CoCoI: Distributed Coded Inference System for Straggler Mitigation

Xing Liu, Mingzhi Chen, Ming Tang, and Chao Huang

***Abstract*—Convolutional neural networks (CNNs) are widely applied in real-time applications on resource-constrained devices. To accelerate CNN inference, prior works proposed to distribute the inference workload across multiple devices. However, they did not address stragglers and device failures in distributed inference, which is challenging due to the devices' time-varying and possibly unknown computation/communication capacities. To address this, we propose a distributed coded inference system, called CoCoI. It splits the convolutional layers of CNN, considering the data dependency of high-dimensional inputs and outputs, and then adapts coding schemes to generate task redundancy. With CoCoI, the inference results can be determined once a subset of devices complete their subtasks, improving robustness against stragglers and failures. To theoretically analyze the tradeoff between redundancy and subtask workload, we formulate an optimal splitting problem to minimize the expected inference latency. Despite its non-convexity, we determine an approximate strategy with minor errors, and prove that CoCoI outperforms uncoded benchmarks. For performance evaluation, we build a testbed with Raspberry Pi 4Bs. The experimental results show that the approximate strategy closely matches the optimal solution. When compared with uncoded benchmarks, CoCoI reduces inference latency by up to 34.2% in the presence of stragglers and device failures.**

***Index Terms*—Convolutional Neural Networks (CNNs), Distributed Inference, Coded Computation**

I. Introduction

Convolutional neural networks (CNNs) have been widely applied to various applications, such as image classification and object detection [1]. To meet the real-time and stable response requirements, inference tasks have been increasingly shifted from the cloud to edge devices in order to reduce the communication overhead [2]. However, CNN inference is typically computationally intensive, while the computation resources of a single edge device are usually too limited to serve inference requests promptly. This limitation is particularly evident when handling complex models. For example, as shown in our technical report [3], the inference of VGG16 [4] and ResNet18 [5] on a Raspberry Pi 4B takes 50.8s and 89.8s, respectively. Meanwhile, the computation of 2-dimensional (2D) convolutional layers are the primary bottleneck of CNN inference. In the aforementioned example, convolutional layers account for more than 99% of the total latency [3].

To address this, distributed edge inference [6]–[10] was proposed to overcome the computation bottleneck on a single edge device by distributing computation workload across multiple devices for collaborative inference. Some of them considered cloud-edge collaboration [6], [7], reducing inference workload on edge by assigning part of the model to the cloud. Some considered edge-edge collaboration [8]–[10]. Such edge-edge collaboration is feasible due to the local dependency of convolutional layers (i.e., each element of convolution output depends on only a local range of input corresponding to its kernel size). In other words, the convolution in CNN can be split for parallel execution at multiple devices, achieving acceleration. Specifically, MoDNN [8] splits convolution layers based on device computation capacity. DINA [9] considers the transmission overhead to optimize the multi-requests of convolutions. CoEdge [10] enables devices to request necessary resources from others to reduce synchronization overhead.

Unfortunately, distributed systems often suffer from stochastic straggling issues [11], [12], i.e., some devices spend much longer time in task completion than others, and device failure. Since distributed inference requires devices to regularly integrate intermediate results [9], straggling effect and device failure can slow down the integration and hence the overall inference process. Existing works have worked on such straggling/failure issues by considering straggler detection (e.g., [13], [14]), which can be slow and ineffective for unpredictable time-varying stragglers. Other works introduced replication-based task redundancy to handle stragglers (e.g., [15], [16]), while these approaches can lead to significant resource waste. To deal with these straggling/failure issues, some existing works have proposed coding schemes for general distributed computation. For example, Lee *et al.* [17] proposed to apply MDS codes to accelerate distributed matrix-vector multiplication. Mallick *et al.* [18] proposed fountain codes that are more robust to accelerate large-scale matrix multiplication. While previous works on coded convolution [19], [20] focused on limit types of convolutions (e.g., vector convolutions), they overlooked the complex high-dimensional input-output dependency when decomposing the convolutions.

To address straggling and device failure issues in distributed inference in general CNNs, we propose Coded Cooperative Inference (CoCoI), a distributed coded inference system that introduces task redundancy to improve system robustness against straggler and device failure. Through task splitting and encoding design, once a subset of devices have accomplished their assigned executions on CNN inference (conditioning on the number of devices which accomplished), their outputs are sufficient to be decoded to obtain the final inference result, thus efficiently eliminating straggler/failure issues. Note that designing a distributed coded computation method for general CNNs can be challenging due to the complex data dependency of high-dimensional inputs. Meanwhile, formulating and analyzing the associated optimal splitting problem is non-trivial

Xing Liu, Mingzhi Chen and Ming Tang are with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. Chao Huang is with the School of Computing, Montclair State University, Montclair, New Jersey, USA.

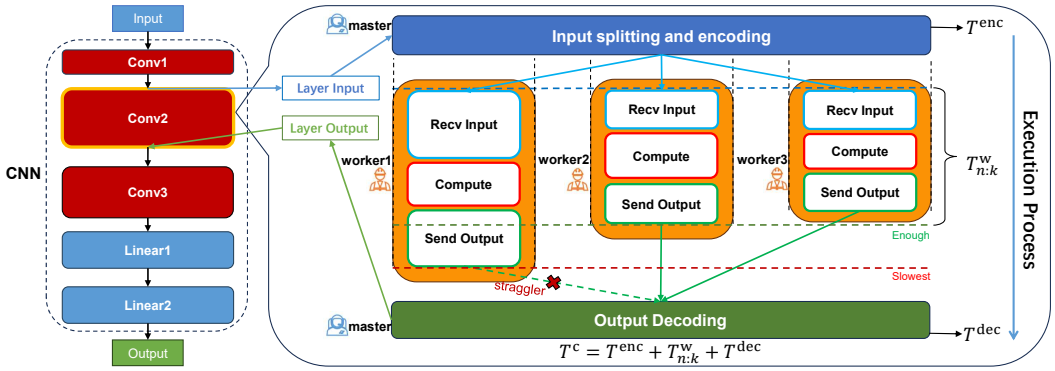


Fig. 1. An illustration of the CoCoI system with $n = 3$ and $k = 2$ and the workflow of our distributed coded inference approach applied to layer *Conv2*. The master first splits the input of layer *Conv2* and extends the input partitions to $n = 3$ encoded input partitions, each corresponding to an encoded subtask. Then, the workers receive their assigned inputs, perform execution, and send their outputs to the master. Once $k = 2$ outputs have been received, the master starts decoding to obtain the layer output. Here, $T^{\text{enc}}, T^{\text{w}}_{n:k}, T^{\text{dec}}$ denote the encoding, execution, and decoding latency (see Section III for details), respectively.

due to the randomness involved in straggling effect and its non-convexity.

We summarize our contributions as follows:

- **Distributed Coded Inference System**: We propose a distributed coded inference system for general CNN inference, called CoCoI. Through task splitting and encoding/decoding design, this system enables devices to cooperatively execute computational-intensive CNN layers and mitigate the straggling effect and device failure, while introducing only minor encoding and decoding overhead.
- **Optimal Splitting and Performance Analysis**: We take into account the straggling effect in CoCoI and formulate an optimal splitting problem. Although its objective function lacks an explicit form and is shown to be non-convex, we approximate the problem into a convex problem, with which we determine an approximate optimal solution. We further analyze how system parameters affect the approximate solution. Moreover, based on the approximate problem, we theoretically prove that CoCoI always achieves a lower inference latency than uncoded scheme in the presence of stragglers.
- **Performance Evaluation on Testbed**: We evaluate our CoCoI system on a real-world testbed with Raspberry-Pi 4Bs. Experimental results show that the performance gap between our approximate optimal splitting strategy and the optimal solution is less than 3.3%. When compared to uncoded and replication-based benchmarks, CoCoI system can reduce the inference latency by up to 34.2% in the presence of straggler or device failure.

The rest of this paper is organized as follows. Section II presents our system. Section III formulates the optimal splitting problem. In Section IV, we analyze the optimal strategy and its performance. Section VI shows our testbed and experimental results. Section VII concludes this work.

II. System Design

In this section, we first propose the CoCoI system. Then, we present our distributed coded inference approach, which is an important component of the CoCoI system.

A. CoCoI System Overview

In CoCoI system, there is a master device[1] and n worker devices. The master tracks the CNN inference process, distributes computation tasks to workers, and performs the related encoding and decoding operations. The workers are responsible for task execution. To complete a CNN inference task, CNN layers should be executed in a specific order. We refer to the execution of each layer as a *computation task*. Based on computational complexity (see Appendix A in our technical report [3]), we classify the layers into two types:

- Type-1 task (high-complexity): Most 2D convolutional layers, which is the bottleneck of CNN inference.
- Type-2 task (low-complexity): Linear, activation, and some light-weight 2D convolutional layers.

The CoCoI adopts a distributed coded inference approach (to be proposed in Section II-B) to execute the high-complexity type-1 tasks in order to eliminate the inference bottleneck. This process contains four phases: *input splitting phase* for input feature map partitioning, *encoding phase* to generate task redundancy, *execution phase* for distributed task execution, and *decoding phase* for recovering the task output. The master device executes those low-complexity type-2 tasks locally to mitigate the communication overhead resulting from intermediate computation result exchange.

An illustration of CoCoI system is shown in Fig. 1. The left part of Fig. 1 shows the layers of a CNN, where convolutional layers Conv1, Conv2, and Conv3 are executed in a distributed manner. The right part of Fig. 1 shows the workflow of the distributed coded inference approach for a type-1 task.

B. Distributed Coded Inference Approach

In the distributed coded inference approach, we partition the type-1 convolutional layers and perform data transform on the

[1]The master can be served by the device which initiates the inference task. Since this device has inference request, it will stay in the system during inference. Meanwhile, since the workload at the master is light, even if the master sometimes fails, re-execution can be completed within a short period of time (e.g., in hundreds of milliseconds). Thus, we do not introduce schemes to address the device failure of the master. This setting is consistent with existing works on coded computation [20], [21].

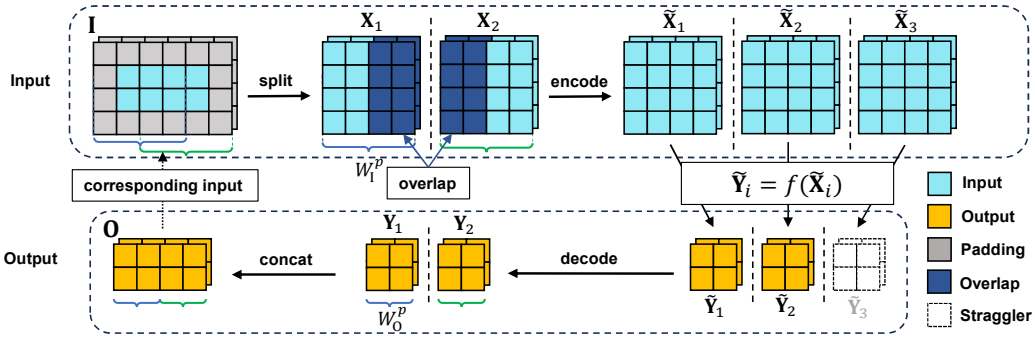


Fig. 2. An illustration of input splitting, encoding, and output decoding for distributed coded convolution with a 3×3 kernel and stride $= 1$. Here, an (n, k)-MDS code ($n = 3$ and $k = 2$) is used to encode the input partitions and to decode the computation task result based on the encoded outputs.

input feature map to ensure distributed inference. Meanwhile, we propose to apply MDS code to introduce task redundancy in order to mitigate straggler/failure issues. Although other codes (e.g., fountain codes) can also be modified to support the encoding and decoding design, they are not suitable to CoCoI due to their overhead and reduced level of successful decoding rate (see Section VI for experimental results).

