Optimization Framework for Splitting DNN Inference Jobs over Computing Networks

Sehun Jung, and Hyang-Won Lee, Member, IEEE

Abstract-Ubiquitous artificial intelligence (AI) is considered one of the key services in 6G systems. AI services typically rely on deep neural network (DNN) requiring heavy computation. Hence, in order to support ubiquitous AI, it is crucial to provide a solution for offloading or distributing computational burden due to DNN, especially at end devices with limited resources. We develop an optimization framework for assigning the computation tasks of DNN inference jobs to computing resources in the network, so as to reduce the inference latency. To this end, we propose a layered graph model with which simple conventional routing jointly solves the problem of selecting nodes for computation and paths for data transfer between nodes. We show that using our model, the existing approaches to splitting DNN inference jobs can be equivalently reformulated as a routing problem that possesses better numerical properties. We also apply the proposed framework to derive algorithms for minimizing the end-to-end inference latency. We show through numerical evaluations that our new formulation can find a solution for DNN inference job distribution much faster than the existing formulation, and that our algorithms can select computing nodes and data paths adaptively to the computational attributes of given DNN inference jobs, so as to reduce the end-to-end latency.

Index Terms—DNN job splitting, computing network, completion time, routing

I. INTRODUCTION

6G system is envisioned to support artificial intelligence (AI) services all over the network from the core to the end hosts, referred to as ubiquitous AI [1]. In many cases, AI services depend on the computation of deep neural network (DNN) and hence require a fair amount of computing power, even only for inference. This can be a significant burden especially for end devices such as mobile phones and IoT devices in which computing resources are highly limited as they run on limited battery power. In order for ubiquitous AI service to be in place, it is thus necessary to provide a solution to overcome limited computing power.

There are several approaches to enable AI at end devices. One is to find a lightweight NN architecture commensurate with available computing resources. SqueezeNet in [2] makes extensive use of 1x1 convolutions to reduce the number of parameters. MobileNetV1 in [3] reduces the number of arithmetic operations by introducing depth-wise separable convolutions. In [4], automated neural architecture search (NAS) is proposed based on reinforcement learning in which the

e-mail: {qhsl1213, leehw}@konkuk.ac.kr

reward reflects the latency of inference. Since the latency depends on the underlying computing resources, such a reward drives the action (i.e., values of hyperparameters) toward the set of architectures that can yield reasonably fast inference for given computing power. There is a large body of work in this context, and refer to [5] for more details.

Another approach is to exploit computing resources distributed over the network. Specifically, the (feedforward) computation of DNN inference job is partitioned into multiple tasks, and these computation tasks are assigned to nodes with computing resources in the physical network. For example, in layer-wise partition, all the neurons in the same layer are assigned to a same node [6], [7]. Once the node finishes computing the tasks (or layers) assigned, it transfers the output data (of the last layer assigned) to the next node and subsequently, the transferred data are fed as the input data to the corresponding layer. In this work, we develop an optimization framework for distributing or splitting computation tasks of DNN inference jobs over the network in which some (or all) nodes are equipped with computing resources.

Such a problem requires to make a joint decision on node selection for computation and path selection for data transfer. One of the challenges in the problem is that the amount of flow can change after passing through a node where computation is carried out, because input and output data size of DNN layer can differ from each other. Hence, the problem is vastly different from conventional routing in which flow conservation holds, i.e., incoming and outgoing flows match at every node except at source and destination. To tackle the problem, we propose a layered graph model in which simple conventional routing jointly solves the node and path selection problems. Using the model, we reformulate the existing approaches to distributing DNN inference jobs as a conventional routing problem that possesses better numerical properties.

Furthermore, we apply the proposed framework for computing DNN inference jobs over the network with minimum end-to-end latency, defined as the duration between the time when the data at source start to be processed and the time when the inference result is delivered to the destination. Obviously, the end-to-end inference latency consists of waiting time and service time. The waiting occurs at link(s) when data need to wait to be transmitted, and also at node(s) when computation tasks need to wait to be processed. The service time is the pure transmission time plus computation time. With this definition, many of the existing works in the context of DNN inference job splitting focus primarily on minimizing the service time while it is also important to take into account the waiting time.

It is generally hard to deal with waiting time as it is a

The authors are with the Department of Computer Science and Engineering, Konkuk University, Seoul, Republic of Korea.

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A2C2012801). (Corresponding author: Hyang-Won Lee)

complex function of arrival and departure processes. Nonetheless, we consider a fictitious system in which the waiting time is an upper bound on the waiting time in the actual system. Our framework enables to efficiently solve the problem of distributing a single DNN inference job such that the end-to-end latency is minimized in the fictitious system. Exploiting the efficient solvability of single-job problem with our framework, we develop algorithms for distributing multiple DNN inference jobs so as to reduce the job completion time defined as the earliest time at which all the inference jobs are finished. The contributions of our work can be summarized as follows:

- We develop an optimization framework for computing DNN inference jobs over distributed computing networks with minimum latency. Our framework enables to select nodes for computation and paths for data transfer jointly via simple conventional routing in the layered graph model.
- Applying our framework, we reformulate the existing approaches as a conventional routing problem that exhibits superior numerical performance.
- We develop algorithms for distributing DNN inference jobs over the network so that the end-to-end latency is reduced.
- We prove the numerical property of our formulation and the performance guarantee of our algorithms.
- We verify the performance of our formulations and algorithms through various numerical evaluations.

The rest of the paper is organized as follows. In Section II, we discuss related work. In Section III, we present the system model and describe the problem, and in Section IV, we propose the layered graph model that simplifies the DNN job distribution as a conventional routing problem. In Section V, we present some applications of our framework including 1) reformulation of existing approaches and 2) algorithms for computing DNN inference jobs with minimum completion time. In Section VI, we present numerical evaluations demonstrating the performance of our formulations and algorithms. We conclude the paper in Section VII.

II. RELATED WORK

Edge computing is one of the most promising solutions in the context of overcoming the computational limit at end hosts [8]. The demand for edge computing mainly arises from video or image analytics that heavily relies on compute-intensive deep learning [9]. In many cases, the focus is on how to select either the local computing resource or remote (edge or cloud) server, so that the entire analysis of an image or frame can be carried out at the chosen resource. For example, in [10], video frames are sent to edge server in which one of neural network based inference models is selected for object detection. In [11], the offloading decision of each video frame is made based on estimated network condition and response time, so that the end-to-end latency of frame analysis can be minimized. In [12], the offloading problem is solved taking into account service time and accuracy in the setting where the local device has a small model while the edge has cumbersome model with better accuracy.

As mentioned above, the demand for computation offloading arises mainly when inference or analysis should be made based on deep neural network (or deep learning). Since deep neural network typically consists of multiple layers and filters (applied block-by-block in data) in some early layers, there has been effort to distribute and parallelize the computation tasks of layers and/or blocks in layers, which is of main interest in this work.

Many of works in this context consider the Inter-of-Things (IoT) environment in which nodes are equipped with limited computing resources. In [13], each hidden neuron is mapped to an IoT node. The goal is to find a mapping that minimizes the network-wide total transmit power or time while the computing resource needed for assigned neurons and daily energy consumption satisfy the specified limit. The work of [6] assumes layer-wise partition, and seeks to find a mapping that minimizes (data) transmission time plus computation time, subject to privacy constraints as well as various resources constraints. This problem is formulated as an integer quadratic program, and a reinforcement learning based mapping algorithm is proposed. In [7], a similar problem and formulation are developed with early-exit convolutional neural network (CNN).

Some work considers vertical partition of convolution operations that take the majority of computation in DNN. Unlike the layer-wise partition where partitioned tasks have precedence constraint, in vertical partition, convolution operations are partitioned into tasks that can be computed independently. In [14], an IoT node distributes such independent computation tasks (formed through vertical partition) to nearby IoT devices and fuses the collected results at the output of convolution layer. In [15], [16], even a layer in the fully connected part is partitioned into multiple groups and distributed to computing nodes. Since parallelization inevitably incurs communication overhead, some works seek to find a partition and allocation of computation tasks with minimum communication demand [17], [18]. There are also some work considering both vertical and layer-wise partitions together with model sparsification [19].

While many of the above solutions are derived based on the formulation of integer optimization [7], [13], [17], [20], there are many results that apply deep reinforcement learning (DRL), circumventing the difficulty of solving such an optimization problem (which is NP-hard in general) [6]. In [21], the classical job-shop scheduling problem is solved via DRL adopting graph neural network, in which a job consists of sequential operations with precedence constraint. This is related to the distributed DNN computation as layers in DNN are computed sequentially with precedence. In [22], a computation task is formed with a group of neuron(s) from a single layer, and the RL agent distributes the tasks to mobile devices with reward being aggregate computation throughput, which has the effect of minimizing latency. This work is extended to account for partition of convolution operations [23]. In [24], DRL-based algorithm makes a decision on each layer of DNN inference computation whether the layer is computed at local, edge, or cloud, so as to minimize the integrated objective of delay, energy and cost. [25] develops an

RL-based algorithm for deferring the assignment of incoming inference request, leaving a room for future requests, which enables a better packing of requests with smaller latency and energy consumption.

The goal of our work is focused on optimization framework for splitting DNN inference jobs over distributed computing networks. In particular, we are primarily interested in developing an efficient formulation of the problem. We believe that such a formulation can be utilized in various facets of methodology to solve the DNN inference job distribution problem.

III. SYSTEM MODEL AND PROBLEM DESCRIPTION

Consider a communication network in which nodes are equipped with computing resources. Let $G_P = (V_P, E_P)$ denote this physical network where V_P is the set of nodes (routers/servers/hosts) and E_P is the set of edges (communication links) connecting the nodes. Let μ_{uv} be the transmission capacity of link $(u, v) \in E_P$. The computation capacity of node u is denoted as μ_u , and its unit can for example be GFLOPs/sec. There is a queue for every transmission link, and Q_{uv} is the queue length at link (u, v) representing the amount of packets waiting to be transmitted. Likewise, Q_u is the amount of computation tasks (e.g., in GFLOPs) waiting to be computed at node u. This computing network is used to process deep neural network (DNN) inference jobs.

A. Inference Jobs

There are J DNN inference jobs, each corresponding to the feedforward computation of a DNN model (We call job and model interchangeably depending on the context). For each model $j \in \mathcal{J} = \{1, ..., J\}$, the input data (e.g., camera images and sensor values) are generated at $s^j \in V_P$ and the inference result needs to be delivered to $t^j \in V_P$. Each model $j \in \mathcal{J}$ has L^j layers. We assume that each layer can be possibly computed at different nodes in V_P . Let c_l^j be the computational load of layer $l(=1,...,L^j)$ of model j. Hence, $\frac{c_l^j}{\mu_u}$ is the *computation time* if layer l of model j is to be processed at node u. Let d_l^j be the output data size of layer l of model j. Similarly, $\frac{d_l^j}{\mu_{uv}}$ is the *transmission time* if the output data of layer l is to be transferred from node u to v. The computation time plus transmission time is called the *service time*. Hence, if layer l of model j is computed at node u, and the output data is transferred to node v, then the service time at this segment is $\frac{c_l^j}{c_l^j} + \frac{d_l^j}{c_l^j}$.

is $\frac{c_i^j}{\mu_u} + \frac{d_i^j}{\mu_{uv}}$. We also consider the *waiting time*. In the above example, suppose that the queue length at node u is Q_u when the computation task arrives at node u, and the queue length at link (u, v) is Q_{uv} when the output of computation is buffered at link (u, v) for transmission. Thus, the waiting time is $\frac{Q_u}{\mu_u} + \frac{Q_{uv}}{\mu_{uv}}$. The duration between the time of entering node u and the time of arriving at node v is waiting time plus service time, i.e., $\frac{Q_u}{\mu_u} + \frac{Q_{uv}}{\mu_{uv}} + \frac{c_i^j}{\mu_u} + \frac{d_i^j}{\mu_{uv}}$. In this work, for simplicity of presentation, we ignore propagation time at any component either inside a node or between nodes (i.e., link), but our results can be readily applied to the scenario with propagation delay since propagation adds constant delay.

