DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters

Zhuoran Zhao, *Student Member, IEEE,* Kamyar Mirzazad Barijough, *Student Member, IEEE,*
Andreas Gerstlauer, *Senior Member, IEEE*

***Abstract*—Edge computing has emerged as a trend to improve scalability, overhead and privacy by processing large-scale data, e.g. in deep learning applications locally at the source. In IoT networks, edge devices are characterized by tight resource constraints and often dynamic nature of data sources, where existing approaches for deploying Deep/Convolutional Neural Networks (DNNs/CNNs) can only meet IoT constraints when severely reducing accuracy or using a static distribution that can not adapt to dynamic IoT environments. In this paper, we propose DeepThings, a framework for adaptively distributed execution of CNN-based inference applications on tightly resource-constrained IoT edge clusters. DeepThings employs a scalable Fused Tile Partitioning (FTP) of convolutional layers to minimize memory footprint while exposing parallelism. It further realizes a distributed work stealing approach to enable dynamic workload distribution and balancing at inference runtime. Finally, we employ a novel work scheduling process to improve data reuse and reduce overall execution latency. Results show that our proposed FTP method can reduce memory footprint by more than 68% without sacrificing accuracy. Furthermore, compared to existing work sharing methods, our distributed work stealing and work scheduling improve throughput by 1.7-2.2x with multiple dynamic data sources. When combined, DeepThings provides scalable CNN inference speedups of 1.7x-3.5x on 2-6 edge devices with less than 23MB memory each.**

I. Introduction

In IoT applications, environmental context analysis is a key but also one of the most challenging tasks given often massively distributed, diverse, complex and noisy sensing scenarios [1]. Deep/Convolutional Neural Networks (DNNs/CNNs) have been intensively researched and widely used in large scale data processing due to their comparable classification accuracy to human experts [2]. As such, DNN/CNN-based data analysis techniques naturally become a solution to deal with the large data streams generated by IoT devices.

However, executing DNN inference locally on mobile and embedded devices requires large computational resources and memory footprints [3] that are usually not available in IoT computing platforms. Cloud-assisted approaches trade off local DNN inference performance for often unpredictable remote server status and network communication latency, in addition to potentially exposing private and sensitive information during data transmission and remote processing. Furthermore, given that the number of IoT devices is predicted to be rapidly increasing and reaching more than 20 billion by 2020 [4], the resulting rapid explosion and scale of collected data is projected to make even centralized cloud-based data processing infeasible in the near future [1], [5].

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2018 and appears as part of the ESWEEK-TCAD special issue.

Alternatively, fog or edge computing has been proposed to make use of computation resources that are closer to data collection end points, such as gateway and edge node devices [6], [7], [8]. However, how to efficiently partition, distribute and schedule CNN inference within a locally connected yet severely resource-constrained IoT edge device cluster is a challenging problem that has not been adequately addressed. Existing approaches for layer-based partitioning of DNN inference applications result in processing a large amount of intermediate feature map data locally. This introduces significant memory footprint exceeding the capabilities of typical IoT devices. Furthermore, existing distributed DNN/CNN edge processing proposals are based on static partitioning and distribution schemes, where dynamically changing data sources and varying availability of computational resources in typical IoT edge clusters can not be optimally explored.

In this paper, we propose DeepThings, a novel framework for locally distributed and adaptive CNN inference in resource-constrained IoT devices. DeepThings incorporates the following main innovations and contributions:

1) We propose a Fused Tile Partitioning (FTP) method for dividing convolutional layers into independently distributable tasks. In contrast to partitioning horizontally by layers, FTP fuses layers and partitions them vertically in a grid fashion, which minimizes memory footprint regardless of the number of partitions and devices, while also reducing communication and task migration overhead.
2) We develop a distributed work stealing runtime system for IoT clusters to adaptively distribute FTP partitions in dynamic application scenarios. Our approach avoids centralized data synchronization overhead of existing work sharing solutions to better exploit available parallelism and computation/communication overlap in the presence of dynamic data sources.
3) Finally, we introduce a novel work scheduling and distribution method to maximize reuse of overlapped data between adjacent FTP partitions and further improve distributed inference performance.

Results show that, in comparison to prior work, DeepThings is able to execute otherwise infeasible CNN inference tasks with significantly improved performance in IoT edge clusters under various dynamic application environments.

The rest of the paper is organized as follows: After an overview of related work, background and a motivational example in Sections II and III, details of DeepThings will be

described in Section IV. Section V then discusses the results of our experiments, while Section VI presents a summary and conclusions of our work.

II. Related Work

Different approaches have been proposed to adopt deep learning frameworks in mobile/IoT applications. In [9], [10], DNN inference is partially offloaded to cloud servers to minimize processing latency and edge device resource consumption. However, in such cloud-assisted approaches, inference performance depends on unpredictable cloud availability and communication latency. Moreover, such offloading schemes can potentially expose privacy issues. Finally, existing cloud approaches use a layer-based partitioning with memory footprints that require aggressive or complete offloading when only severely constrained IoT edge devices are available.

In order to directly deploy DNNs on resource-constrained edge devices, various sparsification and pruning techniques have been proposed [11], [12]. More recently, compression techniques have been developed to further simplify neural network topologies and thus reduce complexities [13], [14], [15]. In all cases, loss of accuracy is inevitable, while available redundancies are also highly application-/scenario-dependent. Our work is orthogonal, where we aim to meet resource constraints by executing unmodified networks in a partitioned and distributed fashion on a cluster of IoT devices.

In the context of mobile systems, the work in [16] proposed to deploy DNNs onto a set of smartphone devices within a Wireless Local Area Network (WLAN). A MapReduce-like distributed programming model is employed to synchronously orchestrate the CNN inference computation among a fixed set of mobile devices. Their approach partitions individual layers into slices to increase parallelism and reduce memory footprint, but still executes slices layer-by-layer in a centrally coordinated, bulk-synchronous and lock-step fashion. By contrast, we fuse layers and apply a finer grid partitioning to minimize communication, synchronization and memory overhead. Layer fusion techniques have been previously applied to CNN accelerators [17]. Such work is, however, specifically aimed at exploiting operator-level parallelism targeting fixed-function hardware implementations. As such, it is limited to fixed-size, single-pixel and static partitions. By contrast, we provide a general and parameterizable grid-based partitioning targeting task-level parallelism in edge devices.

The work in [16] also profiles and partitions the original CNN statically. This assumes that available computation resources are predictable and known a priori. In [18], a work stealing framework is proposed for dynamic load balancing in distributed mobile applications. We adopt a similar but more lightweight approach to execute fine-grain and fused inference stacks in a dynamically distributed, adaptive, asynchronous and low overhead fashion on IoT clusters. In the process, we incorporate a custom work distribution policy to optimize data reuse among inference partitions.

III. Background and Motivation

In CNN-based data analytics, convolutional operations are the key as well as the most resource-demanding components.

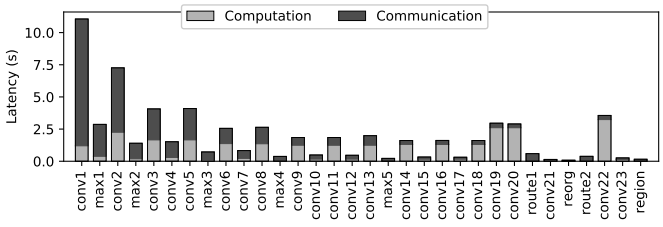


Fig. 1: Per layer execution and comm. latency in YOLOv2.