Specifically, a 2D convolutional layer in CNNs mainly has five configuration parameters, including *in_channels*, *out_channels* (denoted by $C_{\mathrm{I}}, C_{\mathrm{O}}$), *kernel_size*, *stride*, and *paddings*. We use K_W and S_W to represent the values of *kernel_size* and *stride* on the width dimension, respectively. Meanwhile, we set the *kernel_size* on both width and height dimensions to be the same, i.e., K_W. The original input feature map is firstly padded with the *paddings* (the gray area surrounding the input feature map in Fig. 2). Then, the 4D padded input (denoted by $\mathbf{I}$) has a shape of $(B, C_{\mathrm{I}}, H_{\mathrm{I}}, W_{\mathrm{I}})$, where B is the batch size and $H_{\mathrm{I}}, W_{\mathrm{I}}$ denote the height and width of $\mathbf{I}$, respectively. Similarly, let $(B, C_{\mathrm{O}}, H_{\mathrm{O}}, W_{\mathrm{O}})$ denote the shape of output $\mathbf{O}$. Here we set $B = 1$ considering sparse inference requests on edge. Without loss of generality, we assume $W_{\mathrm{I}} \geq H_{\mathrm{I}}$. In this case, we split the feature map in the width dimension (rather than the height dimension) as it can reduce the overlap of input partitions. Based on the principle of 2D convolution, W_{O} can be determined as follows (and H_{O} similarly):

$$W_{\mathrm{O}} = \lfloor (W_{\mathrm{I}} - K_W + 1)/S_W + 1 \rfloor . \tag{1}$$

Key parameters are summarized in Table II in technical report [3].

1) Input Splitting Phase: The master splits the input feature map into k pieces, corresponding to the input of k *source subtasks*. The input partition is determined by computing the target output $\mathbf{O}$ of each source task, and the respective partitions is obtained based on the dependency between input and output. To fulfill the requirement of MDS codes, $\mathbf{O}$ is split into k pieces $\mathbf{Y}_1, \ldots, \mathbf{Y}_k$ on the width dimension in equal, each with a shape of $(B, C_{\mathrm{O}}, H_{\mathrm{O}}, W_{\mathrm{O}}^p(k))$ and an output range of $(a_{\mathrm{O}i}, b_{\mathrm{O}i})$ for $i \in [k]$ (i.e., a segment of $\mathbf{O}$ from width index $a_{\mathrm{O}i}$ to $b_{\mathrm{O}i} - 1$), where $W_{\mathrm{O}}^p(k) = b_{\mathrm{O}i} - a_{\mathrm{O}i} = \lfloor W_{\mathrm{O}}/k \rfloor$[2]. Consider a piece with an output range of $(a_{\mathrm{O}}, b_{\mathrm{O}})$, the width $W_{\mathrm{I}}^p(k)$ and range $(a_{\mathrm{I}}, b_{\mathrm{I}})$ of its input partition is as follows:

$$\begin{aligned} W_{\mathrm{I}}^p(k) &= K_W + (W_{\mathrm{O}}^p(k) - 1) \times S_W, \\ a_{\mathrm{I}} = a_{\mathrm{O}} \times S_W, b_{\mathrm{I}} &= (b_{\mathrm{O}} - 1) \times S_W + K_W. \end{aligned} \tag{2}$$

The master splits $\mathbf{I}$ based on the input ranges derived above to obtain input partitions $\mathbf{X}_1, \ldots, \mathbf{X}_k$ to ensure correct input-output correspondence for each source subtask, i.e., $\mathbf{Y}_i = f(\mathbf{X}_i), i \in [k]$. Note that the adjacent input partitions may overlap (see Fig. 2), therefore $k \times W_{\mathrm{I}}^p(k) \geq W_{\mathrm{I}}$.

2) Encoding Phase: The master generates the inputs for n *encoded subtasks* based on the input of k *source subtasks* to introduce redundancy. Specifically, $\mathbf{X}_1, \ldots, \mathbf{X}_k$ are first flatten[3] to vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ (i.e., $\boldsymbol{x}_i = \mathrm{flatten}(\mathbf{X}_i)$) and concatenated to an input matrix of shape $(k, B \times C_{\mathrm{I}} \times H_{\mathrm{I}} \times W_{\mathrm{I}}^p(k))$. Then, an $n \times k$ generation matrix $\boldsymbol{G}$ is applied on the input matrix to generate the following encoded input matrix, composed of n encoded input vector $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n$:

$$\boldsymbol{G} \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_k \end{bmatrix} = \begin{bmatrix} g_1^{k-1} & g_1^{k-2} & \cdots & g_1^0 \\ \vdots & \vdots & \ddots & \vdots \\ g_n^{k-1} & g_n^{k-2} & \cdots & g_n^0 \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_k \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{x}}_1 \\ \vdots \\ \tilde{\boldsymbol{x}}_n \end{bmatrix}. \tag{3}$$

Here, $\boldsymbol{G}$ should satisfy that every k-row submatrix of $\boldsymbol{G}$ is invertible to ensure successfully decoding, and Vandermonder matrix is usually used as the generation matrix [17]. Then $\tilde{\boldsymbol{x}}_1, \ldots, \tilde{\boldsymbol{x}}_n$ are restored to the original shape as $\tilde{\mathbf{X}}_1, \ldots, \tilde{\mathbf{X}}_n$, serving as the input of n *encoded subtasks* for execution.

3) Execution Phase: The master sends the n encoded input partitions $\tilde{\mathbf{X}}_i$ to n workers. Worker $i \in [n]$ receives the encoded input and feeds $\tilde{\mathbf{X}}_i$ into the on-processing layer for *encoded subtask* execution with the preloaded weights and obtains encoded output $\tilde{\mathbf{Y}}_i = f(\tilde{\mathbf{X}}_i)$ of shape $(B, C_{\mathrm{O}}, H_{\mathrm{O}}, W_{\mathrm{O}}^p(k))$. Afterwards, $\tilde{\mathbf{Y}}_i$ is sent back to master.

[2]For the cases where W_{O} is not divisible by k, the master can keep the subtask associated with the remaining part of the output feature map with width $\mathrm{mod}(W_{\mathrm{O}}, k)$. This subtask is relatively small and has no data transmission latency, thus will not be the bottleneck of the inference process.

[3]An operation to transform a high-dimensional array data to an one-dimensional vector, detailed description can be found in [22].

4) Decoding Phase: When the master receives the encoded outputs from any k workers, it can decode and obtain the final output. Let $\mathcal{S} = \{s_1, \ldots, s_k\} \subset [n]$ denotes the index set of the k fastest workers. Then the k encoded outputs $\tilde{\mathbf{Y}}_{s_1}, \ldots, \tilde{\mathbf{Y}}_{s_k}$ are flatten to vectors $\tilde{\boldsymbol{y}}_{s_1}, \ldots, \tilde{\boldsymbol{y}}_{s_k}$ and concatenated to an encoded output matrix of shape $(k, B \times C_{\mathrm{O}} \times H_{\mathrm{O}} \times W^p_{\mathrm{O}}(k))$. A $k \times k$ sub-matrix $\boldsymbol{G}_{\mathcal{S}}$ is obtained by sequentially selecting the rows of the $\boldsymbol{G}$ based on $\mathcal{S}$. The inverse of $\boldsymbol{G}_{\mathcal{S}}$, denoted by $\boldsymbol{G}_{\mathcal{S}}^{-1}$, is applied on the encoded output matrix to generate the decoded output matrix, composed of k output partition vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k$. That is,

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \vdots \\ \boldsymbol{y}_k \end{bmatrix} = \boldsymbol{G}_{\mathcal{S}}^{-1} \begin{bmatrix} \tilde{\boldsymbol{y}}_{s_1} \\ \vdots \\ \tilde{\boldsymbol{y}}_{s_k} \end{bmatrix}, \text{where } \boldsymbol{G}_{\mathcal{S}} = \begin{bmatrix} g_{s_1}^{k-1} & g_{s_1}^{k-2} & \cdots & g_{s_1}^{0} \\ \vdots & \vdots & \ddots & \vdots \\ g_{s_k}^{k-1} & g_{s_k}^{k-2} & \cdots & g_{s_k}^{0} \end{bmatrix}. \tag{4}$$

Vectors $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_k$ are then restored to the original shape and concatenated to obtain $\mathbf{O} = \text{concat}(\mathbf{Y}_1, \ldots, \mathbf{Y}_k)$. With the MDS mechanism, $\mathbf{O}$ can be perfectly restored and remains consistent with that of the original inference, thus keeping the inference quality unchanged.

III. Optimal Splitting Problem

In CoCoI system, given a total of n workers, the choice of k (i.e., how the task is split) introduces a tradeoff between the computation workload and redundancy. Specifically, a larger k leads to a reduced size of each input partition and thus a reduced workload of each subtask. On the other hand, a smaller k leads to an increased degree of redundancy among workers (i.e., a larger $r \triangleq n-k$), which reduces the execution latency considering the k fastest workers, and slightly reduces the encoding and decoding latency. To investigate this tradeoff, we formulate an optimal splitting problem to minimize the inference latency. When compared with existing works on distributed inference (e.g., [9], [10]), we characterize the straggling/failure effect by modeling the latency of phases as random variables. When compared with existing modeling of stragglers in coded convolution (e.g., [19], [20]), we take into account the modeling of encoding and decoding latency and CNN task-specific execution and transmission latency.

In the following, we first present the overall latency model (of a computation task). Then, we discuss the detailed latency modeling in each phase. Finally, we formulate the problem.

A. Overall Latency Model

Given the choice of splitting strategy k, the overall latency of a computation task consists of three terms: (i) the latency of input splitting and encoding $T^{\text{enc}}(k)$, (ii) the latency of input/output transmission and execution $T^{\text{w}}_{n:k}(k)$, and (iii) the latency of decoding $T^{\text{dec}}(k)$. That is:

$$T^{\text{c}}(k) = T^{\text{enc}}(k) + T^{\text{w}}_{n:k}(k) + T^{\text{dec}}(k), \tag{5}$$

where $T^{\text{enc}}(k)$, $T^{\text{w}}_{n:k}(k)$, and $T^{\text{dec}}(k)$ are random variables.

We now present how random variable $T^{\text{w}}_{n:k}(k)$ is computed. Specifically, suppose worker $i \in [n]$ receives the assigned input with latency $T_i^{\text{rec}}(k)$. It then executes subtask with latency $T_i^{\text{cmp}}(k)$ and sends the output to the master with latency $T_i^{\text{sen}}(k)$. Note that $T_i^{\text{rec}}(k)$, $T_i^{\text{cmp}}(k)$, and $T_i^{\text{sen}}(k)$ are random variables. Thus, the latency that worker i takes to finish the transmission and execution operations mentioned in (ii), denoted by $T_i^{\text{w}}(k)$, can be expressed as follows:

$$T_i^{\text{w}}(k) = T_i^{\text{rec}}(k) + T_i^{\text{cmp}}(k) + T_i^{\text{sen}}(k), \ i \in [n]. \tag{6}$$

Recall that master can decode the final result once receiving k encoded outputs from workers. Thus, the latency in (ii) is equal to the latency $T_i^{\text{w}}(k)$ of the k-th fastest workers, i.e., $T^{\text{w}}_{n:k}(k)$. Note that $T^{\text{w}}_{n:k}(k)$ denotes the k-th order statistics of n random variables in set $\{T_1^{\text{w}}(k), T_2^{\text{w}}(k), ..., T_n^{\text{w}}(k)\}$, i.e., the k-th smallest value among these n random variables.

Suppose these random variables $T_i^{\text{rec}}(k)$, $T_i^{\text{cmp}}(k)$, $T_i^{\text{sen}}(k)$, $T^{\text{enc}}(k)$, and $T^{\text{dec}}(k)$ follow a shift-exponential distribution. This is reasonable because an exponential distribution (with or without a shift) usually indicates the time interval between independent events and is widely adopted in modeling the latency of computation tasks in coded computation [23]. Meanwhile, experiments on our testbed (see Appendix B in technical report [3]) and other research [24], [25] demonstrate that the latency of computation or wireless transmission can be well-fitted by the shift-exponential distribution.