B. Job Completion Time

Suppose that each job j is assigned a path from its source s^{j} to destination t^{j} , including the information of the node at which each layer is computed. Time starts from 0. Let C^{j} be the time when the inference result of job j is delivered to the destination, i.e., t^{j} . Hence, C^{j} is the end-to-end inference latency. This time is obviously equal to waiting time plus service time along the path from s^{j} to t^{j} . Define the *job completion time* C_{max} as

$$C_{\max} = \max_{j} C^{j}.$$

Hence, C_{max} is the time when all the jobs are finished. This time is also called *makespan* in the field of job-shop scheduling. Clearly, the job completion time is determined by path selection for data transfer and node selection for computation. We simply call these two decisions "routing". *In this work, we develop a framework for routing DNN inference jobs for minimum job completion time.*

C. Challenges

There are several challenges in tackling the problem of routing DNN inference jobs for minimum completion time. First, the routing problem in this work is inherently related to the classical job-shop scheduling problem which is known to be NP-complete [26]. In the job-shop scheduling problem, there are multiple jobs, and each job consists of ordered operations. All of the operations must be assigned to some machine, and in each machine, the priority among operations should be decided in order to minimize the job completion time. If the machine for each operation to be computed at is given and fixed, then the problem is called the "job-shop scheduling problem". Otherwise, it is called the "flexible job-shop scheduling problem". In our problem, each layer (corresponding to operation in job-shop scheduling) in every job should be assigned a node for computation, together with path selection. Our problem therefore contains a flexible jobshop scheduling which is hard to solve. Note that this is immensely different from the conventional routing problem which is easy to solve under many circumstances.

Another challenge comes from the difficulty of handling waiting time. It is necessary to predict the waiting time so that the routing decision can be made by taking into account both waiting time and service time. However, the waiting time at a component (i.e., node or link) depends on the departure processes in the preceding components and hence, it is hard to predict the waiting time. To circumvent this issue, we consider a fictitious system in which the waiting time provides an upper bound on the actual waiting time. The completion time in the fictitious system is thus an upper bound on the actual completion time. We apply our framework in order to find a routing decision minimizing completion time in the fictitious system. The details are presented in Section V.

IV. LAYERED GRAPH AND ROUTING

As mentioned above, our problem requires to determine the path as well as the nodes for computation along the path. It is hard to formulate this problem applying the traditional technique involving flow conservation constraints. Inspired by [27] and [28] where service chaining problem in softwaredefined network (SDN) is solved with graph layering, we construct the layered graph with which the problem can be simplified. Suppose for simplicity of presentation that there is only a single model with L layers. Consider L + 1 copies of G_P , denoted by $G_0, G_1, ..., G_L$ with $G_l = (V_l, E_l), \forall l$. For each $l \in \mathcal{L} = \{0, ..., L\}$, denote by $u_l \in V_l$ the replicated node of $u \in V_P$, and hence, $(u_l, v_l) \in V_l$ the replicated link of $(u, v) \in E_P$. There is an edge from node u_{l-1} to u_l for all $u \in V_P$ and l = 1, ..., L. These edges are called *cross-layer* edges, denoted by E_C . Define the layered graph G = (V, E)where $V = V_0 \cup \cdots \cup V_L$ and $E = E_0 \cup \cdots \cup E_L \cup E_C$. Fig. 1 shows an example of the layered graph G derived from the original physical network graph G_P .



Fig. 1. Example of layered graph derived from original network graph G_P for DNN model with L layers

A. Routing in Layered Graph

We now discuss how the routing in the layered graph simply expresses both path selection for data transfer and node selection for computation. Suppose that source and destination nodes are $s \in V_P$ and $t \in V_P$ respectively. Consider finding a path from s_0 to t_L . The cross-layering segment of the path specifies the node where the corresponding layer is computed. For instance, if the path traverses link (u_{l-1}, u_l) , then layer l of the model is computed at node u. The intralayer segment of the path specifies the transfer of the output data of corresponding layer (of model). For instance, if the path traverses the link (u_l, v_l) , then the output data of layer l is transferred from node u to node v. Fig. 2 shows an example of routing in the layered graph, and what each segment in the path represents. It is important to note that the classical routing (that just finds a path from source to destination) in the layered graph determines both path selection and node selection for computing of all the layers of the model simultaneously.

B. Routing Multiple Jobs

The layered graph can easily facilitate the routing of multiple jobs at the same time. Recall that L^j and (s^j, t^j) represent the number of layers in job (or model) j and source-destination pair of job j, respectively. Redefine $L = \max_j L^j$, and construct the layered graph with L + 1 layers as shown in



Fig. 2. Example of routing in layered graph with L = 2

Fig. 1. The source $s^j \in V_P$ and destination $t^j \in V_P$ are mapped to $s_0^j \in V$ and $t_{L^j}^j \in V$, respectively. Finding a path from s_0^j to $t_{L^j}^j$ for each job j in the layered graph gives the node and path selection of all the jobs. Some examples pertaining to this are presented in Section V.

V. APPLICATIONS OF OUR FRAMEWORK

Our framework in the previous section can be applied to various problems of distributing DNN inference jobs. In this section, we present some of these examples.

A. Reformulation of Existing Approaches

The work of [6] and [7] formulates the problem of assigning DNN layers to nodes for minimizing end-to-end latency as an integer quadratic program (IQP). The data transmission time between two nodes, say u and v, is taken into account by assuming that every link has the same capacity and the data are transferred on the shortest hop path from u to v. Although this assumption of data rate simplifies the formulation, the end-to-end latency under the assignment obtained from such a formulation can experience relative large latency when data transmission time contributes to end-to-end latency to a large portion and links can have different capacities. Obviously, such a case occurs when networking resources are scarce while computing resources are abundant.

Our framework can incorporate both node and path selection as an integer linear program (ILP) which has numerical advantages over IQP. To unify the notation in layered graph, let us define the edge capacity and transmission/computation task as follows:

1)
$$\mu_{u_l v_l} = \mu_{uv}, \forall (u, v) \in E_P, l \in \mathcal{L}$$

2) $\mu_{u_{l-1}u_l} = \mu_u, \forall u \in V_P, l = 1, ..., L$
3) $q_{uv}^j = \begin{cases} d_l^j, & \text{if } (u, v) \in E_l, l \in \mathcal{L} \\ c_l^j, & \text{if } (u, v) = (w_{l-1}, w_l) \in E_C, l = 1, ..., L \end{cases}$

where $\mathcal{L} = \{0, 1, ..., L\}$. The first line indicates that every intra-layer edge $(u_l, v_l) \in E_l, \forall l \in \mathcal{L}$ in the layered graph has the same capacity as the original edge $(u, v) \in E_P$. The second line indicates that the capacity $\mu_{u_{l-1}u_l}$ of cross-layer edge $(u_{l-1}, u_l), \forall l \in \mathcal{L}$ is equal to the computation rate of node u which is μ_u . The value q_{uv}^j denotes the amount of task. If (u, v) is the intra-layer edge in G_l , then q_{uv}^j is the data size of layer l of model j. On the other hand, if (u, v) is the cross-layer edge from w_{l-1} to w_l for any w, then q_{uv}^j is the amount of computation tasks required by layer l of model *j*. Hence, $\frac{q_{uv}^j}{\mu_{uv}}$ is the transmission time of output data of layer l if (u, v) is an intra-layer edge in E_l , and the computation time of layer l at node w if (u, v) is a cross-layer edge (w_{l-1}, w_l) .

Define the variable r_{uv}^j to be 1 if the path of job j traverses edge $(u, v) \in E$, and 0 otherwise. Let m_l^j be the memory requirement of layer $l(=1, ..., L^j)$ of model j. Consider the following formulation:

$$\min_{r} \quad \sum_{j \in \mathcal{J}} \sum_{(u,v) \in E} \frac{q_{uv}^j}{\mu_{uv}} r_{uv}^j \tag{1}$$

s.t.
$$\sum_{v:(u,v)\in E} r_{uv}^j - \sum_{v:(v,u)\in E} r_{vu}^j = \begin{cases} 1, & \text{if } u = s_0^* \\ -1, & \text{if } u = t_{L^j}^j \\ 0, & \text{otherwise} \end{cases}$$

$$\forall u \in V, j \in \mathcal{J} \tag{2}$$

$$\sum_{j\in\mathcal{J}}\sum_{l=1}^{L^{j}}c_{jl}r_{u_{l-1}u_{l}}^{j}\leq \bar{c}_{u}, \forall u\in V_{P}$$
(3)

$$\sum_{j \in \mathcal{J}} \sum_{l=1}^{L^{J}} m_{jl} r_{u_{l-1}u_{l}}^{j} \le \bar{m}_{u}, \forall u \in V_{P}$$

$$\tag{4}$$

$$r_{uv}^j \in \{0,1\}, \forall (u,v) \in E, j \in \mathcal{J},$$
(5)

where \bar{c}_u and \bar{m}_u denote the maximum amount of computation tasks that are allowed at node u and the memory (such as RAM) capacity of node u, respectively. The objective function (1) is the total computation plus transmission time of all the jobs. The constraint (2) ensures that for each job j, a path is found from source s_0^j to destination $t_{L^j}^j$, so that the inference on the data generated at the source can be delivered to the destination. The constraints (3) and (4), from [6] and [7], require that the amount of computation tasks and memory requirement at each node should not exceed certain thresholds.

This is an equivalent reformulation of the one¹ in [6] and [7], except that our formulation does not fix a priori the path for data transfer between nodes. Our formulation can take into account the transmission time better, especially when link transmission rates are relatively low. More importantly, our formulation is an ILP which has advantages over IQP in [6] and [7] (Although IQP can be linearized, it introduces a large number of variables and constraints). The number of variables in our formulation is $J \cdot |E_P| \cdot (L+1)$, whereas it is $J \cdot |V_P| \cdot L$ in the formulation of [6] and [7]. Although our formulation has slightly more variables, the linearity of formulation enables to find a solution more quickly. This is demonstrated through numerical evaluation in Section VI.

B. Formulation for Single Job with Waiting Time

Since the above formulation takes into account only the service time, most of the computation tasks may be assigned to fast path with high transmission capacity or node with high computation capacity. For example, if the transmission time is negligible because link capacities are considerably high compared to computation capacity, then the service time can be minimized by assigning all the jobs to the node with highest computation capacity. However, in practice, such an assignment can incur large waiting times, and hence, adversely affect the completion time. Although introducing the budget constraints (3) and (4) has the effect of minimizing the completion by distributing the workload over the network, one may consider the completion time more directly without such constraints.

