Practical Modeling for Split DNN Inference on Near-Edge Accelerators

Hao Liu*, Mohammed E. Fouda†, Ahmed M. Eltawil* and Suhaib A. Fahmy*
*King Abdullah University of Science and Technology (KAUST), Thuwal, Saudi Arabia
hao.liu@kaust.edu.sa
†Compumacy for Artificial Intelligence Solutions, Cairo, Egypt

***Abstract*—Splitting complex model inference between multiple computing devices can overcome latency and energy constraints at the edge. Newer edge accelerator devices with higher computing capacity and energy efficiency, enable more fine-grained offload throughout layers of the network, leading to the potential for multiple split configurations. However, optimizing a DNN for inference across networked devices requires a precise performance model that can guide design choices. In this paper, we demonstrate that existing models for computation and communication latency are inaccurate due to system considerations and propose a new empirical model based on structured benchmarking, considering data ingestion overhead due to transfers between devices as well as within a device for data to reach the GPU. We validate our split inference performance model using VGG16 and ResNet50 networks on two heterogeneous platforms, showing it achieves a mean absolute error of no more than 4% for both DNNs, significantly outperforming previous models with errors of more than 15%. We also validate the practical utility of our model by incorporating it into existing split inference search algorithms under multi-split, dynamic bandwidth, and multi-tenant scenarios, demonstrating its effectiveness in navigating the split inference search space.**

***Index Terms*—Deep learning inference, hardware acceleration, edge computing.**

I. Introduction

Resource-constrained edge devices struggle to perform low-latency inference with complex DNN models. Split inference addresses this by partitioning DNN execution between the edge and cloud, enabling efficient inference on a constrained edge device while leveraging cloud performance [1]–[4]. While the performance benefit of cloud computing can be enough to overcome the newly introduced communication latency, this communication latency remains significant, precluding split inference from use in a variety of latency-sensitive scenarios.

New generation edge hardware based on GPUs such as the Nvidia Jetson Orin Nano [5] and AGX Orin [6], FPGAs [7], or in-memory computing devices [8] dramatically increases computational capability at or near the edge. These offer an intermediate offload target for split inference by constrained edge devices, potentially in combination with a traditional cloud where necessary, or with multiple splits across such accelerators. Unlike traditional "cloudlet" servers further up the network, these can be deployed within a few hops of the edge, potentially integrated into infrastructure such as mobile base-stations [9], [10]. This would allow inference deployments with multiple splits involving edge device(s), near-edge accelerator(s), and potentially, the cloud.

Edge accelerators do not offer as much raw computational performance as cloud servers, and when multiple splits are used, communication overheads must be carefully considered. Given that for many DNNs intermediate activations can be large [1], [11]–[13], inserting splits can have a significant adverse effect on communication latency. Compressing intermediate activation data has been explored to reduce communication latency, using tensor low-rank decomposition [14], quantization [15], [16], and inserting autoencoders [2], [17].

Considering the design space of multiple split positions, the relative computational capabilities of the platforms, and communication compression, optimizing latency and energy is challenging, requiring precise models. Existing split DNN search space optimization algorithms [3], [18]–[20] all rely on such fundamental computation and communication models. In this work, we develop a detailed, structured benchmarking approach to enable more accurate modeling and characterization of split inference systems using near-edge accelerators to support such optimizations, which are orthogonal to our contribution. We focus on single-model inference scenarios as in prior work [2], [4], [19], [21], though we show how the model can be applied to other scenarios (Sec. V-D). Our contributions in this paper are:

- Show experimentally that autoencoder-based communication compression offers higher accuracy and compression ratio than quantization and tensor low-rank decomposition.
- Propose a precise computation and communication formulation that matches measurements from practical deployments, considering system characteristics of GPU data ingestion and packet-based network data transfer.
- Build a real system comprising multiple accelerators and validate our proposed split DNN approach with the VGG16 and ResNet50 models on the CIFAR-100 dataset in a single-split setup with different batch sizes, reporting our model's accuracy.
- Show use of the performance model to guide multi-split, dynamic bandwidth, and multi-tenant search scenarios.

II. Related Work

Accurate end-to-end modeling is required to optimize split DNN inference. Graph-theoretic algorithms [3], [18], deep reinforcement learning [22], dynamic programming [20], and other optimization algorithms [23] have been applied to tame the search space, but the underlying computation and

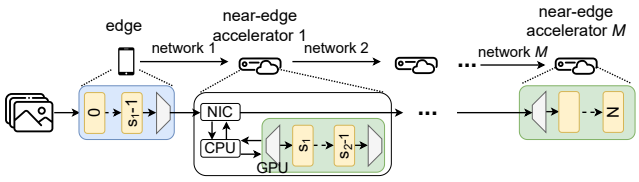


Fig. 1: Split DNN system architecture.

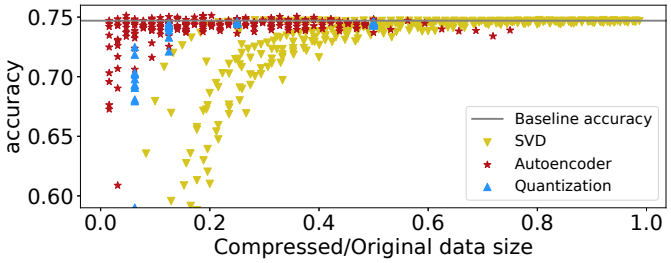


Fig. 2: Accuracy comparison of VGG16 on the CIFAR-100 dataset with autoencoder, low-bit quantization, and SVD.

communication models can be overly simplistic, leading to a discrepancy between theoretical optima and achievable performance. The work in [21] eliminates offline profiling by heuristically adapting the split during runtime to minimize latency, but this approach does not scale to multiple splits.