Definition 1 (Shift-Exponential Distribution)**.** *Consider a random variable T follows a shift-exponential distribution defined by a straggler parameter μ, a shift coefficient θ, and a scaling parameter N. The CDF of the random variable T is given by*

$$F_{\text{SE}}(t; \mu, \theta, N) \triangleq 1 - e^{-\frac{\mu}{N}(t - N\theta)}, t \geq N\theta. \tag{7}$$

In this distributed inference system, a smaller straggler parameter μ implies a stronger straggling effect. The shift coefficient θ indicates the minimum completion time of the corresponding operation. The scaling parameter N describes the scale of the specific operation, e.g., the size of transmission or computation. If a device takes a very long time (e.g., longer than a pre-defined timeout threshold) on a certain operation, we regard the operation as failed. Thus, the scenario of device failure can be captured by shift-exponential distribution.

B. Detailed Latency Model

Let μ^{m}, μ^{cmp}, μ^{rec}, and μ^{sen} denote the straggling parameters of the computation at master, computation, input receiving, and output sending at workers, respectively. Let θ^{m}, θ^{cmp}, θ^{rec}, and θ^{sen} denote the associated minimal completion time. In practical systems, these straggling and shift coefficients can be estimated by prior test and fitting. In the following, we model the latency of each phase by specifying the scaling parameter N in Definition 1, considering the specific partition and coding schemes in CoCoI system.

1) Latency in Input Splitting and Encoding: The master applies the $n \times k$ generation matrix $\boldsymbol{G}$ on the input matrix of shape $(k, B \times C_{\mathrm{I}} \times H_{\mathrm{I}} \times W^p_{\mathrm{I}}(k))$ to generate encoded inputs. Thus, the number of floating point operations (FLOPs) in this phase, denoted by $N^{\text{enc}}(k)$, can be derived by:

$$N^{\text{enc}}(k) = 2k \times n \times B \times C_{\mathrm{I}} \times H_{\mathrm{I}} \times W^p_{\mathrm{I}}(k). \tag{8}$$

The encoding latency $T^{\text{enc}}(k)$ follows $F_{\text{SE}}\left(t;\mu^{\text{m}},\theta^{\text{m}},N^{\text{enc}}(k)\right)$.

2) Latency in Input/Output Transmission and Execution: To execute the subtask, each worker performs a sliding dot-product using a kernel of shape $(B, C_{\text{I}}, K_W, K_W)$ across the received input feature map and generates an output feature map with shape $(B, C_{\text{O}}, H_{\text{O}}, W_{\text{O}}^p(k))$. Then, the FLOPs $N^{\text{cmp}}(k)$ of each subtask on worker is given by:

$$N^{\text{cmp}}(k) = B \times C_{\text{O}} \times H_{\text{O}} \times W_{\text{O}}^p(k) \times 2 \times C_{\text{I}} \times K_W^2. \quad (9)$$

Thus, the computation latency T_i^{cmp} on worker $i \in [n]$ follows a shift-exponential distribution $F_{\text{SE}}(t;\mu^{\text{cmp}},\theta^{\text{cmp}},N^{\text{cmp}}(k))$.

The transmission size for the input distributing $N^{\text{rec}}(k)$ by the master and output forwarding $N^{\text{sen}}(k)$ by each worker can be formulated by the size of the input and output partitions:

$$N^{\text{rec}}(k) = 4 \times B \times C_{\text{I}} \times H_{\text{I}} \times W_{\text{I}}^p(k), \quad (10)$$

$$N^{\text{sen}}(k) = 4 \times B \times C_{\text{O}} \times H_{\text{O}} \times W_{\text{O}}^p(k). \quad (11)$$

The workers share the same $N^{\text{rec}}(k)$ and $N^{\text{sen}}(k)$ due to equal-size subtask partition. The latency $T_i^{\text{rec}}(k)$ and $T_i^{\text{sen}}(k)$ of the input/output sending of worker $i \in [n]$ respectively follow $F_{\text{SE}}(t;\mu^{\text{rec}},\theta^{\text{rec}},N^{\text{rec}}(k))$ and $F_{\text{SE}}(t;\mu^{\text{sen}},\theta^{\text{sen}},N^{\text{sen}}(k))$.

3) Latency in Decoding: The master decodes the execution output matrix with shape $(k, B \times C_{\text{O}} \times H_{\text{O}} \times W_{\text{O}}^p(k))$ with the $k \times k$ inverse matrix $\boldsymbol{G}_{\mathcal{S}}^{-1}$. Similarly, the FLOPs of decoding phase $N^{\text{dec}}(k)$ can be derived by:

$$N^{\text{dec}}(k) = 2k^2 \times B \times C_{\text{O}} \times H_{\text{O}} \times W_{\text{O}}^p(k). \quad (12)$$

The decoding latency $T^{\text{dec}}(k)$ follows $F_{\text{SE}}(t;\mu^{\text{m}},\theta^{\text{m}},N^{\text{dec}}(k))$.

C. Optimal Splitting Problem Formulation

Our objective is to find the optimal choice of k^* in order to minimize the expectation of overall latency $\mathbb{E}[T^{\text{c}}(k)]$:

$$k^* = \arg\min_{k\in\{1,2,\cdots,n\}} \quad \mathbb{E}[T^{\text{c}}(k)]. \quad (13)$$

Remark 1 (Challenges of Solving Problem (13))**.** *It is difficult to represent the objective function as an explicit expression due to the involvement of order statistics. Meanwhile, we empirically validate that the objective function $\mathbb{E}[T^{\text{c}}(k)]$ is non-convex. Both reasons make solving problem* (13) *challenging.*

IV. Optimal Splitting Analysis

In this section, we aim to approximately solve problem (13), with which we investigate (i) how to determine an approximate optimal strategy k°, (ii) how system parameters (e.g., $\mu^{\text{cmp}}, \theta^{\text{cmp}}, \mu^{\text{m}}$) affect the optimal strategy, and (iii) how much is the latency reduction from the CoCoI system.

A. Approximate Optimal Analysis

Since the encoding $T^{\text{enc}}(k)$ and decoding $T^{\text{dec}}(k)$ latency at the master and the latency $T_{n:k}^{\text{w}}(k)$ at workers are independent, the objective function in (13) can be represented as

$$\mathbb{E}[T^{\text{c}}(k)] = \mathbb{E}[T^{\text{enc}}(k) + T^{\text{dec}}(k)] + \mathbb{E}[T_{n:k}^{\text{w}}(k)], \quad (14)$$

where the expected sum of the encoding and decoding latency $\mathbb{E}[T^{\text{enc}}(k) + T^{\text{dec}}(k)] = (N^{\text{enc}}(k) + N^{\text{dec}}(k)) \times (\frac{1}{\mu^{\text{m}}} + \theta^{\text{m}})$. Thus, the main difficulty of solving problem (13) comes from determining the expectation of execution latency $\mathbb{E}[T_{n:k}^{\text{w}}(k)]$. This is because the execution latency $T_{n:k}^{\text{w}}(k)$ is the k-th order statistics of $T_i^{\text{w}}(k)$, where $T_i^{\text{w}}(k)$ is the sum of three random variables $T_i^{\text{rec}}(k)$, $T_i^{\text{cmp}}(k)$, and $T_i^{\text{sen}}(k)$. Calculating the expectation of order statistics over a sum of multiple random variables is still an open problem [26].

Let $T_{n:k}^{\text{rec}}(k)$, $T_{n:k}^{\text{cmp}}(k)$, and $T_{n:k}^{\text{sen}}(k)$ denote the associated k-th order statistics (i.e., the k smallest values) among workers $i \in [n]$. We approximate $\mathbb{E}[T_{n:k}^{\text{w}}(k)]$ as follows:

$$\mathbb{E}[T_{n:k}^{\text{w}}(k)] \approx \mathbb{E}[T_{n:k}^{\text{rec}}(k)] + \mathbb{E}[T_{n:k}^{\text{cmp}}(k)] + \mathbb{E}[T_{n:k}^{\text{sen}}(k)]. \quad (15)$$

Based on the expectation of exponential order statistics [26] and relaxing the floor function in $W_{\text{O}}^p(k) = \lfloor W_{\text{O}}/k \rfloor$, an approximate expected overall latency is given by:

$$\mathbb{E}[T^{\text{c}}(k)] \approx L(k) \triangleq (N^{\text{enc}}(k) + N^{\text{dec}}(k)) \left(\frac{1}{\mu^{\text{m}}} + \theta^{\text{m}}\right) + \theta^{\text{sum}} + \mu^{\text{sum}} \ln \frac{n}{n-k}, \quad (16)$$

where $\mu^{\text{sum}} \triangleq N^{\text{rec}}(k)/\mu^{\text{rec}} + N^{\text{cmp}}(k)/\mu^{\text{cmp}} + N^{\text{sen}}(k)/\mu^{\text{sen}}$ and $\theta^{\text{sum}} \triangleq N^{\text{rec}}(k)\theta^{\text{rec}} + N^{\text{cmp}}(k)\theta^{\text{cmp}} + N^{\text{sen}}(k)\theta^{\text{sen}}$.

An approximate optimal solution k° to problem (13) can be determined by minimizing $L(k)$:

$$k^\circ = \arg\min_{k\in\{1,2,\ldots,n\}} \quad L(k). \quad (17)$$

To understand the approximate optimal solution k°, we temporarily relax integer k to be a real value and eliminate the case of $n = k$ (i.e., no redundancy), i.e., $k \in [1, n)$.

Lemma 1. *When $n \geq 3$, the relaxed problem* (17) *is a convex programming problem under $k \in [1, n)$.*

The proof of Lemma 1 is detailed in Appendix C of technical report [3]. Thus, when compared to $\mathbb{E}[T^{\text{c}}(k)]$, function $L(k)$ can be represented in an explicit form, and it is a convex function given the integer relaxation considered in Lemma 1. To determine k°, we can first solve $k' = \arg\min_{k\in[1,n)} L(k)$ using conventional solvers (e.g., CVX). Then, we can obtain $k^\circ \in \{\lfloor k' \rfloor, \lceil k' \rceil\}$ by comparing $L(\lfloor k' \rfloor)$ and $L(\lceil k' \rceil)$.

In Section VI and Appendix D in [3], we empirically evaluate the gap between the approximate optimal solution k° and the optimal solution k^* and their performance gap. The experimental results show that problem (17) has a good approximation for (i) the objective function and (ii) the optimal k under a wide range of values of μ's and θ's. In most cases, the difference of k^* and k° does not exceed 1.

B. Impact of System Parameters

Let $\hat{k}^\circ$ denote the optimal solution to problem (17) under the relaxation of $k \in [1, n)$. Based on Lemma 1, by checking the first-order condition of $L(k)$, we analyze the impact of system parameters on $\hat{k}^\circ$ (see Appendix E for proof). The analytical results on $\hat{k}^\circ$ are consistent with the empirical optimal solution k^* to problem (13) (see Appendix D).

Proposition 1 (Impact of Straggler and Shift Coefficients)**.** *The approximate optimal $\hat{k}^{\circ}$ increases if (i) any straggler coefficient μ^{cmp}, μ^{m}, μ^{rec}, μ^{sen} increases, (ii) any shift coefficient θ^{cmp}, θ^{rec}, θ^{sen} increases, or (iii) shift coefficient θ^{m} decreases.*

As shown in Proposition 1, if any straggler coefficient μ decreases (i.e., a more severe straggling effect), then the computation task should be partitioned into fewer pieces (i.e., smaller k) to introduce more redundancy (i.e., larger $r \triangleq n-k$). If any shift coefficient θ of workers (i.e., $\theta^{\text{cmp}}, \theta^{\text{rec}}, \theta^{\text{sen}}$) increases (i.e., the minimum completion time increases), then the workload of subtasks become larger, so the computation task should be partitioned into smaller pieces (i.e., a larger k) to reduce the workload. If $\frac{1}{\mu^{\text{m}}}+\theta^{\text{m}}$ is larger, the master has a less powerful processing capacity, so k should be decreased to reduce the encoding and decoding latency.