For example, consider the scenario in Fig. 3 where two models, each with two layers need to be routed. Recall that d_0^j is the input data size. Routing policy 1 seeks to minimize the total service time, thereby processing all the layers at node u. If job 1 is processed first, then job 2 must wait until job 1 is finished, which incurs one second of waiting. This results in the completion time of 2.2s. On the other hand, routing policy 2 aims at minimizing the completion time by assigning job 1 to node v and job 2 to node u. The total service time under this policy is larger than that under policy 1. However, the completion time is reduced to 1.35s as neither job experiences waiting.

al a ² d d		Tab/Madal		Layer l			
5' 5- 100Mb	$t^1 t^2$	Job/Iviodei		0	1	2	
$u \rightarrow v$ $00GFLOPs/sec \qquad 80GFLOPs/sec$ $s^{1} = s^{2} = u, t^{1} = t^{2} = v$		1	d_l^1 (Mbits)	10	0.1	0.001	
			c_l^1 (GFLOPs)		50	50	
		2	d_l^2 (Mbits)	10	0.1	0.001	
			c^2 (GFLOPs)		60	60	

1

Routing	т.1	Latency a	t Each Seg	$S_{ m tot}$	C _{max}		
Policy	100	$u \qquad u \rightarrow v \qquad v \qquad 10^{-5} s cut$				ıt off	
1	1	1	10-5	0	2.2	2.2	
1	2	1.2 (+1)	10-5	0	2.2		
2	1	0	0.1	1.25	2.55	1.35	
	2	1.2	10-5	0	2.55		
Greedy	1	1	10-5	0	2.6	1.6	
(in Sec. 5.3)	2	0	0.1	1.5	2.6		

Fig. 3. Examples of routing policy. The number "+1" in red color indicates waiting time. S_{tot} : total service time

It is important to note that the waiting time needs to be considered in order to minimize the completion time. Furthermore, if one can somehow take into account the waiting time in a more direct manner, some or all of the budget constraints (which are needed to distribute the workload) may be removed. This is because the solution that seeks to minimize waiting time will necessarily distribute the workload over the network. This is also an important aspect of considering waiting time (in a more direct manner) because it is easier to solve the problem with fewer constraints. In the following, we discuss how our framework can be applied in this context.

We start with a simple case with J = 1, i.e., a single job needs to be routed. Recall that the queue lengths Q_u , $\forall u \in V_P$ and Q_{uv} , $\forall (u, v) \in E_P$ are the computation and transmission tasks that are waiting to be computed and transmitted, respectively. These queue lengths in the physical network are reflected into the layered graph as follows:

 $\begin{array}{ll} 1) \quad Q_{u_lv_l} = Q_{uv}, \forall (u,v) \in E_P, l \in \mathcal{L} \\ 2) \quad Q_{u_{l-1}u_l} = Q_u, \forall u \in V_P, l = 1,...,L \end{array}$

¹There are some other constraints, but we only present the constraints that are essential to the discussion

6

For brevity, let us omit the superscript j. Consider the following formulation:

$$\min_{r,z} \quad \sum_{(u,v)\in E} \frac{q_{uv}}{\mu_{uv}} r_{uv} + \sum_{(u,v)\in E\setminus E_C} \frac{Q_{uv}}{\mu_{uv}} r_{uv} + \sum_{u\in V_P} \frac{Q_u}{\mu_u} z_u \quad (6)$$

s.t.
$$z_u \ge r_{u_{l-1}u_l}, \forall u \in V_P, l = 1, \dots, L$$
 (7)

$$\sum_{v:(u,v)\in E} r_{uv} - \sum_{v:(v,u)\in E} r_{vu} = \begin{cases} 1, & \text{if } u = s_0\\ -1, & \text{if } u = t_L, \forall u \in V\\ 0, & \text{otherwise} \end{cases}$$

 $r_{uv} \in \{0,1\}, \forall (u,v) \in E$

(8)

(9)

$$z_u \in \{0, 1\}, \forall u \in V_P \tag{10}$$

Similar to (2), the constraint (8) ensures that the solution finds a path from source to destination, including the node at which each layer of model is to be computed. The first term in (6) is the total service time. The second and third terms represent the waiting time at links and nodes, respectively, provided that upon the arrival at corresponding components, Q_{uv} and Q_u are the actual amount of data waiting for transmission/computation. Specifically, the second term adds the waiting times at all the links traversed. Note that if the solution translates to a path with cycle(s) in the original graph G_P , then the second term may add Q_{uv} multiple times for some $(u, v) \in E_P$. By constraint (7), z_u is 1 in the optimal solution only if node $u \in V_P$ is selected for computing layer(s). The third term is thus the total waiting time at nodes. Likewise, if the solution gives a loopy path, then there may be a node visited twice or more, and hence, the third term may not give the actual waiting time. Otherwise, if the optimal solution gives a simple path, then the objective function value is the actual end-to-end latency. This is summarized below.

Lemma 1: Assume that the optimal solution $r = [r_{uv}, (u, v) \in E]$ of problem (6)-(10) translates to a simple path (i.e., a path with no cycles) in the original graph. Then, the objective function value in (6) is the job completion time. In simulations, we were able to observe that a simple path is found in many of the instances tested.

The problem (6)-(10) is an ILP which is NP-hard in general. However, we show that our formulation has a special structure called total unimodularity, so that the optimal solution to ILP can still be found with the relaxation of binarity constraints in (9) and (10) as $0 \le r_{uv} \le 1$ and $0 \le z_u \le 1$. With this relaxation, the formulation is just a linear program (LP) which is polynomial time solvable. In general, LP introduces fractional solution, which in our problem, implies multiple paths with fractional flow. For example, the solution can have the form of $r_{uv} = 0.7$ along one path and $r_{uv} = 0.3$ along another, while the goal is to find a single path with unit flow. The following theorem shows that our formulation always has an integral optimal solution.

Theorem 1: The LP relaxation of formulation (6)-(10) has an integral optimal solution (i.e., an optimal single path with minimum completion time).

Proof: See Appendix A. By this theorem, one can find a solution of the ILP (6)-(10) by solving its LP relaxation which is much easier. This can be used to develop an efficient algorithm for routing multiple jobs, as presented in the following.

Remark. The quantities Q_u and Q_{uv} can be replaced with expected tasks when the job arrives at the corresponding resource, so that expected waiting time can be taken into account. The computation requirement $q_{u_{l-1}u_l} = c_l$ of layer l can be replaced with $c_l(1 + \alpha \mathbb{I}_{\{m_l > M_u\}})$ where M_u is the physical memory capacity of node u, and \mathbb{I} is the indicator function taking the value 1 if the condition is satisfied, and 0 otherwise. This reflects the slowdown of computation if the memory requirement exceeds the physical memory and techniques such as swapping are activated to accommodate excessive memory demand. This way, one can take into account the impact of memory demand, without budget constraints such as (4), which is likely to make the problem easier to solve.

C. Greedy Algorithm for Priority-based Job Routing

In order to show how the formulation (6)-(10) can be used, we assume preemptive scheduling at links and nodes. Priority is assigned to every inference "job" (not to individual layers), and uniformly applied to all the layers belonging to the same job. For instance, consider two jobs j_1 and j_2 , and suppose that job j_1 has higher priority than job j_2 . The computation/transmission task of a layer in job j_2 can be preempted (while it is being computed/transmitted) up on the arrival of the computation/transmission task of job j_1 .

We consider a greedy algorithm that determines the routing and priority of jobs, one at a time in the order of priority. Suppose now that the queue lengths $Q_u, \forall u \in V_P$ and $Q_{uv}, \forall (u, v) \in E_P$ represent the computation and transmission tasks (that are already routed) with higher priority than the current model to be routed. For routing a job, we use the formulation (6)-(10). Again, the first term in (6) represents the total service time at nodes and links. The third term is the total waiting time at nodes, assuming that the computation of the current job at each node u must wait until all the computation tasks of higher priority. This is obviously an upper bound on the actual waiting time because some of the computation tasks in Q_u may have been finished by the time when the computation task of the current job arrives at the node. Similarly, the second term in (6) is an upper bound on the total waiting time at links. The objective function in (6) is thus an upper bound on the actual service time plus waiting time which is equal to the completion time. Therefore, in this case, the formulation (6)-(10) seeks to find a routing that minimizes an upper bound on the completion time of current job.

Note that due to networking delays (i.e., transmission times), it is hard to express the actual waiting time in a simple form. The upper-bound approach enables a simple formulation. Although there may be a gap between upper bound and actual completion time, we expect that minimizing the upper bound would have the effect of minimizing the actual job completion time. In the following, we consider this fictitious system in which the upper bound is treated as the actual waiting time.

Let $Q = [Q_{uv}, \forall (u, v) \in E]$ which is the vector of unfinished transmission/computation tasks in the network. Recall that this vector determines the waiting time of jobs. Let $C^{j}(Q)$ be the optimal objective function value of formulation (6)-(10) for job j. That is, $C^{j}(Q)$ is the completion time of job j if it is routed based on the formulation in the presence of unfinished tasks Q. Let $r^{*}(j)$ be the optimal routing variable in this case. Note that by Theorem 1, $C^{j}(Q)$ and its solution $r^{*}(j)$ can be found by solving the LP relaxation of (6)-(10).

Algorithm 1 shows the greedy policy. First, it computes the completion time of every job, and selects the job with earliest completion time (line 1). This selected job is given highest priority. Second, the unfinished task vector Q is updated (line 2) so that the remaining jobs with lower priority can be routed with updated waiting time. The same procedure is repeated until all the jobs are routed. As shown in Fig. 3, the greedy algorithm finds a solution that balances the workload over the network, and achieves the completion time close to the minimum possible value which is achieved by routing policy 2. Although the algorithm is derived assuming the fictitious system, we expect that the greedy algorithm performs well in practice as it penalizes the solution utilizing the resources that are highly occupied.

Algorithm 1: Greedy Algorithm					
Given: Jobs $\mathcal{J} = \{1,, J\}$					
Init: $Q_{uv} = 0, \forall (u, v) \in E; U = \mathcal{J}; p = 1;$					
1 while $U \neq \emptyset$ do					
$ j_p = \arg\min_{j \in U} C^j(Q); $					
$3 Q_{uv} \leftarrow Q_{uv} + q_{uv}^{j_p} r_{uv}^*(j_p), \forall (u,v) \in E;$					
4 $p \leftarrow p+1;$					
5 $U \leftarrow U \setminus \{j_p\};$					
6 end					
Output: Priority & Routing: $[j_1 > \cdots > j_J]$ &					
$[r^*(j_p), \forall p = 1,, J]$					

The approximation ratio of this algorithm can be derived. Let us abuse the notation by defining $|V_P|$ and $|E_P|$ as the numbers of nodes with positive computation capacity and edges with finite transmission capacity respectively in the original graph G_P . Let T^* be the minimum possible job completion time in the actual system.

