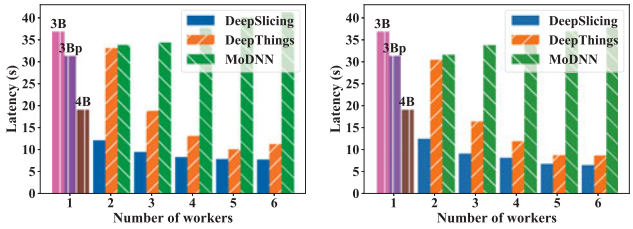
6.2 Results

6.2.1 Overall Improvements

We evaluate DeepSlicing on GoogLeNet under different numbers of workers. The workers are added in the RAM non-increasing order, i.e., 4B-4G, 4B-2G1, 4B-2G2, 4B-2G3, 4B-2G4, and 4B-1G. The results are shown in Fig. 11. For the convenience of comparison, the latency of single device is shown as well, including 3B, 3Bp and 4B. We see, under both wireless and wired networks and different worker number, that

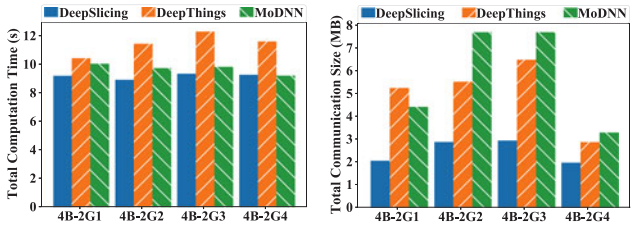
TABLE 2
Specifications of Edge Devices Used in Experiments

	Pi3B	Pi3B plus		Pi4B	
RAM	1GB	1GB	1GB	2GB	4GB
Name	3B	3Bp	4B-1G	4B-2G(1/2/3/4)	4B-4G



(a) Wireless network (b) Wired network

Fig. 11. Comparison on inference latency.



(a) Total computation time (b) Total communication size

Fig. 13. Comparison on total computation time and communication size.

DeepSlicing always performs the best because of its adaptability to the CNN structures and balance of the synchronization and communication cost. On the contrary, poor adaptability affects both DeepThings and MoDNN. For the deep CNN, a single task in DeepThings will cost much time on a worker but others cannot provide any help. In the same case, MoDNN mainly suffers from the layer-wise synchronization. As a consequence, when the number of workers increases, the latency of MoDNN increases rather than decreases and DeepSlicing outperforms MoDNN by up to 5.79×. Besides, due to the frequent synchronization, MoDNN is sensitive to the network environment. For DeepSlicing, as the number of workers increases, the memory footprint of workers is bound to decrease but the improvement of inference latency might be affected by synchronization cost. This is because the more the workers are, the more potential waiting cost will be. Please see the supplemental material for detailed discussion on the scalability, available online.

To compare the memory footprint, we use 3 identical workers (4B-2G1, 4B-2G2, 4B-2G3) to collaboratively execute GoogLeNet. The results are shown in Fig. 12. For each framework, the memory footprints of workers are close. We see the memory footprint of DeepSlicing is always the lowest and keeps below 250 MB. Compared to DeepThings and MoDNN, DeepSlicing has up to 14.72× and 8.13× smaller memory footprint, respectively. In each block in PSS, workers execute their jobs independently. Given that it will not be requested by other workers, much data will become obsolete shortly after it is generated. PSS can mark it timely to trigger the memory reclamation of DeepSlicing. Although there is a large amount of garbage data in MoDNN, it does not have any garbage collection mechanism, resulting in a high memory footprint. In DeepThings, a worker executes multiple independent tasks and hence stores a large amount of intermediate data. Even worse, there exist many overlaps among these data. As a result, its memory footprint is much more than MoDNN.