C. Theoretical Performance Improvement

For analytical simplicity, we use the approximate form $L(k)$ in the following analysis. The expected latency under uncoded approach can be regarded as a special case of that under coded approach, i.e., the master can obtain the execution result of each layer when all n workers send back their output. Thus, it can be computed based on (15) and [26] (see Appendix F in [3]). To provide a clear insight on comparison, we consider a setting where $W_{\text{O}} \gg k$, which matches the case in practical systems. We omit the encoding and decoding latency in coded approach, as those latency are minor (see Section VI). With these approximation, let $\mathbb{E}[T_m^{\text{c}}(n,k)]$ and $\mathbb{E}[T_m^{\text{u}}(n)]$ denote the expected latency of coded and uncoded method, respectively. The proof for the following Propositions 2 and 3 are given in Appendix F of our technical report [3].

First, we consider the straggler scenario without worker failure. Let $R \triangleq (4I_W\theta^{\text{rec}}+4O\theta^{\text{sen}}+N_c\theta^{\text{cmp}})/(\frac{4I_W}{\mu^{\text{rec}}}+\frac{4O}{\mu^{\text{sen}}}+\frac{N_t^{\text{cmp}}}{\mu^{\text{cmp}}})$, where $I_W \triangleq C_{\text{I}}H_{\text{I}}W_{\text{O}}S$, $O \triangleq C_{\text{O}}H_{\text{O}}W_{\text{O}}$, and $N_t^{\text{cmp}} \triangleq 2C_{\text{O}}H_{\text{O}}C_iK^2W_{\text{O}}$. Intuitively, a smaller R implies a higher degree of straggling effect.

Proposition 2 (Straggler Scenario)**.** *When $R \leq 1$ and $n \geq 10$, there exists a $k_{sub}^* \in (1,n)$ such that $\Delta \triangleq \mathbb{E}[T_m^u(n)] - \mathbb{E}[T_m^c(n,k_{sub}^*)] > 0$.*

Proposition 2 implies that when the degree of straggling exceeds a threshold (i.e., $R \leq 1$), our coded inference approach always achieves a lower latency than uncoded method [9]. For example, when $n=20$ and $R=1$, our approach reduces the latency by around 21%.

Second, consider a scenario where one worker eventually fails. For the uncoded approach, we assume that the worker will signal the master when failure happens, and the master will send the subtask to another device for re-execution. Let $R^{\text{cmp}}(n) = \mathbb{E}[T_i^{\text{cmp}}]/\mathbb{E}[T_m^{\text{u}}(n)]$ denote the ratio of the expected computation latency $\mathbb{E}[T_i^{\text{cmp}}]$ of an arbitrary device $i \in [n]$ (where workers have the same latency distribution as in Section III-B) to the expected latency $\mathbb{E}[T_m^{\text{u}}(n)]$.

Proposition 3 (Device Failure Scenario)**.** *Under the conditions in Proposition 2, when $n \geq k+1$, $R^{cmp}(n) > 0.1$, and one device failure occurs, there exists a $k_{sub}^* \in (1,n-1)$ such that $\mathbb{E}[T_m^u(n)] - \mathbb{E}[T_m^c(n,k_{sub}^*)] > \Delta$.*

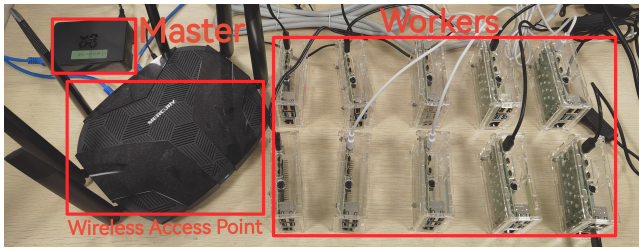


Fig. 3. Testbed for CoCoI system.

Based on Propositions 2 and 3, as the increase of straggling/failure, $\mathbb{E}[T^{\text{c}}(n)] < \mathbb{E}[T^{\text{u}}(n)]$ always holds, and CoCoI offers more latency reduction compared to the uncoded.

V. Non-Linear Coding Scheme

Although distributed inference schemes based on linear erasure codes have demonstrated good performance in addressing load imbalance and improving system robustness [**?**], they are inherently limited by the structural characteristics of linear coding. These limitations often result in frequent encoding and decoding operations, as well as significant communication overhead, which compromises system performance in exchange for flexibility.

To address these challenges, this paper further proposes a non-linear coded inference framework for robust distributed inference. By introducing learning-based non-linear encoders and decoders, our framework aims to encode longer inference tasks, thereby reducing communication overhead and enhancing both the efficiency and robustness of distributed inference in practical edge computing or heterogeneous resource environments.

Different from existing non-linear coding scheme to encode multiple inference tasks together [**?**] which fails to accelerate single-task inference, we propose non-linear distributed coded inference framework to accelerate single-request inference of CNNs, while preserving system robustness against stragglers.

A. Overview

As previously discussed (section I and appendix A), the high computational complexity of convolutional layers dominates the bottleneck in CNN inference. Therefore, we focus on applying distributed coded computation specifically to these convolutional layers. To balance the trade-off between computation and communication overhead in distributed inference, we partition the feature extraction part of the CNN into multiple sequential *conv blocks*. Each *conv block* consists of several convolution layers along with their following activation functions (e.g., ReLU) and normalization layers (e.g., BatchNorm). These *conv blocks* are treated as the minimal unit for task encoding and distributed execution, rather than individual convolutional layers [**?**] or the entire CNN [**?**].

Within the execution of each *conv block*, the forward pass is decomposed into k subtasks, where each subtask requires a

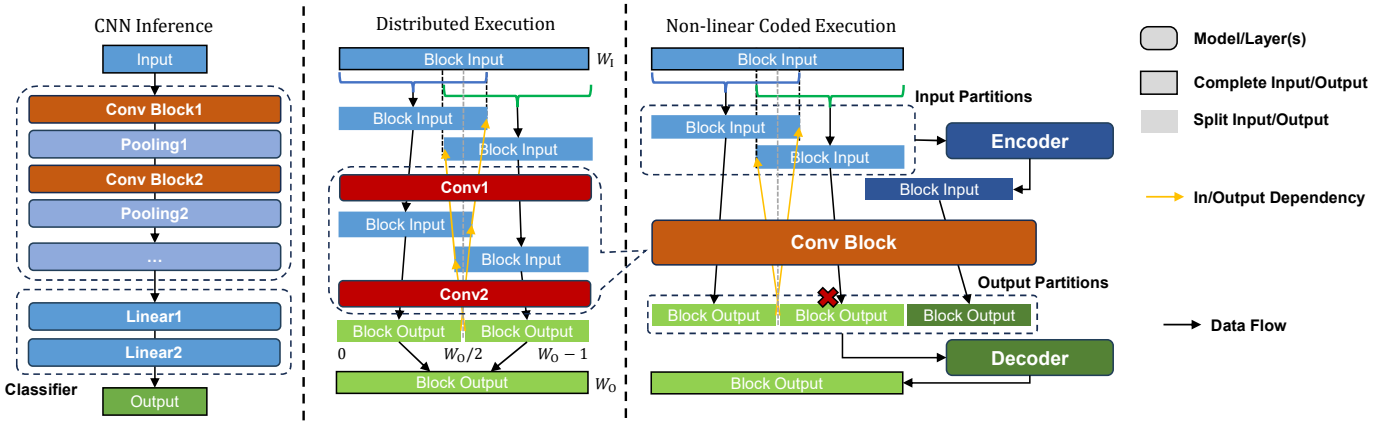


Fig. 4. Overview of non-linear coded execution on conv blocks.

partition of the original block input feature map to compute a shard of the block output. For such decomposition of a *conv block*, we design dedicated encoder-decoder pair to perform non-linear coded computation. The encoder leverages k input partitions to generated r pieces of encoded inputs as r redundant encoded subtasks, obtaining a total of $k+r$ subtasks. The decoder then leverages the encoded output of redundant subtasks to restore the block output when some origin subtasks become unavailable.

To address straggler effects in distributed computing systems, our decoding strategy is designed to be flexible. Unlike linear erasure codes, which can encode linera computations and perfectly reconstruct the original results, non-linear operations involves in deep learning models generally prevent perfect recovery, leading to inevitable accuracy degradation. Therefore, we prioritize utilizing the outputs of all original subtasks, and leveraging the encoded outputs of the redundant subtasks to approximately restore the block output when some original subtasks become stragglers or fail. For example, in the case of $k = 2$ and $r = 1$ in Fig. 4, when the second original subtask become unavailable, the system uses the remaining original subtasks and the redundant subtask to reconstruct the output via the decoder. By applying this non-linear coded computation approach to each *conv block*, we enable efficient and robust distributed inferecne across multiple workers and mitigate the straggling effect. Meanwhile, layers with relatively low computaional complexity, such as pooling layers and the fully-connected layers in the classifier of the CNN, are executed locally on the master device to minimize additional overhead and further improve overall efficiency of distributed inference. As the computational bottleneck in CNN inference primarily , distributed coded computation is applied to the convolutional layers that are computational bottlenecks within CNN inference.

B. Block Partition

Compared to treating individual convolutional layers or the entire CNN as the encoding unit, partitioning the network into *conv blocks* for distributed coded computation achieves a better trade-off between computational and communication overhead. Due to the local dependency of convolution where each element in the output feature map depends only on a local region of the input, subtask partitioning based on output feature maps naturally introduces overlapping between the corresponding input partitions. If the entire CNN is treated as a single unit for partitioning, such distributed inference method has the minimal communication overhead, the cascading effect of too many stacked convolutional layers leads to significant overlapping between the intermediated results in the shallow layers of CNNs, bringing large redundancy computation overhead between the fused-layer subtasks. On the other hand, decomposing ecah individual convolutional layer independently eliminates redundant computation overhead, but it introduces frequent inter-device synchronizations, leading to high communication overhead.

Therefore, to strike a better balance between computation and communication in distributed inference, we propose partitioning the convolutional layers of CNN into *conv blocks* as the minimal unit for distributed coded computation. We observe that pooling layers are a major source of the increasing overlapping across input partitions of original split subtasks. Specifically, pooling operations expands the receptive field of subsequent layers, thereby increasing the dependency range of output partitions on the input. Based on this insight, we partition CNN into multiple *conv blocks* separated by pooling layers.

Each *conv block* consists of multiple convolution layers, along with their associated activation functions and normalization layers. Notably, BatchNorm is essentially equivalent to a linear transform. Together with activation function, they perform element-wise operations that do not alter the input-output dependency structure. Referring to the previously mentioned notations, we define the input feature map of a *conv block* as $\mathbf{I} \in \mathbb{R}^{B\times C_\mathrm{I}\times H_\mathrm{I}\times W_\mathrm{I}}$. Give the shape of the block input $\mathbf{I}$, we can determine the shape of the block output $\mathbf{O} \in \mathbb{R}^{B\times C_\mathrm{O}\times H_\mathrm{O}\times W_\mathrm{O}}$ based on the configuration of convolutional layers within the block and equation (1). Following a similar approach as in prior work, we partition the expected output $\mathbf{O}$ along the

width dimension into k partitions $\mathbf{Y}_1, \ldots, \mathbf{Y}_k$, each serves as the target output of an original subtask. Similarly, an output partition $\mathbf{Y}_i$ ($i \in [n]$) is defined by a range $(a_{\mathrm{O}}, b_{\mathrm{O}})$ along the width dimension of $\mathbf{O}$, with $a_{\mathrm{O}} \geq 0$ and $b_{\mathrm{O}} \leq W_{\mathrm{O}}$. For an output partition $\mathbf{Y}_i$ with range $(a_{\mathrm{O}}, b_{\mathrm{O}})$, we can determined the corresponding partition range $(a_{\mathrm{I}}, b_{\mathrm{I}})$ associated with the required input partition $\mathbf{X}_i$ within the original block input $\mathbf{I}$ by iteratively computing the input range of the previous layer using equation (2) backwards.