Theorem 2: Assume that the graph G_P is k-edge-connected and the formulation (6)-(10) always finds a simple path in G_P . Then, the job completion time under Algorithm 1 in the actual system is at most αT^* , where

$$\begin{split} \alpha &= \max\left\{2\alpha_{tx}, \frac{2(L+1)(|V_P| + |E_P|)\alpha_{tx}}{k} \ , \\ & \left(1 + \frac{|E_P|}{|V_P|}\right)\alpha_{cp}\right\}\left\{2 - \frac{1}{|V_P| + |E_P|}\right\}\\ \alpha_{tx} &= \frac{h_L \cdot \max_{j,l} d_l^j \cdot \max_{(u,v) \in E_P} \mu_{uv}}{h_S \cdot \min_{j,l} d_l^j \cdot \min_{(u,v) \in E_P} \mu_{uv}}, \ \alpha_{cp} = \frac{\max_{u \in V_P} \mu_u}{\min_{u \in V_P} \mu_u}\\ h_L &= \max_j h_L^j, \ h_S = \min_j h_S^j \end{split}$$

 h_L^j = longest path length in G_P in hop count btw. s^j and t^j h_S^j = shortest path length in G_P in hop count btw. s^j and t^j . *Proof:* See Appendix B.

Therefore, the greedy policy is an α -approximation algorithm.

Corollary 1: With zero network delay (or infinite link capacity) and identical computation capacity at all nodes, the greedy policy is a $(2 - 1/|V_P|)$ -approximation algorithm.

Proof: See Appendix C.

Although the approximation ratio in 2 seems loose, in the special case as in Corollary 1, our algorithm approximates the optimal solution within a factor smaller than 2. We thus anticipate our algorithm performs well in many other scenarios. We show in Section VI that in many cases, the greedy approximates the optimal in the fictitious system within a small neighborhood.

Remark. The priority-based scheduling discussed in this section may not be realistic in general communication networks, however such a scheduling policy will likely to gain increasing attention. As the demand for supporting a large spectrum of applications increases, the conventional one-sizefit-all network solution becomes no longer viable. The concept of network slicing has always been one of the key service paradigms that can solve this problem. With network slicing, certain services are assigned logically isolated computing and networking resource blocks, within which customized scheduling policy can be employed. The primary goal of network slicing is to provide a certain level of quality-ofservice (QoS) guarantee through flexible scheduling in the slice. For this reason, priority-based scheduling and admission control (into slice) is an important problem of research [29], [30]. Ubiquitous AI service provider may purchase a network slice, and get subscribers with differentiated service level agreement. Accordingly, we envision that the discussion of priority-based scheduling for inference can potentially be an important issue in practice in the future.

VI. NUMERICAL EVALUATION

In this section, we evaluate numerical advantages of our framework and the performance of the greedy policy in Algorithm 1. We adopt the settings from [7] where IoT environment is considered. The original physical network G_P is generated by random geometric graph. That is, nodes are randomly placed over the plane of $x \times x$ square with x = 30m. For a pair of nodes, there is a bidirectional link if the two nodes are within certain distance, e.g., communication range of each other in wireless settings. Such a communication range is set to 7.5m. If a disconnected graph is generated, it is discarded to consider only connected graphs. The link transmission rate μ_{uv} between two neighbors is randomly set to one of the values in $\{1, ..., 5\} \times \frac{\gamma \cdot \mu_{tx}^{max}}{5}$ where γ is the scaling parameter and μ_{tx}^{max} is set to 72.2 Mbps which is the data rate of WiFi 4. We scale up $(\gamma \uparrow)$ and down $(\gamma \downarrow)$ the entire link transmission rates in order to examine our framework in various regimes where the computation time dominates the completion time over transmission time or the other way around.

We assume three types of IoT nodes including Orange Pi Zero (OPZ), Beaglebone AI (BAI) and Raspberry Pi 3 (RP3). Table I shows the specs of these devices. Note that \bar{c}_u is

 TABLE I

 NODE TYPES AND CAPACITIES. MM: MILLION MULTIPLICATIONS

Node type	\bar{c}_u (MM)	\bar{m}_u (MB)	μ_u (MM/s)
OPZ	100,000	524.288	360
BAI	100,000	131.072	480
RP3	100,000	524.288	560

just the parameter introduced by the constraint (3), not a real hardware spec. Each node in G_P is randomly set to one of the three types.

We consider three CNN models including SevenLayerNet (SLN), AlexNet (AN) and ResNet101 (RN). The memory m_l^j and computation c_l^j requirement and output data size d_l^j of these models are identical to those in [7] where compressed form of layers is assumed (see Table II). For each job, its source and destination are chosen randomly from G_P , and one of the three models is randomly selected.

A. Example of Solution Found by Our Framework

We first present a small example of how our framework finds a solution for distributing DNN inference jobs. There are 20 nodes and 5 jobs. Fig. 4(a) shows the original network graph with the source-destination pair of each job to be routed. Different markers indicate different node types (circle: OPZ, diamond: BAI, square: RP3). Note that the source and destination of job 4 are the same node, implying that the corresponding node generates the data and needs the inference result on the data.

Next, the layered graph is constructed as shown in Fig. 4(b). Note that the 3D geometry is only for visualizing the graph, and not needed in the process of finding a solution. The sources are placed in layer 0, i.e., G_0 , while the destinations are placed in the final layer of corresponding DNN model. For instance, job 1 has 5 layers in the model, and hence, its destination is placed at t_5^1 in layer 5. By solving the formulation, the path is found for each source-destination pair in the layered graph.

Fig. 4(c) depicts the five paths found. The straight down arrow from upper layer to lower layer indicates that the corresponding layer is computed at the corresponding node (in the original graph). Note that job 4 is processed at the the source node up to layer 4, and then at the neighboring square node up to the final layer. The result is then delivered back to the source (which is also the destination). Every other job is processed at a single node. These paths in the layered graph are then mapped back to the original graph as shown in Fig. 4(d).

B. Numerical Advantages of Our Framework

We compare our ILP formulation (1)-(5) with the IQP formulation in [7]. The two formulations solve the same problems. We use Gurobi Optimizer running on Intel Xeon Gold 5220R CPU @ 2.2GHz with 24 cores and 256GB of RAM. Recall that in the formulation of [7], the transmission latency between nodes is assumed to be shortest hop distance * (data size \div constant rate). Since links can have different



(a) Original graph with source-destination pairs



(b) Layered graph with source-destination pairs





(d) Paths mapped to original graph

Fig. 4. Example of how the solution is found. Formulation (1)-(5) is used to solve a problem with $|V_P| = 20, J = 5$.

TABLE II CNN models and parameters. MM: million multiplications, KB: KiloBytes

Mo	del	Requirement of each layer $l = 0,, L^j$ (d_0^j : input data size)							Unit			
	d_l^j	9.41	50.18	12.54	1.54	0.77	0.04	-	-	-	-	KB
SLN	c_l^j	-	3.81	20.08	1.20	0.07	0.002	-	-	-	-	MM
	m_l^j	-	19.20	409.60	4816.90	294.91	7.68	-	-	-	-	KB
AN	d_l^j	618.35	279.94	173.06	259.58	259.58	36.86	16.38	16.38	4.00	-	KB
	c_l^j	-	105.73	224.34	149.52	112.14	74.84	37.75	16.78	4.10	-	MM
	m_l^j	-	139.78	1229.82	3540.48	2655.74	1770.50	151011.39	67125.25	16388.00	-	KB
RN	d_l^j	602.12	802.82	802.82	200.71	50.18	50.18	50.18	50.18	12.54	4.00	KB
	c_l^j	-	118.01	616.56	757.86	950.53	1156.06	1156.06	1156.06	565.18	2.20	MM
	m_l^j	-	37.63	786.43	2228.22	21757.95	26738.69	26738.69	26738.69	51380.22	8388.61	KB

rates, the constant rate in the IQP formulation is set to the average rate of links on the shortest-hop path. The number of nodes $|V_P|$ is fixed to 50, while the number of jobs J and the data rate scaling γ are varied as $J \in \{5, 10, 20, 30, 40\}$ and $\gamma \in \{0.2, 0.5, 1.0, 2.0\}$.

Fig. 5 compares the solutions found by our ILP formulation and IQP formulation of [7]. In Fig. 5(a), the service times (i.e., the objective function value in (1)) are shown in the form of relative gap defined as $\frac{O_{IQP} - O_{ILP}}{O_{ILP}} \times 100(\%)$ where O_F is the optimal objective function value of formulation F. For small values of γ , link transmission rates are small, and hence, the transmission latency takes a significant portion of service time. Since our formulation takes into account both transmission and computation latency, the solution found by our formulation achieves smaller service time especially when γ is small. On the other hand, for large values of γ , the transmission latency becomes less significant, and the performance gap between ours and IQP decreases. Observe also that the performance gap increases as the number of jobs J increases. The gap in each job's service time cumulates more with a large number of jobs.

The IQP formulation can be easily fixed to better take into account transmission time and hence, the service time of IQP formulation can be improved. We believe what is more important is the runtime to solve formulation. Since it takes nearly indefinite time for some problem instances, the time limit of the solver is set to 100s for both formulations. Fig. 5(b) shows the runtime of both formulations. As the number of jobs increases, the IQP quickly hits the time limit, meaning that in 100 seconds, it fails to find a solution which can be claimed to be optimal. On the other hand, our formulation finds an optimal solution much faster than IQP, with slowly growing runtime in the number of jobs.

C. Integrality Gap: Comparison with LP Relaxation

In order to further investigate the numerical advantages of our framework, we evaluate the integrality gap. Consider the LP relaxation of formulation (1)-(5), i.e., $r_{uv}^j \in \{0,1\}$ relaxed as $0 \le r_{uv}^j \le 1$. Let $O_{\rm LP}$ be the optimal objective function value of LP relaxation. Clearly, we have $O_{\rm LP} \le O_{\rm ILP}$ because LP relaxation optimizes over the superset of what is considered in ILP. For minimization problem, the integrality gap is defined as $\frac{O_{\rm ILP}}{O_{\rm LP}}$, which is always no smaller than 1. If the solver finds an integral solution which yields objective



(a) Relative gap of total service time (objective function value in (1)) defined as $\frac{O_{\rm IQP} - O_{\rm ILP}}{O_{\rm ILP}} \times 100(\%)$



(b) Runtime to solve formulation

Fig. 5. Comparison of ILP (ours) and IQP, with $|V_P| = 50$. Each point is the average of the results from 50 random problem instances

function value equal to $O_{\rm LP}$, it means that it has found an optimal solution. However, this happens only when the integrality gap is 1. Therefore, the integrality gap is a crucial property of formulation from numerical perspectives (as well as approximation algorithms).

Fig. 6 shows the integrality gap under various circumstances². For most of the problem instances, the integrality gap is nearly 1, and we postulate that the gap mostly results from numerical precision issue as the objective function values of ILP and LP are often equal upto a few digits after the decimal point. As mentioned above, the integrality gap of 1 can accelerate the solver to find an optimal solution. The results in Fig. 6 explain why our formulation is able to find an optimal

²In contrast with the above evaluation, some problem instances turn out to be infeasible especially when the network with small size (i.e., small $|V_P|$) needs to serve a large number of jobs. We omit the results for the combination of $(|V_P|, J)$ with which there are fewer than 10 feasible instances out of 50



Fig. 6. Integrality gap of our ILP formulation (1)-(5). Each point is the average of the results from 50 random problem instances. The results with other values of γ are omitted due to limited space.

solution much faster than IQP.

