LOPInfer: A High-Performance Local-Operator Parallel Inference System for Service Workloads in Mobile Edge Computing

Zekai Sun, Xiuxian Guan, Zheng Lin, Zihan Fang, Xiangming Cai, Zhe Chen, Fangming Liu, Heming Cui, Jie Xiong, Wei Ni, *Fellow, IEEE*, and Jun Luo, *Fellow, IEEE*

***Abstract*—Real-time mobile edge computing (MEC) services rely on deep neural network (DNN) inference under stringent per-request latency and energy budgets on resource-constrained mobile devices. While MEC enables offloading to GPU-equipped edge servers, layer-wise model partitioning creates a transmission bottleneck: the sequential execution of DNN layers forces stop-and-wait transfers of large activation tensors, leading to transmission delay. To address this problem, we present LOPInfer, a local-operator parallel inference system for MEC service workloads that enables fine-grained overlap of computation and transmission within a single inference. LOPInfer exploits local operators (i.e., operators whose computation can proceed on partial inputs rather than the entire input tensor), decomposes their computation (operation) into independent sub-operations. By pipelining intra-layer and cross-layer sub-operations, LOPInfer reduces idle time, hides transmission delay, and thereby shortens end-to-end inference latency and lowers device energy consumption without altering the DNN's semantics. Evaluation shows that LOPInfer reduces per-inference latency by up to 50% and energy consumption by up to 75% compared with state-of-the-art baselines, without sacrificing accuracy.**

***Index Terms*—Model inference, Service workloads, Mobile edge computing, Distributed system and network**

I. Introduction

Mobile edge computing (MEC) underpins real-time intelligent services (e.g., autonomous navigation [1], robot perception and control [2], [3], and IoT systems [4]) that rely on deep neural networks (DNNs) to turn sensor streams into timely decisions. These applications run in an event-driven, per-request regime with strict latency targets and tight device power budgets: missed deadlines degrade control freshness and safety, while excessive on-device computation breaches energy and thermal limits. Achieving high-performance (low end-to-end latency and high energy efficiency) therefore requires MEC to jointly exploit the compute of GPU-equipped edge servers (GPU servers) and the locality of on-device execution (e.g., smartphones, IoT devices), while mitigating the dominant, highly variable cost of wireless transmissions.

Z. Sun, X. Guan, and H. Cui are with the Department of Computer Science, The University of Hong Kong, Pok Fu Lam, Hong Kong SAR, China. Z. Sun and H. Cui are also with Shanghai AI Laboratory, Shanghai, China. (e-mail: zksun@cs.hku.hk; xxguan@cs.hku.hk; heming@cs.hku.hk).

Z. Lin is with the Department of Electrical and Electronic Engineering, The university of Hong Kong, Pok Fu Lam, Hong Kong SAR, China (e-mail: linzheng@eee.hku.hk).

Z. Fang is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong SAR, China (e-mail: zihanfang3-c@my.cityu.edu.hk).

Z. Chen is with the Institute of Space Internet, Fudan University, Shanghai 200438, China, and also with the School of Computer Science, Fudan University, Shanghai 200438, China (e-mail: zhechen@fudan.edu.cn).

X. Cai is with the School of Physics and Information Engineering, Fuzhou University, Fuzhou 350108, China (e-mail: xiangming.cai@fzu.edu.cn).

F. Liu is with the Peng Cheng Laboratory, Shenzhen, China, and Huazhong University of Science and Technology, Wuhan, China (e-mail: fmliu@hust.edu.cn).

J. Xiong is with the School of Computing and Data Science, Nanyang Technological University, Singapore 639798 (e-mail: jie.xiong@ntu.edu.sg).

W. Ni is with the School of Engineering, Edith Cowan University, Perth, WA 6027, Australia, and the School of Computing Science and Engineering, The University of New South Wales, Kennington, NSW 2052, Australia (e-mail: wei.ni@ieee.org).

J. Luo is with the College of Computing and Data Science, Nanyang Technological University, Singapore (e-mail: junluo@ntu.edu.sg).

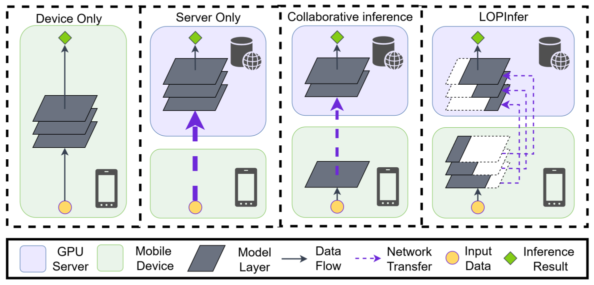


Fig. 1: Comparison between conventional MEC inference paradigms with LOPInfer. Inference results computed on the GPU server must be transmitted back to the mobile device for application utilization.

Conventional MEC inference follows three paradigms (Fig. 1): device-only, server-only, and collaborative inference via layer-wise partitioning [5]–[9]. In device-only inference, all DNN layers run on the mobile device, eliminating network transfers but constrained by on-device compute and energy. In server-only inference, the raw input is uploaded to a GPU server for end-to-end execution, leveraging accelerators at the cost of network dependence. In collaborative inference, a partition point between layers splits execution across device and server; device-only and server-only are degenerate cases with the cut after the last layer and before the first, respectively. Because many intermediate layers produce activations smaller than the raw input [6], offloading features rather than the input can substantially reduce transmission payloads, thereby improving end-to-end latency and energy efficiency (Sec. II-E).

These MEC paradigms based on layer-wise partitioning face fundamental obstacles to meeting stringent per-request latency and device-energy targets. Device-only inference often incurs high latency and sustained power draw due to limited on-device compute (Sec. II-B). Server-only inference is highly sensitive to uplink conditions: under volatile wireless bandwidth, uplink transfers frequently dominate end-to-end delay and produce long-tailed latency (Sec. II-C). In layer-partitioned collaborative inference, computation and communication are serialized within a single request (early layers on the device, intermediate activations uploaded, remaining layers on the GPU server). This sequential execution forces stop-and-wait transfers of large activation tensors, creating a transmission bottleneck and incurring delay under limited, time-varying uplink bandwidth (Sec. II-C). In our experiments, transmission accounts for 50% of end-to-end latency and 40% of device energy.

Motivated by the transmission-dominated, stop-and-wait behavior of layer-wise partitioning, we pursue intra-request compute–communication overlap for real-time MEC inference. Pipeline execution [10] overlaps computation and communication across requests to mitigate transmission overheads, but it primarily increases throughput and does not reduce the single-request latency required by real-time MEC services. This limitation motivates overlapping computation and transmission within a single inference: partial results are transmitted and consumed as soon as they are produced, so the device and edge avoid idle waiting, hide transmission delay, shorten end-to-end latency. Shorter idle periods also lower device energy because idle-time power is dominated by core components (CPU, GPU, memory) [11]; in our setup, these components account for about 95% of device energy during waiting, whereas the wireless network interface cards contributes only about 1.5%.

In this paper, we propose **LOPInfer** (**L**ocal-**O**perator **P**arallel **Infer**ence), a high-performance MEC inference system that enables fine-grained, intra-request overlap of computation and transmission without altering DNN semantics. Realizing such overlap, however, poses three key challenges. ① Traditional parallel computing methods (data, tensor, and pipeline parallelism) are tailored to data centers and perform poorly in MEC due to batch-size limits, high synchronization costs, and nontrivial transmission delays (Sec. II-D), necessitating finer-grained scheduling units within layers. ② Fine-grained streaming raises consistency concerns: loss, reordering, corruption, or stale inputs in any unit can compromise the final result, so the execution must ensure correctness. ③ Balancing computation and transmission requires a new scheduling paradigm whose optimization is substantially harder: the search space expands from $O(n)$ (layers) to $O(n^2)$ (intra-layer units), and the solution must handle MEC-specific transmission and computing constraints.

To this end, LOPInfer systematically addresses these challenges in turn. ① LOPInfer defines scheduling units as local operators whose outputs depend only on a subset of the input tensor (e.g., element-wise inputs for ReLU and tensor-block inputs for convolution), enabling their computations (operations) to be decomposed into independent sub-operations (local operations). This locality allows downstream local operations to start as soon as their required inputs are ready, without waiting for all operations in the current layer, thereby enabling intra- and cross-layer compute–communication overlap within a single inference. ② To ensure correctness and completeness, we introduce Local Operation Parallelism (LOP), which explicitly tracks producer–consumer data dependencies so that each local operation consumes and produces the correct data within ①, preserving semantic equivalence to sequential execution. ③ To attain low-latency and energy-efficient inference under MEC conditions, we propose the Local Operation Scheduling Strategy (LOSS), which places local operations across the mobile device and the GPU server while respecting MEC-specific transmission and compute constraints, employing a fast heuristic to materialize the overlap exposed by ②.

In summary, this paper makes the following contributions:

- We propose LOPInfer, a local-operator parallel inference system for MEC service workloads that enables intra-request compute–communication overlap.
- We develop Local Operation Parallelism (LOP), a parallel computing technique that guarantees inference correctness at local-operation granularity.
- We design the Local Operation Scheduling Strategy (LOSS), which formulates device–server placement of local operations under LOP as a constrained optimization and produces intra- and cross-layer compute–communication overlap schedules for high-performance (low-latency and energy-efficient) inference.
- We implement an adaptive control mechanism that tracks and responds to real-time network bandwidth fluctuations in MEC to sustain overlap and performance.
- We conduct real-world experiments on commercial robots and GPU servers in MEC settings, showing that LOPInfer outperforms state-of-the-art baselines in terms of per-request end-to-end latency and on-device energy efficiency; the code is released at https://github.com/hku-systems/LOPInfer.

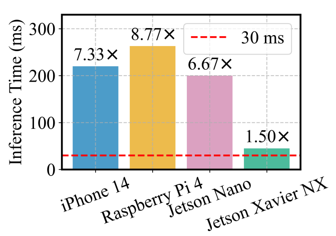
II. Background and Motivation

A. MEC Service Workloads

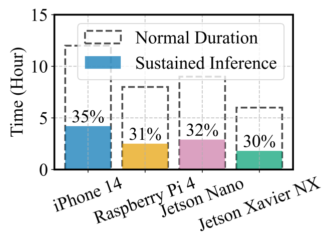
We target real-time intelligent MEC services (e.g., autonomous navigation [1], robot perception and control [2], [3], and IoT systems [4]) that map sensor inputs to actions under tight latency and energy constraints. These workloads share three properties: i) they require immediate per-request inference upon the arrival of device-generated sensor input, thus the operational batch size is 1 to avoid queuing delay and output staleness; ii) they must preserve model accuracy (accuracy-critical tasks such as localization and tracking are sensitive to small errors that can cause target loss, drift, or misalignment); iii) they must deliver high performance (low end-to-end latency and high energy efficiency) within the tight compute and power budgets of mobile devices. In this event-driven regime, the system objective is to minimize per-request on-device latency while strictly preserving accuracy and DNN semantics. Doing so increases the effective update rate, reduces decision staleness, and shortens the closed-loop horizon, thereby improving control fidelity, responsiveness to environmental change, and operational safety within the device's power envelope.