By applying this process to all output partitions, we decompose a *conv block* into k parallelizable subtasks, each associated with a matched input-output pair of equal size. This subtask partitioning strategy enables efficient and balanced distributed execution of CNN inference, while maintaining a nice trade-off for both computational redundancy and communication overhead.

C. Encoder and Decoder

Encoding and Decoding. To generate redundant subtasks based on the original split subtasks of a *conv blocks*, we introduce specially designed encoder-decoder pair using *sparseMLP*. In general, the encoder takes k equal-size split input partitions $\mathbf{X}_1, \ldots, \mathbf{X}_k$ of the *conv block* as input and generates r encoded input partitions $\tilde{\mathbf{X}}_1, \ldots, \tilde{\mathbf{X}}_r$, each of them has a shape of $(B, C_{\mathrm{I}}, H_{\mathrm{I}}, W_{\mathrm{I}}^p)$ due to the partition on the width dimension. As for the decoder, we need a unified decoding scheme to handle different straggler cases of the subtask execution. Specifically, stragglers may occur to various number and different individual block subtasks. Therefore, the decoder takes $k + r$ encoded output partitions $\mathbf{Y}_1, \ldots, \mathbf{Y}_k$ and $\tilde{\mathbf{Y}}_1, \ldots, \tilde{\mathbf{Y}}_r$ as input, each with shape $(B, C_{\mathrm{O}}, H_{\mathrm{O}}, W_{\mathrm{O}}^p)$, and generates the decoded output feature map of size $(B, C_{\mathrm{O}}, H_{\mathrm{O}}, W_{\mathrm{O}})$, served as the block output $\mathbf{O}$. When some stragglers occur, the corresponding subtask outputs fed to the decoder are transformed to all-zero tensor, therefore the decoder does not obtain any information about the desired output corresponding to the straggler subtasks. Similar settings are also adopted in [27].

Architecture of Encoder and Decoder. Encoder composed of multiple convolutional layers (conv encoder) and decoder based on MLP (mlp decoder) have been proposed to approximate a non-linear erasure code to encode multiple CNN inference tasks [27]. However, such design leads to inefficient coded inference in edge scenarios, especially for block-wise encoding. Specifically, the high FLOPs encoding through conv encoder leads to significant encoding overhead. While the MLP encoder/decoder necessitate a large scale of parameters to encode the intermediate *conv blocks* due to the large intermediate results, even larger than the original CNN model, accounting for too much memory resources.

While linear erasure codes essentially perform simple linear transform with the input symbols, which motivates us that a non-linear erasure code may not need the full mapping from all element of the input symbols. To this end, we propose *sparseMLP* encoder/decoder that incorporates the expressiveness of MLP and the weight-sharing feature of convolutional layers. A *sparseMLP* encoder/decoder consists of multiple *sparse-linear* layers, which specifies the output size of the encoder/decoder. Different from convolutional layers, a *sparse-linear* layer applies different weight to compute the output of different positions (i.e., computing)

Training Encoder-Decoder Pairs. We simulate the straggling effects during the training of encoder-decoder pairs.

VI. Performance Evaluation

We build a testbed to implement CoCoI system, as shown in Fig. 3. This testbed consists of a wireless access point Mercury D128 (2.4/5GHz WiFi, 1200 Mbps) to provide WIFI access and 11 Raspberry-Pi 4Bs with Quad Core ARM Cortex-A72 1.5GHz. One of the Raspberry-Pi 4Bs is set as the master device with wired connection, and the remaining $n = 10$ devices are workers with WIFI connection. The operating system of the master and worker devices is Raspberry Pi OS (64-bit) based on linux and the architecture is AArch64. We apply the widely-used deep learning framework PyTorch-CPU (version 1.11.0) to support CNN inference on Raspberry-Pi. We evaluate well-known CNNs VGG16 and ResNet18 on testbed for inference with $224 \times 224 \times 3$ images as input data. Each experiment has been conducted for 20 times.

We consider three scenarios to evaluate the performance of CoCoI. Since Raspberry-Pi 4Bs originally have similar processing and communication capacities, we manually put devices into sleep in scenario-1 for evaluating the impact of straggling effect. Scenario-2 and scenario-3 are more practical scenarios without manually introduced sleeping behaviors.

- Scenario 1 (Straggling): We introduce additional random delay to wireless transmission, which follows exponential distribution with a scale of $\lambda_{\mathrm{tr}}\overline{T^{\mathrm{tr}}}$. Here, $\overline{T^{\mathrm{tr}}}$ is the original expected transmission latency.
- Scenario 2 (Device Failure): n_f workers randomly fail in each turn of subtask execution.
- Scenario 3 (Straggling and Failure): Based on scenario-2, one worker is a "high-probability" straggler, which usually has larger execution latency then other workers.

For reference, we measure the latency when CNN inference is performed locally on a single Raspberry Pi 4B. For VGG16, the average latency of normal worker and "high-probability" straggler are 50.8s and 85.2s, respectively. For ResNet18, the associated latency are 89.8s and 148.8s, respectively.

We consider the following methods for comparison. In all methods, type-1 tasks are executed in a Map-Reduce style.

- CoCoI-k^*: CoCoI with the optimal k^*, which is obtained by testing all feasible k's and choose the best one.
- CoCoI-k°: CoCoI with the approximate optimal k°.
- Uncoded [9]: Each type-1 task is split into n subtasks and allocated to workers. If any worker fails, the subtask will be re-assigned to another worker for execution.
- Replication [16]: Each type-1 task is split into $k = \lfloor n/2 \rfloor$ subtasks, with each subtask assigned to 2 workers for repeated execution. The master determines the final result upon receiving one copy of each subtask's output.

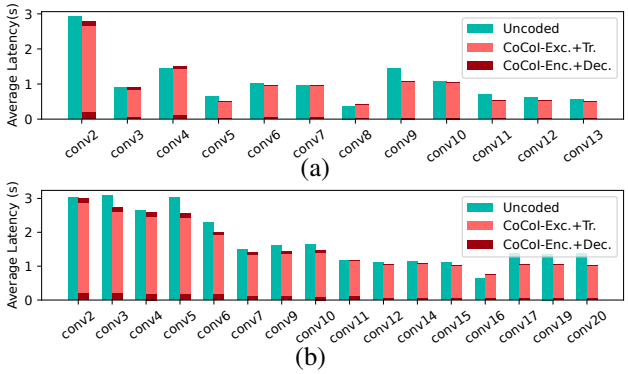


Fig. 5. Inference latency of convolutional layers of (a) VGG16 and (b) ResNet18 under secnario-1 with $\lambda^{\text{tr}} = 0.5$.

TABLE I
STATISTIC OF k^* AND k° UNDER SCENARIO-1

CNN	Statistics \ λ^{tr}	0.2	0.4	0.6	0.8	1.0
VGG16	$\max_{l\in\mathcal{L}_d} \lvert k_l^* - k_l^\circ \rvert$	1	1	1	1	1
	$\sum_{l\in\mathcal{L}_d} \lvert k_l^* - k_l^\circ \rvert / n_l$	0.42	0.5	0.5	0.42	0.5
	$\sum_{l\in\mathcal{L}_d} \lvert t_l^* - t_l^\circ \rvert$ (in sec.)	0.10	0.08	0.11	0.49	0.51
ResNet18	$\max_{l\in\mathcal{L}_d} \lvert k_l^* - k_l^\circ \rvert$	1	1	2	1	1
	$\sum_{l\in\mathcal{L}_d} \lvert k_l^* - k_l^\circ \rvert / n_l$	0.19	0.31	0.37	0.37	0.6
	$\sum_{l\in\mathcal{L}_d} \lvert t_l^* - t_l^\circ \rvert$ (in sec.)	0.35	0.15	0.97	0.37	1.3

- LtCoI-k_l: CoCoI while incorporating a Luby Transform (LT)-based coding scheme [21]. The splitting strategy $k_l = W_{\text{O}}$ represents the finest-grained splitting of type-1 tasks. Note that k_l can be larger than n.
- LtCoI-k_s: CoCoI while incorporating an LT-based coding scheme [21], with the optimal splitting strategy k_s under $k_s \leq n$. This is introduced to better compare with MDS codes in a similar number of source tasks k.

Note that LT codes (see Appendix G in [3]) is a type of rateless fountain codes [21], which is the recent state-of-the-art coding scheme. Comparing our approach with LtCoI-k_l and LtCoI-k_s shows the rationale of using MDS coding scheme.

A. Encoding and Decoding Overhead

Fig. 5 compares the total latency of each convolutional layer in CNN models between CoCoI and uncoded approach. The dark and light red areas indicate the encoding/decoding latency at the master and the execution and input/output transmission latency at workers in CoCoI, respectively. In Fig. 5, the encoding and decoding latency occupies around $2\% - 9\%$ of the total latency in each layer. In addition, despite such encoding/decoding overhead, our proposed CoCoI can usually achieve a lower average latency than the uncoded approach.

B. Approximate Optimal Splitting Strategy

Table I compares the difference between optimal and our proposed approximate optimal splitting strategies. Specifically, $\mathcal{L}_d$ denotes the set of type-1 tasks in CNN models, where the set has a size of n_l. For each layer $l \in \mathcal{L}_d$, let k_l^* and k_l° denote the optimal and approximate optimal splitting strategies, respectively. Let t_l^* and t_l° denote the associated latency under k_l^* and k_l°, respectively. As shown in Table I, the maximum difference $|k_l^* - k_l^\circ|$ is usually not larger than one, and the average difference is around 0.5. Meanwhile, the average latency difference is no larger than 1.3 seconds.

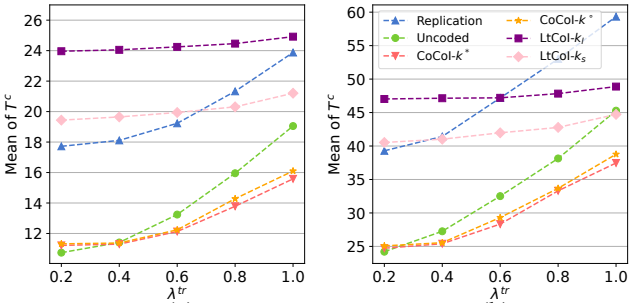


Fig. 6. CNN inference latency under scenario-1: (a) VGG16; (b) ResNet18.

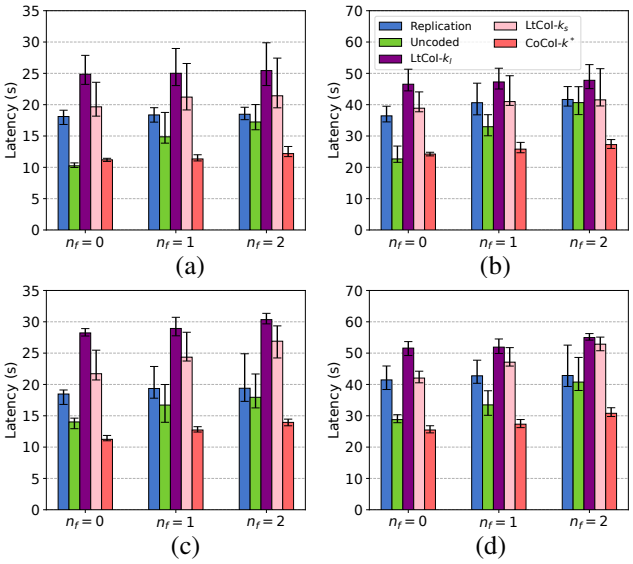


Fig. 7. Inference of (a) VGG16 and (b) ResNet18 in scenario-2; inference of (c) VGG16 and (d) ResNet18 in scenario-3.