Note that for some instances, the integrality gap is relatively large. For example, with $(J = 50, |V_P| = 50)$ in Fig. 6(a), the average integrality gap is about 3. We found that for only two instances out of 50, the ILP finds an extraordinarily long path with cycles in the layered graph, after the time limit 100s of the solver is reached. The objective function values in the two cases are about 9724s and 7725s respectively, while the LP relaxation attains about 169s and 181s respectively. Such a solution of loopy path can never be optimal because one can immediately remove cycle(s) in the path to obtain a solution with smaller objective function value and reduced burden in the budget constraints. Although we believe this is an issue of solver, such an extraordinary solution can be avoided by adding a small penalty such as $\delta \sum_{j,(u,v)} r_{uv}^j$ with small positive δ . This penalizes the solution with cycle(s), forcing the solver to find an acyclic path.

Fig. 7 compares the runtime. As the number of jobs increase, the runtime of ILP increases relatively fast compared to that of LP. Notice that it takes more time for ILP to find a solution when the number of jobs is large and the network size is small. For instance, the runtime with $(|V_P| = 50, J = 40)$ is smaller than that with $(|V_P| = 20, J = 40)$. This is because when the resources are abundant in the large network, it is easier to find a solution to support all the jobs. On the other hand, if the resources are scarce compared to the demand, the solution tends to be complicated, which obviously takes the solver a long time to discover such a solution. Fig. 8 compares the solutions under small and large values of $|V_P|$ with fixed J, where only the first (in job-index-wise) 5 jobs' paths are shown. When $|V_P|$ is small, the paths tend to be complicated, and indeed, many of the paths are cyclic in the original graph. It is easy to guess from the form of the solution that it tries to pack the paths to stay below the budgets. On the other hand, when $|V_P|$ is large, all the paths are simple in the original graph. This example clearly shows why it may take longer to solve the problem when $|V_P|$ is small and J is large.

D. Evaluation of Greedy Algorithm

We now evaluate the greedy policy in Algorithm 1. The following two algorithms are considered for comparison with greedy, denoted by GRD.

1) Optimal algorithm (denoted by OPT) that finds a path for each job such that the completion time is minimized



Fig. 7. Runtime of ILP and LP vs. number of jobs J



(a) $|V_P| = 20, J = 40$: only 5 jobs' paths are shown.



(b) $|V_P| = 20, J = 40$: only 5 jobs' paths are shown. Nodes are omitted to avoid congested figure.

Fig. 8. Integrality gap of our ILP formulation (1)-(5). Each point is the average of the results from 50 random problem instances. The results with other values of γ are omitted due to limited space.

in the fictitious system. The formulation is given in Appendix D.

2) Node-first selection algorithm (denoted by NFS) that selects a single node, say u, for each job with earliest computation completion, and takes the shortest (in transmission completion time) path from source to u and from u to destination. The details are given in Appendix E.

While OPT finds an optimal solution, it may take an excessively long time. In contrast, NFS may find a solution quickly, but at the expense of increased completion time.

We first compare greedy and optimal algorithms. Let C_{ALG}

be the completion time of algorithm ALG in the fictitious system. We found that OPT fails to find a solution in 30 minutes if either J > 10 or N > 30. In addition, even if it finds a solution after the time limit of 30 minutes, it achieves even larger completion time than GRD or NFS. Hence, the results from those instances were dropped. In other words, we only consider the results in which the completion time of OPT is no greater than that of GRD and NFS. Fig. 9 shows those results. In Fig. 9(a) the relative gap of completion time, defined as $\frac{C_{\text{GRD}} - C_{\text{OPT}}}{C_{\text{OPT}}} \times 100(\%)$, remains below 7%, implying that on average, GRD achieves completion time at most 1.07 times that of OPT. As mentioned above, OPT introduces a large number of variables and constraints, and thus, it takes a long time to find an optimal solution (see Fig. 9(b)). Note that when OPT hits the time limit of 30 minutes, most of the solutions (if found) give meaningless completion time. As mentioned above, those cases were omitted, and thus, the runtime of OPT shown in Fig. 9(b) somewhat underestimates the true runtime. This result shows that the greedy policy is able to find nearly optimal solutions in reasonable time (at least for the cases shown in the figure).







Fig. 9. Comparison of greedy and optimal algorithms

Next, we compare greedy and node-first selection algorithms. In Fig. 10(a), the relative gap of completion time increases as the data transmission rate (small γ) decreases. This is because NFS selects a node (and thus a path) with an emphasis on computation time, while with small γ , the transmission time becomes a substantial element affecting completion time. The greedy policy finds a path adaptively to the circumstances regarding computation and transmission, thereby achieving better completion time performance.

On the other hand, as the numbers of nodes and jobs increase, the runtime of greedy scales poorly compared to the node-first selection algorithm. The greedy requires to solve the LP relaxation of (6)-(10) $O(J^2)$ times. We found that as the iteration continues, it tends to take longer to solve the LP. With J = 30, even one second of average solver runtime for single LP can result in several hundreds of seconds of total runtime. Therefore, it may be worthwhile to delve into the algorithmic solution for (6)-(10), which we leave as future study.



(a) Relative gap of completion time defined as $\frac{C_{\rm NFS} - C_{\rm GRD}}{C_{\rm GRD}} \times 100(\%)$

Fig. 10. Comparison of greedy and node-first selection algorithms

VII. CONCLUSION

In this paper, we proposed a framework for routing DNN inference jobs over the distributed computing network. Our framework uses a novel layered graph model to simplify and integrate the communication and computation problems via conventional routing. Our algorithms presented in this paper show an example of what one can do by exploiting the proposed framework. Namely, a simple but effective solution for enjoying the capability of DNN even with the lack of computing power can be derived. In addition to the results in this paper, we believe that our framework can facilitate practical solutions to the problem of routing inference jobs. Our framework will therefore be able to help overcome limited computing power at end devices when ubiquitous AI is about to be in service.

APPENDIX A Proof of Theorem 1

We start with some background needed to show the aforementioned property. The definitions and facts in the following can be found in [31].

Definition 1: An $m \times n$ matrix A is totally unimodular (TU) if the determinant of each square submatrix is equal to 0, 1 or -1.

Definition 2: A polyhedron $P \subseteq \mathbb{R}^n$ is the set of points that satisfy a finite number of linear inequalities, that is, $P = \{x \in \mathbb{R}^n : Ax \leq b\}$ where $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$.

Definition 3: A nonempty polyhedron is said to be *integral* if all of its extreme points³ are integral.

The following two lemmas are the key to our analysis.

Lemma 2: If $A \in \mathbb{R}^{m \times n}$ is TU, then $P(b) = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$ is integral for all $b \in \mathbb{Z}^m$ for which it is nonempty (the same is true for $P(b) = \{x \in \mathbb{R}^n_+ : Ax = b\}$).

Lemma 3: For a polyhedron $P \subseteq \mathbb{R}^n$, a solvable LP $(\max\{c^T x : x \in P\})$ has an integral optimal solution for all $c \in \mathbb{R}^n$ if and only if P is integral.

By Lemmas 2 and 3, if the constraint matrix is TU, then solving LP with integral vector b in P gives an integral optimal solution (or if not, the fractional solution can be rounded to an integral solution without losing optimality). In the case of binary ILP, LP relaxation replaces the constraints ($\cdot \in \{0, 1\}$) with ($0 \le \cdot \le 1$). If the polyhedron with this relaxation is integral, then solving the LP relaxation gives a binary optimal solution because all the variables are constrained to be between 0 and 1 and hence integrality implies binarity. We exploit this fact to show that LP relaxation of our formulation has a binary optimal solution which in our case is a single path.

A. Matrix Representation of Our Formulation

We first represent the formulation in a matrix form. Let $y = [z; r_1; r_2]^4 \in \mathbb{R}^{(L+1)(|V_P|+|E_P|)}$ where $z = [z_u, \forall u \in V_P] \in \mathbb{R}^{|V_P|}$, $r_1 = [r_{u_{l-1}u_l}, \forall u \in V_P, l = 1, 2, ..., L] \in \mathbb{R}^{L \cdot |V_P|}$ and $r_2 = [r_{uv}, \forall (u, v) \in E_l, l = 0, 1, 2, ..., L] \in \mathbb{R}^{(L+1) \cdot |E_P|}$. Let A_1 and A_2 be the matrices corresponding to constraints (7) and (8), respectively. Hence, we have $A_1 \in \mathbb{R}^{L \cdot |V_P| \times (L+1)(|V_P| + |E_P|)}$ and $A_2 \in \mathbb{R}^{(L+1) \cdot |V_P| \times (L+1)(|V_P| + |E_P|)}$. The formulation (6)-(10) can be written as

where c_1 is a vector whose inner product with y gives the service plus waiting time as in (6), and b_2 is a integral vector having 1, -1 or 0.

³A point in polyhedron P is extreme if it cannot be expressed as a nontrivial convex combination of two distinct points in P.

⁴The notation ; is used to indicate the beginning of a new row. For example, [1; -1; 1] is a 3-dimensional column vector.

Lemma 4: LP relaxation of the above formulation can be written as

$$\begin{array}{ll}
\min_{y} & c^{T}x \\
\text{subject to} & \begin{bmatrix} A_{1} & I_{12} & 0 \\ A_{2} & 0 & 0 \\ I_{31} & 0 & I_{33} \end{bmatrix} \begin{bmatrix} y \\ s_{1} \\ s_{2} \end{bmatrix} = \begin{bmatrix} 0 \\ b_{2} \\ 1 \end{bmatrix} \quad (12) \\
y, s_{1}, s_{2} \ge 0
\end{array}$$

where x is the vector $[y; s_1; s_2]$, and $s_1 \in \mathbb{R}^{(L+1)(|V_P|+|E_P|)}$ and $s_2 \in \mathbb{R}^{L|V_P|}$ are slack variables. The vector c is the augmented vector of c_1 in (11) padded with zero to match the dimension of x. The matrices I_{mn} are identity matrices of appropriate sizes.

Proof: To replace inequality constraint in (11) with equality constraint, we introduce the slack variable $s_1 \ge 0$. The formulation (11) is equivalent to

min
$$c_1^T y$$

s.t. $A_1 y + s_1 = 0$
 $A_2 y = b_2$ (13)
 y binary vector
 $s_1 \ge 0$

Now relax the binarity constraint as

min
$$c_1^T y$$

s.t. $A_1 y + s_1 = 0$
 $A_2 y = b_2$ (14)
 $y \le 1$
 $y, s_1 \ge 0$

The slack variable s_2 is introduced to replace the inequality constraint with equality constraint again, i.e., replace $y \le 1$ with $y+s_2 = 1$ and $s_2 \ge 0$. Putting all the equality constraints together obtains the desired result (12).