A common approach for modeling the computation latency of DNN segments is to sum profiled latencies of individual layers. This approach is used in various split methods, including graph-theoretic split search [3], [12], [18], co-optimization of wordlengths and split points [15], and Neural Architecture Search (NAS) [24]. However, this approximation is inaccurate due to compilers applying cross-layer inference optimization in hardware. To avoid the cost of profiling, several works instead build latency models. [25] modeled execution latency considering resource availability, partition point, and cross-DNN interference, but only considered CPUs. Other approaches predict per-layer execution latency using regression models over layer types [1], [26], fully-connected neural networks across platforms [27], collaborative filtering [4], or formulations based on layer computational load and device peak performance [28]. These approaches all predict the latency of an individual layer, so when summed to determine segment latency are inherently less accurate than profiling. Offline profiling of all possible DNN segments was used in [2], but this is not feasible when considering a large number of DNN segments and multiple split positions with various autoencoder configurations.

Communication latency is often unrealistically modeled as data size divided by bandwidth. Round-trip latency was additionally considered in [13], [16], [19], where [16] quantized inputs at each split, [13] co-optimized financial costs, and [19] used autoencoders, extending to two splits. However, such a communication model does not accurately reflect the packet-based nature of network transmission. Network wake-up time and transmission time were considered in [29], but without a general formulation. DNN segments exchange data that exceeds the size of a single packet so transport layer characteristics can have a significant impact on model accuracy.

Finally, the latency of data movement between the network interface (NIC) and GPU/accelerator within a compute platform is not negligible, especially for resource-limited edge and near-edge accelerators [30]. This has been considered in the cloud GPU setting [31], but not in near-edge accelerators, where this factor is even more significant.

III. Motivation

A. Overview

We consider a system comprising an edge device and M near-edge accelerator devices, as shown in Fig. 1. The DNN is split at M positions leading to $(M+1)$ partitions, which are executed on the edge and M near-edge accelerators in a chain. Input data is captured by the edge, where the first DNN partition is executed. Intermediate activations are transmitted from the edge to the first near-edge accelerator that executes the second DNN partition. This process continues sequentially, with activations forwarded to subsequent accelerators until the Mth near-edge accelerator executes the $(M+1)$th partition to complete inference. Autoencoders are used to compress intermediate activations at each split.

B. Communication Compression

Splitting more than once can introduce significant communication latency. Hence, it is critical to compress intermediate activations with minimal accuracy loss in order for such a multi-split setup to be feasible. Quantization and Singular Value Decomposition (SVD) have been proposed in the past, and require no modifications to the model [14], [15]. Alternatively, autoencoders can be added and fine-tuned [2], [17], [32]. In this setting, an encoder is deployed at the source to compress the transmitted data, and a decoder is deployed at the sink to decompress. We explore the lightweight convolutional autoencoder design in [17], and adjust convolutional layer parameters to control the compression ratio and accuracy trade-off.

To understand how well these compression methods perform in practice, we evaluate their compression ratios and impact on model accuracy for a single split applied to VGG16 on the CIFAR-100 dataset in Fig. 2. We evaluate quantization to 2, 4, 8, and 16 bits with fine-tuning, explore a range of tensor ranks for SVD, and include a range of autoencoder configurations. Autoencoders offer much better compression with marginal accuracy drop, while other methods suffer much more significant accuracy drop.

C. Limitations of Previous DNN Split Models

Edge-cloud split inference has traditionally been considered in the context of edge devices with low computational capability. Newer edge accelerator hardware with much higher

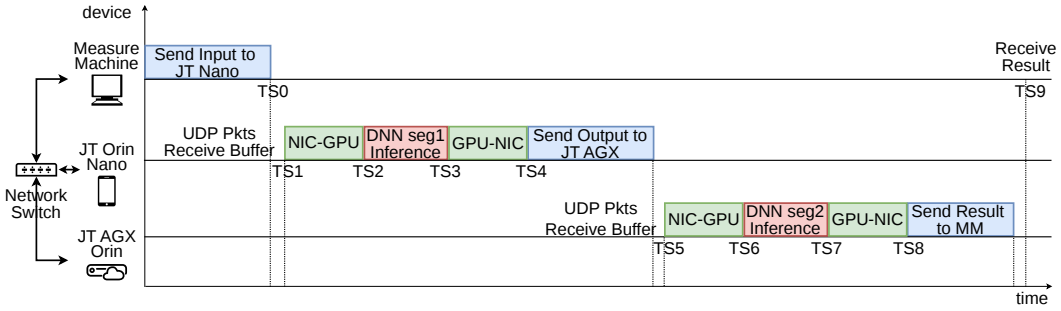


Fig. 3: Illustration of measurement setup showing the measurement machine that initiates measurements and the measured components.

capability, introduces more opportunities for latency and energy optimization, but also demands a more accurate model to guide DNN split inference. Previous performance models neglect important system considerations such as cross-layer DNN inference optimizations in hardware, data movement between the network and accelerator hardware, and packet-based network transmission, leading to inaccurate predictions, which are more problematic in a multi-split setting.

To demonstrate this, we measure the real inference latency of VGG16 pretrained on the CIFAR-100 dataset with a single split (thus two DNN segments) in an edge and near-edge accelerator system as shown in Fig. 3. The edge device is an Nvidia Jetson Orin Nano, which executes the first DNN segment, while the near-edge accelerator is an Nvidia Jetson AGX Orin, which executes the second DNN segment. Intermediate activations are transmitted over a 1 Gbps wired network. Consistent with prior work [33], [34], we use User Datagram Protocol (UDP) for activation transmission. While these studies consider loss-aware split DNN inference, we adopt a simple policy that discards an input if any of its UDP packets are lost. Since DNN segments are statically determined, the number and size of input and output UDP packets for communication are known upfront. A separate measurement machine with an Intel Core i7-11700 CPU at 2.50GHz running Ubuntu 20.04, wired to the same switch, is used to measure the end-to-end latency. This measurement machine sends the DNN input to the edge device via consecutive UDP packets and records the timestamp of the last packet ($TS0$). This is to reduce the influence of the measurement machine on forming and sending data, and allows us to characterize the edge device's ingestion characteristics, should it also be used for offload. When the edge device receives a full input, it moves the data to its GPU for inference on the first DNN segment, before moving the resulting intermediate activations to its NIC for transmission to the near-edge accelerator. Meanwhile, the edge device records the start and end times of each step ($TS1$ to $TS4$) as shown in Fig. 3. A similar process occurs at the near-edge accelerator, which records timestamps $TS5$ to $TS8$. Once the near-edge accelerator completes execution, the result is transmitted to the measurement machine which records the timestamp of receiving the result ($TS9$). Note we only compare