Fig. 6 compares the CNN inference latency. In particular, CoCoI-k^* and CoCoI-k° lead to similar inference latency, which validates the rationale of using k° in practical systems.

C. Method Comparison

Fig. 6 shows the CNN inference latency in scenario-1. When λ^{tr} is small (e.g., $\lambda^{\text{tr}} \leq 0.2$), "uncoded" is slightly faster than CoCoI due to the smaller workload at each worker. However, under a moderate degree of straggling effect (e.g., $\lambda_{\text{tr}} \geq 0.4$),[4] both CoCoI-$k^*$ and CoCoI-k° can reduce the inference latency, comparing to the benchmarks. The latency reduction can be up to 20.2% when $\lambda^{\text{tr}} = 1$. Although LT codes may be more robust under stragglers [21], it achieves worse performance than "uncoded" and CoCoI in scenario-1.

[4]Even under the most severe straggling case (i.e., $\lambda^{\text{tr}} = 1$) considered in scenario-1, it corresponds to a moderate degree of straggling effect in practical systems (less severe than that considered in [19], [20]). This case corresponds to the case with $R > 1.3$ in Section IV-C. It validates that in addition to our theoretical results on severe straggling scenarios with $R \leq 1$, our approach can reduce the latency under less severe scenarios with R larger than one.

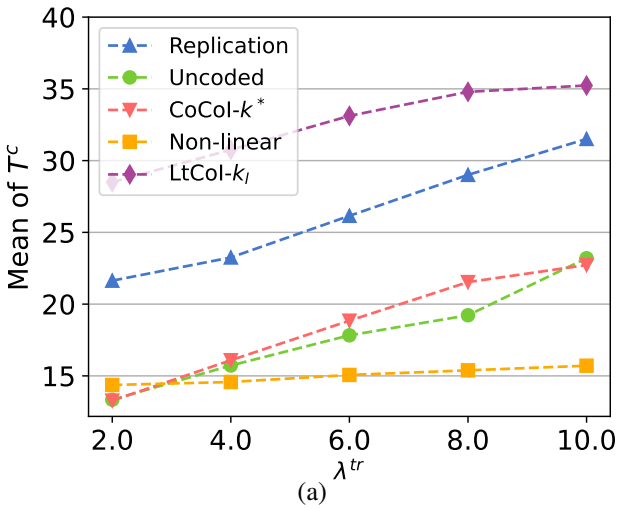


Fig. 8. CNN inference latency under scenario-1 of VGG16

Fig. 7 shows the CNN inference latency in scenario-2 and scenario-3, both of which consider device failure. The error bars here imply the variance of inference latency. When there is no device failure or straggler (i.e., $n_f = 0$ in scenario-2), CoCoI is slightly slower than "uncoded" due to the system redundancy. As n_f increases from 0 to 2 in scenario-2, the latency of "uncoded" increase by 68.3%-79.2%, while CoCoI and LtCoI show better resistance to failure. However, the fine-grained splitting of LtCoI-k_l introduces excessive transmission overhead, and LtCoI-k_s incurs a higher subtask computation overhead due to the inevitable higher redundancy resulting from the small k in LT codes. Above all, CoCoI achieves lower and more stable inference latency with less variance (according to the length of error bars) under the existence of device failure. When compared with "uncoded", the latency reduction can be up to 34.2% in scenario-2 and 26.5% scenario-3.

D. Non-Linear

Fig. 8 shows the CNN inference latency of VGG16 in scenario-1. When λ^{tr} is small (e.g., $\lambda^{tr} = 2.0$), "Uncoded" and CoCoI-k^* are slightly faster than Nonlinear due to the additional computational overhead of the neural encoder/decoder at each worker. However, under a moderate degree of straggling effect (e.g., $\lambda^{tr} \geq 4.0$), the proposed Non-linear scheme significantly outperforms the linear benchmarks due to its high stability against transmission delays. The latency reduction can be up to 47.8% when $\lambda^{tr} = 10$ compared to the Uncoded scheme (and 44.7% compared to CoCoI-k^*).

Fig. 9 presents the CNN inference latency in scenario-2 and scenario-3, both of which consider device failure. The error bars indicate the variance of the inference latency. When there is no device failure or straggler (i.e. $n_f = 0$ in scenario-2), the proposed Non-linear scheme is slightly slower than CoCoI-k^* and "Uncoded" benchmarks. This is attributed to the additional computational overhead introduced by the neural encoder and decoder inference. However, as n_f increases from 0 to 2 in scenario-2, the latency of "Uncoded" and

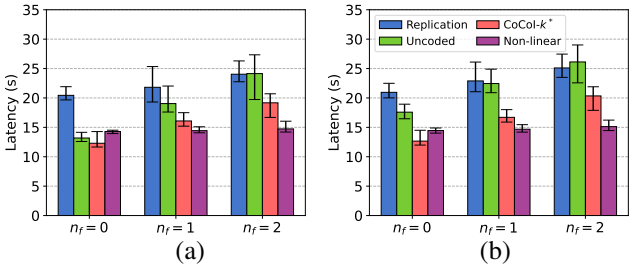


Fig. 9. Inference of VGG16 in (a) scenario-2 and (b) scenario-3.

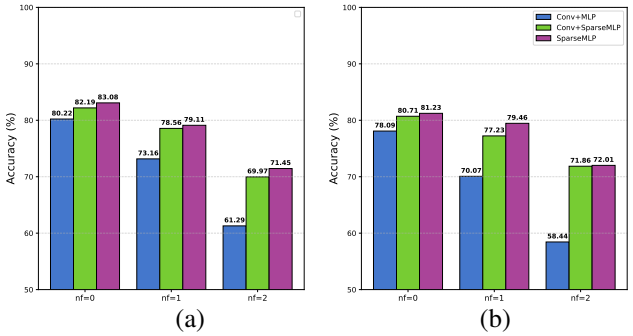
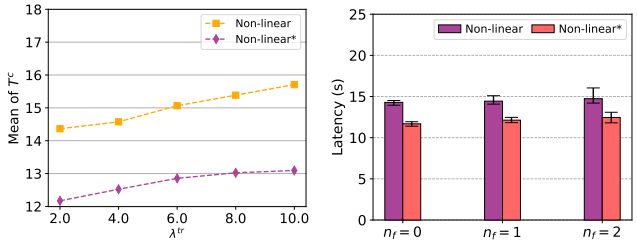


Fig. 10. Accuracy of en/decoders on (a)CIFAR10 and (b)Imagenet10.

Replication significantly deteriorates, increasing noticeable due to the necessary re-transmission or waiting for replicas. While CoCoI-k^* mitigates this rise better than the baselines, it still exhibits a noticeable latency increase as it waits for the k-th fastest result. In contrast, the Non-linear scheme demonstrates superior robustness; its latency remains around 15s regardless of the number of failures. Unlike linear codes which rely on exact reconstruction, the Non-linear approach effectively approximates the output using available partitions, eliminating the need for re-execution. Consequently, the Non-linear scheme achieves the most stable performance with negligible variance (indicated by the minimal error bars). When compared to the "Uncoded" baseline under severe failure conditions ($n_f = 2$), the Non-linear approach reduces inference latency by up to 39.0% in scenario-2 and 42.1% in scenario-3, and reduces inference latency by up to 23.1% in scenario-2 and 25.7 in scenario-3 compared to the CoCoI-k^*.

Fig. 10 compares the inference accuracy of three encoder-decoder configurations (Conv+MLP, Conv+SparseMLP, and pure SparseMLP) under varying partition loss scenarios ($n_f = 0, 1, 2$). In the failure-free scenario ($n_f = 0$), the SparseMLP architecture achieves the highest baseline accuracy of 83.08%, slightly outperforming Conv+SparseMLP (82.19%) and Conv+MLP (80.22%), indicating effective information preservation. As the number of stragglers increases, the advantage of the SparseMLP-based designs becomes more pronounced. This validating that the SparseMLP decoder significantly enhances feature map recovery when significant partitions are missing.

As illustrated in Appendix A (Fig. 12), the latency of convolutional layers in VGG16 is not uniform. The last convolutional block (conv11, conv12, and conv13) exhibits significantly lower inference latency compared to the pre-



(a) (b)

Fig. 11. Inference latency of Non-linear*

ceding blocks. For these lightweight layers, the overhead introduced by distributed coded inference, including input encoding, wireless transmission, and output decoding, which may outweigh the acceleration benefits of parallel execution.

To further optimize inference latency, we propose a Hybrid Execution Scheme Non-linear*. In this scheme, the high-complexity layers (Conv Blocks 1 through 4) are executed using our distributed coded inference approach (CoCoI or Non-linear), and the final convolutional block (Block 5) is executed locally on the Master device.

Fig. 11 shows the inference latency of Non-linear*, this reduces the total inference latency of VGG16 by approximately 15% compared to the fully distributed Non-linear approach.

VII. Conclusion

In this work, we proposed a novel distributed coded inference system, called CoCoI. This system mitigates the straggling and device failure issue by splitting 2D convolution layers, considering the data dependency between high-dimensional input and output data, and employing MDS codes to generate redundancy in distributed inference. To understand the optimal splitting strategy, we formulated an expected inference latency minimization problem. Despite its non-convexity and lack of an explicit expression, we derived an approximate solution achieving near-optimal empirical performance. Theoretical analysis demonstrates CoCoI's superiority over uncoded methods a certain degree of straggling and device failure. Evaluations on the real-world testbed shows that CoCoI can effectively reduce inference latency in the presence of stragglers and device failure. We further extended the system with a non-linear coding framework based on SparseMLP, which trades minor accuracy loss for robustness against stragglers and failures. An interesting future direction is to optimize the subtask allocation across heterogeneous workers for further inference latency reduction.

Acknowledgement

This work was supported in part by the National Natural Science Foundation of China under Grant 62202214 and Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515012819.

References

[1] Y.-J. Zhang, "Computer vision overview," in *3-D Computer Vision: Principles, Algorithms and Applications*. Springer, 2023, pp. 1–35.

[2] M. M. H. Shuvo, S. K. Islam, J. Cheng, and B. I. Morshed, "Efficient acceleration of deep learning inference on resource-constrained edge devices: A review," *Proceedings of the IEEE*, vol. 111, pp. 42–91, 2023.

[3] X. Liu, C. Huang, and M. Tang, "CoCoI: Distributed Coded Inference System for Straggler Mitigation," *arXiv e-prints*, p. arXiv:2501.06856, 2025.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015*, Y. Bengio and Y. LeCun, Eds., 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[6] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic adaptive DNN surgery for inference acceleration on the edge," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 1423–1431.

[7] L. Yang, X. Shen, C. Zhong, and Y. Liao, "On-demand inference acceleration for directed acyclic graph neural networks over edge-cloud collaboration," *Journal of Parallel and Distributed Computing*, vol. 171, no. C, p. 79–87, Jan. 2023.

[8] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1396–1401.

[9] T. Mohammed, C. Joe-Wong, R. Babbar, and M. D. Francesco, "Distributed inference acceleration with adaptive DNN partitioning and offloading," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, 2020, pp. 854–863.

[10] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2021.

[11] R. Bitar, M. Wootters, and S. El Rouayheb, "Stochastic gradient coding for straggler mitigation in distributed learning," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 277–291, 2020.

[12] H. Zhu, L. Chen, X. Chen, and W. Wang, "Heterogeneous secure coded matrix multiplication: Straggler problem versus information leakage," in *2023 IEEE 98th Vehicular Technology Conference (VTC2023-Fall)*, 2023, pp. 1–6.