Note that the polyhedron in LP relaxation (12) is the same form as in Lemma 2 with equality Ax = b, where

$$A = \begin{bmatrix} A_1 & I_{12} & 0 \\ A_2 & 0 & 0 \\ I_{31} & 0 & I_{33} \end{bmatrix}, \quad x = \begin{bmatrix} y \\ s_1 \\ s_2 \end{bmatrix}, \quad b = \begin{bmatrix} 0 \\ b_2 \\ 1 \end{bmatrix}$$
(15)

Note that b is an integral vector. Consequently, by Lemmas 2 and 3, we just need to show the total unimodularity of A.

B. Total Unimodularity

We start with some background on total unimodularity [31]. Lemma 5: For an $m \times n$ integral matrix A, the following are equivalent:

- 1) A is TU
- 2) A^T is TU
- 3) For each $C \subseteq \{1, ..., n\}$, there exists a partition (C_1, C_2) of C such that

$$\left| \sum_{j \in C_1} a_{ij} - \sum_{j \in C_2} a_{ij} \right| \le 1 \text{ for } i = 1, ..., m$$
 (16)

4) For each R ⊆ {1,...,m}, there exists a partition (R₁, R₂) of R such that

$$\left|\sum_{i \in R_1} a_{ij} - \sum_{i \in R_2} a_{ij}\right| \le 1 \text{ for } j = 1, ..., j \quad (17)$$

Condition (16) requires that for each row, the partitioned sum (called partitioned row sum) must differ by at most 1. Note that statement 4) in Lemma 5 is an immediate consequence of 1), 2) and 3).

We can show the following lemma.

Lemma 6: If $[A_1; A_2]$ is TU, then A is TU.

Proof: Suppose that $[A_1; A_2]$ is TU. Then, by statement 3) in Lemma 5, any subset of the first block column (i.e., block containing $[A_1; A_2]$) can be partitioned into two sets C_1 and C_2 so that condition (16) is satisfied. Note that the identity matrix I_{31} in the first block column of A does not affect the forming of C_1 and C_2 because any column partition of identity matrix satisfies condition (16).

Next, consider adding an arbitrary column, say *i*th column, from the second block column of A. Note that the only nonzero entry (which is 1) is in the *i*th row of this column. So, if the current partitioned sum difference of ith row is zero, then put the column into either C_1 or C_2 , which does not violate condition (16). If the partitioned row sum difference is 1, put the column into a subset with smaller row sum so that the partitioned row sum difference is balanced, i.e., zero. The third block column can be treated exactly the same as the second block in order to keep the condition in (16). This completes the proof.

We can now focus on proving the total unimodularity of $[A_1; A_2]$ which is indeed totally unimodular.

Lemma 7: The matrix $[A_1; A_2]$ is TU.

Proof: Consider the breakdown of $[A_1; A_2]$ as shown in Fig. 11. Let $A_1 = [A_{11} \ A_{12} \ A_{13}]$. The first $|V_P|$ columns correspond to the variables $z_u, \forall u \in V_P$. The next $L|V_P|$ columns correspond to the variables $r_{u_{l-1}u_l}, \forall l = 1, ..., L, \forall u \in V_P$. The last $(L+1)|E_P|$ columns correspond to the variables $r_{uv}, \forall (u,v) \in E_l, l = 0, 1, ..., L$.

Each row of A_1 in the top block corresponds to a constraint in (7). Constraints are arranged such that the first block of Lrows corresponds to some node u, and each row in the block indicates whether each cross-layer edge along replicated nodes of u is visited. This is repeated for every other node in V_P . This forms the matrix A_1 .

Each row of A_2 in the bottom block corresponds to a flow conservation constraint in (8). For instance, for some node $u \in V_P$, the first L + 1 rows correspond to the flow conservation at replicated nodes u_l in each layer l = 0, 1, ..., L. As in A_1 , the second block A_{22} corresponds to the variables $r_{u_{l-1}u_l}, l = 1, ..., L$ of cross-layer edges. Constraints are arranged in the order of $u_0, u_1, ..., u_L$. Since the edges are between consecutive nodes, A_{22} are written as in Fig. 11. Note that this A_{22} is a network matrix. The third block A_{23} corresponds to the variables $r_{uv}, \forall (u, v) \in E_l, l = 0, 1, ..., L$ of intra-layer edges. Clearly, A_{23} is a network matrix of L+1 connected components $G_0, ..., G_L$.

Fig. 11. Breakdown of matrix A. A_{23} is the network matrix of L+1 layers $G_0, ..., G_L$ without cross-layer edges

Consider partitioning the columns of the matrix A. Let C_1 and C_2 be a partition of an arbitrary subset C of the entire columns. The following rule is applied for the partition.

- 1) Put any column from the first $(L+1)|V_P|$ columns into C_1 regardless of J
 - Without the third column block (i.e., the columns of $[A_{13}; A_{23}]$, this guarantees that each row sum is 0, 1, or -1. Furthermore, the row sums corresponding to A_{22} have exactly the same numbers of 1s and -1s. Let $s \in \mathbb{R}^{(L+1)|V_P|}$ be the vector of these row sums from the columns of A_{22} . Again, s has exactly same numbers of 1s and -1s. Consider the matrix $[s A_{23}]$. This matrix is obviously TU because i) s has exactly the same numbers of 1s and -1sand ii) each column of A_{23} has exactly one 1 and one -1. Consequently, any subset of rows can be partitioned into two subsets so that the condition in (17) can be satisfied. That partition is indeed putting all the rows in the same subset, with which the partitioned column sum difference is always 0, 1, or -1. Consequently, $[s A_{23}]$ is TU and any subset of columns of the matrix can be partitioned so that the condition in (16) is satisfied.
- 2) Treat any columns from the first two block columns as a single column by adding those columns. Let d be the resulting column vector. Let E be the matrix of arbitrary column(s) from [A₁₃; A₂₃]. Partition the columns of [d E] in the same way as partitioning an arbitrary subset of columns of [s, A23] discussed above.
 - As shown in 1), all of the top block row sums satisfy condition (16) since the third block column in the top block row (i.e., A_{13}) is a zero matrix. By the argument in 1) on the TU of $[s A_{23}]$, all of the bottom block row sums also satisfy (16).

This completes the proof.

Proof of Theorem 2: By Lemmas 7 and 6, A is TU. By Lemmas 3 and 2, LP relaxation (12) has an integral optimal

solution. It is clear that LP relaxation (12) is equivalent to (18).

min
$$c_1^T y$$

s.t. $A_1 y \leq 0$
 $A_2 y = b_2$
 $0 \leq y \leq 1$
(18)

Therefore, LP relaxation (18) has an integral optimal solution which is a single path with minimum completion time.

APPENDIX B

PROOF OF THEOREM 2

We first introduce various routing policies below.

- Shortest completion (SC): optimal routing policy that achieves minimum job completion time T^* in the actual system
- Shortest service (SS): routing policy that achieves shortest service time for each job (same in the fictitious and actual systems)
- Greedy (GR): Algorithm 1 (in the fictitious system)
- Shortest waiting (SW): given that p-1 jobs are routed by greedy, the *p*th job is assigned to a node with positive computation capacity, say u^* , with shortest computation waiting time in the fictitious system. Then, a shortest transmission waiting (in the fictitious system) path is assigned from source to u^* and from u^* to destination.

The following notation is used:

- W_j^{π} : total waiting time of job j under routing policy π
- S_j^{π} : total service time of job j under routing policy π
- $W_i^{\pi}(tx)$: total transmission waiting time of job j under routing policy π
- $W_i^{\pi}(cp)$: total computation waiting time of job j under routing policy π
- $S_j^{\pi}(tx)$: total transmission (service) time of job j under routing policy π
- $S_i^{\pi}(cp)$: total computation (service) time of job j under routing policy π

It is clear from the definition that $W_i^{\pi} + S_i^{\pi}$ is the completion time of job j under routing policy π . We also have $S_i^{\pi} =$ $S_{j}^{\pi}(tx) + S_{j}^{\pi}(cp)$ and $W_{j}^{\pi} = W_{j}^{\pi}(tx) + W_{j}^{\pi}(cp)$.

We need a few lemmas to prove Theorem 2. For simplicity of presentation, we break the results into small lemmas and integrate these results for the proof of Theorem 2. Denote by j_p the *p*th job routed under greedy routing policy.

Lemma 8: The job completion time under greedy algorithm in the actual system is upper-bounded by $W_{j_J}^{GR} + S_{j_J}^{GR}$ where j_J is the last job routed under greedy algorithm.

Proof: By assumption of Theorem 2, the greedy policy finds only a simple path for every job, and hence, each job's completion time in the fictitious system is the objective function value of the iteration in which the job's route is fixed. The greedy policy routes a job with earlier completion time first, and hence, the waiting time plus service time $W_{i_I}^{GR} + S_{i_I}^{GR}$ of last job routed under greedy is the job completion time of the greedy policy in the fictitious system. The completion time in the actual system is at most the completion time in the fictitious system, and consequently, upper-bounded by $W_{i_I}^{\text{GR}} + S_{i_I}^{\text{GR}}$.

Consequently, in order to analyze the job completion time of greedy policy, we just need to derive a bound on $W_{j_J}^{GR} + S_{j_J}^{GR}$. Lemma 9: The optimal job completion time T^* is lower-

bounded as

$$S_j^{\rm SS} \le T^*, j = 1, ..., J$$
 (19)

$$\frac{1}{|V_P| + |E_P|} \sum_{j=1}^{J} S_j^{SS} \le T^*$$
(20)

Proof: The LHS in inequality (19) is the fastest possible service time of job j. The entire job is completed only after every job has been served. Hence, the minimum completion time T^* cannot be smaller than the fastest possible service time of every job.

To show (20), we have

$$\frac{1}{|V_P| + |E_P|} \sum_{j=1}^J S_j^{SS} \le \frac{1}{|V_P| + |E_P|} \sum_{j=1}^J S_j^{SC} \le T^* \quad (21)$$

The first inequality holds since the total service time of SC cannot be smaller than that of SS. In the second inequality, the LHS is the average busy time of network components (nodes+links) under SC. It is clear that job cannot be completed as long as there remains a busy component under SC, and consequently, optimal job completion time T^* is no smaller than the LHS. Note that the service time at the link with infinite capacity is zero, and hence, those links should be excluded when computing the average. Similarly, under SS or SC, the computation will not be carried out at the node with zero computation capacity, and hence, those nodes should be excluded when computing the average. This completes the proof.

The above lemma enables to compare GR and SC via SS. Let h_L^j and h_S^j be the longest and shortest path lengths (in G_P) in hop count between s^j and t^j , respectively. Define $h_L =$ $\max_{j} h_{L}^{j}$ and $h_{S} = \max_{j} h_{S}^{j}$.