B. Device-Only Inference

Device-only inference (deploying DNNs entirely on mobile devices) operates under tight compute and power budgets that constrain both performance and energy efficiency. As shown



(a) Average Inference Latency



(b) Battery Life

Fig. 2: The performance of VGG-16 under device-only inference across different mobile devices [13]–[15].

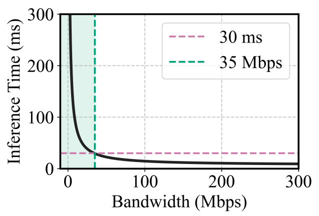


Fig. 3: The inference time of VGG-16 under server-only inference across different network bandwidth.

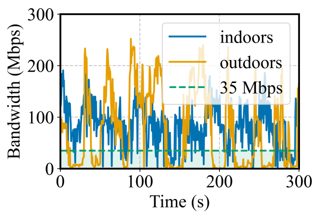


Fig. 4: The wireless transmission instability of TCP between our robot and the base station in MEC networks.

in Fig. 2a, the average per-request latency exceeds the 30 ms threshold for smooth video fluency [12] (red dotted line). Fig. 2b further shows that sustained inference reduces device standby time to 30–35% of normal, degrading user experience and undermining overall device practicality.

C. Server-Only Inference

Server-only inference (offloading inputs to a remote GPU) makes end-to-end performance network-bound, inducing long-tail latency and heavy uplink bandwidth demand. As shown in Fig. 3, on the same testbed as Sec. V (a GPU server with an NVIDIA GeForce GTX 3080), inference time increases sharply as available bandwidth decreases, and its dispersion widens, amplifying deadline-miss risk.

In contrast to data-center interconnects that offer orders-of-magnitude higher capacity and stability (e.g., 100–400 Gbps InfiniBand [16] within clusters and high-bandwidth intra-server fabrics such as PCIe), mobile devices rely on wireless links whose bandwidth and jitter are fundamentally limited in both theory and practice. Although Wi-Fi 6 can reach a peak of 1.2 Gbps per stream [17], commodity mobile hardware cannot fully utilize this capacity [18]; the actually available throughput further varies with device mobility [19], signal obstruction [20], and channel contention [21].

To quantify wireless variability in MEC, we ran a robot-surveillance experiment: four-wheeled robots traversed an indoor lab and an outdoor garden at 5–40 cm/s. Using iperf [22] over TCP, we measured available throughput between the robot and a base station, sampling every 0.5 s for five minutes. As shown in Fig. 4, average throughput was 93 Mbps indoors and 73 Mbps outdoors; the outdoor trace showed larger fluctuations and occasional near-zero dips due to obstacles and reduced signal reflections. These dynamics make server-only inference unlikely to meet real-time per-request latency targets under realistic MEC wireless conditions.

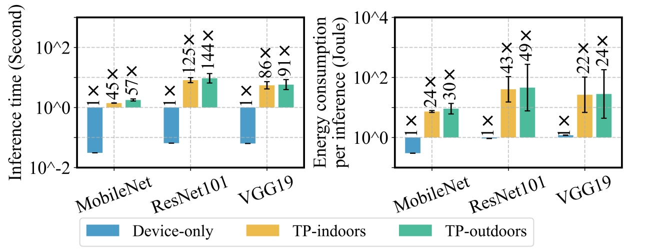


Fig. 5: The performance of TP for different models. The cross marker (×) denotes the mean value.

D. Related Parallel Computing Techniques

Parallel computing has been extensively studied and is effective in modern data centers. However, it is ill-suited to MEC, where tight compute and power budgets and stringent per-request latency dominate system design. Data centers primarily use three forms of parallelism: i) data parallelism (DP), which replicates the model across devices and processes different samples from a mini-batch in parallel; ii) tensor parallelism (TP), which partitions intra-layer computations across devices; iii) pipeline parallelism (PP), which partitions layers across devices and executes them as a pipeline across multiple requests to boost throughput.

DP scales by sharding sufficiently large batches across replicas to increase throughput. In data centers, large batches (e.g., 16 images) are split into per-replica mini-batches to keep accelerators well utilized. In MEC, inference is event-driven with online arrivals and batch size = 1, leaving no inter-request concurrency to exploit. Consequently, DP at batch size = 1 degenerates to single-device execution and further slicing a single input into sub-mini-batches (e.g., 1/4 of an image) is impractical, rendering DP ineffective for MEC inference.

TP exposes intra-layer parallelism but requires frequent cross-device all-reduce collectives to synchronize partial results, making communication dominant on device/server interconnects. We evaluate DINA [23], a state-of-the-art TP method, on our testbed (Sec. V) with batch size 1. As shown in Fig. 5, TP's synchronization overhead inflates per-request end-to-end latency by $45\times$-$144\times$ and on-device energy by $22\times$-$49\times$ relative to device-only inference. Although recent work reduces TP communication in data centers (e.g., joint spatial/temporal partitioning [24]), the all-reduce barrier is intrinsic to TP and remains prohibitive in MEC. In contrast, LOPInfer avoids cross-device collectives with local operators and enables intra-request compute–communication overlap, hiding communication and reducing per-request latency.

PP overlaps computation and communication across requests to boost throughput, but in the single-request regime, it leaves per-request latency (makespan) essentially unchanged because each request's device-to-server uplink remains on the critical path of layer partitioning [10]. SPINN [25] combines layer partitioning with early exits to reduce activation traffic and average cost, at the expense of accuracy and with instance-dependent accuracy variability that requires application-specific calibration. In contrast, LOPInfer exposes intra-request parallelism, overlapping each request's uplink with its on-device computation to mask transmission delay and reduce per-request latency, especially when the uplink

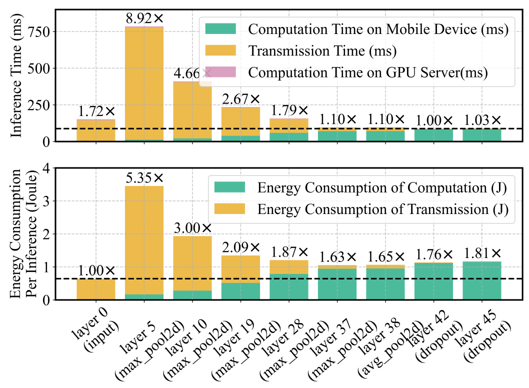


Fig. 6: The performance of different layer partitioning strategies for VGG-19 under 35 Mbps in our experiments. The X-axis represents various partitioning points, where 'layer i' denotes that all layers up to and including the i_{th} layer are executed on the robot, while the remaining layers are offloaded to the GPU server. Note that transmission time depends on network bandwidth.

dominates, without modifying the model.

In conclusion, although conventional parallel computing techniques are effective in data centers, they are ill-suited to MEC: DP offers no benefit at batch size 1; TP is dominated by cross-device all-reduce synchronization; and PP optimizes throughput rather than per-request latency. As a result, collaborative inference in MEC typically falls back to layer-wise partitioning between the device and server, which serializes computation and transmission within each request and keeps the device-to-server uplink on the critical path, the dominant bottleneck for low-latency inference.

E. Existing Collaborative Inference

Existing collaborative inference methods [5]–[9] primarily use layer partitioning to trade off per-request latency and energy (Fig. 6). Device-only and server-only are the two extremes, corresponding to splitting after the final layer or before the first; in Fig. 6, "layer 0" denotes server-only with all layers on the GPU server. Prior work typically targets either faster inference [5]–[7] or lower energy under deadlines [8], [9]. However, because layer partitioning executes layers sequentially across the device/server boundary, each split inserts a transmission on the critical path, which dominates both the mean and variability of per-request latency. In contrast, LOPInfer decomposes models into local operations and schedules them in parallel across the device and GPU server, preserving DNN semantics (and thus accuracy) and still subsuming device-only and server-only as special cases. By enabling intra-request compute–communication overlap, it hides the transmission delay relative to layer partitioning.

Some efforts explore finer-grained partitions to mitigate transmission delay but prove ineffective in GPU-enabled MEC. Input-splitting methods partition only the first-layer input (i.e., a special case of local-operation splitting restricted to layer 0) across devices in proportion to their compute capacity [26]. Designed for IoT collectives, they are ill-suited to MEC: given the large device–edge performance gap, allocations often degenerate to server-only inference. [27] overlaps computation and communication by issuing numerous small tiles, but excessive fragmentation incurs substantial kernel-launch, RPC/driver, and scheduling overheads. Its host-driven, asynchronous first-come-first-served dispatch is CPU-oriented and not tuned for GPU-equipped MEC, leading to low GPU kernel occupancy and limited end-to-end speedup. By contrast, LOPInfer uses GPU-aware scheduling that explicitly models compute and transmission efficiency (Sec. IV-C4), respects MEC constraints, and sustains high parallelism—capabilities absent from existing fine-grained approaches in this setting.

III. System Overview

In this section, we present LOPInfer, a local-operator parallel inference system optimized for MEC service workloads. In this setting, resource-constrained mobile devices (e.g., robots and smartphones) collaborate with GPU servers over wireless networks (e.g., Wi-Fi 6 and 5G) to deliver real-time, energy-efficient inference on device-generated sensor inputs.

A. Key Insight

Existing MEC inference paradigms predominantly adopt layer partitioning, treating each operator as an atomic unit that can run only after the entire input tensor is available. Because operator sequences must respect the computational graph's topological order, this design imposes layer-wise barriers and limits parallelism across devices and servers.

We observe that many operators are local (i.e., their outputs depend only on a subset of the input tensor) so their computations (opeartions) can be decomposed into independent sub-operations (local operations). Leveraging this locality, LOPInfer schedules local operations once their required inputs are ready, allowing downstream local operations to start without waiting for the entire current layer to finish. This enables fine-grained intra- and cross-layer scheduling and overlaps computation with communication within a single request, thereby hiding communication and reducing per-request latency. For global operators whose outputs depend on the entire input tensor (a special case where the subset equals the whole tensor, e.g., Softmax), the formulation naturally collapses to operator-level synchronization for that operator, preserving correctness.