[13] S. A. Said, S. M. Habashy, S. A. Salem, and E. M. Saad, "An optimized straggler mitigation framework for large-scale distributed computing systems," *IEEE Access*, vol. 10, pp. 97 075–97 088, 2022.

[14] H. Tran-Dang and D.-S. Kim, "Disco: Distributed computation offloading framework for fog computing networks," *Journal of Communications and Networks*, vol. 25, no. 1, pp. 121–131, 2023.

[15] A. Behrouzi-Far and E. Soljanin, "Data replication for reducing computing time in distributed systems with stragglers," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 5986–5988.

[16] F. Ciucu, F. Poloczek, L. Y. Chen, and M. Chan, "Practical analysis of replication-based systems," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[17] K. Lee, M. Lam, R. Pedarsani, D. Papailiopoulos, and K. Ramchandran, "Speeding up distributed machine learning using codes," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1514–1529, 2018.

[18] A. Mallick, M. Chaudhari, U. Sheth, G. Palanikumar, and G. Joshi, "Rateless codes for near-perfect load balancing in distributed matrix-vector multiplication," *Communications of the ACM*, vol. 65, no. 5, p. 111–118, Apr. 2022.

[19] S. Dutta, V. Cadambe, and P. Grover, "Coded convolution for parallel and distributed computing within a deadline," in *2017 IEEE International Symposium on Information Theory (ISIT)*, 2017, pp. 2403–2407.

[20] B. Zhou, J. Xie, and B. Wang, "Dynamic coded distributed convolution for UAV-based networked airborne computing," in *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*, 2022, pp. 955–961.

[21] B. Fang, K. Han, Z. Wang, and L. Chen, "Latency optimization for Luby Transform coded computation in wireless networks," *IEEE Wireless Communications Letters*, vol. 12, no. 2, pp. 197–201, 2023.

[22] PyTorch, "PyTorch Documentation," 2019, [Online]. Available: https://pytorch.org/docs/1.11/index.html.

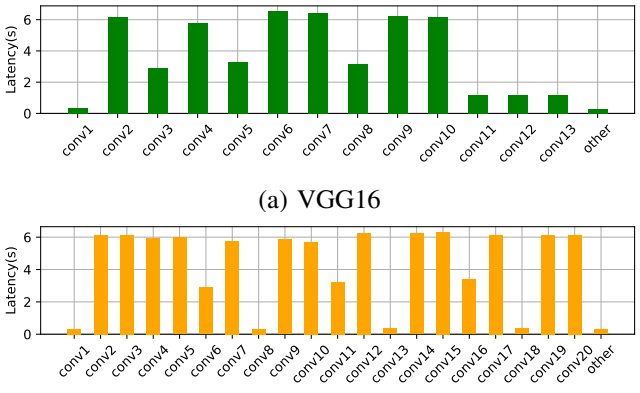
[23] A. W. Marshall and I. Olkin, "A multivariate exponential distribution," *Journal of the American Statistical Association*, vol. 62, no. 317, pp. 30–44, 1967.

[24] S. Kianidehkordi, N. Ferdinand, and S. C. Draper, "Hierarchical coded matrix multiplication," *IEEE Transactions on Information Theory*, vol. 67, no. 2, pp. 726–754, 2021.
[25] Y. Sun, F. Zhang, J. Zhao, S. Zhou, Z. Niu, and D. Gündüz, "Coded computation across shared heterogeneous workers with communication delay," *IEEE Transactions on Signal Processing*, vol. 70, pp. 3371–3385, 2022.
[26] H. A. David and H. N. Nagaraja, *Order statistics*. John Wiley & Sons, 2004.
[27] J. Kosaian, K. V. Rashmi, and S. Venkataraman, "Learning-based coded computation," *IEEE Journal on Selected Areas in Information Theory*, vol. 1, no. 1, pp. 227–236, 2020.

APPENDIX

A. Bottleneck of CNN Inference

Figure 12 shows the local inference latency of VGG16 [4] and ResNet18 [5] **by layer** on a single Raspberry Pi 4B under the same software settings as that of Section VI. Here, "other" denotes the inference latency of all layers except for convolutional layers, including pooling, activation, normalization, and linear layers. It takes 50.8s for a complete local inference of VGG16, and 89.8s for ResNet18. While convolutional layers account for 99.43% and 99.68% of total inference latency, respectively, demonstrating the evidently high computational complexity of convolutional layers. Note that **not all** convolutional layers are type-1 layers, for example, "conv1" in VGG16 and "conv1", "conv8", "conv13", "conv18" in ResNet18. We classify a layer to be a type-1 layer according to whether performing distributed execution on that layer **can** accelerate its completion latency.



(a) VGG16

(b) ResNet18

Fig. 12. Inference latency at different layers. The total latency on VGG16 and ResNet18 are 50.8s and 89.8s, respectively.

B. Stochastic Latency and Exponential Distribution Fitting

To justify the modeling of the wireless transmission or computation latency using a shift-exponential distribution, we evaluate the wireless device-to-device transmission latency with a bandwidth limitation of 100Mbps and the convolution execution latency on workers on our testbed (with Raspberry Pi 4B using python 3.9 and cpu version of torch-1.11).

To evaluate the wireless transmission latency, the master sends a 2MB torch tensor to workers for 500 times, and the workers response with a short message immediately for each received data. The master records the RTT until receiving the corresponding response. To evaluate the execution latency, we let all 10 workers repeatedly execute the same 2D-convolution task (specifically, the third convolutional layer of VGG16) for 100 times, recording each computation delay.

Figure 13 shows the CDF curves of the collected transmission and computation latency. We use the shift-exponential distribution to fit these CDF curves. As shown in Figure 13, the empirically measured transmission and computation latency distributions show a high degree of consistency with the shift-exponential distribution. Thus, we use shift-exponential distribution to model the latency in CoCoI.

C. Proof for Lemma 1

The objective function in (17) can be represented by function $P(k)$ plus a constant value:

$$P(k) \triangleq h_1(\mu^{\mathrm{m}}, \theta^{\mathrm{m}})k + h_2(\theta^{\mathrm{rec}}, \theta^{\mathrm{sen}}, \theta^{\mathrm{cmp}})\frac{1}{k} + h_3(\mu^{\mathrm{rec}}, \mu^{\mathrm{sen}}, \mu^{\mathrm{cmp}})\frac{1}{k}\ln\frac{n}{n-k} + h_4(\mu^{\mathrm{rec}})\ln\frac{n}{n-k}. \quad (18)$$

Recall $h_1(\mu^{\mathrm{m}}, \theta^{\mathrm{m}}) \triangleq 2(\frac{1}{\mu^{\mathrm{m}}}+\theta^{\mathrm{m}})(nI_{ov}+O)$, $h_2(\theta^{\mathrm{rec}}, \theta^{\mathrm{sen}}, \theta^{\mathrm{cmp}}) \triangleq 4I_W\theta^{\mathrm{rec}}+4O\theta^{\mathrm{sen}}+N_c\theta^{\mathrm{cmp}}$, $h_3(\mu^{\mathrm{rec}}, \mu^{\mathrm{sen}}, \mu^{\mathrm{cmp}}) \triangleq \frac{4I_W}{\mu^{\mathrm{rec}}}+\frac{4O}{\mu^{\mathrm{sen}}}+\frac{N_t^{\mathrm{cmp}}}{\mu^{\mathrm{cmp}}}$, $h_4(\mu^{\mathrm{rec}}) \triangleq \frac{4I_{ov}}{\mu^{\mathrm{rec}}}$, with $I_{ov} \triangleq C_{\mathrm{I}}H_{\mathrm{I}}(K-S)$, $I_W \triangleq C_{\mathrm{I}}H_{\mathrm{I}}W_{\mathrm{O}}S$, $O \triangleq C_{\mathrm{O}}H_{\mathrm{O}}W_{\mathrm{O}}$, and $N_t^{\mathrm{cmp}} \triangleq 2C_{\mathrm{O}}H_{\mathrm{O}}C_iK^2W_{\mathrm{O}}$.

Now, we are ready to prove Lemma 1. In $P(k)$, the terms of k, $1/k$, and $\ln(n/(n-k))$ are obviously convex by checking their second-order derivatives in $k \in [1, n)$. In this proof, we aim to prove that $f(k) \triangleq \ln(n/(n-k))/k$ is also convex for $k \in [1, n)$, under which problem (17) is convex.

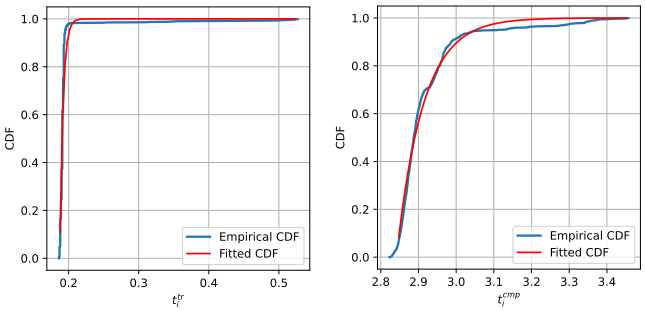
To prove the convexity of $f(k)$, we need to show that the second-order derivative $f''(k) = (3k-2n)/(k^2(n-k)^2) + 2\ln(n/(n-k))/k^3 > 0$ for $k \in [1, n)$. Since $k \geq 1$, $n > k$, and $n \geq 3$, it is equivalent to prove $g(k, n) \triangleq 3k^2 - 2nk + 2\ln(n/(n-k))(n-k)^2 > 0$. To prove $g(k, n) > 0$, we need to show that (i) $g(1, n) > 0$ and (ii) $g(k, n)$ is increasing in $k \in [1, n)$. (i) When $k = 1$, $g(1, n) = 3 - 2n + 2\ln\frac{n}{n-1}(n-1)^2$. It is easy to check that $g(1, n_l) > 0$ for a large n_l. Then, since $\partial g(1, n)/\partial n < 0$, $g(1, n) > 0$ is proven. (ii) We denote $g(k, n)$ as $g(k)$ and show $g(k)$ is increasing in $k \in [1, n)$. We need to prove $g'(k) = 4((k-n)n/(n-k) + k) > 0$. This is equivalent to prove $n/(n-k) - 1 > \ln(n/(n-k))$. Let $x \triangleq n/(n-k), x \in [n/(n-1), \infty)$. Since $x - 1 > \ln x$ for $x \in [n/(n-1), \infty)$, we have $g'(k) > 0$.

D. Empirical Evaluation for Approximation

Since the original latency formulation (13) does not have an explicit expression, we perform large-scale numerical simulations with the scale of 3×10^5 to verify its property.

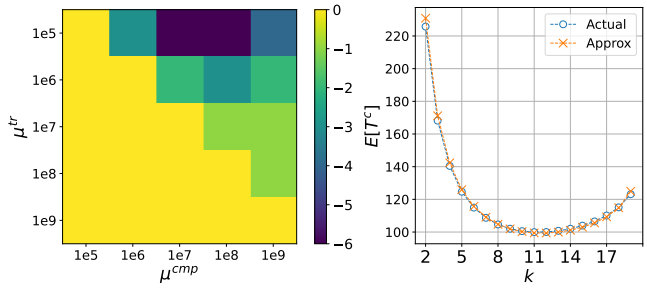
TABLE II
NOTATIONS FOR KEY PARAMETERS

Notation	Explanation
n, k	Number of worker devices/source subtasks
$f(\cdot)$	2D convolution with convolution kernel
$\mathbf{I}, \mathbf{O}$	Input/Output feature map of convolution, $\mathbf{O} = f(\mathbf{I})$
$C_{\mathrm{I}}, C_{\mathrm{O}}$	*In/Out_channels* of convolution kernel
K_W, S_W	*Kernel_size/Stride* values (on the width dimension)
$W_{\mathrm{I}}, W_{\mathrm{O}}$	Width of $\mathbf{I}, \mathbf{O}$
$W_{\mathrm{I}}^p, W_{\mathrm{O}}^p$	Width of input/output partition
$\mu^{\mathrm{m}}, \theta^{\mathrm{m}}$	Straggling and shift coefficients of comp. on the master
$\mu^{\mathrm{cmp}}, \theta^{\mathrm{cmp}}$	Straggling and shift coefficients of comp. on workers
$\mu^{\mathrm{rec}}, \theta^{\mathrm{rec}}$	Straggling and shift coefficients of recv. on workers
$\mu^{\mathrm{sen}}, \theta^{\mathrm{sen}}$	Straggling and shift coefficients of send. on workers
$T_i^{\mathrm{rec}}, T_i^{\mathrm{cmp}}, T_i^{\mathrm{sen}}$	Recv./Comp./Send. latency of subtask $i \in [n]$
$T^{\mathrm{w}}_{n:k}$	Completion latency of the k^{th} fastest subtask among n
T^{c}	Overall latency of a distributed execution
k^*, k°	Optimal k to achieve lowest $\mathbb{E}[T^{\mathrm{c}}]$ and $L(k)$



(a) (b)

Fig. 13. CDF of Empirical (a) transmission and (b) computation latency.