Lemma 10: The transmission times are bounded as

$$S_j^{\rm SW}(tx) \le 2\alpha_{tx} S_j^{\rm SS}(tx), \forall j$$
(22)

$$S_j^{\rm GR}(tx) \le (L+1)\alpha_{tx}S_j^{\rm SS}(tx), \forall j$$
(23)

where $\alpha_{tx} = \frac{h_L \cdot \max_{j,l} d_l^j \cdot \max_{(u,v) \in E_P} \mu_{uv}}{h_S \cdot \min_{j,l} d_l^j \cdot \min_{(u,v) \in E_P} \mu_{uv}}.$

Proof: Under routing policy SW, link transmission occurs only in layers 0 and L because all the NN layers are computed in a single node with smallest computation waiting time. Furthermore, in layers 0 and L, the transmission path is simple because SW takes a shortest transmission waiting path. Consequently, in both of layers 0 and L, the transmission time is at most $\frac{\prod_{j,l}^{h_L \cdot \max_{j,l} d_l^j}}{\prod_{(u,v) \in E_P} \mu_{uv}}$. The transmission time under SW is

thus upper-bounded as h_{τ} , may d^{j}

$$S_j^{\text{SW}}(tx) \le 2 \frac{n_L + \max_l u_l}{\min_{(u,v) \in E_P} \mu_{uv}}.$$
(24)

On the other hand, the transmission time under SS is at least $\frac{h_S \cdot \min d_l^j}{\max_{(u,v) \in E_P} \mu_{uv}}$. Combining this with inequality in (24) yields the desired result (22).

To prove (23), the same argument as above is applied to each of L + 1 layers in G since the greedy policy finds a simple path in each layer. This completes the proof.

Lemma 11: For any routing policy π , the computation service time is upper-bounded as

$$S_j^{\pi}(cp) \le \alpha_{cp} S_j^{\rm SS}(cp), \forall j$$
(25)

where $\alpha_{cp} = \frac{\max_{u} \mu_{u}}{\min_{u} \mu_{u}}$.

Proof: The worst case is when all the layers are computed at the node with smallest (positive) computation capacity, and the best case is when computed at the node with largest computation capacity. The lemma immediately follows from this observation.

Lemma 12: The service time under routing policy SW is bounded as

$$S_j^{\rm SW} \le \alpha_1 S_j^{\rm SS}, \forall j, \tag{26}$$

where $\alpha_1 = \max(2\alpha_{tx}, \alpha_{cp})$.

Proof: We have

$$S_j^{\rm SW} = S_j^{\rm SW}(tx) + S_j^{\rm SW}(cp) \tag{27}$$

$$\leq 2\alpha_{tx}S_j^{\rm SS}(tx) + \alpha_{cp}S_j^{\rm SS}(cp) \tag{28}$$

$$\leq \alpha_1 S_j^{\rm SS}$$
 (29)

The first inequality follows from Lemmas 10 and 11, and the second inequality follows from the definition of α_1 .

The above lemmas characterize the bounds on the service time. We now derive the bounds on waiting time. Recall that given p-1 jobs routed by GR, the policy SW routes the *p*th job j_p as mentioned in the beginning of this section.

Lemma 13: Suppose that the original graph G_P is k-edgeconnected. Then,

$$W_{j_J}^{SW}(tx) \le \frac{2(L+1)\alpha_{tx}}{k} \sum_{p=1}^{J-1} S_{j_p}^{SS}(tx).$$
(30)

Proof: Recall that when job j_J is to be routed by SW, the rest of J-1 jobs have already been routed by GR. By assumption, there are k disjoint paths between any pair of nodes. For the path segment from source to u^* , the average transmission waiting time along each of k disjoint paths is at most⁵ $\frac{1}{k} \sum_{p=1}^{J-1} S_{j_p}^{\text{GR}}(tx)$. This shows that there is a path from source to u^* with transmission waiting time no greater than $\frac{1}{k} \sum_{p=1}^{J-1} S_{j_p}^{\text{GR}}(tx)$. Since the routing policy SW selects a path from source to u^* that has the smallest transmission waiting time, the transmission waiting time $W_{j_J}^{\text{SW}}(tx)$ is upperbounded by $\frac{1}{k} \sum_{p=1}^{J-1} S_{j_p}^{\text{GR}}(tx)$. By Lemma 10, this is in turn

⁵This is an upper bound as there may edges not in k disjoint paths.

upper-bounded by $\frac{(L+1)}{k} \sum_{p=1}^{J-1} S_{j_p}^{SS}(tx)$. Applying the same argument to the segment from u^* to destination proves the lemma.

Lemma 14: We have

$$W_{j_J}^{SW}(cp) \le \frac{\alpha_{cp}}{|V_P|} \sum_{p=1}^{J-1} S_{j_p}^{SS}(cp)$$
 (31)

Proof: This lemma immediately follows from the following inequalities:

$$W_{j_J}^{\text{SW}}(cp) \le \frac{1}{|V_P|} \sum_{p=1}^{J-1} S_{j_p}^{\text{GR}}(cp) \le \frac{\alpha_{cp}}{|V_P|} \sum_{p=1}^{J-1} S_{j_p}^{\text{SS}}(cp) \quad (32)$$

Recall that both GR (by nature of formulation (6)-(10)) and SW (by definition) do not compute at the node with zero computation capacity. In addition, $|V_P|$ is the number of nodes with positive computation capacity. The RHS of the first inequality is thus the average computation waiting time at a node. Since the routing policy SW selects a node with minimum waiting time, the node waiting time under SW should be no greater than the average node waiting time, which shows the first inequality. The second inequality follows from Lemma 11. This completes the proof.

Lemma 15: We have

$$W_{j_J}^{\rm GR} + S_{j_J}^{\rm GR} \le W_{j_J}^{\rm SW} + S_{j_J}^{\rm SW}.$$
(33)

Proof: Recall that given J - 1 jobs routed by GR, SW finds a path such that the last job is processed at a single node, say u^* , with minimum waiting time, and a shortest (in link transmission waiting) path from source to u^* and a shortest path from u^* to destination are concatenated to form a path from source to destination of the last job. On the other hand, for the last job, GR finds a path with minimum waiting plus service time, given J - 1 jobs routed by GR. Therefore, the inequality holds.

Proof of Theorem 2: By Lemmas 8 and 15, the job completion time under GR in the actual system is at most $W_{j_J}^{\text{SW}} + S_{j_J}^{\text{SW}}$. Let $\alpha_2 = \max\left(\frac{\alpha_{cp}}{|V_P|}, \frac{2(L+1)\alpha_{tx}}{k}\right)$. By Lemmas 13 and 14, we have

$$W_{j_{J}}^{SW} \leq \frac{\alpha_{cp}}{|V_{P}|} \sum_{p=1}^{J-1} S_{j_{p}}^{SS}(cp) + \frac{2(L+1)\alpha_{tx}}{k} \sum_{p=1}^{J-1} S_{j_{p}}^{SS}(tx) \quad (34)$$
$$\leq \alpha_{2} \sum_{p=1}^{J-1} S_{j_{p}}^{SS} \qquad (35)$$

This together with Lemma 12 leads to

$$W_{j_J}^{\text{SW}} + S_{j_J}^{\text{SW}} \le \alpha_2 \sum_{p=1}^{J-1} S_{j_p}^{\text{SS}} + \alpha_1 S_{j_J}^{\text{SS}}$$
(36)

$$\leq \alpha_3 \left\{ \frac{1}{|V_P| + |E_P|} \sum_{p=1}^{J-1} S_{j_p}^{SS} + S_{j_J}^{SS} \right\}$$
(37)

$$= \alpha_3 \left\{ \frac{1}{|V_P| + |E_P|} \sum_{p=1}^{J} S_{j_p}^{SS} + \left(1 - \frac{1}{|V_P| + |E_P|}\right) S_{j_J}^{SS} \right\} (38)$$

$$\leq \alpha_3 \left(2 - \frac{1}{|V_P| + |E_P|} \right) T^* \tag{39}$$

where $\alpha_3 = \max(\alpha_2(|V_P| + |E_P|), \alpha_1)$. The last inequality follows from Lemma 9. This completes the proof.

APPENDIX C

PROOF OF COROLLARY 1

By assumption of zero network delay, we have $|E_P| = 0$, and also $\alpha_{tx} = 0$ because Lemma 10 holds with $\alpha_{tx} = 0$. In addition, $\alpha_{cp} = 1$. It immediately follows that $\alpha_1 = 1$, $\alpha_2 = \frac{1}{|V_P|}$ and $\alpha_3 = 1$. Therefore, the approximation ratio is $2 - \frac{1}{|V_P|}$, which completes the proof.

APPENDIX D

FORMULATION OF OPTIMAL ROUTING

Consider the binary variable t_p^j that takes the value 1 if job j has priority p, and 0 otherwise. For two jobs j and k, if $t_p^j = 1$ and $t_{p'}^k = 1$ with p' < p, then it means that job k has higher priority than j. As mentioned in Section V, the waiting time of j at a component (link or node) in the fictitious system is the sum of the computation or transmission tasks of all the jobs passing through the component with higher priority than j. The problem of minimizing the completion time in the fictitious system can be formulated as follows:

$$\min_{r,z,t} \max_{j \in \mathcal{J}} \sum_{(u,v) \in E} \frac{q_{uv}^{\prime}}{\mu_{uv}} r_{uv}^{j} + \sum_{(u,v) \in E \setminus E_{C}} \frac{1}{\mu_{uv}} \left(Q_{uv} + \sum_{p \in \mathcal{J}} \sum_{p' < p} \sum_{j' \in \mathcal{J}} t_{p'}^{j} t_{p'}^{j'} q_{uv}^{j'} r_{uv}^{j'} \right) r_{uv}^{j} + \sum_{u \in V_{P}} \frac{1}{\mu_{u}} \left(Q_{u} + \sum_{\substack{(v,w) \in E_{C} \\ v \in V_{u}}} \sum_{p \in \mathcal{J}} \sum_{p' < p} \sum_{j' \in \mathcal{J}} t_{p}^{j} t_{p'}^{j} q_{vw}^{j'} r_{vw}^{j'} \right) z_{u}^{j}$$
(40)

s.t. Constraint (2)

$$z_u^j \ge r_{u_{l-1}u_l}^j, \forall u \in V_P, l = 1, \dots, L^j, j \in \mathcal{J}$$

$$(4)$$

)

$$\sum_{p \in \mathcal{J}} t_p^j = 1, \forall j \in \mathcal{J}$$
(42)

$$\sum_{j \in \mathcal{J}} t_p^j = 1, \forall p \in \mathcal{J}$$
(43)

$$r_{uv}^j \in \{0,1\}, \forall (u,v) \in E, \forall j \in \mathcal{J}$$

$$(44)$$

$$z_u^j \in \{0,1\}, \forall u \in V_P, \forall j \in \mathcal{J}$$

$$(45)$$

$$t_{jp} \in \{0, 1\}, \forall j, p \in \mathcal{J}$$

$$\tag{46}$$

The first term in the objective function is the total service time. The second term is the waiting time at the traversing links, adding up the transmission tasks of all the jobs with higher priority. Similarly, the third term is the waiting time at the node where the processing of job j is carried out, adding up the computation tasks of all the jobs with higher priority. The constraints (42) and (43) ensure that every job is assigned a unique priority index p, and no priority index p is assigned to more than one job. The maximum of this objective function is the completion time, and the goal is to find a routing with minimum completion time. This formulation introduces a large number of variables. We found that it takes excessively long time to preprocess for constructing objective function and constraints. Furthermore, the solver often fails to find a solution even in about half an hour. This led us to exclude problem instances with large network/job size.