B. Overall Workflow

As shown in Fig. 7, LOPInfer comprises three stages: Offline Profiling, Offline Schedule Synthesis, and Runtime. Offline Profiling (①) runs once per model–hardware pair and extracts key execution characteristics, including: i) operator types and input/output sizes; ii) execution times of operations on the mobile device and GPU server; iii) data dependencies between operations (i.e., the output of one operation serves as the input for another).

Using these profiles, Offline Schedule Synthesis (②) formulates a local-operation scheduling problem (Sec. IV-C) that jointly selects partitioning, placement, and pipeline order to minimize end-to-end latency under data dependency and MEC constraints. To eliminate online solving overhead, LOPInfer precomputes a family of bandwidth-indexed schedules (e.g., 0–30 MB/s in 1 MB/s bins) and builds a lookup map from the measured bandwidth to the corresponding schedule.

At Runtime, LOPInfer uses Operator Interceptors (Sec. IV-A, ③) for per-operator tensor partitioning and

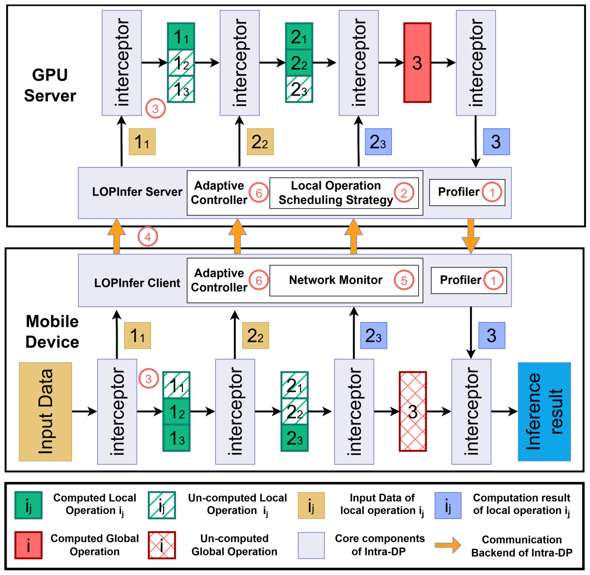

GPU Server
interceptor
LOPInfer Server
Adaptive Controller ⑥
Local Operation Scheduling Strategy ②
Profiler ①
LOPInfer Client
Adaptive Controller ⑥
Network Monitor ⑤
Profiler ①
Mobile Device
Input Data
Inference result
③
④
Computed Local Operation i_j
Un-computed Local Operation i_j
Input Data of local operation i_j
Computation result of local operation i_j
Computed Global Operation
Un-computed Global Operation
Core components of Intra-DP
Communication Backend of Intra-DP

Fig. 7: Overview and inference workflow of LOPInfer. Local operation i_j represents the j-th local operation within the i-th operator.

reconstruction to enable partial-tensor execution, while Local-Operation Parallelism (Sec. IV-B, ④) pipelines local operations with dependency-preserving scheduling. A lightweight network monitor (⑤) continuously estimates network conditions using standard tools [28]; an adaptive controller (Sec. IV-D, ⑥) selects a precomputed schedule via table lookup based on the current bandwidth, ensuring stable performance under fluctuations. To support instantaneous switching, LOPInfer maintains a warm replica of the model on the GPU server and keeps state consistent across switches.

We quantify LOPInfer's runtime overhead over baseline inference and find it minimal in steady state. The only added primitives are per-operator tensor partitioning and aggregation, implemented by Operator Interceptors (Sec. IV-A). Partitioning runs asynchronously and overlaps with network transmission and other local operations via LOP, so its latency stays off the critical path. Aggregation can incur waits at operator-level barriers due to data dependencies; LOSS (Sec. IV-C) explicitly models these waits and, during offline synthesis, minimizes end-to-end makespan under data-dependency and MEC constraints. Consequently, the amortized overhead remains small, and LOPInfer maintains high execution efficiency.

IV. Design of LOPInfer

In this section, we present the design of LOPInfer, a high-performance inference system that exploits fine-grained parallelism at the level of local operators and is optimized for MEC workloads. We first classify operators as local or global (Sec. IV-A) and establish the conditions under which operator-level parallel execution is provably correct. Building on this taxonomy, Sec. IV-B introduces LOP, a parallel computing technique that guarantees inference correctness at the granularity of local operations. To orchestrate these parallel local operations, Sec. IV-C presents LOSS, which formulates operator partitioning, placement, and ordering as a constrained optimization to minimize end-to-end latency under data-dependency and MEC constraints. Finally, to maintain robust performance under time-varying network conditions, Sec. IV-D develops an adaptive control mechanism that tracks bandwidth fluctuations in real time and adjusts scheduling decisions accordingly. Together, these components deliver fast, energy-efficient inference in MEC environments.

A. Local Operators and Global Operators

As LOPInfer's speedups come from parallelizing local operators, we formalize each operator's minimal input unit to enable local-operation execution. LOPInfer uses Operator Interceptors (Fig. 7 ③) to i) derive operator-specific dependency footprints, ii) slice and reconstruct inputs/outputs, iii) and classify operators as local or global. We model an operator as a (possibly parameterized) mapping $\mathrm{op} : (X, W) \to Y$ with input index set I_X, output index set I_Y, and optional parameters W. Let $D \subseteq I_Y \times I_X$ be the data-dependency relation, where $(i, j) \in D$ iff $y[i]$ depends on $x[j]$. For any $T \subseteq I_Y$, the input footprint is $R(T) = \{\, j \in I_X \mid \exists\, i \in T \text{ s.t. } (i, j) \in D \,\}$. The minimal input unit for any non-empty output subset T is X restricted to $R(T)$; the atomic unit is $R(\{i\})$. An operator is local if each output can be written as $y[i] = g_i\big(X[R(\{i\})], W\big)$ with $R(\{i\}) \subsetneq I_X$ (i.e., it does not require the entire input tensor) and the dependencies factorize with respect to a partition of I_X (i.e., each $R(\{i\})$ lies within a single partition without cross-partition aggregation). Otherwise, the operator is global (e.g., when computing $y[i]$ aggregates over an entire axis). In models commonly used in MEC services (MEC models), three local classes are prevalent:

- **Element-wise local operators.** Here $I_Y \equiv I_X$ and $D = \{(i, i) \mid i \in I_Y\}$, so $R(T) = T$ and the atomic unit is a single element. Typical activations (e.g., ReLU, Sigmoid, and SiLU) are element-wise local. In contrast, Softmax aggregates over that entire axis along an axis; thus $R(\{i\})$ equals all inputs on the axis, making it a global operator.
- **Block-wise local operators.** Each output element is computed from a finite spatial window of the input. For 2D convolution or pooling with kernel (k_h, k_w), stride (s_h, s_w), dilation (d_h, d_w), and padding (p_h, p_w), the footprint for $y[o, i, j]$ (over all input channels c) is
$$R(\{(o, i, j)\}) = \big\{(c, u, v) \;\big|\; r \in [0, k_h - 1],\ t \in [0, k_w - 1],\\ u = i\, s_h - p_h + d_h\, r,\ v = j\, s_w - p_w + d_w\, t\big\}.$$
This is the standard $k_h \times k_w$ (dilated) window aligned to (i, j) and replicated across input channels. Convolution (including depthwise/group variants) and max/average pooling follow this rule; the window and halo are determined by kernel size, padding, dilation, and stride [26].
- **Row-wise local operators.** Let $A \in \mathbb{R}^{m\times k}$ be the input and W a shared parameter (e.g., $W \in \mathbb{R}^{k\times n}$). If the operator is row-separable, each output row depends only on the corresponding input row and W: $y_r = f(a_{r:}, W)$ for $r = 1, \ldots, m$. For matrix multiplication $C = AW$,
$$\begin{pmatrix} c_{11} & \cdots & c_{1n} \\ \vdots & \ddots & \vdots \\ c_{m1} & \cdots & c_{mn} \end{pmatrix} = \begin{pmatrix} a_{1:} \\ \vdots \\ a_{m:} \end{pmatrix} \times \begin{pmatrix} | & & | \\ w_1 & \cdots & w_n \\ | & & | \end{pmatrix},$$
we have $c_{r:} = a_{r:}W$. The data-dependency relation is $D = \{\, ((r, \cdot), (r, \cdot)) \,\}$, so $R(T)$ selects the same set of

rows in A. No cross-row aggregation is required; subsequent row-separable operators (e.g., bias add, element-wise activation) can consume rows in the same manner. TP, in contrast, partitions the layer's parameter matrix W and replicates the full input matrix A across devices, which necessitates all-reduce to aggregate partial results. Moreover, LOP treats operators whose layer-parameter matric contains only one row as global operators.

Model	Local / Global	Model	Local / Global
DenseNet [29]	428 / 3	RegNet [30]	231 / 3
ResNet [31]	341 / 3	VGGNet [32]	73 / 5
ResNeXt [33]	342 / 3	ConvNeXt [34]	340 / 6

TABLE I: Number of local/global operators in MEC models.

Models with a high proportion of local operators, common in MEC models (e.g., CNNs for vision and point cloud processing [2], [3]), benefit from LOPInfer via fine-grained operator parallelism. Table I quantifies the share of local versus global operators across representative MEC models under our formal definition, measured by operator count. These locality classes directly determine the slicing granularity used by LOP and expose safe local-operator parallelism.

B. Local Operation Parallelism

This section details how LOPInfer guarantees inference correctness within local operators, and how LOP remove the transmission bottleneck in existing methods (Fig. 7, ④).

LOP ensures correctness via dependency-aware scheduling along with the execution properties of local operators. Specifically, it maintains the following invariants: i) graph consistency: operators (and their local operations) execute in a topological order that respects all data dependencies; ii) footprint closure: for any operator, each local operation consumes exactly the input footprint required by its output and produces the corresponding output; iii) parameter consistency: operator parameters are shared across local operations; iv) global completeness: global operators run only after synchronization barriers ensure that all required input tensors are fully assembled. Under these invariants, the parallel execution is functionally equivalent to the original inference by preserving correctness for both local and global operators.