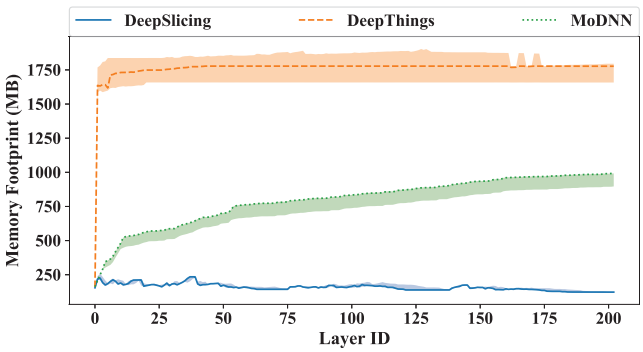
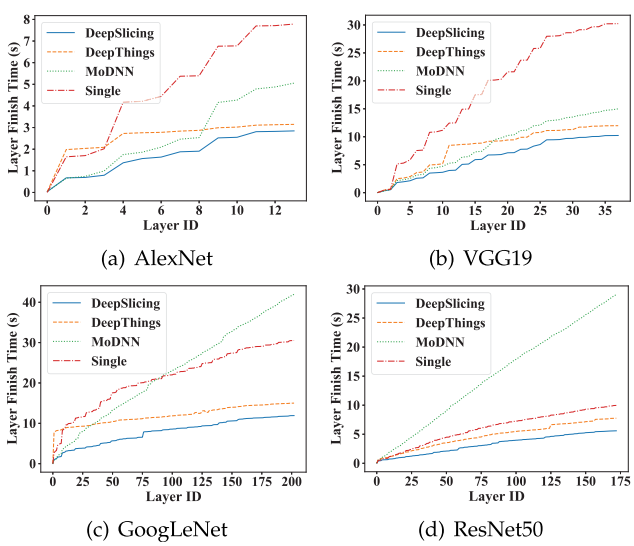


Fig. 12. Comparison on memory footprint.

Fig. 13 shows the comparison results on the computation time and communication size when using 4 identical devices (4B-2G1, 4B-2G2, 4B-2G3, 4B-2G4). Generally speaking, DeepSlicing reduces computation time by 20 percent compared to DeepThings and has 58 percent less communication data compared to MoDNN. In DeepThings, workers take more time to request data than DeepSlicing, thus it has a larger computation time than DeepSlicing. Based on MapReduce, MoDNN transfers intermediate data via a coordinator device and such data relay brings about redundant transmission. By block based synchronization, DeepSlicing requires fewer data requests than DeepThings and transmits less data than MoDNN.

6.2.2 Supporting a Variety of CNNs

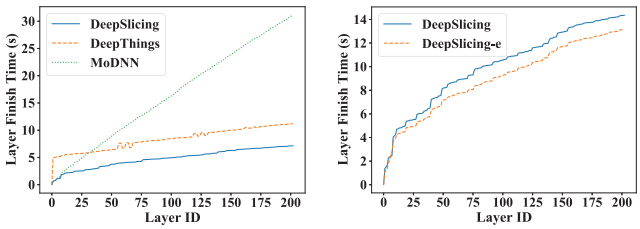
In this subsection, we compare the layer finish times of three frameworks on four typical CNNs (AlexNet, VGG19, GoogLeNet, and ResNet50). Inference latency of a single machine is also provided for comparison. Four identical workers (4B-2G1, 4B-2G2, 4B-2G3, 4B-2G4) are used. Fig. 14 shows the advantage of DeepSlicing. When the CNN is simple (AlexNet and VGG19), MoDNN outperforms the single machine. However, when the CNN is complex and deep (GoogLeNet and ResNet50), MoDNN is worse than a single machine. On



(a) AlexNet (b) VGG19

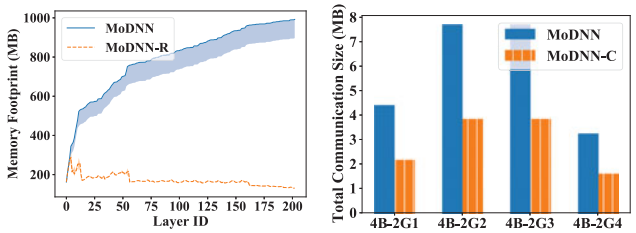
(c) GoogLeNet (d) ResNet50

Fig. 14. Layer finish time of four typical CNNs.



(a) Layer finish time (b) The effect of the explore step

Fig. 15. Layer finish time under heterogeneous computing capabilities.