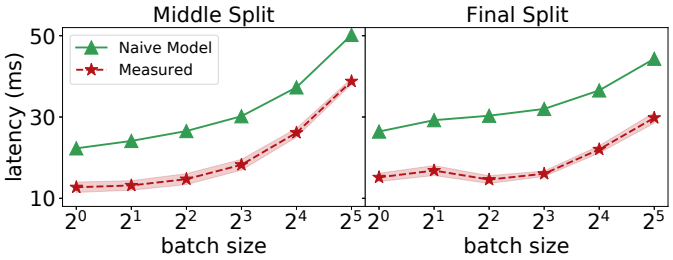


Fig. 4: Comparison between measured latency and predicted latency of the naive model on VGG16.

timestamps recorded by the same device. The measurement machine measures the interval between sending the last UDP packet to the edge and receiving the result from the near-edge accelerator as the end-to-end latency ($TS9 - TS0$). During inference, the measurement machine does not interfere with the edge and near-edge accelerator, completing 200 closed-loop measurements. Without loss of generality, we select both the middle and last split positions among all the split candidates for a single split and conduct the measurement with various batch sizes. Fig. 4 compares the average measured end-to-end latency with the latency predicted by a naive model as used in previous work [19]. The results show a significant discrepancy due to the inaccuracy of the computation and communication models, and NIC-to-GPU data movement being ignored.

IV. Proposed Approach

Based on this analysis, we propose an end-to-end latency model for the system in Fig. 1, that accounts for packet-based communication, cross-layer inference optimization, and NIC-GPU data movement. To demonstrate that our formulation is not restricted to a single hardware architecture, our empirical measurements, baseline profiling, and subsequent model validations are conducted and verified across two heterogeneous computing platforms, namely the Nvidia Jetson Orin Nano and the Jetson AGX Orin. The overall split DNN execution process is shown in Fig. 5. We map a DNN comprising $N + 1$ layers, and therefore N potential split positions to M near-edge accelerators. A split $s_m \in \{1..N\}$ can be implemented using an autoencoder $k_m \in K$, where K is the

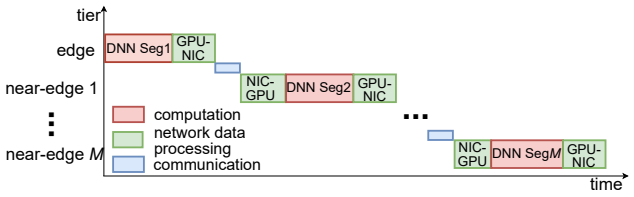


Fig. 5: Overall split DNN execution process. The edge executes the first DNN segment, moves intermediate activations from GPU to NIC then sends them as UDP packets to the first near-edge accelerator. This receives packets, moves data to the GPU to execute the second DNN segment, and transfers the resulting activations to the second near-edge accelerator. This process continues sequentially across the chain of M near-edge accelerators until inference completion.

TABLE I: Model coefficients for computation. All coefficients are in $\times 10^{-4}$ ms/layer.

Platform (Y)	**VGG16**		**ResNet50**	
	$C_{0,Y}$	$C_{1,Y}$	$C_{0,Y}$	$C_{1,Y}$
Orin Nano ($\mathcal{E}$)	6.75	-2.39	5.16	0.77
AGX Orin ($\mathcal{N}$)	5.34	1.86	5.20	-0.54

set of possible autoencoder configurations and $m \in \{1..M\}$ since there is a split for each near-edge accelerator. We consider the computation latency, the communication latency, and the data processing latency between the NIC and GPU in each device. Our framework provides a general model for these three latency components. Specifically, the computation model in Sec. IV-A explicitly incorporates the compressing autoencoder, where the encoder and decoder are simply considered as additional DNN layers. Correspondingly, the communication model in Section IV-B and the network data processing model in Section IV-C account for the data volume after compression.

A. Computation Latency Model

First, we profile execution of the individual $N+1$ DNN layers (or portions where splits are feasible), as well as different autoencoders, K, on the different hardware devices: edge ($\mathcal{E}$) and near-edge accelerators ($\mathcal{N}$). We denote the execution latency of the nth layer or portion on the edge and near-edge as $Texec_{n,\mathcal{E}}$, and $Texec_{n,\mathcal{N}}$, respectively. For a given autoencoder configuration k, we denote the execution time of the encoders on the edge and accelerators as $Texec_{enc_k,\mathcal{E}}$ and $Texec_{enc_k,\mathcal{N}}$ respectively. We also denote the execution time of the decoders on the accelerators as $Texec_{dec_k,\mathcal{N}}$ (the edge is never a sink). Second, we profile the execution latency of combined DNN segments with varying numbers of layers on each device, to determine the latency discrepancy between summed and combined executions. Based on this, we derive linear regression constants $C_{\{0,1\},\mathcal{E}}$ (in seconds per layer), and $C_{\{0,1\},\mathcal{N}}$ (in seconds) for the edge and near-edge accelerators, respectively. $C_{0,\mathcal{E}} \times s_1 + C_{1,\mathcal{E}}$ represents a latency correction term (in seconds) for a segment with s_1 layers. These constants

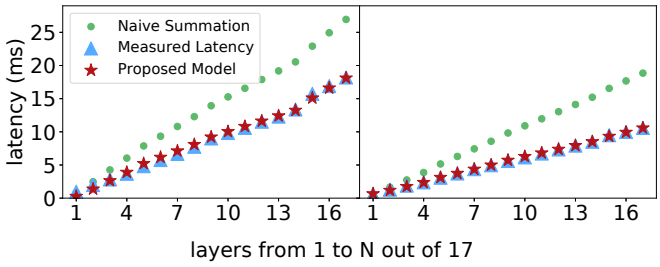


Fig. 6: Comparison of measured, naive summation, and our model's predicted latencies for ResNet50 (batch size 1) on Jetson Orin Nano (←) and AGX Orin (→).

are DNN and device dependent. Fig. 6 shows that this linear regression offers an accurate approximation of actual segment latency, compared to traditional summed latency approaches, with mean absolute percentage errors of 9.28% and 2.07% for Jetson Orin Nano and AGX Orin, respectively. These derived values for the computation model are shown in Table I.