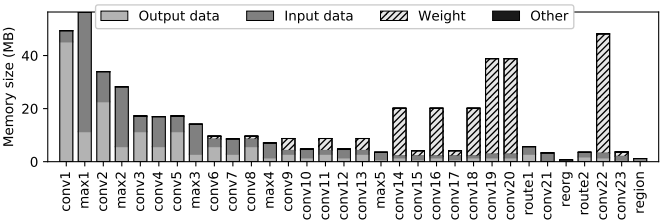


Fig. 2: Inference memory footprint of YOLOv2.

Figure 1 shows the per layer execution duration and output data transmission time over a WiFi network for a widely used CNN-based object detection application[1]. During CNN inference, large dimensional visual data is generated in early convolutional layers in order to extract fine-grained visual context from the original images. In the following inference layers, visual information is gradually compressed and down-sampled using high-dimensional filters for high-level feature reasoning. Such data compression behaviour along CNN inference layers motivates existing partitioning schemes at layer granularity [9]. As shown in Figure 1, delays for communication of output data between layers are gradually decreasing with inference progress, where an intermediate layer with lightweight feature data output can serve as a partition point between edge and gateway devices. However, this straightforward approach can not explore the inference parallelism within one data frame, and significant computation is still happening in each layer. Apart from the long computation time, early convolutional layers also leave a large memory footprint, as shown by memory profiling results in Figure 2. Taking the first 16 layers in YOLOv2 as an example, the maximum memory footprint for layer execution can be as large as 70MB, where the input and output data contribute more than 90% of the total memory consumption. Such large memory footprint prohibits the deployment of CNN inference layers directly on resource-constrained IoT edge devices.

In this paper, we focus on partition and distribution methods that enable execution of early stage convolutional layers on IoT edge clusters with lightweight memory footprint. The key idea is to slice original CNN layer stacks into independently distributable execution units, each with smaller memory footprint and maximal memory reuse within each IoT device. With these small partitions, DeepThings will then dynamically balance the workload among edge clusters to

[1]YOLOv2 (608x608) is instantiated in Darknet with NNPACK acceleration [19], [20], [21]. Execution time is collected based on the single-core performance of a Raspberry Pi 3.

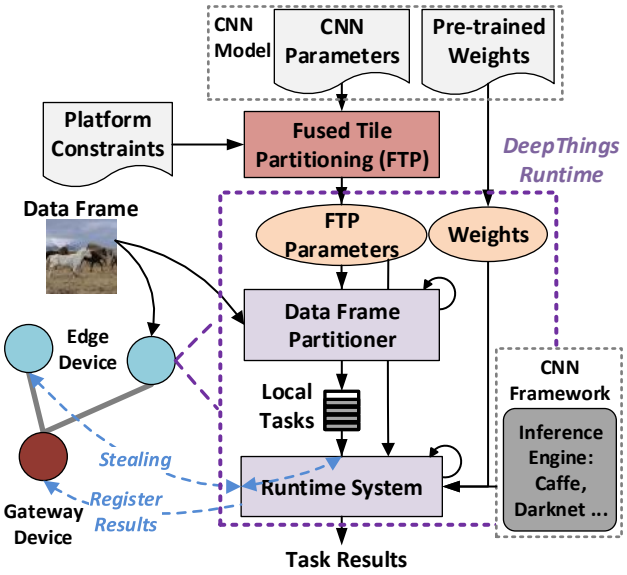


Fig. 3: Overview of the DeepThings framework.

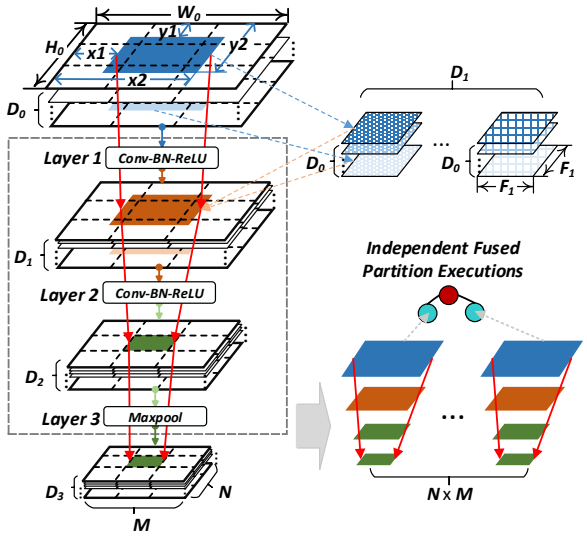


Fig. 4: Fused Tile Partitioning for CNN.

enable efficient locally distributed CNN inference under time-varying processing needs.

IV. DeepThings Framework

An overview of the DeepThings framework is shown in Figure 3. In general, DeepThings includes an offline CNN partitioning step and an online executable to enable distributed adaptive CNN inference under dynamic IoT application environments. Before execution, DeepThings takes structural parameters of the original CNN model as input and feeds them into a *Fused Tile Partitioning (FTP)*. Based on resource constraints of edge devices, a proper offloading point between gateway/edge nodes and partitioning parameters are generated in a one-time offline process. FTP parameters together with model weights are then downloaded into each edge device. For inference, a DeepThings runtime is instantiated in each IoT device to manage task computation, distribution and data communication. Its *Data Frame Partitioner* will partition any incoming data frames from local data sources into distributable and lightweight inference tasks according to the pre-computed FTP parameters. The *Runtime System* in turn loads the pre-trained weights and invokes an externally linked CNN inference engine to process the partitioned inference tasks. In the process, the *Runtime System* will register itself with the gateway device, which centrally monitors and coordinates work distribution and stealing. If its task queue runs empty, an IoT edge node will poll the gateway for devices with active work items and start stealing tasks by directly communicating with other DeepThings runtimes in a peer-to-peer fashion. Finally, after edge devices have finished processing all tasks for a given data source associated with one of the devices, the gateway will collect and merge partition results from different devices and finish the remaining offloaded inference layers. A key aspect of DeepThings is that it is designed to be independent of and general in supporting arbitrary pre-trained models and external inference engines.

A. Fused Tile Partitioning

Figure 4 illustrates our FTP partitioning on a typical example of a CNN inference flow. In deep neural network architectures, multiple convolutional and pooling layers are usually stacked together to gradually extract hidden features from input data. In a CNN with L layers, for each convolutional operation in layer $l = 1...L$ with input dimensions $W_{l-1} \times H_{l-1}$, a set of D_l learnable filters with dimensions $F_l \times F_l \times D_{l-1}$ are used to slide across D_{l-1} input feature maps with a stride of S_l. In the process, dot products of the filter sets and corresponding input region are computed to generate D_l output feature maps with dimensions $W_l \times H_l$, which in turn form the input maps for layer $l + 1$. Note that in such convolutional chains of deep neural networks, the depth of intermediate feature maps (D_l) are usually very high to be able to encode a large amount of low level feature information. This results in extremely large memory footprints and potential communication overhead when following a layer-based partitioning. However, we can observe that each output data element only depends on a local region in the input feature maps. Thus, each original convolutional layer can be partitioned into multiple parallel execution tasks with smaller input and output regions and hence reduced memory footprint each. At the same time, regions that are connected across layers can be fused into a larger task that processes intermediate data locally and thus reduce communication overhead while maintain the same reduced memory footprint.