(a) (b)

Fig. 14. (a) The difference between the approximate optimal solution k° and the optimal splitting strategy k^* to problem (13). We set $\mu^{\text{rec}} = \mu^{\text{sen}} = \mu^{\text{tr}}$. (b) The approximation gap of the latency under different values of k ($\mu^{\text{tr}} = 10^7$ and $\mu^{\text{cmp}} = 10^8$).

We empirically show the gap between the k^* from (13) and k° from (17) under different values of $\mu^{\text{rec}} = \mu^{\text{sen}} = \mu^{\text{tr}}$ and μ^{cmp} in Fig. 14.[5] Let $n = 20$. In Fig. 14 (a), the approximation gap is around zero when the processing capacities of workers have larger variance (i.e., smaller values of μ^{cmp}). Fig. 14 (b) shows that the difference between the objective function in (13) (denoted by "Actual") and approximate objective function in (17) (denoted by "Approx") is negligible. A similar observation holds for all μ^{tr} and μ^{cmp} falling in the yellow range of Fig. 14 (a).

E. Proof and Analysis for Proposition 1

1) Proof: According to Lemma 1, by checking the first-order condition of $L(k)$, we obtain the following result.

Lemma 2 (Optimal Solution to Relaxed Problem (17))**.** *Under the relaxation of $k \in [1, n)$, the optimal solution to the relaxed problem* (17)*, denoted by $\hat{k}^\circ$, satisfies*

$$-h_1(\mu^m, \theta^m) + \frac{h_2(\theta^{rec}, \theta^{sen}, \theta^{cmp})}{(\hat{k}^\circ)^2} = \frac{h_4(\mu^{rec})}{n - \hat{k}^\circ} + h_3(\mu^{rec}, \mu^{sen}, \mu^{cmp})\left(-\frac{1}{(\hat{k}^\circ)^2}\ln\frac{n}{n-\hat{k}^\circ} + \frac{1}{\hat{k}^\circ(n-\hat{k}^\circ)}\right). \quad (19)$$

According to Lemma 2, $\hat{k}^\circ$ is the solution that satisfies equality (19). Let $l(k)$ and $r(k)$ denote the left-hand and right-hand sides of equality (19), then $\hat{k}^\circ$ is the intersection of $l(k)$ and $r(k)$. First, we show the intersection is unique. According to lemma 1, all the terms in (16) are convex regarding k, then $-l(k)$ and $r(k)$ are monotonically increasing functions. Besides, $l(k)$ and $r(k)$ have same codomin of $(0, \infty]$ for $k \in [1, n)$, thus their intersection $\hat{k}^\circ$ is unique. Then, the increasing of μ^{cmp} leads to a decreased $h_3(\cdot)$ and hence a smaller $r(k)$ under each $k \in [1, n)$. Based on the monotonicity of $l(k), r(k)$, their intersection $\hat{k}^\circ$ increases accordingly. In addition, the increasing of θ^{cmp} leads to an increased $h_2(\cdot)$ and hence a larger $l(k)$, which causes the increased $\hat{k}^\circ$. Similar proofs hold for other system parameters and are omitted.

2) Insights and Empirical Results: Proposition 1 provides insights in selecting splitting strategy in practical systems. If any straggler coefficient μ decreases, then the overall latency has a larger variance and means more severe straggling/failure effect. Thus, the computation task should be partitioned into fewer pieces (i.e., smaller k) to introduce more redundancy (i.e., larger $r \triangleq n - k$). If any shift coefficient θ of workers (i.e., $\theta^{\text{cmp}}, \theta^{\text{rec}}, \theta^{\text{sen}}$) increases, the minimum completion time of each subtask increases. As a result, the workload of subtasks become larger, so the computation task should be partitioned into smaller pieces (i.e., a larger k) to reduce the workload. Finally, if $\frac{1}{\mu^{\text{m}}} + \theta^{\text{m}}$ is larger, the master has a less powerful processing capacity, so k should be decreased to reduce the encoding and decoding latency.

In Figure 15, we evaluate the impact of system parameters on the optimal splitting strategy for problem (13) (see subfigures (a) and (c)) and the approximate optimal strategy k^* determined by problem (17) (see subfigures (b) and (d)). First, we observe that the experimental results of the approximate optimal k^* is consistent with the analytical results in Proposition 2. Second, as the total number of workers n increases, the optimal splitting strategy increases under both cases. This is reasonable because a larger n provides a larger worker pool for parallel task execution.

F. Proof for Propositions 2 and 3

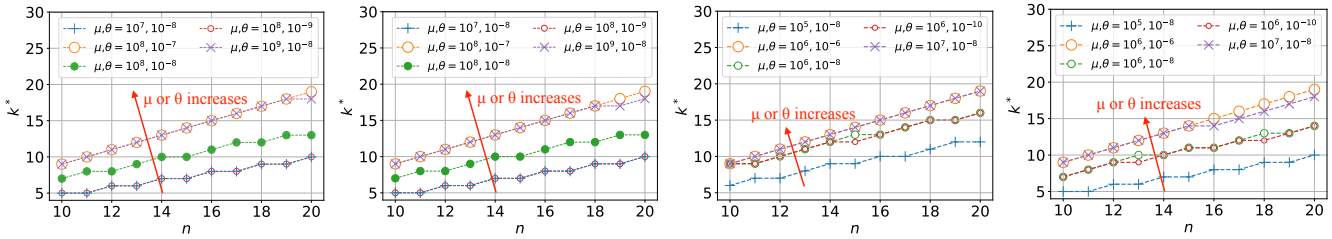
In the following, we first derive the expected latency under uncoded approach. Then, we prove Propositions 2 and 3.

1) Latency under Uncoded Approach: As in conventional uncoded distributed convolution approach (e.g., [9]), the master partitions the feature map into n pieces and sends them to workers for execution. The master can obtain the result of each layer only when all n workers send back their output. Based on (15) and [26], the expectation of latency under uncoded approach $\mathbb{E}[T^{\text{u}}]$ is determined as follows:

$$\mathbb{E}[T^{\text{u}}(n)] \approx h_2(\theta^{\text{rec}}, \theta^{\text{sen}}, \theta^{\text{cmp}})\frac{1}{n} + h_3(\mu^{\text{rec}}, \mu^{\text{sen}}, \mu^{\text{cmp}})\frac{1}{n}\ln n + h_4(\mu^{\text{rec}})\ln n + h_5(\theta^{\text{rec}}), \quad (20)$$

where $h_5(\theta^{\text{rec}}) = 4I_{ov}\theta^{\text{rec}}$. Recall the other notations have been defined in Appendix C. In the following proofs, to provide a clear insight on comparison, we omit two terms in the expected latency for both methods: we omit $h_4(\mu^{\text{rec}})$, as it is negligible for high-complexity cases with $W_{\text{O}} \gg k$; we omit $T^{\text{enc}} + T^{\text{dec}}$, as it is minor compared with T_i^{cmp}. With

[5] Since the θ's correspond to the shift coefficients, they have minor effect on the approximation error. Thus, we omit the results related to θ's.



(a) (b) (c) (d)

Fig. 15. Impact of $\mu = \mu^{\text{cmp}}$ and $\theta = \theta^{\text{cmp}}$ on (a) actual expected latency and (b) approximate expected latency; impact of $\mu = \mu^{\text{rec}} = \mu^{\text{sen}}$ and $\theta = \theta^{\text{rec}} = \theta^{\text{sen}}$ on (c) actual expected latency and (d) approximate expected latency.

these terms omitted, let $\mathbb{E}[T_m^{\text{c}}(n,k)]$ and $\mathbb{E}[T_m^{\text{u}}(n)]$ denote the expected latency of coded and uncoded method, respectively.

2) Proof of Proposition 2: Based on (16) and (20), comparing $\mathbb{E}[T_m^{\text{c}}(n,k)]$ and $\mathbb{E}[T_m^{\text{u}}(n)]$ is equivalent to comparing R and $\max_k h(n,k) \triangleq (k\ln n - n\ln n/(n-k))(n-k)$. Given n, $h(n,k)$ is maximized at $\partial h(n,k)/\partial k = 0$, where the optimal splitting strategy is $k^*_{sub}(n) = n - e$, where e is the natural base, and $h(k^*_{sub}(n)) = n/e - \ln n$. Since $h(k^*_{sub}(n))$ is monotonically increasing in $n \geq 10$ and $h(k^*_{sub}(10)) = 1.38 > R$, we have $h(k^*_{sub}(n)) > R$ for $n \geq 10$. Thus, $\mathbb{E}[T_m^{\text{c}}(n,k^*_{sub})] < \mathbb{E}[T_m^{\text{u}}(n)]$, which is $\Delta > 0$.

3) Proof of Proposition 3: Based on Proposition 2, there exists a k^*_{sub} such that $\mathbb{E}[T_m^{\text{c}}(n,k^*_{sub})] < \mathbb{E}[T_m^{\text{u}}(n)]$ holds when there is no device failure. When one device fails, $\mathbb{E}[T_m^{\text{c}}(n,k^*_{sub})]$ becomes the k^*_{sub}-th order statistics of $n-1$ random variables, thus is increased by $\frac{1}{k^*_{sub}}(\ln\frac{k^*_{sub}-1}{n-k^*_{sub}-1} - \ln\frac{n}{n-k^*_{sub}})$, which is less than 0.09 for $n \geq 10$. On the other hand, $\mathbb{E}[T_m^{\text{u}}(n)]$ is increased by at least $R^{\text{cmp}}(n) > 0.1$. Therefore, $\mathbb{E}[T_m^{\text{u}}(n)] - \mathbb{E}[T_m^{\text{c}}(n,k^*_{sub})] > \Delta$.

G. Detailed Implementation of LtCoI

We implement LtCoI using `asyncio` in python. To begin with, a type-1 convolution task is split into k pieces (where k can be larger than n), each is known as a *source symbol*. Then, for the encoding process, LT codes generate *encoded symbols* continuously, corresponding to the rateless property. Each time a random degree d is sampled from Robust Soliton distribution [18], then d *source symbols* are selected uniformly and summed up to generate an encoded symbol, and the corresponding *encoding vector* has a length of k with 1's at the selected indices of the *source symbols* and 0's elsewhere.

For each coded computation, the master creates n coroutines to distribute subtasks to n workers, and starts receiving the encoded outputs and the respective *encoding vectors*. In each coroutine, encoded symbols are continuously created and sent to the worker. Upon receiving subtasks, the workers simply execute the received subtasks and send back the results. As for decoding, since the required number of encoded outputs n_d for decoding is a random variable, we use Gaussian Elimination to judge the completeness of the received results. Once the rank of the *encoding matrix* formed by encoding vectors of the received results is k, n_d is determined and $\mathbf{O}$ can be resolved.