The total computation latency of the first portion of DNN layers and the first split encoder on the edge is hence:

$$Texec_{\mathcal{E}} = \sum_{i=0}^{s_1-1} Texec_{i,\mathcal{E}} + Texec_{enc_{k1},\mathcal{E}} - \left(C_{0,\mathcal{E}} \times s_1 + C_{1,\mathcal{E}}\right) \tag{1}$$

where s_1 is the first split position and k_1 is the autoencoder configuration for the first split.

The execution latency of the decoder, the portion of DNN layers, and the encoder on the mth near-edge accelerator is then:

$$\begin{aligned} Texec_{\mathcal{N}_m} = Texec_{dec_{k_m},\mathcal{N}} + \sum_{i=s_m}^{s_{m+1}-1} Texec_{i,\mathcal{N}} \\ + Texec_{enc_{k_m},\mathcal{N}} \\ - \left(C_{0,\mathcal{N}} \times (s_{m+1} - s_m + 2) + C_{1,\mathcal{N}}\right) \end{aligned} \tag{2}$$

where $m \in \{1, 2, \ldots, M\}$, s_m is the chosen mth split position and k_m is the chosen autoencoder configuration for the mth split. The Mth near-edge accelerator does not have an encoder, while Eq. 2 can be readily adapted accordingly.

Hence, the total execution latency is:

$$Texec = Texec_{\mathcal{E}} + \sum_{m=1}^{M} Texec_{\mathcal{N}_m} \tag{3}$$

B. Communication Latency Model

Previous work failed to consider the latency overhead due to packet-based data communication. Our latency model comprises two components: the latency for the first packet to be sent and arrive at the receiver and the latency from that first packet until reception of the last packet. We measure communication latency without DNN execution in Fig. 7. A measurement machine (receiver) exchanges UDP packets with the edge device over a wired connection. The receiver sends UDP packets to the sender, and records $TS10$ for the last packet

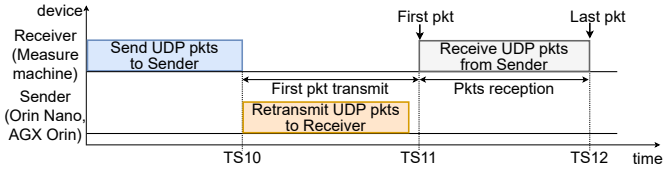


Fig. 7: Communication measurement setup.

TABLE II: Model coefficients for communication.

Platform (Y)	$Q_{0,Y}$ ($\times 10^{-3}$ **ms/KB**)	$Q_{1,Y}$ **(ms)**	$T_{FP,Y}$ **(ms)**
Orin Nano ($\mathcal{E}$)	8.58	-0.57	1.59
AGX Orin ($\mathcal{N}$)	8.32	-0.54	1.67

sent, which approximates the sender's first packet transmission due to negligible single-packet latency over a wired connection. After receiving packets, the sender retransmits them to the receiver. The receiver records $TS11$ and $TS12$ for the first and last received packets. We compute the first packet transmission latency as $TS11 - TS10$, and packet reception latency as $TS12 - TS11$. Fig. 8 (a) illustrates these latencies across varying data sizes for sender-receiver systems tested separately. We employ linear regression to model packet reception latency ($TS12 - TS11$) as a function of data size, while modeling first packet transmission latency ($TS11 - TS10$) as a constant averaged across data sizes. The mean absolute percentage errors of the packet reception latency are 2.54% and 3.12% for Jetson Orin Nano and AGX Orin, respectively. Fig. 8 (b) compares the absolute percentage prediction error of our model with a naive communication model based on bandwidth measured using iperf3 [35], demonstrating the accuracy of our model. The Jetson Orin Nano and AGX Orin exhibit comparable communication latencies, due to their similar CPU performance.

We profile the first packet transmission latency from the upstream edge device or $(m-1)$th near-edge accelerator to the downstream mth near-edge accelerator, denoted as $T_{FP,\mathcal{N}}$. We also profile the packet reception latency for the mth near-edge accelerators, and derive the linear regression constants as $Q_{\{0,1\},\mathcal{N}}$. We denote the output data size of autoencoder k_m as d_{k_m}. These derived values for the communication model are shown in Table II.

Thus, we model the communication latency of the network from the edge device or $(m-1)$th near-edge accelerator to the mth near-edge accelerator as follows:

$$Tcomm_{\mathcal{N}_m} = \left(Q_{0,\mathcal{N}} \times d_{k_m} + Q_{1,\mathcal{N}}\right) + T_{FP,\mathcal{N}} \quad (4)$$

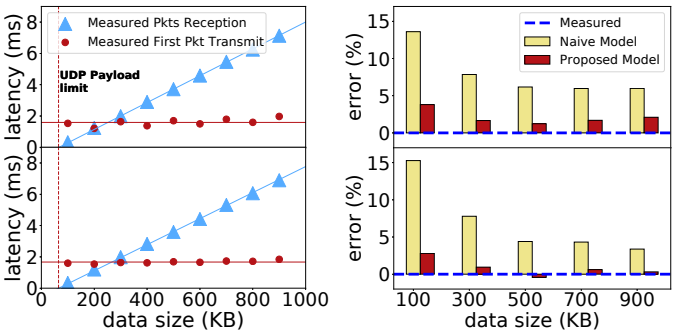
where $m \in \{1, 2, \ldots, M\}$.

The total communication latency is then:

$$Tcomm = \sum_{m=1}^{M} Tcomm_{\mathcal{N}_m} \quad (5)$$

C. Network Data Processing Model

Network data processing latency is another significant source of latency. It is the time required to transfer input data from the



(a) Measured first packet transmission and reception latency of the Jetson Orin Nano (↑) and AGX Orin (↓).

(b) Comparison of absolute percentage error between naive and proposed model for Jetson Orin Nano (↑) and AGX Orin (↓).

Fig. 8: Communication model for near-edge platforms.