Based on these properties, we propose a Fused Tile Partitioning (FTP) method to parallelize the convolutional operation and reduce both the memory footprint and communication overhead for early stage convolutional layers. In FTP, the original CNN is divided into tiled stacks of convolution and pooling operations. The feature maps of each layer are divided into small tiles in a grid fashion, where corresponding feature map tiles and operations across layers are vertically fused together to constitute an execution partition and stack. As shown in Figure 4, the original set of convolutional and

pooling layers are thus partitioned into $N \times M$ independent execution stacks. By fusing consecutive convolutional layers together, the intermediate feature maps will remain within the edge node device that is assigned such a partition, where only input feature maps are potentially migrated among edge nodes, while output feature maps need to be transmitted to the gateway. Furthermore, by partitioning fused stacks along a grid, the sizes of intermediate feature maps associated with each partition can be reduced to any desired footprint based on the grid granularity. Multiple partitions can then be iteratively executed within one device, where the memory requirement is reduced to only fit a maximum of one partition at a time instead of the entire convolutional feature maps.

In the process, we also need to consider that, due to the nature of the convolutions, data regions of different partitions will overlap in the feature space. In order to distribute the inference operations and input data, each partition's intermediate feature tiles (marked in orange in Figure 4) and input region (marked in blue in Figure 4) need to be correctly located based on the output partition (marked in green in Figure 4) and cropped out of the original feature maps. In FTP, the region for a tile $t_{l,(i,j)} = (t1_{l,(i,j)}, t2_{l,(i,j)})$ at grid location (i, j) in the output map of layer l is represented by the x and y index coordinates $t1_{l,(i,j)} = (x1_{l,(i,j)}, y1_{l,(i,j)})$ of its top left and $t2_{l,(i,j)} = (x2_{l,(i,j)}, y2_{l,(i,j)})$ of its bottom right elements in the original feature maps. During FTP, the output data is first partitioned equally into non-overlapping grid tiles with a given grid dimension. Then, with the output offset parameters, a recursive backward traversal is performed for each partition (i, j) to calculate the required tile region in each layer as follows:

$$\begin{aligned} x1_{l-1,(i,j)} &= \max(0,\ S_l \times x1_{l,(i,j)} - \lfloor \tfrac{F_l}{2} \rfloor) \\ y1_{l-1,(i,j)} &= \max(0,\ S_l \times y1_{l,(i,j)} - \lfloor \tfrac{F_l}{2} \rfloor) \\ x2_{l-1,(i,j)} &= \min(S_l \times x2_{l,(i,j)} + \lfloor \tfrac{F_l}{2} \rfloor,\ W_{l-1} - 1) \\ y2_{l-1,(i,j)} &= \min(S_l \times y2_{l,(i,j)} + \lfloor \tfrac{F_l}{2} \rfloor,\ H_{l-1} - 1) \end{aligned} \quad (1)$$

for convolutional layers, and for pooling layers:

$$\begin{aligned} x1_{l-1,(i,j)} &= S_l \times x1_{l,(i,j)} \\ y1_{l-1,(i,j)} &= S_l \times y1_{l,(i,j)} \\ x2_{l-1,(i,j)} &= \min(S_l \times x2_{l,(i,j)} + S_l - 1,\ W_{l-1} - 1) \\ y2_{l-1,(i,j)} &= \min(S_l \times y2_{l,(i,j)} + S_l - 1,\ H_{l-1} - 1). \end{aligned} \quad (2)$$

For a CNN with L layers, the final input offsets for each partition in the CNN input map at $l = 1$ can be obtained by applying Equations (1) and (2) recursively starting from partition offsets in the output map at $l = L$ with initial coordinates of $(\ x1_{L,(i,j)}, y1_{L,(i,j)}\ ) = (\frac{W_L \times j}{M}, \frac{H_L \times i}{N})$ and $(\ x2_{L,(i,j)}, y2_{L,(i,j)}\ ) = (\frac{W_L \times (j+1)}{M} - 1, \frac{H_L \times (i+1)}{N} - 1)$. The steps of this FTP process are summarized in Algorithm 1.

An illustrative example of a 2×2 FTP partition with one convolutional layer is demonstrated in Figure 5. The input and output data footprints of partition $(0, 1)$ and $(1, 0)$ at layer $l = 1$ are highlighted with colors, where concrete values of corresponding parametrized variables are annotated. A set of $3 \times 3 \times 3$ filters are applied to each partitioned input feature map tile with a stride of 1. Under such convolutional parameters, an input tile needs an 1-element wide extra boundary to generate the corresponding output tile. Taking partition $(0, 1)$ and $(1, 0)$ as examples, the output region index coordinates are $t_{1,(0,1)} = ((3, 0), (5, 2))$ and $t_{1,(1,0)} = ((0, 3), (2, 5))$ after applying a non-overlapping grid partition. The required input regions will be $t_{0,(0,1)} = ((2, 0), (5, 3))$ and $t_{0,(1,0)} = ((0, 2), (3, 5))$, respectively, according to Equation (1).

Algorithm 1 Fused Tile Partitioning (FTP) algorithm

```
procedure FTP({W_i, H_i, F_i, S_i, Type_i}, N, M)
    i ← 0; j ← 0;
    while i < N do
        while j < M do
            t_{L,(i,j)} ← Grid(W_L, H_L, N, M, i, j)
            l ← L
            while l > 0 do
                t_{l-1,(i,j)} ←
                    Traversal(t_{l,(i,j)}, W_{l-1}, H_{l-1}, F_l, S_l, Type_l)
                l ← l − 1
            j ← j + 1
        i ← i + 1
    return {t_{l,(i,j)} | 0≤l≤L, 0≤i<N, 0≤j<M}
```

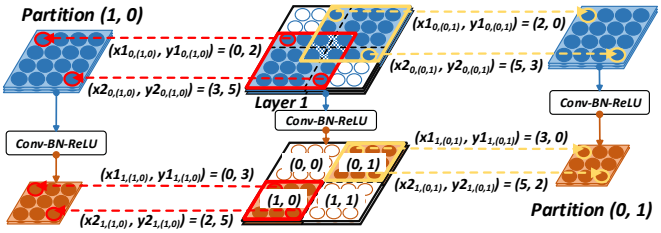


Fig. 5: FTP example ($L = 1, N = M = 2, F_1 = 3, S_1 = 1$).

B. Distributed Work Stealing

During inference, the FTP-partitioned early convolutional layers need to be distributed among edge node devices in a load balancing manner. To deal with the dynamic nature of workloads in IoT clusters, we employ a work stealing scheme to adaptively distribute inference tasks, where idle devices can steal partitioned input data from peers to accelerate their inference processing.

An example message activity and synchronization flow diagram for our stealing approach is shown in Figure 6. Whenever

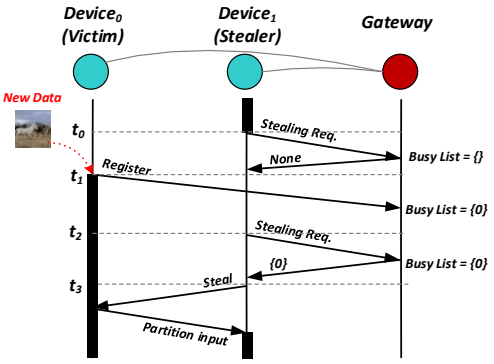
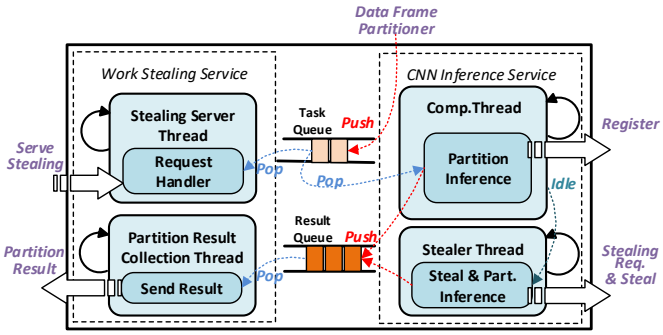
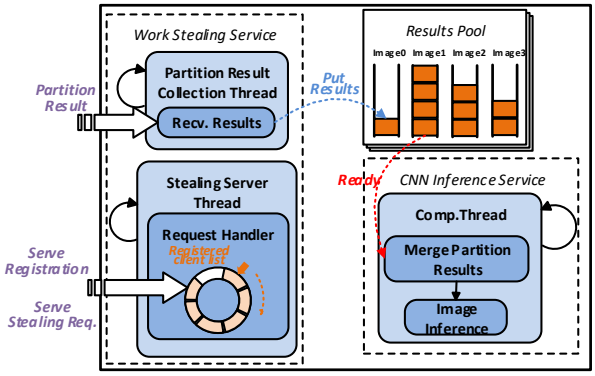


Fig. 6: Work stealing flow in DeepThings.