(a) MoDNN with memory reclamation (b) MoDNN with optimized communication

Fig. 17. Performance improvement of MoDNN by DeepSlicing.

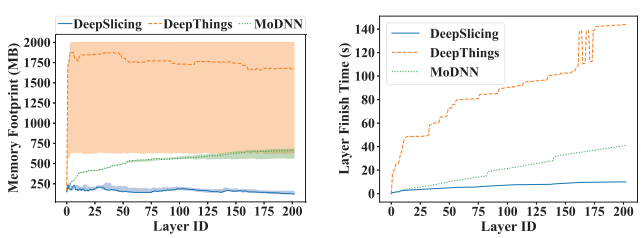
the contrary, DeepSlicing outperforms a single machine, DeepThings, and MoDNN by up to 2.95× (VGG16), 1.39× (ResNet50) and 5.20× (ResNet50), respectively. Whe-ther the architecture has a lot parameters (AlexNet), residual connections (ResNet) or other, DeepSlicing always have a significant performance improvement.

6.2.3 Impact of Heterogeneity

We first evaluate the impact of the heterogeneous computing capabilities of workers. We use 3 workers (3B, 3Bp, 4B-1G) with different CPUs and the same RAM size (i.e., 1 GB), connected using the switch. To avoid the influence of memory shortage, we use the 960×450 resolution. Fig. 15a shows the results. In spite of the prior knowledge of the computing capabilities, the frequent synchronization slows down the MoDNN. Owing to the complexity of CNN, a single task in DeepThings involves a lot computation. When a computationally weak worker is trapped in such a task, others cannot help until this task is finished. Coarse granularity affects the performance of work stealing. Different from them, DeepSlicing learns the available computing capabilities of workers at each synchronization point and adjusts the workloads correspondingly. Compared with DeepThings and MoDNN, DeepSlicing reduces the inference latency by 1.57× and 4.37×, respectively.

We are also interested in evaluating the effect of the explore step. The results in Fig. 15b show that the explore step (DeepSlicing-e) improves the latency by 8.4 percent. For the layers with IDs smaller than 12, their layer finish time is pretty close, but the advantage of DeepSlicing-e becomes apparent after layer 12. With the help of the explore step, DeepSlicing-e learns the computing capabilities of workers earlier than the original DeepSlicing.

We then evaluate the impact of the heterogeneous memories of workers. We use 3 workers (4B-1G, 4B-2G, 4B-4G) with the same configurations except the RAM size. Fig. 16a shows that DeepSlicing and MoDNN have a lower memory footprint than DeepThings. In DeepThings, the memory footprints of devices are dramatically unbalanced because the performance of 4B-1G is affected by memory shortage early and hence most tasks are assigned to others. Fig. 16b shows the comparison results on layer finish time. Because of the low memory footprint, DeepSlicing is almost not affected by the memory heterogeneity and has a 14.32× and 4.08× smaller latency compared to DeepThings and MoDNN, respectively. We note that there are many steep rises in the layer finish time of MoDNN and DeepThings, indicating that the 4B-1G suffers from memory shortage.



(a) Memory footprint (b) Layer finish time

Fig. 16. Performance under memory heterogeneity.

6.2.4 Contribution of Each Component

The layer-based memory reclamation mechanism of DeepSlicing supports PSS to keep the memory footprint at a low level. Same as PSS, MoDNN also synchronizes workers periodically (with different periods). This indicates that there is also lots of useless data which can be deleted directly.

We use the APIs in DeepSlicing to mark the garbage layers for MoDNN (denoted by MoDNN-R) and observe its memory footprint. In this experiment, we use 3 identical workers (4B-2G1, 4B-2G2, 4B-2G3) and the results are shown in Fig. 17a, in which the reduction in memory footprint is up to 7.58×. Besides, the memory footprints become close and thus the memory load of workers is balanced.