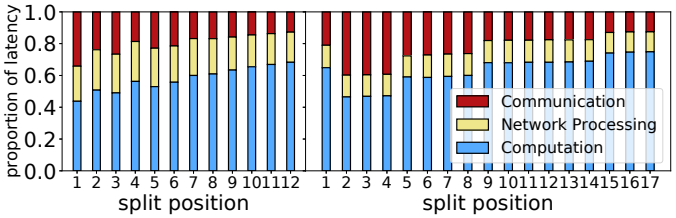


Fig. 9: Proportions of total end-to-end latency due to computation, network data processing, and communication for VGG16 (←) and ResNet50 (→).

NIC to CPU for packet ingestion and tensor reshaping, then to the GPU for DNN execution via on-chip interconnect (NIC-GPU latency), and the reverse (GPU-NIC latency). To quantify its significance, we split VGG16 and ResNet50 at various single split positions, and measured end-to-end execution latency and network data processing latency using the experimental setup in Fig. 3. The edge records $TS1$ upon receiving all UDP packets from the measurement machine, and $TS2$ after moving data into the GPU for DNN execution. NIC-GPU latency is $TS2 - TS1$. It records DNN execution completion ($TS3$), and the point where output data is ready for transmission ($TS4$). GPU-NIC latency is thus $TS4 - TS3$. The same is done at the near-edge accelerator for $TS5$ to $TS8$. The proportion of total end-to-end latency consumed by data processing is thus $\frac{(TS2-TS1)+(TS4-TS3)+(TS6-TS5)+(TS8-TS7)}{TS9-TS0}$. Fig. 9 shows this constitutes approximately 25% and 15% for VGG16 and ResNet50, respectively, across different split configurations.

To confirm that this measurement is not DNN-specific, we use a simple single ReLU layer for profiling. We validate this approximation by comparing the latencies of this setup against a convolutional DNN with five layers, adjusting the channel configurations of the first and last layers to achieve different data sizes. As shown in Fig. 10, the ReLU measurement shows acceptable correspondence to a full convolutional segment on both the Jetson Orin Nano and AGX Orin, for both NIC-GPU and GPU-NIC latencies, across different data sizes. Consequently, we can compute a linear regression for

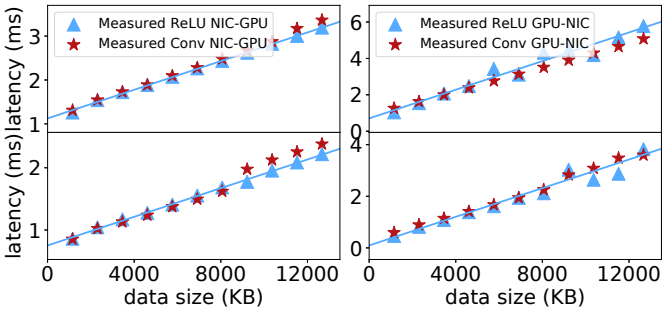


Fig. 10: Comparison of ReLU-based and Conv-based NIC-GPU (←) and GPU-NIC (→) latencies on Jetson Orin Nano (↑) and AGX Orin (↓).

the NIC-GPU and GPU-NIC latencies using a single-layer ReLU network as a function of data size. The mean absolute percentage errors are 2.21% and 4.66% for NIC-GPU latency on Jetson Orin Nano and AGX Orin, and 8.34% and 7.53% for GPU-NIC latency, respectively.

We profile the NIC-GPU and GPU-NIC latency on the edge and the near-edge accelerators, and employ linear regression for each. The NIC-GPU regression constants are denoted as $P_{\{N_0,N_1\},\mathcal{N}}$ for the near-edge accelerators, and the GPU-NIC constants as $P_{\{G_0,G_1\},\{\mathcal{E},\mathcal{N}\}}$ for the edge and the near-edge accelerators. These derived values for the network data processing model are shown in Table III.

We incorporate this network data processing latency into the model. At the edge:

$$Tproc_{\mathcal{E}} = P_{G_0,\mathcal{E}} \times d_{k_1} + P_{G_1,\mathcal{E}} \tag{6}$$

At the near-edge accelerator:

$$\begin{aligned} Tproc_{\mathcal{N}_m} = &\left(P_{N_0,\mathcal{N}} \times d_{k_m} + P_{N_1,\mathcal{N}}\right) \\ &+ \left(P_{G_0,\mathcal{N}} \times d_{k_{m+1}} + P_{G_1,\mathcal{N}}\right) \end{aligned} \tag{7}$$

where $m \in \{1, 2, \ldots, M\}$.

The total network processing latency is:

$$Tproc = Tproc_{\mathcal{E}} + \sum_{m=1}^{M} Tproc_{\mathcal{N}_m} \tag{8}$$

And the total latency of the system is:

$$T_{total} = Texec + Tcomm + Tproc \tag{9}$$

Recall that this can be extended to arbitrary splits and the devices can be homogeneous or heterogeneous.

We calculate the total energy consumption as

$$\begin{aligned} E_{total} = &\left(Texec_{\mathcal{E}} + Tproc_{\mathcal{E}}\right) \times p_{\mathcal{E}} + \\ &\sum_{m=1}^{M} \left(Texec_{\mathcal{N}_m} + Tproc_{\mathcal{N}_m} + Tcomm_{\mathcal{N}_m}\right) \times p_{\mathcal{N}_m} \end{aligned} \tag{10}$$

where $p_{\mathcal{E}}$ and $p_{\mathcal{N}}$ represent the power consumption during execution of the edge and near-edge respectively. These power consumption values are profiled with `jetson-stats` [36] using the same structured benchmarking approach, which allows for accurate energy prediction given the precision of our runtime model.

TABLE III: Model coefficients for network data processing.

Platform (Y)	**NIC→GPU**		**GPU→NIC**	
	$P_{N_0,Y}$ ($\times 10^{-4}$ **ms/KB**)	$P_{N_1,Y}$ **(ms)**	$P_{G_0,Y}$ ($\times 10^{-4}$ **ms/KB**)	$P_{G_1,Y}$ **(ms)**
Orin Nano ($\mathcal{E}$)	1.63	1.12	3.94	0.70
AGX Orin ($\mathcal{N}$)	1.15	0.75	2.77	0.09

D. Profiling New DNNs and Devices

Our model can be applied to a new DNN and/or device $\mathcal{D}$ as follows. For the computation latency model, profile execution latency of individual DNN layers and a limited number of selected combined DNN segments—this is both DNN and device dependent. In practice, we find that selecting the same number of combined segments as individual layers is sufficient to accurately model the discrepancy between summed and combined latencies. Specifically, for a DNN with N potential splits, profiling $2N + 1$ configurations is adequate. Based on these measurements, obtain $C_{\{0,1\},\mathcal{D}}$.