APPENDIX E NODE-FIRST SELECTION ALGORITHM

Consider the routine [u, C] = selectNode(V, W) that selects the smallest-weight node in V with node weights $W = [w_v, \forall v \in V]$. The output u is the selected node, and C is the weight of the selected node. Likewise, consider the routine [H, C] = findShoPath(G, u, v, W) that finds the shortest path from $u \in V$ to $v \in V$ over the graph G = (V, E) with link weights $W = [w_{uv}, \forall (u, v) \in E]$. The output H is the shortest path found, and C is the total weight of the shortest path. Consider the following algorithm:

A	Algorithm 2: Node-first Selection Algorithm					
_	Given: Jobs $\mathcal{J} = \{1,, J\}$					
	Init: $Q_u = 0, \forall u \in V_P; Q_{uv} = 0, \forall (u, v) \in E_P;$					
	$U = \mathcal{J}; p = 1;$					
1	while $U \neq \emptyset$ do					
2	$W_{cp}^{j} = \begin{bmatrix} \frac{Q_{u} + \sum_{l=1}^{J} c_{l}^{j}}{\mu_{u}}, \forall u \in V_{P} \end{bmatrix}, \forall j \in U;$					
3	$\left[u^{j}, C_{cp}^{j}\right] = selectNode(V_{P}, W_{cp}^{j}), \forall j \in U;$					
4	$W_{tx}^{j} = \left \frac{Q_{uv} + d_{0}^{j}}{\mu_{uv}}, \forall (u, v) \in E_{P} \right , \forall j \in U;$					
5	$[H_1^j, C_{tx,1}^j] = findShoPath(G_P, s^j, u^j, W_{tx}^j), \forall j \in U;$					
6	$W_{tx}^{j} = \left \frac{Q_{uv} + d_{Lj}^{j}}{\mu_{uv}}, \forall (u, v) \in E_{P} \right , \forall j \in U;$					
7	$[H_2^j, C_{tx,2}^j] = findShoPath(G_P, u^j, t^j, W_{tx}^j), \forall j \in U;$					
8	$j_p = \arg\min_{j \in U} C_{tx,1}^j + C_{cp}^j + C_{tx,2}^j;$					
9	$Q_{u^j} \leftarrow Q_{u^j} + \sum_{l=1}^{L^{j_p}} c_l^{j_p};$					
10	$Q_{uv} \leftarrow Q_{uv} + d_0^{j_p}, \forall (u,v) \in H_1^{j_p};$					
11	$Q_{uv} \leftarrow Q_{uv} + d_{L^j}^{j_p}, \forall (u,v) \in H_2^{j_p};$					
12	$p \leftarrow p + 1;$					
13	$\bigcup U \leftarrow U \setminus \{j_p\};$					
14	4 end					
	Output: Priority & Routing: $[j_1 > \cdots > j_J]$ &					
	$[H_1^{\jmath_p} \to H_2^{\jmath_p}, \forall p = 1,, J]$					

In line 2, W_{cp}^{j} is the vector of computation completion time when the entire layer of job j is assigned to each node. In line 3, for each job, the node with shortest computation completion time is selected together with the corresponding completion time. In line 4, the weight vector W_{tx}^{j} contains the transmission completion time at each link if the input data of each job is transferred over the link. In line 5, the path from source to the best node (selected in line 3) found such that the input data are delivered to the best node with minimum transmission latency. Similarly, in lines 6-7, the path from the best node to the destination is found with respect to the output data, i.e., the shortest path from the best node to the destination in terms of transmission latency. In line 8, the job with the earliest completion selected assuming that each job is computed at the node selected in line 3, and the input and output data are delivered along the paths found in lines 5 and 7, respectively. The route of the selected job is fixed with priority corresponding to p (the lower, the higher priority), and the unfinished tasks Q are updated based on the fixed path of the job. In addition, the selected job is removed from the set U that contains unassigned jobs. This is repeated until all the jobs are assigned. Therefore, the output of this algorithm gives the priority and the path (with single node selection) for every job.

This algorithm puts an emphasis on the computation by first selecting the node with earliest completion if the entire layer of a job is assigned to a single node. Although the algorithm may be able to give out a solution quickly, there are two drawbacks. First, it assigns the entire layer to a single node, and hence, in situations where some layers need to be split, the algorithm may perform poor. Second, in the setting where data rates are low, the transmission time may become a substantial element in the completion time. In this case, the node-first selection strategy may incur large transmission latency, eventually leading to large completion time.

REFERENCES

- K. B. Letaief, W. Chen, Y. Shi, J. Zhang, and Y.-J. A. Zhang, "The Roadmap to 6G: AI Empowered Wireless Networks," *IEEE Communications Magazine*, vol. 57, no. 8, pp. 84–90, 2019.
- [2] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size," *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: http://arxiv.org/abs/1602.07360
- [3] 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," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/ abs/1704.04861
- [4] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for mobilenetv3," in 2019 IEEE/CVF International Conference on Computer Vision (ICCV), 2019, pp. 1314–1324.
- [5] P. Ren, Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," vol. 54, no. 4, 2021. [Online]. Available: https://doi.org/10.1145/3447582
- [6] E. Baccour, A. Erbad, A. Mohamed, M. Hamdi, and M. Guizani, "RL-PDNN: Reinforcement Learning for Privacy-Aware Distributed Neural Networks in IoT Systems," *IEEE Access*, vol. 9, pp. 54872–54887, 2021.
- [7] S. Disabato, M. Roveri, and C. Alippi, "Distributed Deep Convolutional Neural Networks for the Internet-of-Things," *IEEE Transactions on Computers*, vol. 70, no. 8, pp. 1239–1252, 2021.
- [8] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.
- [9] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of Edge Computing and Deep Learning: A Comprehensive Survey," *IEEE Communications Surveys Tutorials*, vol. 22, no. 2, pp. 869–904, 2020.
- [10] Q. Liu and T. Han, "DARE: Dynamic Adaptive Mobile Augmented Reality with Edge Computing," in *ICNP*, 2018, pp. 1–11.
- [11] T. Tan and G. Cao, "Deep learning video analytics through edge computing and neural processing units on mobile devices," *IEEE Transactions* on *Mobile Computing*, pp. 1–1, 2021.
- [12] W. Wu, P. Yang, W. Zhang, C. Zhou, and X. Shen, "Accuracy-guaranteed collaborative dnn inference in industrial iot via deep reinforcement learning," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 7, pp. 4988–4998, 2021.
- [13] E. Di Pascale, I. Macaluso, A. Nag, M. Kelly, and L. Doyle, "The Network As a Computer: A Framework for Distributed Computing Over IoT Mesh Networks," *IEEE Internet of Things Journal*, vol. 5, no. 3, pp. 2107–2119, 2018.
- [14] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "DeepThings: Distributed Adaptive Deep Learning Inference on Resource-Constrained IoT Edge Clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.
- [15] 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, 2017, pp. 1396–1401.

- [16] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Toward collaborative inferencing of deep neural networks on internet-of-things devices," *IEEE Internet of Things Journal*, vol. 7, no. 6, pp. 4950–4960, 2020.
- [17] R. Stahl, Z. Zhao, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Fully distributed deep learning inference on resourceconstrained edge devices," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, D. N. Pnevmatikatos, M. Pelcat, and M. Jung, Eds., 2019, pp. 77–90.
- [18] F. Xue, W. Fang, W. Xu, Q. Wang, X. Ma, and Y. Ding, "EdgeLD: Locally Distributed Deep Learning Inference on Edge Device Clusters," in *IEEE International Conference on High Performance Computing and Communications*, 2020, pp. 613–619.
- [19] Y. Chang, X. Huang, Z. Shao, and Y. Yang, "An Efficient Distributed Deep Learning Framework for Fog-Based IoT Systems," in 2019 IEEE Global Communications Conference (GLOBECOM), 2019, pp. 1–6.
- [20] W. He, S. Guo, S. Guo, X. Qiu, and F. Qi, "Joint dnn partition deployment and resource allocation for delay-sensitive deep learning inference in iot," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9241–9254, 2020.
- [21] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi, "Learning to dispatch for job shop scheduling via deep reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1621–1632.
- [22] Y. Huang, X. Qiao, S. Dustdar, J. Zhang, and J. Li, "Toward decentralized and collaborative deep learning inference for intelligent iot devices," *IEEE Network*, vol. 36, no. 1, pp. 59–68, 2022.
- [23] Y. Huang, X. Qiao, W. Lai, S. Dustdar, J. Zhang, and J. Li, "Enabling dnn acceleration with data and model parallelization over ubiquitous end devices," *IEEE Internet of Things Journal*, pp. 1–1, 2021.
- [24] M. Xue, H. Wu, G. Peng, and K. Wolter, "Ddpqn: An efficient dnn offloading strategy in local-edge-cloud collaborative environments," *IEEE Transactions on Services Computing*, vol. 15, no. 2, pp. 640–655, 2022.
- [25] Z. Xu, L. Zhao, W. Liang, O. F. Rana, P. Zhou, Q. Xia, W. Xu, and G. Wu, "Energy-aware inference offloading for dnn-driven applications in mobile edge clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 4, pp. 799–814, 2021.
- [26] M. R. Garey and D. S. Johnson, Computers and Intractability; A Guide to the Theory of NP-Completeness. USA: W. H. Freeman & Co., 1990.
- [27] Z. Cao, S. S. Panwar, M. Kodialam, and T. V. Lakshman, "Enhancing mobile networks with software defined networking and cloud computing," *IEEE/ACM Trans. on Netw.*, vol. 25, no. 3, pp. 1431–1444, 2017.
- [28] J. Zhang, A. Sinha, J. Llorca, A. M. Tulino, and E. Modiano, "Optimal control of distributed computing networks with mixed-cast traffic flows," *IEEE/ACM Trans. on Networking*, vol. 29, no. 4, pp. 1760–1773, 2021.
- [29] Q. Wang, J. Fu, J. Wu, B. Moran, and M. Zukerman, "Energy-efficient priority-based scheduling for wireless network slicing," in 2018 IEEE Global Communications Conference (GLOBECOM), 2018, pp. 1–6.
- [30] M. Jiang, M. Condoluci, and T. Mahmoodi, "Network slicing management amp: prioritization in 5g mobile systems," in 22th European Wireless Conference, 2016, pp. 1–6.
- [31] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Opti*mization. New York, NY, USA: Wiley-Interscience, 1999.

PLACE PHOTO HERE Sehun Jung is an Undergraduate Student in the Department of Computer Science and Engineering, Konkuk University, Seoul, South Korea. His research interests include reinforcement learning and its applications to neural network training and computation.

Hyang-Won Lee (S'02-M'08) received the B.S., M.S., and Ph.D. degrees all in Electrical Engineering and Computer Science from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 2001, 2003, and 2007, respectively. He was a Post-Doctoral Research Associate with the Massachusetts Institute of Technology (MIT) from 2007 to 2011, and a Research Assistant Professor with KAIST from 2011 and 2012. He was also a Visiting Research Scientist at the Laboratory of Information and Decision Systems (LIDS), MIT

from 2017 to 2018. He is presently a Professor with the Deputtment of Computer Science and Engineering, Konkuk University, Seoul, South Korea. His research interests are in the areas of network optimization and algorithms, reinforcement learning and formal verification.