(a) Edge node runtime system



(b) Gateway runtime system

Fig. 7: Distributed work stealing runtime systems.

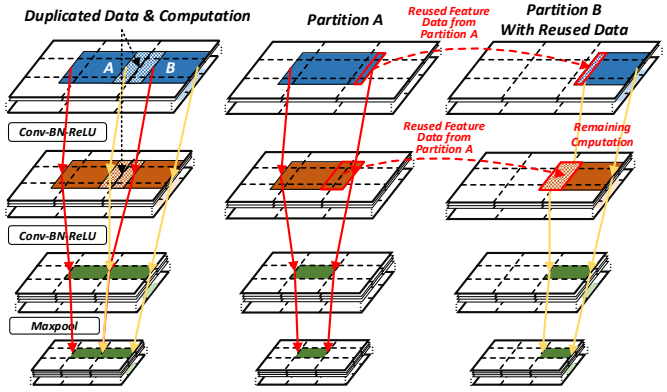


Fig. 8: Feature reuse between adjacent partitions.

a new data frame arrives in a source device, the corresponding device will register itself with the gateway ($Device_0$ at time t_1) and start processing work queue items. Once a device's work queue runs empty, it will notify and regularly poll the gateway for other busy devices to steal work from. An idle devices will first get a victim ID from the gateway before performing the actual task stealing. If the gateway responds that no other devices are busy, the idle device will sleep for a certain period before the next request. In Figure 6, $Device_1$ will sleep after the first request at time t_0. Then, at t_2, $Device_1$ polls the gateway again, which responds with $Device_0$'s ID as potential victim. As a result, $Device_1$ sends a request and steals a pending task from $Device_0$ at t_3.

To deploy such a dynamic execution framework, we design corresponding edge node and gateway runtime systems. Our proposed distributed work stealing system architecture for partitioned CNN inference is detailed in Figure 7 and in the following subsections.

1) Edge Node Service: Figure 7a shows the architecture of the distributed work stealing runtime system in edge node devices. It consists of two major functional components, a *Work Stealing Service* and a *CNN Inference Service*, which are encapsulated within the edge node runtime. Each service in turn consists of several threads and related software libraries to perform the task processing and distribution. As described above and in Figure 3, in an edge node, each incoming data frame will be first sliced into smaller tasks according to FTP parameters, and independent input data partitions are pushed into a task queue. In the *CNN inference service*, the *Computation Thread* will register itself with the gateway whenever the task queue is no longer empty. It will then repeatedly fetch work items from the task queue and process each partition. If there is no more incoming data and the task queue runs empty, the *Computation Thread* will report its status to the gateway and notify the *Stealer Thread* to start stealing and processing work items from other edge node devices. Note that in order to reduce the number of unsuccessful stealing attempts, the *Stealer Thread* will first request the IP address of an edge node with pending tasks from the gateway before performing the actual stealing. If no other edge node has pending input partitions according to the gateway, the *Stealer Thread* will sleep for a certain period before polling again. Finally, all output data from the *Computation Thread* and *Stealer Thread* will be pushed into a result queue.

To serve incoming stealing requests from other devices, a *Stealing Server Thread* is continuously running as part of the *Work Stealing Service* in each edge device. Once a steal request has been received, the *Request Handler* inside the stealing server will get a task from the local task queue and reply with the corresponding partition input data to the stealer. Additionally, a *Partition Result Collection Thread* will collect all the stolen and local partition results from the result queue and send them to the gateway.

2) Gateway Service: The gateway runtime architecture for distributed work stealing is shown in Figure 7b. In our setup, the gateway is responsible for the collection, merging and further processing of results from edge nodes. In addition, the gateway will also keep a record of edge devices with pending partitions and give out stealing hints to stealer devices. Similar to the edge node device, two components are included in gateway runtime systems: In the *Work Stealing Service*, a *Stealing Server Thread* receives registration requests from edge nodes that have just forked pending tasks. The IP addresses of these edge nodes are put into a ring buffer, where a pointer will rotate and pick up a registered IP address as the stealing victim upon arrival of each stealing request. We employ such a round-robin approach to improve work stealing fairness and workload balancing among edge nodes. The *Partition Result Collection Thread* collects partition data from all edge nodes within the network, where the collected partition results will be reordered and merged into a result pool according to the data source

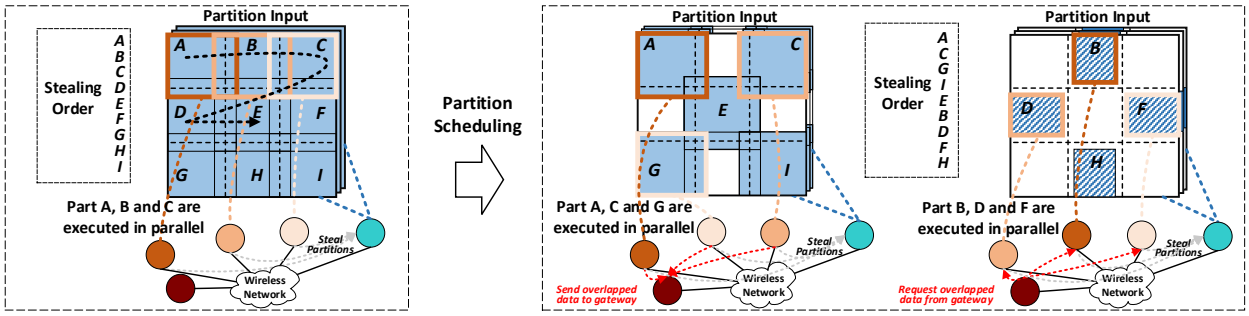


Fig. 9: Data reuse-aware work scheduling and distribution.

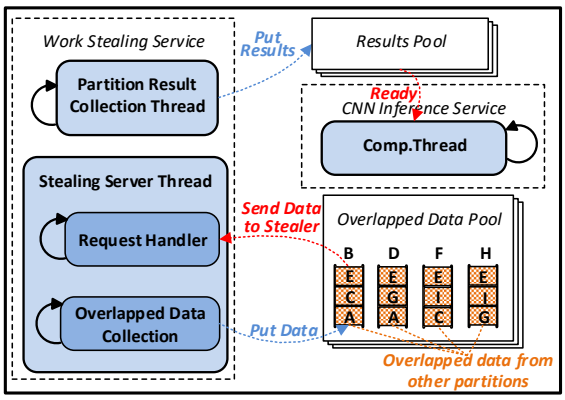


Fig. 10: Reuse data management.