For the communication model, profile first packet transmission latency $T_{FP,\mathcal{D}}$ and packet reception latency per device $Q_{\{0,1\},\mathcal{D}}$. For the network data processing model, profile the NIC-GPU and the GPU-NIC latency for each device, denoted as $P_{\{N_0,N_1\},\mathcal{D}}$ and $P_{\{G_0,G_1\},\mathcal{D}}$ respectively. We profiled for 20 data sizes from 1–10 UDP packet(s) to cover all possible intermediate activation sizes. The communication and network data processing models are device-specific rather than DNN-specific, thus, evaluating new compression mechanisms, autoencoders, or even different DNN architectures requires no re-profiling of these infrastructural delays. Consequently, only the execution times of the modified DNN layers require re-profiling and modeling. Ultimately, the profiling overhead of communication and network data processing is effectively amortized across subsequent explorations of new split configurations.

As in prior work [11], we consider the deployed DNN applications to be predetermined, and profiling individual layers and combined segments can be fully automated through scripting, making profiling effort acceptable for an accurate model that can also be used for dynamic resource allocation.

V. Evaluation

A. Experimental Setting

Our system is based on GStreamer [37] and NNStreamer [38], [39], using PyTorch version 1.11.0 as deep learning backend. The hardware configuration comprises an Nvidia Jetson Orin Nano 8GB as the edge and an Nvidia Jetson AGX Orin 64GB as the near-edge accelerator. We employ Linux Traffic Control to simulate dynamic bandwidth, which is the underlying tool of WonderShaper [40] used in [13], [41].

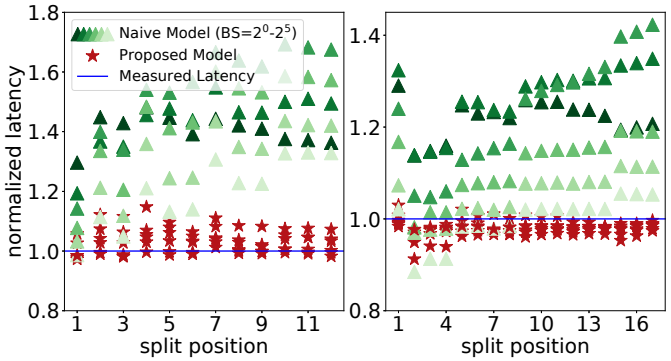


Fig. 11: Normalized latency comparison of proposed model and naive model for VGG16 (←) and ResNet50 (→) on CIFAR-100 across batch sizes and split positions.

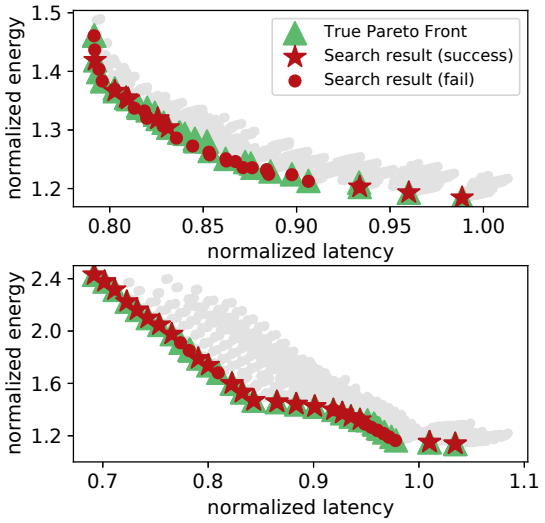


Fig. 12: Experimental results for multiple splits on VGG16 (↑) and ResNet50 (↓) for CIFAR-100 (batch size=1).

B. Validation of Proposed Model

We compare our model with previous models such as the one in [19], using VGG16 and ResNet50 pretrained on the CIFAR-100 dataset for a single split across an edge and near-edge accelerator. We evaluate inference batch sizes of 1, 2, 4, 8, 16, and 32, measuring end-to-end latency in a closed-loop manner. Fig. 11 shows predicted latencies normalized against measured latency. Our proposed model achieves mean absolute percentage errors of 3.7% and 2.2% on VGG16 and ResNet50, respectively, significantly outperforming previous models with errors of 38.9% and 15.7%. Furthermore, the maximum absolute percentage error of our model across all split positions and batch sizes remains below 15%, compared to over 60% for previous models.

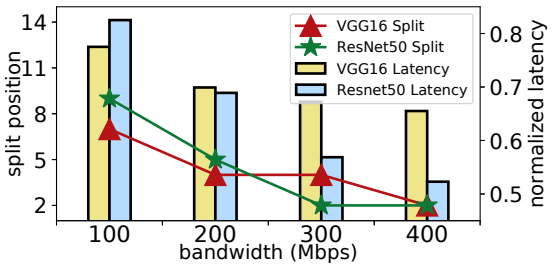
C. Extending to Multiple Split Positions

Our goal in this section is not to propose a new split-search algorithm, but rather to demonstrate that the proposed performance model can be seamlessly incorporated into existing optimization frameworks to identify near-optimal multi-split configurations. We modify the multi-split search approach in [19], which identifies the near-optimal two-split configurations with autoencoder compression under a user-defined acceptable accuracy loss threshold, optimizing end-to-end latency and energy. Specifically, [19] utilizes single-split accuracy as a proxy for two-split accuracy to shrink the search space, and then exhaustively searches the remaining space using the latency and energy models to obtain near-optimal trade-offs. Here, the second split offloads to an Nvidia A100 GPU. Fig. 12 shows determined split configurations with the proposed model for VGG16 and ResNet50, with energy and latency normalized against full execution on the edge. Grey points show the full set of possible configurations, with green points representing the true Pareto front, identified through exhaustive search. Integrating the proposed model into the existing search algorithm successfully identifies near-optimal split strategies, shown in red, the vast majority of which meet the accuracy loss threshold after fine-tuning (red stars), while a few do not (red circles). For VGG16, 28% of split strategies meet an accuracy loss threshold of 0.01, while for ResNet50, 66% of split strategies satisfy this threshold. For both DNNs, the average additional latency and energy overhead introduced by the autoencoders in these near-optimal split strategies is below 5% and 7% of the total, respectively.