As mentioned earlier, communication via a coordinator device leads to redundant transmissions in MoDNN. On the contrary, data transmission in DeepSlicing is directly sending from the source to the destination without any third-party relay. This mechanism can also be used to improve MoDNN. Similarly, we use related APIs in DeepSlicing to help MoDNN obtain the data location and transmit data between workers (denoted by MoDNN-C). We use 4 identical workers. The comparison results are shown in Fig. 17b, in which the communication mechanism in DeepSlicing helps MoDNN reduce the communication data size by up to 2.0×.

7 Related Work

A variety of approaches have been proposed to accelerate the inference of DNN at the edges, including model compression [12], [13], [14], model early-exit [15], [16], [17], model partitioning [18], [19], [20], [21], [22], [23], [24], [25], data partitioning [26], [27], [28], [29], [30] and domain specific hardware/tools [31], [32].

Model Compression and Early-Exit. Model compression tries to prune the redundant connections by learning which

connections are unimportant [12] and apply the L1-norm channel pruning and Fisher pruning [13]. There is also new structure designed to reduce the computational requirement [14]. In face of large input data, even a compressed model generates large intermediate data which might overwhelm an IoT device. DeepSlicing is able to handle this situation and orthogonal to model compression.

Model early-exit argues that it is not necessary to execute a whole DNN to get the result [17]. BranchyNet modifies several well-known DNNs by adding exit branches to the original models [16] and DeepIns achieves the early-exit based collaboration [15]. These approaches require the developers to scrutinize the weights and structures of DNNs and retrain models, which is time-consuming. DeepSlicing has a series of developer-friendly APIs, facilitating the rapid deployment of pre-trained CNNs, even with new structures.

Model Partitioning and Data Partitioning. Model partitioning splits the DAG and distributes partitions to different devices. [18] and [19] partitioned between edge devices and cloud servers. Jeong *et al.* performed computation and DNN distribution in parallel [20]. Ko *et al.* reduced the communication cost by encoding [21]. This method is mainly used to execute DNN between edge and cloud, bringing about a large amount of intermediate data transmissions.

Data partitioning distributes data partitions to devices and executes them in parallel. MoDNN proposed two partition schemes for convolutional and fully-connected layers, respectively [26]. A single task in DeepThings referred to the whole CNN, leading to overlapped computation and redundant tasks [27]. Stahl *et al.* focused on the partition of fully-connected layers to achieve the fully distributed execution [30]. Hadidi *et al.* studied the optimal partitioning method for each layer [29]. These systematic researches barely paid attention to the variety of CNN structures or scheduling strategies while theoretical researches tended to ignore the synchronization cost between devices.

Domain Specific Hardware/Tools. The Intel OpenVino modifies the model and accelerates the computation by underlying libraries [31]. The Google TPU is designed to provide enough computational power for deep learning [32]. DeepSlicing is orthogonal to them and can run with these specified technologies after necessary adaptions. The detailed discussion is presented in the supplemental material, available online.

DeepSlicing combines both data partitioning and model partitioning, leading to a flexible fine-grained partitioning method that finally translates into low latency.

8 Discussion

We discuss several limitations of DeepSlicing that may motivate future work.

CNN Distribution. DeepSlicing distributes pre-trained CNN models to workers in the initialization phase. This may prolong the inference time, but once distribution is complete, workers can process every input with it. So distribution dost not cost much time in the long term. Moreover, we can split the entire CNN model into multiple parts and make the distribution and execution taking place in parallel.

Memory Reclamation. At present, the basic unit of memory reclamation in DeepSlicing is a single layer. Such granularity is enough for PSS to keep the memory footprint at a pretty low level. However, more fine-grained memory reclamation can bring better performance to work-stealing based schedulers and is left as our future work.

Continuous Inputs. DeepSlicing currently only supports CNN inference with a single input. If continuous input is supported, the Scheduler can learn the characteristics of workers and CNNs from the previous runs, leading to more judicious decisions. Nonetheless, a single run involves $\sim$800 tasks (GoogLeNet with 4 devices), which still can feed a lot information to Scheduler for sensing the environment.

Graph Convolutional Network (GCN). GCN is an effective tool to extract the feature in the graph-structured data. Although CNN and GCN both involve convolution, the input of GCN has no *local translational invariance*, which enables the data partitioning for CNN. As a consequence, it may be hard to apply DeepSlicing to GCN directly.