ID and partition number. Note that the received results are the non-overlapping output data tiles of the last FTP layer mapped to edge devices, which can be simply concatenated together to retrieve the entire output tensor. Whenever a new result is collected, the total number of received data tiles for the corresponding data frame will be checked. When all the data outputs for a particular input data frame are collected, the *Computation Thread* of the *CNN Inference Service* in the gateway will fetch the corresponding data from the results pool, concatenate different partition outputs according to their region indexes to reconstruct the original output feature map of the early convolutional layers, and further feed the data into the remaining CNN layers to complete the inference.

C. Work Scheduling and Distribution

The Fused Tile Partitioning approach can largely reduce the communication overhead by localizing the high-dimensional intermediate feature map data within the edge node devices while still supporting partitions of selected size and memory footprint. However, the FTP scheme requires the overlapped input data between adjacent partitions to be locally replicated in different distributed devices and to locally reproduce the overlapped output feature regions. As such, despite of a reduction in memory footprint and inter-device communication, FTP may result in additional transmission overhead and duplicated computation effort. Furthermore, the amount of overlapped data and redundant computation is amplified by the fusing of consecutive convolutional and pooling layers. As a result, the processing and communication redundancy caused by the FTP method needs to carefully considered and pruned. To address this problem, we introduce an optimized work scheduling and distribution enhancement to our runtime system.

1) Overlapped Data Reuse: A concrete data overlapping example is shown in Figure 8. In this example, two adjacent partitions A and B share an overlapped data region at the inputs of their first two convolutional layers. Assuming A and B will be distributed onto two different devices, the overlapped region at the input layer (blue dotted) needs to be duplicated and each device needs to have an individual copy after task distribution. Then, during execution of each partition, both devices need to calculate any overlapped intermediate feature map vertically along the inference layers. For example, in Figure 8, the overlapped region (orange dotted) required by the input of the second convolutional layer needs to be computed by the first layer during the execution of both A and B.

In order to prune such data transmission and computation redundancy, overlapped intermediate feature map data produced by one partition can be cached and later reused by other adjacent partitions. Taking the example above, a data reuse scenario between partitions A and B is depicted in Figure 8. After the execution of partition A, the overlapped data is cached for reuse by partition B. During execution of B, the first layer only needs to receive and generate the non-overlapping regions of input and intermediate feature maps. As such, both the required input data transmission and computation workload of the first layer can be reduced.

Such a reuse scheme, however, creates dependencies among adjacent partitions and thus limits the inference parallelism. A straightforward solution would be to collect the required overlapped data after every layer and hand it out to dependent devices during partition execution time. However, in such an approach, the execution of adjacent partitions must be synchronized on a layer by layer basis, which introduces extra overhead and limits performance and scalability. In order to parallelize partitions while maximizing data reuse and minimizing synchronization overhead in a dynamically distributed work stealing environment, we propose a novel partition scheduling method.

2) Partition Scheduling: To enable data reuse between FTP partitions without the need for synchronization, we need to schedule parallel work items while considering their data reuse dependencies. Figure 9 on the left shows an example in which the data source node serves incoming work stealing requests with FTP tiles in a default left-to-right, top-to-bottom order. In

this example, adjacent partitions A, B and C are distributed into different edge devices to be executed in parallel, where overlapped intermediate results between A, B and B, C can not be reused without extra synchronization and thus wait time between partitions. If we instead serve requests and thus schedule and distribute work items in a different stealing order that minimizes dependencies and thus maximizes parallelism, independent partitions with minimal or no overlap should be distributed and executed first, such that their adjacent partitions can have a higher probability to be executed when their predecessors have finished and overlapped intermediate results are available. In the scheduled scenario in Figure 9 on the right, A, C and G are being distributed and executed in parallel with a largely reduced or no amount of overlapped data. Partitions B, D and F will only be processed after all their adjacent partitions ($A - E$) have been distributed or processed locally, where they will have a better chance to reuse the overlapped data.

We implement such an approach in DeepThings by inserting items into each device's task queue and thus serving both local and stealing requests in dependency order, starting from even grid rows and columns, and by least amount of overlap first. In order to efficiently share overlapped results between different edge node devices, the gateway collects and manages overlapping intermediate data among all involved partitions. During inference, an IoT edge node will receive the necessary input data from peer devices or the local task queue, request any overlapped intermediate or input data from the gateway, and compute all unique intermediate and output results locally. If the required overlapped data is not collected by the gateway yet, the partition will execute without reuse and locally compute the overlapped data. Note that if adjacent partitions from different groups are executed within the same device, intermediate results can be stored locally and the overhead of communication with the gateway is avoided.

In order to support such a distributed partition scheduling, the gateway device needs to be augmented with corresponding components to collect and hand out overlapped data for reuse. Figure 10 shows the enhanced gateway runtime system architecture. The *Stealing Server Thread* is now also responsible for the *Overlapped Data Collection*. The collected overlapped data will be stored and aligned according to the region information in a data pool. In each edge node, a fetcher is then added before partition execution to request the overlapped data from the pool in the gateway device.

3) Overlapped Region Computation: To deploy such a scheduling method, each FTP partition needs to record the overlapped regions with other partitions in each convolutional layer. To calculate these new parameters, we extend the FTP process from Algorithm 1 to compute both the set of regions overlapped with all adjacent partitions as well as the non-overlapping region local to each partition (i, j) in each layer l. Note that although the input data communication and intermediate feature map computation amount can both be reduced by data reuse, the exchange of overlapped intermediate data can introduce extra communication overhead. We will discuss and evaluate different design trade-offs in detail in Section V.

TABLE I: Comparison of different frameworks.

	DeepThings	MoDNN [16]
Partition Method	FTP	BODP-MR
Partition Dimensions	3x3∼5x5	1x1∼1x6
Distribution Method	Stealing/Sharing	Sharing
Edge Node Number	1∼6	

V. Experimental Results

We implemented the DeepThings framework in C/C++, with network communication modules in runtime systems realized using TCP/IP with socket APIs. DeepThings is available for download in open-source form at [22]. Without loss of generality, we evaluate DeepThings using the You Only Look Once, version 2 (YOLOv2) object detection framework [19] with its C-based Darknet neural network library as external CNN inference engine. YOLO and Darknet are widely used in the embedded domain because of their lightweight structure compared to other CNN-based object detection approaches [23], [24]. We use Darknet with NNPACK as backend acceleration kernel to provide state-of-the-art convolution performance as baseline for embedded devices without GPU support [21]. In our experiments, we deploy YOLOv2 on top of DeepThings on a set of IoT devices in a wireless local area network (WLAN). Our network setup consists of up to six edge and a single gateway device. We use Raspberry Pi 3 Model B (RPi3) as both edge and gateway platforms. Each RPi3 has a quad-core 1.2 GHz ARM Cortex-A53 processor with 1GB RAM. In order to provide a realistic evaluation of low-end IoT device performance, we limit each edge device to only a single RPi3 core. We apply our partitioning and distribution approach to the first 16 layers in YOLOv2 (12 convolutional layers and 4 maxpool layers), which contribute more than 49% of the computation and 86.6% of the memory footprint to the overall inference (see Figures 1 and 2). All results in the following experimental sections refer to these 16-layer stages.