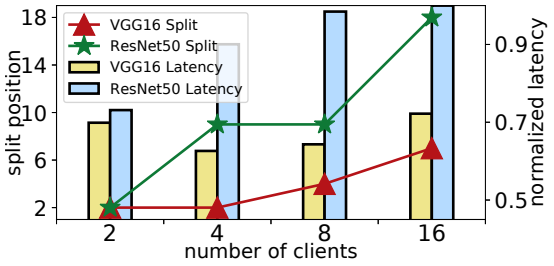
D. Use in Dynamic Bandwidth and Shared Offload Settings

Our proposed model can be used in a dynamic bandwidth scenario, adapting the split position to available network bandwidth. Fig. 13 (a) shows how different optimal split positions are identified in response to network bandwidth changes. As an example, we fix the autoencoder compression ratios at the split positions (5× for VGG16 and 20× for ResNet50), thereby avoiding an exhaustive search over all compression configurations. From 100 Mbps to 400 Mbps, the optimal split position of VGG16 moves from 7 to 4, and subsequently to 2, offloading more layers from the edge to the near-edge accelerator, due to reduced communication overhead, achieving a 22.5–34.5% reduction in total latency compared to the edge-only execution. The optimal split position for ResNet50 moves from 9 to 5, and subsequently to 2, achieving a 17.5–47.7% reduction.

Previous work has established a linear relationship between batch size and execution latency on GPUs [42]–[44], and our own measurements confirm this for these near-edge devices. By using this insight, the near-edge accelerator can dynamically modify the split position depending on the number of offloading clients. Fig. 13 (b) shows our model identifies optimal split positions in a multi-tenant scenario where multiple edge clients offload to a shared near-edge accelerator connected via a 200 Mbps network. Each client processes a single input batch through the first DNN segment and offloads the second segment to the near-edge accelerator, which aggregates intermediate activations from multiple clients to form a larger batch for inference of the second DNN segment. Our model accurately



(a) Dynamic bandwidth splitting.



(b) Multiple client sharing.

Fig. 13: Experimental results for single split on VGG16 and ResNet50 for CIFAR-100 under dynamic bandwidth and with multiple clients.

predicts execution latency for batch size 1 on both edge and near-edge accelerator, and the linear relationship provides a prediction for batched execution on the near-edge accelerator. As the number of edge clients increases from 2 to 16, the optimal split position of VGG16 moves from 2 to 4, then to 7, achieving a 27.8–37.4% reduction in total latency compared to the edge-only execution. The optimal split position of ResNet50 moves from 2 to 9, achieving a 4.7–23.2% reduction. (Note that more complex considerations on batching windows are beyond scope.)

VI. Conclusions

We have proposed an accurate end-to-end latency model for split DNN inference. By modeling cross-layer DNN optimizations, packet-based communication, and network data processing, combined with structured profiling, we were able to demonstrate accurate modeling of split DNN inference on different edge and near-edge devices. Experiments on VGG16 and ResNet50 show that our proposed model dramatically reduces latency prediction error for DNN segments, achieving mean absolute percentage errors of 3.7% and 2.2% on VGG16 and ResNet50, respectively, down from 38.9% and 15.7% for previous models. We also showed that this model can be incorporated into more complex scenarios including multiple splits, dynamic bandwidth, and multi-tenant offload to find feasible splitting strategies to guide split search. Our model can be used as a drop-in replacement within existing split optimization frameworks, enhancing their capability to make accurate splitting decisions. We plan to investigate DNN split inference across multiple homogeneous or heterogeneous edge devices with consideration of dynamic batching, as well as applying it to more complex DNNs, such as vision transformers. We believe the new embedded ML accelerators will further enable efficient distributed DNN execution by facilitating computation at the edge or near-edge, thereby reducing dependency on the cloud and improving both performance reliability and energy efficiency.

VII. Acknowledgment

This work was supported by the King Abdullah University of Science and Technology through the Competitive Research Grant program under grant URF/1/4704-01-01.

References

[1] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, pp. 615–629, 2017.

[2] A. E. Eshratifar, A. Esmaili, and M. Pedram, "BottleNet: A deep learning architecture for intelligent mobile cloud computing services," in *IEEE/ACM International Symposium on Low Power Electronics and Design*, 2019.

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

[4] A. K. Kakolyris, M. Katsaragakis, D. Masouros, and D. Soudris, "RoaD-RuNNer: Collaborative DNN partitioning and offloading on heterogeneous edge systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023.

[5] NVIDIA Corporation, "Jetson Orin Nano super developer kit," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/nano-super-developer-kit/, 2025, accessed: 2025-09-01.

[6] ——, "Jetson AGX Orin for next-gen robotics," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, accessed: 2025-09-01.

[7] Advanced Micro Devices (AMD), "AMD Versa AI edge series," https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vek280.html, 2025, accessed: 2025-09-01.

[8] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, 2020.

[9] A. Cartas, M. Kocour, A. Raman, I. Leontiadis, J. Luque, N. Sastry, J. Nuñez-Martinez, D. Perino, and C. Segura, "A reality check on inference at mobile networks edge," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2019, pp. 54–59.

[10] R. A. Cooke and S. A. Fahmy, "A model for distributed in-network and near-edge computing with heterogeneous hardware," *Future Generation Computer Systems*, vol. 105, pp. 395–409, 2020.

[11] A. E. Eshratifar, M. S. Abrishami, and M. Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, 2019.

[12] S. Wang, X. Zhang, H. Uchiyama, and H. Matsuda, "HiveMind: Towards cellular native machine learning model splitting," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 2, pp. 626–640, 2021.