9 Conclusion

In this work, we propose DeepSlicing, a collaborative adaptive CNN inference system. Our key findings are as follows: 1) DeepSlicing is general: on the one hand, it supports various CNN structures and customized scheduling strategy; on the other hand, it is a combination of model and data partitioning and traditional partitioning methods can be seen as special cases of DeepSlicing. 2) DeepSlicing is adaptive: it optimizes memory reclamation and communication, and it distributes computation workloads to workers based on their available resources. 3) DeepSlicing is configurable: it allows developers to trade off between computation and synchronization via specifying the number of synchronized points. 4) DeepSlicing is efficient: experimental results show that DeepSlicing with PSS reduces the inference latency and memory footprint by up to 5.79$\times$ and 14.72$\times$ than state-of-the-art frameworks. Moreover, by incorporating fully-connected layer acceleration, DeepSlicing can be extended to support more applications.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant 2017YFB1001801, in part by NSFC under Grant 61872175, 61832008, in part by NSF of Jiangsu Province under Grant BK20181252, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015.
[2] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2015, pp. 1–9.
[3] 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.
[4] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 91–99.
[5] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.*, 2016, pp. 21–37.

[6] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2017, pp. 7263–7271.
[7] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 377–392.
[8] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *Proc. Conf. ACM Special Interest Group Data Commun.*, 2018, pp. 253–266.
[9] K. Hsieh *et al.*, "Focus: Querying large video datasets with low latency and low cost," in *Proc. 13th USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 269–286.
[10] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, and J. Zhang, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1738–1762, Aug. 2019.
[11] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.
[12] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
[13] E. J. Crowley, J. Turner, A. Storkey, and M. O'Boyle, "A closer look at structured pruning for neural network compression," 2019.
[14] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2018, pp. 6848–6856.
[15] L. Li, K. Ota, and M. Dong, "Deep learning for smart industry: Efficient manufacture inspection system with fog computing," *IEEE Trans. Ind. Informat.*, vol. 14, no. 10, pp. 4665–4673, Oct. 2018.
[16] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *Proc. 23rd Int. Conf. Pattern Recognit.*, 2016, pp. 2464–2469.
[17] S. Scardapane, M. Scarpiniti, E. Baccarelli, and A. Uncini, "Why should we add early exits to neural networks?," *Cogn. Comput.*, vol. 12, no. 5, pp. 954–966, 2020.
[18] Y. Kang *et al.*, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Comput. Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.
[19] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *Proc. IEEE Conf. Comput. Commun.*, 2019, pp. 1423–1431.
[20] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon, "IONN: Incremental offloading of neural network computations from mobile devices to edge servers," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 401–411.
[21] J. H. Ko, T. Na, M. F. Amir, and S. Mukhopadhyay, "Edge-host partitioning of deep neural networks with feature space encoding for resource-constrained Internet-of-Things platforms," in *Proc. 15th IEEE Int. Conf. Adv. Video Signal Based Surveillance*, 2018, pp. 1–6.
[22] S. Dey, J. Mondal, and A. Mukherjee, "Offloaded execution of deep learning inference at edge: Challenges and insights," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops*, 2019, pp. 855–861.
[23] M. Xu, F. Qian, and S. Pushp, "Enabling cooperative inference of deep learning on wearables and smartphones," 2017, *arXiv: 1712.03073*.
[24] W. He, S. Guo, S. Guo, X. Qiu, and F. Qi, "Joint DNN partition deployment and resource allocation for delay-sensitive deep learning inference in IoT," *IEEE Internet of Things J.*, vol. 7, no. 10, pp. 9241–9254, Oct. 2020.
[25] D. Hu and B. Krishnamachari, "Fast and accurate streaming CNN inference via communication compression on the edge," in *Proc. IEEE/ACM 5th Int. Conf. Internet-of-Things Des. Implementation*, 2020, pp. 157–163.
[26] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proc. Des. Autom. Test Eur. Conf. Exhib.*, 2017, pp. 1396–1401.
[27] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.
[28] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, 2019, pp. 195–208.
[29] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Collaborative execution of deep neural networks on Internet of Things devices," 2019, *arXiv: 1901.02537*.
[30] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Fully distributed deep learning inference on resource-constrained edge devices," in *Proc. Int. Conf. Embedded Comput. Syst.*, 2019, pp. 77–90.
[31] Intel distribution of OpenVINO toolkit. [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html
[32] Cloud TPU. [Online]. Available: https://cloud.google.com/tpu
[33] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[34] K. Yang, M. Yang, and J. H. Anderson, "Reducing response-time bounds for DAG-based task systems on heterogeneous multicore platforms," in *Proc. 24th Int. Conf. Real-Time Netw. Syst.*, 2016, pp. 349–358.
[35] M. Han, T. Zhang, Y. Lin, and Q. Deng, "Federated scheduling for typed DAG tasks scheduling analysis on heterogeneous multicores," *J. Syst. Archit.*, vol. 112, 2020, Art. no. 101870.
[36] S. Yao *et al.*, "FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proc. 16th ACM Conf. Embedded Netw. Sensor Syst.*, 2018, pp. 278–291.