In order to validate the benefits of our proposed framework, we implement and compare DeepThings against the MoDNN framework from [16]. MoDNN uses a Biased One-Dimensional Partition (BODP) approach with a MapReduce-style (MR) task model and a work sharing (WSH) distribution method in which processing tasks in each layer are first collected by a coordination device and then equally distributed to existing edge nodes. We compare the MoDNN approach against DeepThings's FTP partitioning using either a work sharing (WSH) or our proposed work stealing (WST) framework with (FTP-WST-S) or without (FTP-WST) data reuse and reuse-aware work scheduling. All frameworks are applied to YOLOv2, where different partitioning parameters are included in our evaluation to demonstrate corresponding design trade-offs. A summary of our experimental setup is shown in Table I.

We first evaluate the memory reduction under different partitioning methods. Then, communication overhead, inference latency and throughput are investigated for different combinations of partitioning methods and their possible work

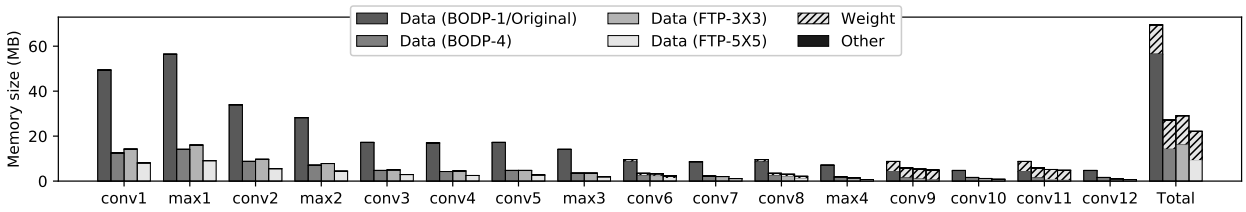
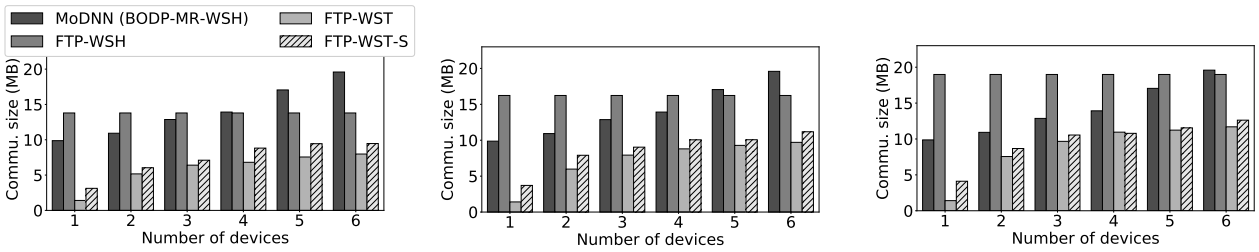


Fig. 11: Memory footprint per device for early convolutional layers.



(a) 3x3 Fused Tile Partition (b) 4x4 Fused Tile Partition (c) 5x5 Fused Tile Partition

Fig. 12: Communication data size for a single data frame.

distribution strategies. Results show that the major benefit of our distributed work stealing and work scheduling framework is the improvement of throughput with multiple dynamic data sources.

A. Memory Footprint

Per device memory footprints of each layer in different partitioning approaches are shown in Figure 11. In all cases, partitioning convolutional layers will only reduce the input/output data memory footprint, while the weights remain the same and need to be duplicated in each device. As such, the total memory requirement within one device is bounded by the sum of the largest layer data size and the total weight amount in partitioned CNN inference. Comparing with the memory requirement of the original CNN model, the reduction of total per device memory usage in BODP with 4 partitions is 61%, while the reductions are 58% and 68% for FTP with 3x3 and 5x5 grid dimensions, respectively. In general, total per device memory footprint shrinks proportionally with an increasing number of partitions in both BODP and FTP. However, since each FTP partition includes additional overlapped input data to guarantee independently distributable and executable partitions, an FTP with more tiles than BODP partitions can end up having a slightly larger (3%) memory footprint, as shown by the comparison between FTP 3x3 and BODP-4.

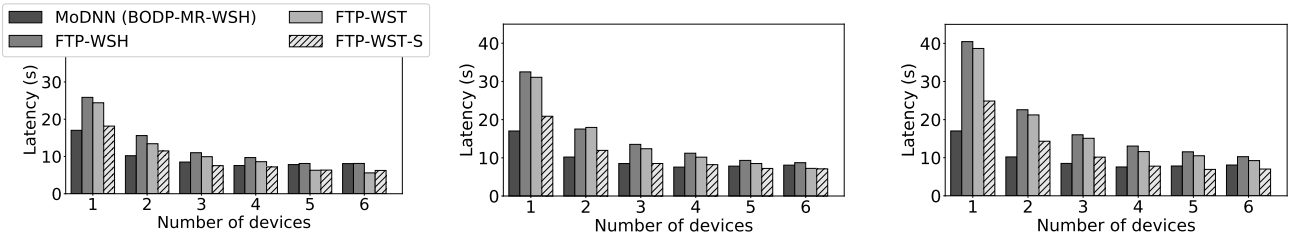
Comparing the memory footprint for each layer to the original CNN model, memory requirements can be reduced by an average of 69% and 79% for FTP-3x3 and FTP-5x5, while in BODP-1x4, the average reduction is 67%. Specifically, in the first 7 layers, FTP with a 3x3 grid dimension will have a 7% larger memory footprint than the 4-partition BODP. However, overlapped intermediate feature map regions are smaller in later layers, as shown in the comparison between FTP-3x3 and BODP-4 after the *max3* layer.

In general, both BODP and FTP provide a configurable range of memory budget options and significantly alleviate the memory consumption problem of the original CNN inference or any purely layer-by-layer partitioning. Comparing FTP with BODP, FTP requires extra memory because of the overlapped data. However, in general, partitioning parameters will also affect communication overhead, latency and throughput, which will be further discussed in the following sections. For a BODP- and MR-based execution model, an optimal approach is thereby to match the number of partitions to the number of available devices [16], which we will use for all further comparisons.

B. Communication Overhead

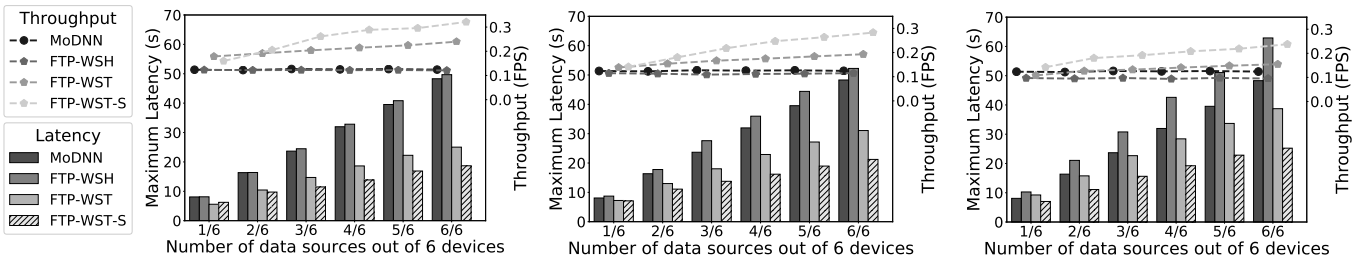
Communication overhead is dependent on both partitioning methods and distribution strategies. In FTP, only the input and output data of each partition needs to be communicated. As such, the total communication size is upper bounded by the sum of inputs and outputs from all partitions. The overlapped input data included by each FTP partition will result in extra communication, where the overhead will increase with finer partitioning granularity because of additional overlapped boundaries. By contrast, the BODP-MR based execution model relies on a centralized scheduler to exchange overlapped intermediate results after the execution of each layer. Such data exchanges between partitions in each layer will have a linearly increasing communication overhead when more collaborative devices are introduced. Note that intermediate feature maps can have a much larger depth compared to input ones. As a result, although BODP-MR can avoid additional transmission overhead of overlapped partition input, the communication size of BODP-MR will exceed the one of FTP with more than 5 devices because of these intermediate data exchanges.