Fig. 8 contrasts the workflow of LOP with TP and PP (layer partitioning). LOP decomposes each local operator into multiple local operations; we denote the 'j'-th local operation of the 'i'-th operator by 'LOi_j'. It then applies two optimizations to remove the transmission bottleneck and shorten the critical path. First, it performs computation–communication overlap over the device-to-server link across both intra- and cross-layer local operations (e.g., while computing 'LO1_2', 'LO1_3', and 'LO2_3' on the mobile device, it concurrently transmits the inputs required by upcoming local operations 'LO1_1'), thereby hiding communication. Second, it exploits data locality via dependency-aware placement, prioritizing execution on the node that already holds the required inputs (e.g., scheduling 'LO2_1' on the GPU server to directly consume the output of 'LO1_1' resident there), eliminating avoidable transfers. Unlike TP, which incurs frequent all-reduce across tensor shards, LOP restricts synchronization to global operators only,

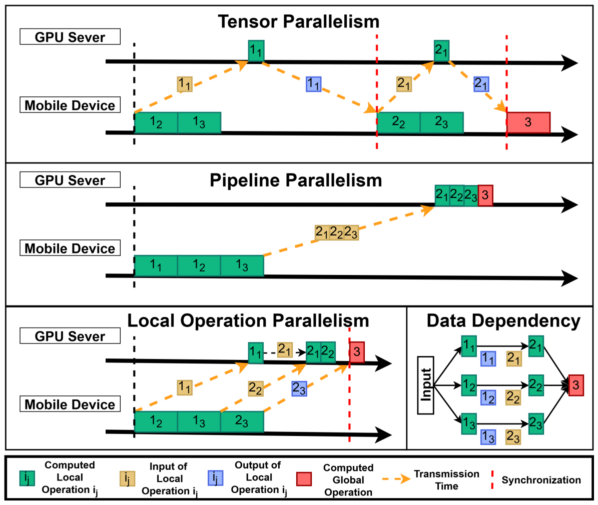

Tensor Parallelism
GPU Sever
Mobile Device
Pipeline Parallelism
Local Operation Parallelism
Data Dependency
Input
Computed Local Operation i_j
Input of Local Operation i_j
Output of Local Operation i_j
Computed Global Operation
Transmission Time
Synchronization

Fig. 8: Workflow of TP, PP and LOP. In the three cases above, each local operator executes three operations with identical computation times on both the mobile device and the GPU server, along with the corresponding transmission time, while the lower right corner illustrates the data dependencies between operations.

inserting barriers solely at their boundaries. Compared with layer partitioning, LOP issues finer-grained communications and may increase total bytes, but its early-started transmissions and computation–communication overlap across local operations hide communication for single-request inference, overcoming the sequential transmission bottleneck of layer partitioning. By overlapping computation with communication and exploiting locality, LOPInfer reduces mobile-device idle time and improves energy efficiency; in mobile settings where CPU/GPU/DRAM activity often dominates the energy budget [11], shorter idle periods reduce energy per inference even with an active wireless network interface cards.

C. Local Operation Scheduling Strategy

This section explains how LOPInfer schedules computation and transmission for local operations to achieve fast, energy-efficient inference (Fig. 7, ②). LOSS leverages the fact that wireless bandwidth fluctuates on seconds-scale [35], whereas a single inference completes within tens to hundreds of milliseconds. It therefore treats the wireless bandwidth as constant over an individual request. Schedules are computed offline by solving a constrained optimization with a fast heuristic under the assumed bandwidth. Within this unified abstraction, global operators are modeled as special cases that aggregate all outputs from preceding local operations.

The performance gains stem from three mechanisms. First, computation–communication overlap across intra- and cross-layer local operations shortens the critical path by transmitting required inputs while upstream operations execute. Second, dependency-aware placement exploits data locality by prioritizing execution on the node that already holds the required inputs, avoiding unnecessary transfers. Third, when multiple downstream local operations on the mobile device and GPU server depend on the same producer, LOSS replicates it on both endpoints so its output is produced locally on each side, eliminating cross-link transfers; the model jointly accounts for replication overhead and reduced transmission. Collectively, these mechanisms reduce device idle time, yielding lower latency and energy per inference.

1) DNN Execution Model

We model a DNN as a directed acyclic graph $G = (V, E)$, where each vertex $v \in V$ denotes an operator (layer) and each edge $(u, v) \in E$ represents a data dependency from producer u to consumer v. Two virtual vertices, v_{in} and v_{out}, denote the model input and final output; they are computation-free boundary nodes. The DNN executes collaboratively across a resource-constrained mobile device M and a GPU server R.

2) Variables and Functions

For each operator v, we partition its computation into a set of local operations $OP(v)$ based on its minimum input unit. We allocate them to M and R as $x_M(v)$, $x_R(v) \subseteq OP(v)$; overlaps ($x_M(v) \cap x_R(v) \neq \emptyset$) occur when local operations are replicated (e.g., for block-wise operators). On each device, the local operations of v start at $s_M(v), s_R(v) \geq 0$, and the computation time are estimated as $C_M(v)$ and $C_R(v)$. If no operation is performed on a device, its computation time is zero. Data transmission times include $T_{\text{MR}}(u, v)$ for sending data from M to R and $T_{\text{RM}}(u, v)$ for the reverse direction, both determined by network bandwidth and the size of the inputs/outputs associated with the corresponding operations.

3) Objective Function

The objective of LOSS is to minimize end-to-end latency for a single request, with the final output delivered to the mobile device. As $s_M(v_{\text{out}})$ equals the earliest time when all outputs are available on M, the optimization problem is

$$\min T = s_M(v_{\text{out}}), \tag{1}$$

subject to the scheduling, causality, and communication constraints defined in the execution model.

The framework also supports alternative objectives and constraints by replacing the latency objective with $\min L = f_M(v_{\text{out}})$. For multi-objective settings, one can trade off latency and energy (e.g., $f_M(v_{\text{out}}) = s_M(v_{\text{out}}) + \lambda E_M$), or minimize energy subject to a latency deadline (e.g., $s_M(v_{\text{out}}) \leq \tau$), following prior work [8]. Exploration of such trade-offs is left to future work.

4) Constraints

Data Partitioning. To ensure correctness, all operations must be covered: $x_M(v) \cup x_R(v) = OP(v), \forall v \in V$. For any global operator u, the computation is atomic, so $|OP(u)| = 1$. To avoid redundant computation for global operators, we impose disjoint placement: $x_M(u) \cup x_R(u) = OP(u)$, ensuring the global operator runs entirely on exactly one device.

Location. We fix the residency of the virtual vertices: the model input originates on the mobile device ($x_M(v_{\text{in}}) = \textit{input}$ and $x_R(v_{\text{in}}) = \emptyset$) and the final result is consumed on the mobile device ($x_M(v_{\text{out}}) = \textit{output}$ and $x_R(v_{\text{out}}) = \emptyset$).

Data Dependency. Operations on a device can start only after all required inputs for the assigned portion are available on that device. For each v with parent connections $e = (u, v)$, execution on M follows:

$$s_M(v) = \begin{cases} s_M(u) + C_M(u), \\ \quad \text{if } x_M(v) - \textit{child}(x_M(u)) = \emptyset, \\ \max\big(s_R(u) + C_R(u) + T_{\text{RM}}(u, v), \\ \quad s_M(u) + C_M(u)\big), \\ \quad \text{if } x_M(v) - \textit{child}(x_M(u)) \neq \emptyset. \end{cases} \tag{2}$$

Here, $\text{child}(x(u))$ denotes the set of operations that consume the outputs of $x(u)$ according to data dependencies, and transmission volume in $T_{\text{MR}}(u, v)$ is given by $x_M(v) - \text{child}(x_M(u))$. When $x_M(v) - \text{child}(x_M(u)) = \emptyset$, all inputs required by $x_M(v)$ are produced locally by operations on M; otherwise, $x_M(v)$ also depends on outputs from operations on R. These constraints let LOSS exploit locality and selective replication by prioritizing execution on the device that already holds required inputs and only transferring the missing portions. The same constraints apply symmetrically to R.

Computational Efficiency. To reduce kernel-launch overhead [27] caused by excessively fine-grained local operations and improve GPU utilization, LOSS batches adjacent local operations per operator instead of launching per-local-operation kernels. Batching preserves the existing placement and replication semantics and does not alter data dependencies or previously defined constraints. Packing/unpacking of input/output tensors occurs within each batch, retaining fine-grained placement without redundant kernel invocations. For each operator v, we index its local operations in a canonical order $OP(v) = \{1, 2, \ldots, |OP(v)|\}$. We enforce per-device contiguity of indices: for all operators $v \in V$: $x_M(v) = \{i \mid i \in [\min(x_M(v)), \max(x_M(v))]\}$, $x_R(v) = \{i \mid i \in [\min(x_R(v)), \max(x_R(v))]\}$.

Transmission Efficiency. In MEC, communication is a dominant cost. Inspired by layer-partitioning methods showing that some intermediate layers produce activations smaller than the raw input [6], we constrain producer–consumer placement to avoid transmitting expansive intermediate tensors. Let $\Pi \subseteq V_c$ denote operators whose output size exceeds the raw input size. For any $u \in \Pi$ and edge $(u, v) \in E$, we enforce locality on both devices: $x_M(v) - \text{child}(x_M(u)) = \emptyset$, and $x_R(v) - \text{child}(x_R(u)) = \emptyset$. Equivalently, these constraints imply zero cross-device transfer on edges from $u \in \Pi$. This pruning eliminates transmissions of expansive tensors and shrinks the search space, accelerating LOSS while preserving correctness under the local-operation partitioning semantics.

5) Solution Algorithm

LOPInfer schedules at the granularity of local operations within each local operator, expanding the search space from the layer level to the operation level. Because attaining global optimality in non-convex optimization remains challenging, we adopt a two-stage heuristic (Alg. 1) to plan quickly under a given bandwidth while satisfying the constraints in Sec. IV-C4. Device-only, server-only, and layer-partitioning paradigms serve as baselines; we retain them as safe fallbacks and as pruning bounds during search (line 1). For models

Algorithm 1 LOSS heuristic solver (Offline Schedule)

```
    Input: bandwidth $b$; time budget $\tau$; iterations times $K$
    Output: operation-level schedule $X^b = (x_M^b, x_R^b)$
    Parameters: baselines placements of device-only $X_{DO}^b$, server-only $X_{SO}^b$ and layer-partitioning $X_{LP}^b$ under the same objective function.
 1: $X_{DO}^b$, $X_{SO}^b$, $X_{LP}^b$ ← Baselines(model,$b$)
 2: $X^b$ ← Best($X_{DO}^b$, $X_{SO}^b$, $X_{LP}^b$); Beam ← $\{X^b\}$
 3: for all local operator $v$ in topological order do
 4:     Beam ← ExpandBeam(Beam, $v$, $b$)
 5: end for
 6: $X^b$ ← Best(Beam); $t$ ← Now(); count ← 0
 7: while Now()$-t < \tau$ and count $< K$ do
 8:     $X^b$ ← LNSRefine($X^b$, $b$)
 9:     count ← count +1
10: end while
11: $X^b$ ← Best($X^b$, $X_{DO}^b$, $X_{SO}^b$, $X_{LP}^b$);
12: return $X^b$
```

without any local operations (i.e., containing only global operators), the solver defaults to layer partitioning by design.