Shuai Zhang received the BS degree from the Department of Computer Science and Technology, Nanjing University, China, in 2019, where he is currently working toward the master's degree under the supervision of associate professor Sheng Zhang. He is a member of the State Key Laboratory for Novel Software Technology. He has published two papers including IEEE GLOBECOM 2018 and CCF TON. Currently, his research interests include edge computing and cloud computing.

Sheng Zhang (Member, IEEE) received the BS and PhD degrees from Nanjing University, China, in 2008 and 2014, respectively. He is an associate professor with the Department of Computer Science and Technology, Nanjing University, China. He is also a member of the State Key Laboratory for Novel Software Technology. His research interests include cloud computing and edge computing. To date, he has published more than 80 papers, including those appeared in the *IEEE Transactions on Mobile Computing*, *IEEE/ACM Transactions on Networking*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *MobiHoc*, *ICDCS*, *INFOCOM*, *SECON*, *IWQoS*, and *ICPP*. He received the Best Paper Award of IEEE ICCCN 2020 and the Best Paper Runner-Up Award of IEEE MASS 2012. He is the recipient of the 2015 ACM China Doctoral Dissertation Nomination Award. He is a senior member of CCF.

Zhuzhong Qian (Member, IEEE) received the PhD degree in computer science, in 2007. He is an associate professor with the Department of Computer Science and Technology, Nanjing University, P. R. China. Currently, his research interests include cloud computing, distributed systems, and pervasive computing. He is the chief member of several national research projects on cloud computing and pervasive computing. He has published more than 30 research papers in related fields.

Jie Wu (Fellow, IEEE) is the director of the Center for Networked Computing and Laura H. Carnell professor at Temple University, Philadelphia, Pennsylvania. He also serves as the director of International Affairs at College of Science and Technology. He has served as a chair of Department of Computer and Information Sciences from the summer of 2009 to the summer of 2016 and associate vice Provost for International Affairs from the fall of 2015 to the summer of 2017. Prior to joining Temple University, Philadelphia, Pennsylvania, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University, Boca Raton, Florida. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including the *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Service Computing*, *Journal of Parallel and Distributed Computing*, and *Journal of Computer Science and Technology*. He was general co-chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, ACM MobiHoc 2014, ICPP 2016, and IEEE CNS 2016, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society distinguished visitor, ACM distinguished speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). He is a CCF distinguished speaker. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.

Yibo Jin (Student Member, IEEE) received the BS degree from the Department of Computer Science and Technology, Nanjing University, China, in 2017, where he is currently working toward the PhD degree under the supervision of Professor Sanglu Lu. He was a visiting student with the Hong Kong Polytechnic University, Hong Kong, in 2017. His research interests include big data analytics, edge computing and distributed machine learning.

Sanglu Lu (Member, IEEE) received the BS, MS, and PhD degrees from Nanjing University, China, in 1992, 1995, and 1997, respectively, all in computer science. She is currently a professor with the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology. Her research interests include distributed computing, wireless networks, and pervasive computing. She has published more than 80 papers in referred journals and conferences in the above areas.