We first compare the communication overhead of MoDNN and FTP using the same work sharing (WSH) distribution strategy. In a WSH strategy, the input data and, in the MoDNN case, the intermediate data after each layer, is transferred



(a) 3x3 Fused Tile Partition (b) 4x4 Fused Tile Partition (c) 5x5 Fused Tile Partition

Fig. 13: Inference latency for a single data frame.



(a) 3x3 Fused Tile Partition (b) 4x4 Fused Tile Partition (c) 5x5 Fused Tile Partition

Fig. 14: Inference latency and throughput with multiple data sources.

to the gateway, where it will be further partitioned and distributed into existing edge devices. As shown in Figure 12, for distributed inference of one data frame, MoDNN communicates an average of 14.0MB of data. When the device number increases from 1 to 6, the communication overhead of MoDNN increases linearly to up to 19.6MB. For FTP, the communication overheads are constantly 13.8MB, 16.2MB, and 18.9MB for FTP-3x3, FTP-4x4 and FTP-5x5, respectively. As the results show, the pre-partitioning method in FTP has a better scalability but may result in more overhead with a smaller number of devices.

An important aspect of FTP is that its independent task model enables other distribution strategies to be applied, which can have a large influence on the communication overhead. To investigate the effect of the distribution strategy on communication overhead, we apply both WSH and WST to FTP. Figure 12 shows that WST can reduce the communication overhead by an average of 52% while also having better scalability. In WST, tasks will be transferred directly to idle devices through stealing activities without centralized data collection and distribution. As a result, potential overlaps between communication and computation can be better explored. At the data source, more tasks will be consumed locally in parallel with task distribution, and tasks will only be distributed as necessary, where overall data movement is reduced and overlapped with local processing. Finally, our data reuse approach (FTP-WST-S) introduces additional communication overhead because of the transmission of intermediate feature map data. In FTP-WST-S, a data source will always notify the gateway of completion and will transfer the intermediate data to the gateway for possible data reuse by other stealing devices. As such, communication overhead of FTP-WST-S is larger than that of FTP-WST in all cases. However, such overhead can be amortized by the larger reduction of partition data at the input to the first layer with reuse (Figure 8). As a result, the overall communication overhead of FTP-WST-S decreases with finer partitioning granularity because of an increasing probability of data reuse. In a 3x3 grid (Figure 12a), FTP-WST-S introduces an average of 1.4MB more communication data than FTP-WST, while the overhead is reduced to 0.9MB in a 5x5 grid (Figure 12c).

C. Latency and Throughput

We finally compare overall latency and throughput of different frameworks. Single data frame latencies of MoDNN vs. DeepThings are shown in Figure 13. With the same partitioning method, WST generally results in shorter inference latency for one data frame because of the reduced communication overhead. As shown by the comparison between FTP-WSH and FTP-WST, an average of 10.3% reduction can be achieved by WST across different partitioning parameters. The single-frame inference scalability is, however, dominated by both partitioning methods and distribution approaches. In MoDNN, the computation time is inversely proportional while the communication overhead will grow linearly with the number of devices because of the centralized data distribution and processing synchronization. As a result, the latency will reach a minimum and increase after 4 devices. By contrast, FTP-WST and FTP-WST-S can adaptively explore the available communication bandwidth. As such, the performance improves until the bandwidth is maximally exploited, after which additional nodes will not be fully utilized and performance will saturate at a minimal value.

A finer grid granularity can expose more parallelism, but additional grid cells will also result in more overlapped boundaries and redundant communication/computation. As the

grid dimension increases from 3x3 to 5x5, latency increases by an average of 43% because of more overlapped regions. However, such overhead can be largely reduced by reuse-aware partition scheduling. For the same distribution strategy and number of devices, FTP-WST-S reduces latency by more than 27% on average compared to FTP-WST because of the reduced amount of duplicated computation. The efficiency of work scheduling also increases with partitioning dimension, where finer grid granularity will provide higher probability of data reuse. In Figure 13a, FTP-WST-S will reduce the latency by an average of 16% for a 3x3 grid, while an average of 33% reduction is achieved in a 5x5 grid (Figure 13c).

Overall, DeepThings has similar scalability but more redundant communication and computation as compared to MoDNN. With only one edge device, FTP-WST-S has an average of 25% larger latency. However, as more devices are added, the latency of FTP-WST-S will become similar to or lower than MoDNN. When processing a single data frame with 6 devices, FTP-WST-S has an average latency of 6.8s with a 3.5x speedup, while the latency and speedup in MoDNN are 8.1s and 2.1x.

In order to evaluate our approach under more realistic application scenarios, we further evaluate the maximum inference latency and overall throughput with multiple data sources and a fixed number of devices. For WSH-based approaches, when multiple data sources are present, the data source devices will first register themselves at the gateway. The gateway will then fetch and distribute the input data according to the registered device list in a round-robin manner. During execution, the WSH gateway will always treat edge nodes as collaborative workers and orchestrate the execution of data frames from different sources within the cluster in a serialized manner. As such, WSH can only explore the execution parallelism within a single data frame. In case of WST, task distribution will only be initiated in case of availability of idle resources, where a busy edge device is not forced to share resources with peer devices. As a result, the parallelism and overlap of communication and computation between different data frames are better explored.

Maximum single-frame latency and multi-frame throughput with multiple data sources are shown in Figure 14. In general, performance is mainly decided by the task distribution strategies. The processing latency in WSH-based approaches increases linearly when more data sources are involved. WST-based approaches, however, have a much smaller growth in maximum inference latency and are upper bounded by the single-device latency because of the adaptive stealing-based distribution. As the number of data sources increase from 1 to 6, the maximum latency will be increased by an average of 6.1x and 6.0x for FTP-WSH and MoDNN, respectively, while the increase is only 4.2x and 3.1x for FTP-WST and FTP-WST-S, respectively.

Figure 14 also shows the maximum throughput for different configurations. Because of the centralized overhead and serialized execution of WSH, the throughput can not be improved when parallelism from multiple data frames in different edge nodes exists. In MoDNN, the throughput is constantly around 0.12 frames per second (FPS) with changing number of data

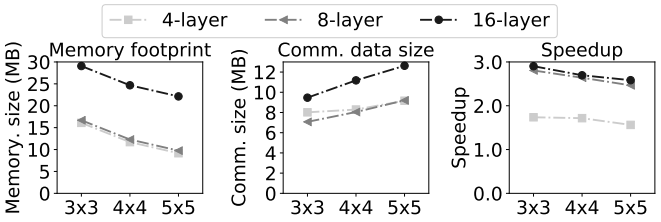


Fig. 15: Variations of design metrics under different FTP partitioning parameters with 6 IoT edge devices.

sources. Similarly, throughput for FTP-WSH is 0.11 FPS on average. By contrast, in the case of WST, throughput increases as more data sources are introduced because of the reduced communication amount and coordinator-free workload balancing scheme. As the number of data sources increase from 1 to 6, the average throughput will increase from 0.14 to 0.20 for FTP-WST. The data reuse-aware scheduling (FTP-WST-S) will have large benefits under finer partitioning granularity compared to the unoptimized approach (FTP-WST). As shown in Figure 14a, reuse-aware work scheduling provides 20% and 22% improvement for latency and throughput. In Figure 14c, with finer partitioning granularity, 32% and 45% latency reduction and throughput improvement are achieved. When the number of data sources increases, the efficiency of WST-S also manifest itself as fewer idle devices and increased data reused probability. In summary, when multiple data sources exist within a IoT cluster, our proposed FTP-WST-S approach will outperform MoDNN by an average of 41% latency reduction and 80% throughput improvement.