Phase I constructs a high-quality initial plan seeded by baselines via a critical-path-aware beam search (line 4). We traverse local operators in topological order and, for each, consider only a few candidates: a contiguous portion on M, a contiguous portion on R, and a small amount of replication when it clearly improves locality. Candidates that violate the constraints in Sec. IV-C4 are discarded immediately. Feasible expansions are evaluated by a lightweight, event-driven simulator that estimates end-to-end makespan using calibrated compute and transfer models parameterized by given bandwidth; the score is biased by critical-path slack to prioritize latency-critical choices. We retain the top-K partial plans and prune the rest using the strongest baseline (the best of device-only, server-only, and layer partitioning) as the initial upper bound, tightening it whenever the beam finds a better complete plan.

Phase II refines the Phase I seed plan via a time-bounded Large Neighborhood Search (LNS) guided by the current critical path (line 8). In each iteration, we select a bounded neighborhood around latency-critical vertices and communication edges in the execution DAG (prioritized by low slack or high transfer contribution) temporarily revoke their placements and portion decisions, and repair the induced subgraph using the same constrained candidate generator as in Phase I. Each repaired plan is checked for feasibility (Sec. IV-C4) and then evaluated by the lightweight simulator. We accept a modification only if it strictly reduces the end-to-end makespan, breaking ties by lower cross-device transfer volume or compute cost. After each acceptance, we recompute the critical path to refresh guidance. The search terminates when the time budget T is reached or after K consecutive non-improving iterations (line 7). The procedure is anytime: larger T typically yields lower makespan at higher planning cost, whereas smaller budgets return feasible plans quickly.

Compared with Differential Evolution (DE) and reinforcement learning (RL), our fast heuristic solver better matches the dependency-aware, constraint-heavy nature of LOP. DE [36] typically requires a large evaluation budget (population size × generations) and additional decoding/repair, inflating planning time. RL [37] entails substantial training cost and confronts a extremely vast operation-level state–action space; enforcing

Algorithm 2 LOPInfer client at runtime stage

```
    Input: Data input for inference input; DNN model model
    Output: The inference result ret
    Parameter: Input(i),Output(i): input and output of layer $i$; $x_M^b$, $x_R^b$: schedule plan under the $b$ bandwidth.
 1: $b$ ← TestBandwidth(); Input(0) ← input
 2: for all layer $u$ in model do
 3:     if $Input(u) \neq \emptyset$ and $x_M^b(u) \neq \emptyset$ then
 4:         output(u) ← COMPUTE(Input(u),$x_M^b(u)$)
 5:     end if
 6:     for all edge $e = (u, v) \in E$ do
 7:         if $(x_M^b(v) - \mathtt{child}(x_M^b(u)) \neq \emptyset$ then
 8:             Input(v) ← COMBINE(Output(u),Receive())
 9:         end if
10:         if $x_R^b(v) - \mathtt{child}(x_R^b(u)) \neq \emptyset$ then
11:             Send(Output(u),$x_R^b(v) - \mathtt{child}(x_R^b(u))$)
12:         end if
13:     end for
14: end for
15: return $ret$ ← Output(t)
```

hard constraints often relies on rejection or penalty shaping, inducing high exploration and sampling overhead. In contrast, our solver is training-free, integrates feasibility by construction via constrained candidate generation with immediate checks, is deterministic given a seed, and is explicitly time-bounded, yielding competitive schedules within the allotted planning time and bandwidth budget. We leave exact solvers to future work to obtain optimal schedules and move toward global optimality in LOSS, since LOPInfer's performance hinges on the solution quality delivered by LOSS.

D. Adaptive Control Mechanism

The adaptive control mechanism in LOPInfer runs on both mobile device and GPU server (Fig. 7, ⑥). Offline, LOPInfer builds a schedule table by precomputing high-quality plans over a representative MEC bandwidth range (0–30 MB/s) at 1 MB/s resolution, eliminating online optimization overhead. At runtime, a lightweight background monitor on mobile device estimates effective TCP bandwidth using standard tools [28] (Fig. 7, ⑤); its energy cost is negligible relative to inference. Before each inference, the client and server agree on current bandwidth bucket via a small control message and deterministically index the same schedule-table entry, yielding a stateless per-inference decision that depends only on current bandwidth budget and fixed model/hardware profile. Server-side model replicas enable plan switching with negligible latency. Consequently, LOPInfer adapts instantly to bandwidth fluctuations and maintains robust performance across various networks, as scheduling is driven solely by measured bandwidth.

The adaptive control procedure is realized by Alg. 2 (client) and Alg. 3 (server). At runtime, both sides synchronize the current bandwidth (line 1 in both) and deterministically index the corresponding precomputed schedule. For each operator u, the device assigned to u's local operations executes them, if any; otherwise, it skips execution and waits for remote inputs. To reduce launch overhead, all local operations of an operator on a device are batched into a single kernel invocation ($x_M^b(u)$ and $x_R^b(u)$), as required by LOSS's computational efficiency constraint. Kernels launch once all required inputs

Algorithm 3 LOPInfer server at runtime stage

```
1:  $b \leftarrow$ RECEIVE(); Input(0) $\leftarrow \emptyset$
2:  for all layer $u$ in model do
3:      if $Input(u) \neq \emptyset$ and $x_R^b(u) \neq \emptyset$ then
4:          output(u) $\leftarrow$ COMPUTE(Input(u), $x_R^b(u)$)
5:      end if
6:      for all edge $e = (u, v) \in E$ do
7:          if $x_M^b(v)$ − child($x_M^b(u)$) $\neq \emptyset$ then
8:              SEND(Output(u), $x_M^b(v)$ − child($x_M^b(u)$))
9:          end if
10:         if ($x_R^b(v)$ − child($x_R^b(u)$) $\neq \emptyset$ then
11:             Input(v) $\leftarrow$ COMBINE(Output(u),Receive())
12:         end if
13:     end for
14: end for
```

are available; the scheduled start times ($s_M(u)$ and $s_R(u)$) are treated as soft targets rather than hard barriers, preserving correctness under runtime variability. The remaining blocking cost, dominated by tensor packing/combining, is minimized by the parallel execution of LOP.

When the schedule requires a cross-device transfer, the producer sends the output tensors to the peer upon kernel completion; the receiver materializes them into its input buffers and continues with downstream operators (lines 8 and 11 in Alg. 2 and Alg. 3). Otherwise, outputs are forwarded locally to subsequent operators without transmission. After all operators finish, the final inference result resides on the client for consumption by upper-layer applications (line 15 in Alg. 2).

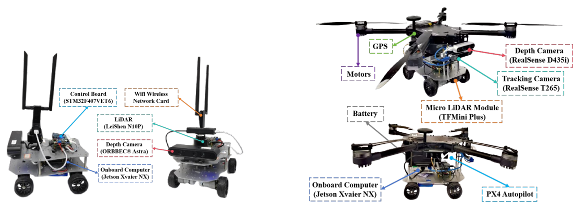
V. Implementation

In this section, we first describe the implementation of LOPInfer and then the experimental setup.

A. System Implementation

Software. We implement LOPInfer in Python on PyTorch as a lightweight, drop-in runtime embedded in the model's forward pass. During the first forward pass, LOPInfer uses PyTorch's profiler (torch.profiler) to collect operator-level execution traces and tensor sizes. From these traces, LOPInfer constructs a data dependency DAG among local operations and derives an optimized execution schedule. Subsequent inferences reuse this schedule, launching kernels across multiple CUDA streams and issuing non-blocking device–server transfers to expose inter-operator concurrency and, when dependencies allow, overlap computation with communication. Integration requires only three lines of application code (a context manager and two API calls) with no changes to the model architecture or inference workflow.

Hardware Testbed. We evaluated LOPInfer on two custom robotic platforms: a four-wheeled ground robot (Fig. 9a) and an air–ground hybrid robot (Fig. 9b). Each platform integrates an NVIDIA Jetson Xavier NX (8 GB) [14] for on-board inference. Both run Ubuntu 20.04 with ROS Noetic and use a dual-band USB Wi-Fi adapter (MediaTek MT76x2U) for wireless communication. Detailed hardware and sensor configurations are shown in Fig. 9. The GPU server is a desktop with an Intel i7-7700K CPU and an NVIDIA GeForce RTX 3080 GPU, connected to the robots over Wi-Fi 6 on a 5 GHz, 80 MHz channel.


Control Board (STM32F407VET6)
Wifi Wireless Network Card
LiDAR (LeiShen N10P)
Depth Camera (ORBBEC® Astra)
Onboard Computer (Jetson Xvaier NX)
GPS
Motors
Depth Camera (RealSense D435i)
Tracking Camera (RealSense T265)
Micro LiDAR Module (TFMini Plus)
Battery
Onboard Computer (Jetson Xvaier NX)
PX4 Autopilot

(a) Four-wheeled robot (b) Air-ground robot

Fig. 9: The detailed composition of the robot platforms.

	inference	communication	standby
Energy (Watt)	13.35	4.25	4.04

TABLE II: Power draw (Watt) of our robot in different states.

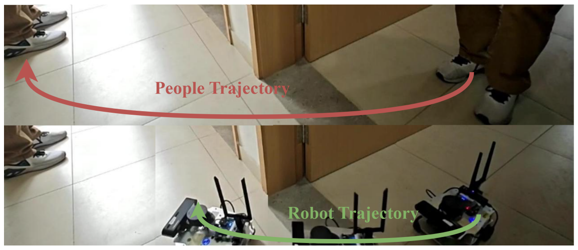


Fig. 10: Kapao [2], a real-time people-tracking application on our four-wheeled robot with a CNN-based keypoint detection model.

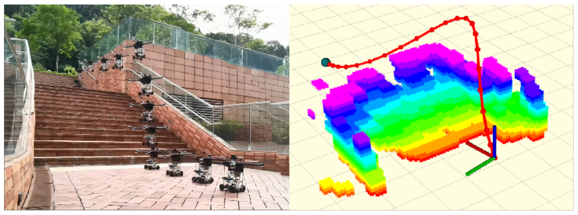


Fig. 11: AGRNav [3], a navigation application on our air-ground robot with a CNN-based 3D semantic scene completion model.