[13] D. Luger, A. Aral, and I. Brandic, "Cost-aware neural network splitting and dynamic rescheduling for edge intelligence," in *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2023, pp. 42–47.

[14] J. Chen, H. Xu, and M. Li, "CollabTrans: Device-cloud collaborative inference framework for transformer-based models," in *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*, 2025.

[15] A. Banitalebi-Dehkordi, N. Vedula, J. Pei, F. Xia, L. Wang, and Y. Zhang, "Auto-Split: A general framework of collaborative edge-cloud AI," in *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2021, pp. 2543–2553.

[16] M. Almeida, S. Laskaridis, S. I. Venieris, I. Leontiadis, and N. D. Lane, "DynO: Dynamic onloading of deep neural networks from cloud to device," *ACM Transactions on Embedded Computing Systems*, 2022.

[17] J. Shao and J. Zhang, "BottleNet++: An end-to-end approach for feature compression in device-edge co-inference systems," in *IEEE International Conference on Communications Workshops*, 2020.

[18] H. Liang, Q. Sang, C. Hu, D. Cheng, X. Zhou, D. Wang, W. Bao, and Y. Wang, "DNN surgery: Accelerating DNN inference on the edge through layer partitioning," *IEEE Transactions on Cloud Computing*, pp. 3111–3125, 2023.

[19] H. Liu, M. E. Fouda, A. M. Eltawil, and S. A. Fahmy, "Split DNN inference for exploiting near-edge accelerators," in *IEEE International Conference on Edge Computing and Communications*, 2024, pp. 84–91.

[20] M. F. AlShams, K. S. Smagulova, S. A. Fahmy, M. E. Fouda, and A. M. Eltawil, "DONNA: Distributed optimized neural network allocation on CIM-Based heterogeneous accelerators," in *IEEE International Conference on Edge Computing and Communications (EDGE)*, 2024.

[21] S. K. Ghosh, A. Raha, V. Raghunathan, and A. Raghunathan, "PArtNNer: Platform-agnostic adaptive edge-cloud DNN partitioning for minimizing end-to-end latency," *ACM Transactions on Embedded Computing Systems*, 2024.

[22] Y. Shi, L. Li, Y. Zeng, P. Cong, and J. Zhou, "Joint DNN partition and thread allocation optimization for energy-harvesting MEC systems," in *Design, Automation Test in Europe Conference (DATE)*, 2025.

[23] J.-A. Lim, J. Lee, J. Kwak, and Y. Kim, "Cutting-edge inference: Dynamic DNN model partitioning and resource scaling for mobile AI," *IEEE Transactions on Services Computing*, 2024.

[24] Y. Tian, Z. Zhang, Z. Yang, and Q. Yang, "JMSNAS: Joint model split and neural architecture search for learning over mobile edge networks," in *IEEE International Conference on Communications Workshops*, 2022, pp. 103–108.

[25] J. Wu, L. Wang, Q. Pei, X. Cui, F. Liu, and T. Yang, "HiTDL: High-throughput deep learning inference at the hybrid mobile edge," *IEEE Transactions on Parallel and Distributed Systems*, pp. 4499–4514, 2022.

[26] B. Huang, A. Abtahi, and A. Aminifar, "Energy-aware integrated neural architecture search and partitioning for distributed internet of things (IoT)," *IEEE Transactions on Circuits and Systems for Artificial Intelligence*, 2024.

[27] G. Liu, F. Dai, X. Xu, X. Fu, W. Dou, N. Kumar, and M. Bilal, "An adaptive DNN inference acceleration framework with end-edge-cloud collaborative computing," *Future Generation Computer Systems*, 2023.

[28] D. Jung, J. Lee, H. Jeong, D. Cha, H. Kim, and S. Pack, "Split computing for mobile devices: Energy and latency perspective," *IEEE Transactions on Services Computing*, 2025.

[29] 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 and Test in Europe Conference Exhibition (DATE)*, 2017, pp. 1396–1401.

[30] Y. Gao, "High-performance network data transfers to GPU: A study of NVIDIA GPU direct RDMA and GPUNetIO," 2023.

[31] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "iGniter: Interference-aware GPU resource provisioning for predictable DNN inference in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, pp. 812–827, 2022.

[32] M. Jankowski, D. Gündüz, and K. Mikolajczyk, "Joint device-edge inference over wireless links with pruning," in *IEEE International Workshop on Signal Processing Advances in Wireless Communications*, 2020.

[33] S. Itahara, T. Nishio, and K. Yamamoto, "Packet-loss-tolerant split inference for delay-sensitive deep learning in lossy wireless networks," in *IEEE Global Communications Conference (GLOBECOM)*, 2021.

[34] Y. Cheng, Z. Zhang, and S. Wang, "RCIF: Toward robust distributed DNN collaborative inference under highly lossy IoT networks," *IEEE Internet of Things Journal*, 2024.

[35] ESnet, "iPerf3: A TCP, UDP, and SCTP network bandwidth measurement tool," https://iperf.fr, 2014, accessed: 2025-09-01.

[36] R. Bonghi, "jetson_stats," https://github.com/rbonghi/jetson_stats/, accessed: 2025-09-01.

[37] The GStreamer Project, "GStreamer: Open source multimedia framework," https://gstreamer.freedesktop.org/, accessed: 2025-09-01.

[38] M. Ham, J. Moon, G. Lim, J. Jung, H. Ahn, W. Song, S. Woo, P. Kapoor, D. Chae, G. Jang *et al.*, "NNStreamer: Efficient and agile development of on-device AI systems," in *IEEE/ACM International Conference on Software Engineering*, 2021, pp. 198–207.

[39] M. Ham, S. Woo, J. Jung, W. Song, G. Jang, Y. Ahn, and H. Ahn, "Toward among-device AI from on-device AI with stream pipelines," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 285–294.

[40] J. Geul, S. Shier, and B. Hubert, "WonderShaper," https://github.com/magnific0/wondershaper, 2020, accessed: 2025-09-01.

[41] Y. Duan and J. Wu, "Joint optimization of DNN partition and scheduling for mobile cloud computing," in *Proceedings of the International Conference on Parallel Processing*, 2021.

[42] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 322–337.

[43] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2019.

[44] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 613–627.