D. Sensitivity Analysis of FTP Parameters

We further investigate the sensitivity of various design metrics to FTP partitioning parameters in a 6-device network using a FTP-WST-S approach in which the first 4, 8 or 16 layers are executed on the edge devices, while other layers are offloaded to the gateway (Figure 15). In terms of memory footprint, more than 23% memory reduction can be achieved when increasing the partitioning granularity from 3x3 to 5x5 with 16 fused layers (see Section 5.1). In terms of communication overhead, as described in Section 5.2, due to more redundant input data overlap between partitions, the amount of transmitted data increases overall by 25% from 3x3 to 5x5 partitions in case of 16-layer fusion. Finally, the speedup for different number of fusing layers is calculated by comparing against the corresponding fastest single-device performance, which is 3x3 FTP in all cases. Coarser partitioning achieves better inference performance because of less overlapped and duplicated computation and communication.

Fewer fused layers will result in the same memory footprint for input/output data, which is bounded by early intensive convolutional layers, while the reduced memory consumption is mainly from a smaller amount of weight parameters, which are mainly concentrated in layers 8-16. Fewer fused layers can also have smaller overlapped partition regions and less partition input data. However, their output layers will have larger amount of output feature maps to be transferred to the

gateway. As a result, more fused layers do not necessarily result in larger communication overhead, as shown by the case of 3x3 partitions for 4 vs. 8 fused layers. Nevertheless, despite the generally higher communication overhead, due to a larger computation to communication ratio for each individual partition, deeper fusion can efficiently explore task-level parallelism by hiding more communication overhead in longer partition execution durations, and therefore has a larger speedup on inference latency.

In conclusion, trade-offs do exist for FTP partitioning parameters. Coarser partitioning granularity and deeper fusion can provide better inference speedup with larger memory footprint. The communication demand will also be larger with more partitions and fused layers. However, finer partitioning parameters inherently provide more parallelism with smaller task units. As such, when there is enough communication bandwidth, finer granularity can potentially provide better scalability or workload balancing in large and heterogeneous IoT edge clusters.

VI. Summary and Conclusions

In this paper, we presented DeepThings, a lightweight framework for adaptively distributed CNN inference among IoT gateway and resource-constrained edge node devices. We mainly focus on the distribution of early convolutional layers, which largely contribute to the overall inference latency and are the most memory-intensive stages. In order to create independently distributable processing task parallelism and minimize memory footprint from convolutional layers, a Fused Tile Partitioning (FTP) method is first proposed. We then develop a distributed work stealing runtime system for IoT clusters to adaptively distribute FTP partitions in dynamic application scenarios. Finally, the proposed framework is augmented by a partition scheduling and distribution policy to more efficiently reuse overlapped data between adjacent CNN partitions. Results show that FTP significantly reduces memory footprint compared to traditional layer-based partitioning, while the fusion strategy across multiple convolutional layers avoids communication overhead from intermediate feature map data. When combined with a distributed work stealing and reuse-aware work scheduling and distribution framework, scalable CNN inference performance is significantly improved compared to existing distributed inference methods under varying static or dynamic application scenarios. We have released DeepThings in open-source form at [22]. Future work will include further investigating dynamic, heterogeneous partitioning and locality-aware work scheduling schemes, optimizing the energy efficiency of distributed inference, as well as an evaluation on larger and heterogeneous IoT edge clusters.

Acknowledgements

We sincerely thank the reviewers for their comments to help improve the paper. This work was supported by NSF grant CNS-1421642.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems (FGCS)*, vol. 29, no. 7, pp. 1645 – 1660, 2013.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[3] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, and F. Kawsar, "An early resource characterization of deep learning on wearables, smartphones and Internet-of-Things devices," in *International Workshop on Internet of Things Towards Applications (IoT-A)*, 2015.

[4] Gartner, https://www.gartner.com/newsroom/id/3598917, 2017.

[5] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The cloud is not enough: Saving IoT from the cloud," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2015.

[6] S.-Y. Chien, W.-K. Chan, Y.-H. Tseng, C.-H. Lee, V. S. Somayazulu, and Y.-K. Chen, "Distributed computing in IoT: System-on-a-chip for smart cameras as an example," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2015.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal (IoT-J)*, vol. 3, no. 5, pp. 637–646, 2016.

[8] F. Samie, V. Tsoutsouras, S. Xydis, L. Bauer, D. Soudris, and J. Henkel, "Distributed QoS management for Internet of Things under resource constraints," in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2016.

[9] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[10] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[11] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2016.

[12] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. F. Abdelzaher, "Compressing deep neural network structures for sensing systems with a compressor-critic framework," *arXiv preprint arXiv:1706.01215*, 2017.

[13] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[14] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[15] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," *arXiv preprint arXiv:1707.01083*, 2017.

[16] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2017.

[17] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[18] N. Fernando, S. W. Loke, and W. Rahayu, "Computing with nearby mobile devices: a work sharing algorithm for mobile edge-clouds," *IEEE Transactions on Cloud Computing (TCC)*, vol. Pre-printed, 2017.

[19] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," *arXiv preprint arXiv:1612.08242*, 2016.

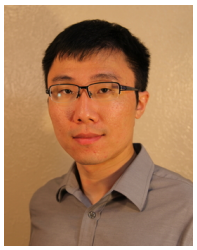
[20] J. Redmon, "Darknet: Open source neural networks in C," http://pjreddie.com/darknet/, 2013–2016.

[21] M. Dukhan, "NNPACK," https://github.com/Maratyszcza/NNPACK, 2018.

[22] Z. Zhao, "DeepThings," https://github.com/SLAM-Lab/DeepThings, 2018.

[23] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European Conference on Computer Vision (ECCV)*, 2016.



Zhuoran Zhao received the B.S. in Electrical Engineering from Zhejiang University, Zhejiang, China in 2012, and the M.S. degree in Electrical and Computer Engineering from The University of Texas at Austin, Austin, TX, USA, in 2015, where he is currently working toward his Ph.D. degree. His current research interests include source-level simulation for software/hardware codesign, performance modeling of distributed embedded systems, and acceleration of deep learning algorithms on IoT clusters.

Kamyar Mirzazad Barijough received the B.S. in Electrical Engineering from Sharif University of Technology, Tehran, Iran in 2015, and the M.S. degree in Electrical and Computer Engineering from The University of Texas at Austin, Austin, TX, USA, in 2017, where he is currently working toward the Ph.D. degree. His current research interests include design methodology and optimization methods for distributed and embedded systems.

Andreas Gerstlauer received his Ph.D. degree in Information and Computer Science from the University of California, Irvine (UCI) in 2004. He is currently an Associate Professor in the Electrical and Computer Engineering Department at The University of Texas at Austin. Prior to joining UT Austin, he was an Assistant Researcher with the Center for Embedded Computer Systems at UCI. He is co-author on 3 books and more than 100 refereed conference and journal publications. His research interests are in the area of embedded systems, cyber-physical systems (CPS) and the Internet of Things (IoT), specifically with a focus on electronic system-level design (ESL/SLD) methods, system modeling, system-level design languages and design methodologies, and embedded hardware and software synthesis.