Table II reports on-board energy (excluding actuator/motor power) for the robot under three mutually exclusive modes: inference (on-device model execution, including CPU/GPU power), communication (data transmission, including the wireless network interface card) and standby (no application tasks, idle). Each Jetson Xavier NX is powered by a 21.6 Wh battery, supporting up to 1.6 hours of continuous model inference. We log instantaneous on-board power (in Watts) at 1 Hz using the back-end power and performance monitoring methodology in [14]. Energy per inference (in Joules) is computed by integrating the power trace over the exact inference interval, from the recorded start timestamp to the completion timestamp.

B. Experiment Setup

Task. We evaluate two real-world robotic workloads on our physical robot platforms: KAPAO for people tracking [2] (Fig. 10) and AGRNav for autonomous navigation [3] (Fig. 11). Low per-request (per-frame) inference latency is critical for closed-loop operation; as illustrated in Figs. 10 and 11, faster inference shortens perception–action delay and mitigates target loss and localization error. For evaluation, we run KAPAO on CrowdPose [38] and AGRNav on SemanticKITTI [39] using our testbed. To assess generality beyond robotics workloads, we also benchmark widely used

MEC models (VGGNet [32], ConvNeXt [34], ResNet [31], and DenseNet [29]), using their Torchvision implementations [40] on CIFAR-100 [41]. Across all experiments, the batch size is 1 to satisfy real-time MEC constraints. We define inference time as the wall-clock time from input arrival at the device to when the final inference output becomes available on the device, and use its mean and standard deviation as the primary summary, because elevated medians or heavy tails degrade responsiveness and can violate real-time deadlines.

Emulation Environments. We evaluated two deployment environments: indoor (a lab with desks and partitions that disrupt wireless signals) and outdoor (a garden with trees and bushes, yielding lower bandwidth). To control variability and ensure repeatability, we emulated the measured bandwidth traces in Fig. 4 on the Wi-Fi 6 link using Linux Traffic Control (tc). The traffic shaper updated the robot–server throughput cap every 0.5 s to track the target profile, and the same profiles were replayed for all methods and runs.

Baselines. To comprehensively evaluate LOPInfer, we compare against the following baselines:

- **Device-only inference ("Device-only")**: All layers execute on the mobile device.
- **Server-only inference ("Server-only")**: All layers execute on a GPU server. These two configurations serve as bounds for layer partitioning and contextualize latency and energy results.
- **DSCCS** [5]: A state-of-the-art (SOTA) layer-partitioning method for accelerating inference. For a fair comparison under time-varying wireless conditions, we pair DSCCS with LOPInfer's adaptive controller so it can react to bandwidth variability without altering the model or DSCCS's optimization objective.
- **SPSO-GA** [8]: An SOTA energy-optimized layer-partitioning method under timing constraints. We configure SPSO-GA with a 1 Hz control deadline (one-second control period, the minimum frequency required for effective robotic motion control) and likewise integrate LOPInfer's adaptive controller for real-time adaptation.

All systems use the same uncompressed DNN and transmit raw, lossless intermediate activations to preserve accuracy. Profiling and scheduling decisions for LOPInfer and all baselines are computed offline before experiments. As discussed in Sec. II-D, DP is inapplicable because the batch size is 1. TP is also unsuitable: on our hardware, its end-to-end latency (Fig. 5) is substantially higher than device-only. Given the batch size of 1 and the absence of throughput gains from pipelining for single-request inference, PP degenerates to layer partitioning; accordingly, we evaluate DSCCS and SPSO-GA as the SOTA PP baselines in this regime (without pipeline execution). Methods that trade accuracy for efficiency are beyond the scope of this paper.

VI. Performance Evaluation

In this section, we evaluate LOPInfer along three axes: i) head-to-head comparisons with baselines on representative MEC service workloads to quantify end-to-end latency and energy gains; ii) a micro-event analysis of LOSS that profiles per-operator timelines to expose compute–transfer overlap and bottlenecks; and iii) sensitivity studies across bandwidth budgets, diverse model architectures, and varying device/server compute capacities to assess robustness and scalability.

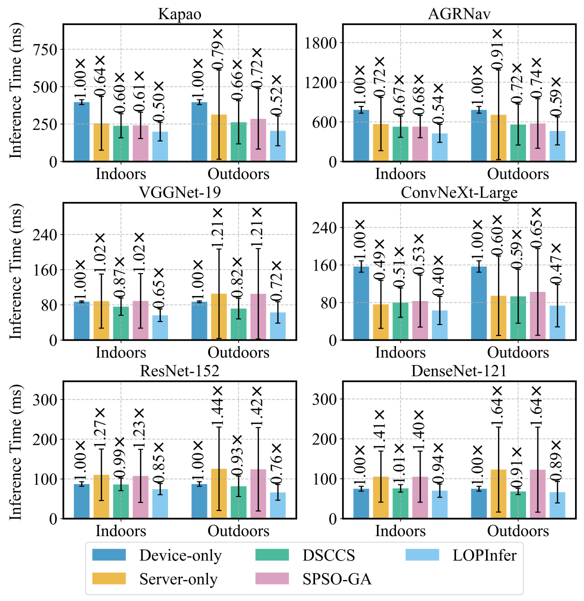


Fig. 12: Inference time for different models across various environments and systems. The cross marker (×) denotes the mean value.

A. Superiority of LOPInfer

Inference Time. Fig. 12 shows LOPInfer outperforming four baselines across diverse tasks and environments, cutting end-to-end latency by up to 50% indoors and 48% outdoors. Against the closest competitor, DSCCS, LOPInfer delivers 8–26% lower latency indoors and 8–22% outdoors. These gains come from LOP, which exposes operator-level parallelism and overlaps computation with communication within a single request, and LOSS, which produces low-latency, energy-efficient placement of local operations. All offloading-based methods (including server-only, the two layer-partitioning baselines, and LOPInfer) exhibit larger variance and longer tails outdoors due to severe bandwidth volatility (Fig. 4). The improvement on DenseNet is less pronounced; we analyze model-structure sensitivity in Sec. VI-C. A micro-event study (Sec. VI-B) further explains LOPInfer's efficiency.

Energy Consumption. Fig. 13 reports device-side power draw across systems and scenarios. As expected, device-only draws the most power because all computation runs on the robot, whereas server-only minimizes client power by fully offloading to the GPU server. Among partial-offloading methods, SPSO-GA achieves lower energy per inference than DSCCS due to its energy-oriented placement; LOPInfer incurs slightly higher device energy than SPSO-GA because it occasionally recomputes local operations to enable compute–communication overlap and controlled replication. The average power of LOPInfer and DSCCS is nearly identical, indicating that LOPInfer's overlap adds negligible instantaneous power; energy differences stem primarily from execution time and minor replication. Consistent with this, Tab. II shows the robot sustains roughly 95% of its active power even when

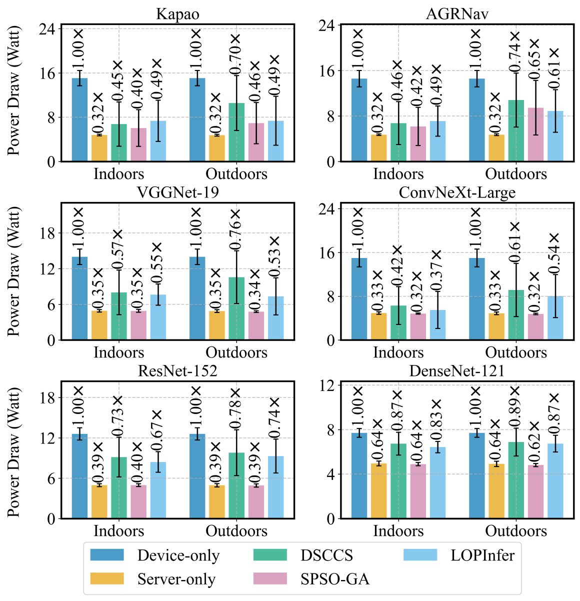


Fig. 13: Power draw for different models across various environments and systems. The cross marker (×) denotes the mean value.

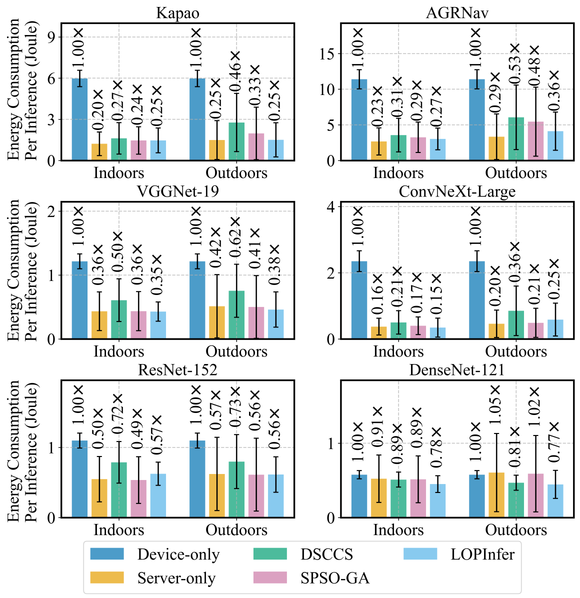


Fig. 14: Energy consumption per inference request for different models across various environments and systems. The cross marker (×) denotes the mean value.

idle, dominated by CPU/GPU/memory static leakage [11]; the wireless NIC adds only 0.21 W during transmission versus 13.35 W during computation.

Fig. 14 reports device energy per inference across systems and scenarios. Consistent with the power profiles in Fig. 13, LOPInfer may draw slightly higher average power than DSCCS, but its shorter runtime yields lower energy than all partial-offloading baselines, reducing device energy by up to 75% indoors and 72% outdoors. Versus the most energy-efficient baseline, server-only, LOPInfer raises device energy by at most 20% while cutting average latency by up to 42% and substantially shortening tail latency, offering a favorable latency–energy trade-off. This gap arises because LOPInfer optimizes latency, not energy:additional local computation

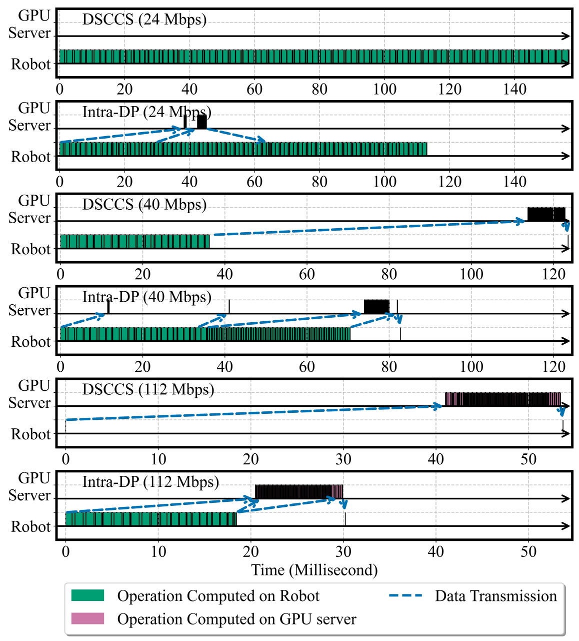


Fig. 15: Snapshots of schedule plan of LOPInfer and baseline during runtime under various network bandwidth.

raises device power, which the latency reduction cannot fully offset in energy terms. On lower-power mobile devices, such local computation can be particularly energy-inefficient, potentially increasing absolute energy per inference. Making LOPInfer energy-aware (Sec. IV-C3) is left to future work.

B. Micro-Event Analysis

Next, we examine why LOPInfer achieves lower inference latency via a micro-event analysis, with a focus on the effectiveness of LOSS relative to baseline methods.

Detailed Schedule Plan. We profile ConvNeXt under time-varying bandwidth and log per-operator micro-events (device/server kernel launches/completions and network send/receive) to reconstruct execution timelines (Fig. 15). Relative to DSCCS, LOPInfer's primary benefit comes from operation-level parallelism via LOP, enabling compute–communication overlap within a single request. The adaptive controller selects bandwidth-aware layer partitions for both DSCCS and LOPInfer. At low bandwidth, DSCCS tends to allocate more layers to the device (often near device-only) to curb transfer cost, reducing tails but increasing device compute. In contrast, LOPInfer preserves distributed execution at low bandwidth with a finer-grained operation-level schedule (parallel execution and limited replication of local operations), achieving useful server acceleration at lower bandwidth via increased overlap and reduced idle time. At high bandwidth, DSCCS often converges to server-only to avoid device compute, whereas LOPInfer leverages LOSS to balance network and GPU server utilization, sustaining high overlap and improving end-to-end latency. These traces align with higher overlap ratios and shorter idle time on LOPInfer's timelines versus DSCCS.

Breakdown. Fig. 16 breaks down per-request time by phase for ConvNeXt. In DSCCS, network transfer accounts for up to 42% of end-to-end latency, revealing a persistent transmission

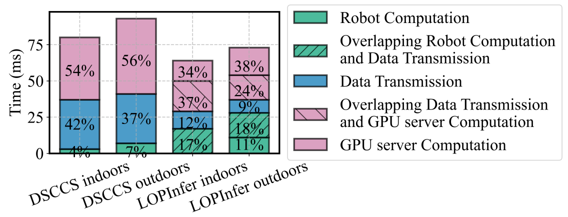


Fig. 16: Breakdown of each phase of the inference process.

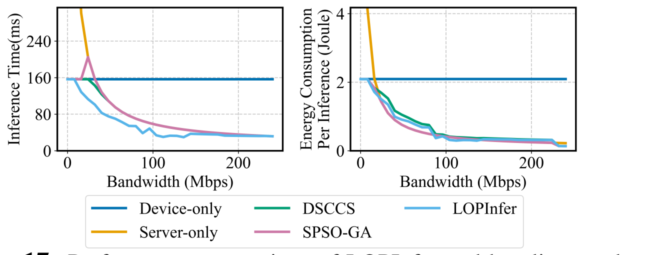


Fig. 17: Performance comparison of LOPInfer and baselines under different network bandwidth conditions.

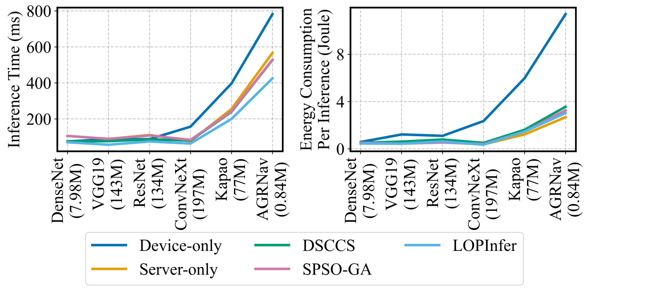


Fig. 18: Performance comparison of LOPInfer and baselines with varying model structures.

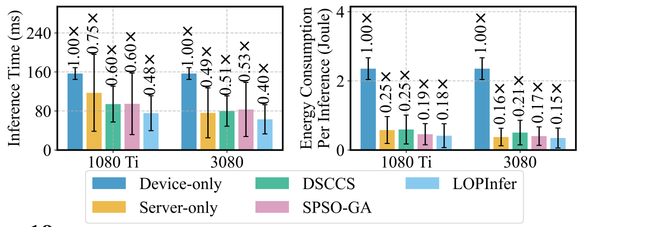


Fig. 19: Performance comparison of LOPInfer and baselines on GPU servers with varying computational power.

bottleneck in SOTA layer-partitioning baselines. In contrast, LOPInfer runs transfer, robot compute, and GPU-server compute in parallel within a single request, reducing end-to-end latency. Despite LOPInfer issues fine-grained, operation-level transfers with higher volume, this overhead is offset by compute–communication overlap and early send (Fig. 15), shortening the makespan. Consequently, LOPInfer shows lower idle time and higher overlap than layer-partitioning methods under the same bandwidth.

Offline Cost for Precomputing Plans. For fairness, we equip all baselines with the same adaptive controller as LOPInfer and precompute a schedule table over TCP bandwidth buckets from 0–30 MB/s at 1 MB/s resolution (31 entries per model), generating entries in parallel across all CPU cores. For each bucket, the planner solves a schedule under the fixed bandwidth; Tab. III reports the total offline time per model to populate the table. DSCCS completes table generation within seconds, whereas LOPInfer takes longer due to operation-level partitioning, feasibility checks, and optional replication under the constraints in Sec. IV-C4. This one-time cost is paid per (model, hardware) configuration and has no impact on runtime latency, since adaptation reduces to a cached table lookup.

TABLE III: Offline time (seconds) to precompute plans.

	Kapao	AGRNav	VGGNet-19
DSCCS	5.09	1.01	2.96
LOPInfer	1149.32	34.94	53.63
	ConvNeXt-Large	**ResNet-152**	**DenseNet-121**
DSCCS	3.63	3.63	4.75
LOPInfer	127.61	73.87	162.53

C. Sensitivity Studies

Network Bandwidth. To assess LOPInfer across bandwidths, we use Linux Traffic Control (tc) on a wired Ethernet link to emulate controlled bandwidth budgets, enabling fair baseline comparisons. As shown in Fig. 17, LOPInfer attains lower end-to-end latency over a wider bandwidth range by combining LOP (exposing local-operation-level parallelism and compute–communication overlap) with LOSS (producing bandwidth-aware placements), seeded/pruned by device-only, server-only, and layer-partitioning plans. At very low bandwidths (near 0 MB/s), both LOPInfer and DSCCS converge to device-only, but LOPInfer's device-only threshold is substantially lower. At very high bandwidths, both converge to server-only, and LOPInfer's server-only threshold is higher. These thresholds—the smallest (largest) bandwidth at which the planner selects server-only (device-only)—depend on model architecture and device/server compute capacities. Finally, the improvements in Figs. 12 and 14 are smaller than in Fig. 17 because runtime bandwidth estimation introduces noise and temporal mismatch that can occasionally select suboptimal buckets.

Model Structure. As shown in Fig. 18, offloading methods (excluding device-only) deliver larger latency reductions on compute-intensive models (more model parameters) but can underperform device-only on DenseNet. This reflects the compute–communication tradeoff: when activations are large and per-layer kernels small (low compute-to-communication ratio), transfer overhead dominates. LOPInfer consistently outperforms other offloading baselines, yet its margin narrows on architectures with limited local operators, since its gains stem from operation-level parallelism and compute–communication overlap. Thus, with modest compute or minimal operation-level concurrency, fewer overlap opportunities limit LOPInfer's headroom over layer partitioning.

Device/Server Compute Capacity. To assess sensitivity to device/server compute imbalance, we evaluate LOPInfer with a GTX 1080 Ti server (Fig. 19). As the gap of device/server compute capacity widens, offloading methods deliver larger latency reductions over device-only. For any gap, LOPInfer achieves the largest reduction among offloading baselines. The gains stem from LOP and LOSS, which expose local-operation-level parallelism and sustain compute–communication overlap, boosting server utilization at the same bandwidth. Because our client (Fig. 2a) is far more capable than typical smartphones, LOPInfer should yield even greater benefits on more resource-constrained mobile devices.

VII. Related work and discussion

Limited bandwidth. Due to hardware availability, our real-world evaluation used commodity Wi-Fi typical in robotic deployments rather than a broader set of radio access technologies (e.g., 5G). Despite differing peak rates and coverage, practical wireless links often exhibit variable quality and fluctuating transport-layer bandwidth [19]–[21]. Under such conditions, LOPInfer remains effective because its controller adapts to measured bandwidth and its variability, making the mechanism transport-layer-agnostic. When GPU servers run in public clouds, end-to-end congestion and suboptimal routing can further reduce available bandwidth [42], increasing the relative benefit of LOPInfer. Finally, consistent with our sensitivity analysis (Sec. VI-C), LOPInfer's gains grow with model compute intensity and server capability, yielding larger improvements over baselines.

Inference Request Scheduling. Prior work [43]–[45] schedules concurrent DNN inference at the request level, assigning different layer-partitioning strategies per request based on optimization goals and current system state, thereby improving overall latency–energy trade-offs. While these system-level schedulers coordinate across requests, LOPInfer provides a high-performance per-request primitive via local-operation-level scheduling. Thus, existing and future inference request schedulers can integrate LOPInfer as a drop-in request-level primitive to boost per-request efficiency and, in turn, enhance overall system performance.

Model Compression. Quantization and knowledge distillation reduce compute and memory footprints—and often the amount of transferred activations—by lowering numerical precision or training compact student models [46]. Hybrid offloading [47] integrates model compression with layer partitioning via split-aware compressed representations. Orthogonal to compression, LOPInfer accelerates inference without accuracy loss by keeping the model architecture and precision unchanged and instead redistributing exact computation between device and server through operation-level scheduling. As a result, it applies directly to already compressed models and exploits their smaller activations to further reduce transfer overhead. Building on LOPInfer's gains, future work will jointly optimize accuracy–efficiency trade-offs (e.g., selecting quantization levels and early-exit policies [25] alongside local operations) under bandwidth, latency, energy, and accuracy constraints to further improve MEC performance.

Future Work. We will extend LOPInfer to a broader class of local operators, including Transformer primitives (e.g., attention softmax, multi-head aggregation, and normalization layers) and design lightweight synchronization for global operators by exchanging compact sufficient statistics instead of full tensors. For example, in softmax, synchronizing the global maximum (for numerical stability) and the sum of exponentials (via log-sum-exp accumulation) allows each side to complete normalization locally, reducing bandwidth and latency without loss of accuracy. This operator-aware synchronization generalizes to attention score aggregation, normalization, and other reductions; its feasibility is supported by FlashAttention [48]. In particular, in vision transformers, patch-wise computations (e.g., patch embedding/projection) are also block-local operations under our definition.

VIII. Conclusion

In this paper, we presented LOPInfer, a high-performance local-operator parallel inference system for MEC service workloads. LOPInfer combines LOP, which exposes local-operation-level parallelism, with LOSS, which produces intra- and cross-layer compute–communication overlap schedules for high-performance inference Across representative workloads and network conditions, LOPInfer consistently reduces end-to-end latency and device energy versus baselines, expanding the ML applications deployable on resource-constrained mobile devices.

References

[1] R. Zhou, Y. Huang, Y. Wang, L. Jiao, H. Tan, R. Zhang, and L. Wu, "User Preference Oriented Service Caching and Task Offloading for UAV-Assisted MEC Networks," *IEEE Trans. Serv. Comput.*, vol. 18, no. 2, pp. 1097–1109, 2025.

[2] W. McNally, K. Vats, A. Wong, and J. McPhee, "Rethinking Keypoint Representations: Modeling Keypoints and Poses as Objects for Multi-Person Human Pose Estimation," in *Proc. ECCV*, 2022, pp. 37–54.

[3] J. Wang, Z. Sun, X. Guan, T. Shen, Z. Zhang, T. Duan, D. Huang, S. Zhao, and H. Cui, "AGRNav: Efficient and Energy-Saving Autonomous Navigation for Air-Ground Robots in Occlusion-Prone Environments," in *Proc. ICRA*, May 2024.

[4] Z. Wang, M. Goudarzi, and R. Buyya, "TF-DDRL: A Transformer-Enhanced Distributed DRL Technique for Scheduling IoT Applications in Edge and Cloud Computing Environments," *IEEE Trans. Serv. Comput.*, vol. 18, no. 2, pp. 1039–1053, 2025.

[5] 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 Trans. Cloud Comput.*, May 2023.

[6] C. Hu, W. Bao, D. Wang, and F. Liu, "Dynamic Adaptive DNN Surgery for Inference Acceleration on the Edge," in *Proc. INFOCOM*, Apr. 2019, pp. 1423–1431.

[7] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge," in *Proc. ASPLOS*, Apr. 2017, pp. 615–629.

[8] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter, and G. Min, "Energy-Efficient Offloading for DNN-Based Smart IoT Systems in Cloud-Edge Environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 3, pp. 683–697, Mar. 2021.

[9] Y. Chen, Q. Zhang, R. Xing, Y. Li, X. Ma, Y. Zhang, A. Zhou, and S. Wang, "SLICE: Energy-Efficient Satellite-Ground Co-Inference via Layer-Wise Scheduling Optimization," *IEEE Trans. Serv. Comput.*, vol. 18, no. 4, pp. 2388–2402, 2025.

[10] L. Gao, J. Liu, H. Xu, S. Xu, Q. Ma, and L. Huang, "Accelerating End-Cloud Collaborative Inference via Near Bubble-Free Pipeline Optimization," *arXiv preprint arXiv:2501.12388*, Jan. 2024.

[11] N. S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Law Meets Static Power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003.

[12] A. Ghosh, S. Iyengar, S. Lee, A. Rathore, and V. N. Padmanabhan, "REACT: Streaming Video Analytics on the Edge with Asynchronous Cloud Support," in *Proc. IoTDI*, 2023, pp. 222–235.

[13] "MLPerf Mobile Benchmarks," https://mlcommons.org/working-groups/benchmarks/mobile/, 2023.

[14] NVIDIA, "Jetson Xavier NX Series: The World's Smallest AI Supercomputer," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[15] Z. Ning, M. Vandersteegen, K. Van Beeck, T. Goedemé, and P. Vandewalle, "Power Consumption Benchmark for Embedded AI Inference," in *Proc. Int. Conf. Applied Comput. WWW/Internet*. IADIS Press, Mar. 2024, pp. 3–10.

[16] NVIDIA, "InfiniBand Networking Solutions," https://www.nvidia.com/en-us/networking/products/infiniband/, 2024.

[17] R. Liu and N. Choi, “A First Look at Wi-Fi 6 in Action: Throughput, Latency, Energy Efficiency, and Security,” in *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, no. 1, Mar. 2023, pp. 1–25.

[18] X. Yang, H. Lin, Z. Li, F. Qian, X. Li, Z. He, X. Wu, X. Wang, Y. Liu, Z. Liao *et al.*, “Mobile Access Bandwidth in Practice: Measurement, Analysis, and Implications,” in *Proc. SIGCOMM*, Aug. 2022, pp. 114–128.

[19] A. Masiukiewicz, “Throughput Comparison between The New HEW 802.11 ax Standard and 802.11 n/ac Standards in Selected Distance Windows,” *Int. J. Electron. Telecommun.*, vol. 65, no. 1, pp. 79–84, 2019.

[20] M. Ding, P. Wang, D. López-Pérez, G. Mao, and Z. Lin, “Performance Impact of LoS and NLoS Transmissions in Dense Cellular Networks,” *IEEE Trans. Wireless Commun.*, vol. 15, no. 3, pp. 2365–2380, Mar. 2015.

[21] Y. Ren, C.-W. Tung, J.-C. Chen, and F. Y. Li, “Proportional and Preemption-Enabled Traffic Offloading for IP Flow Mobility: Algorithms and Performance Evaluation,” *IEEE Trans. Veh. Technol.*, vol. 67, no. 12, pp. 12 095–12 108, Dec. 2018.

[22] “iPerf - Download iPerf3 and Original iPerf Pre-Compiled Binaries,” https://iperf.fr/iperf-download.php, 2022.

[23] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco, “Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading,” in *Proc. INFOCOM*, Jul. 2020, pp. 854–863.

[24] H. Wang, L. Wang, H. Xu, Y. Wang, Y. Li, and Y. Han, “PrimePar: Efficient Spatial-Temporal Tensor Partitioning for Large Transformer Model Training,” in *Proc. ASPLOS*, Apr. 2024, pp. 801–817.

[25] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, “SPINN: Synergistic Progressive Inference of Neural Networks Over Device and Cloud,” in *Proc. MobiCom*. Association for Computing Machinery, Sep. 2020.

[26] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, “DeepSlicing: Collaborative and Adaptive CNN Inference with Low Latency,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 9, pp. 2175–2187, Sep. 2021.

[27] K. Bin, J. Park, C. Park, S. Kim, and K. Lee, “CoActo: CoActive Neural Network Inference Offloading with Fine-grained and Concurrent Execution,” in *Proc. MobiSys*, Jun. 2024, pp. 412–424.

[28] J. Yao, S. S. Kanhere, and M. Hassan, “An Empirical Study of Bandwidth Predictability in Mobile Computing,” in *Proc. WiNTECH*, Sep. 2008, pp. 11–18.

[29] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” *arXiv preprint arXiv:1608.06993*, Aug. 2018.

[30] J. Xu, Y. Pan, X. Pan, S. Hoi, Z. Yi, and Z. Xu, “RegNet: Self-Regulated Network for Image Classification,” *IEEE Trans. Neural Netw. Learn. Syst.*, Aug. 2022.

[31] S. Targ, D. Almeida, and K. Lyman, “ResNet in ResNet: Generalizing Residual Architectures,” *arXiv preprint arXiv:1603.08029*, Mar. 2016.

[32] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *arXiv preprint arXiv:1409.1556*, Apr. 2015.

[33] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, “Aggregated Residual Transformations for Deep Neural Networks,” *arXiv preprint arXiv:1611.05431*, Jan. 2017.

[34] S. Woo, S. Debnath, R. Hu, X. Chen, Z. Liu, I. S. Kweon, and S. Xie, “ConvNeXt V2: Co-Designing and Scaling ConvNets with Masked Autoencoders,” in *Proc. CVPR*, Jun. 2023, pp. 16 133–16 142.

[35] R. Poorzare and A. C. Augé, “How Sufficient Is TCP When Deployed in 5G mmWave Networks over the Urban Deployment?” *IEEE Access*, vol. 9, pp. 36 342–36 355, Mar. 2021.

[36] A. K. Qin, V. L. Huang, and P. N. Suganthan, “Differential Evolution Algorithm with Strategy Adaptation for Global Numerical Optimization,” *IEEE Trans. Evol. Comput.*, vol. 13, no. 2, pp. 398–417, Apr. 2008.

[37] J. Ji, K. Zhu, and L. Cai, “Trajectory and Communication Design for Cache-Enabled UAVs in Cellular Networks: A Deep Reinforcement Learning Approach,” *IEEE Trans. Mobile Comput.*, vol. 22, no. 10, pp. 6190–6204, Oct. 2023.

[38] J. Li, C. Wang, H. Zhu, Y. Mao, H.-S. Fang, and C. Lu, “CrowdPose: Efficient Crowded Scenes Pose Estimation and a New Benchmark,” in *Proc. CVPR*, Jun. 2019, pp. 10 855–10 864.

[39] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall, “SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences,” in *Proc. ICCV*, Oct. 2019, pp. 9296–9306.

[40] S. Marcel and Y. Rodriguez, “Torchvision the Machine-Vision Package of Torch,” in *Proc. ACM Multimedia*. Association for Computing Machinery, Oct. 2010, pp. 1485–1488.

[41] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” University of Toronto, Tech. Rep., 2009.

[42] M. Noormohammadpour and C. S. Raghavendra, “Datacenter Traffic Control: Understanding Techniques and Tradeoffs,” *IEEE Commun. Surv. Tutor.*, vol. 20, no. 2, pp. 1492–1525, Jun. 2017.

[43] C. Sun, X. Li, C. Wang, Q. He, X. Wang, and V. C. M. Leung, “Hierarchical Deep Reinforcement Learning for Joint Service Caching and Computation Offloading in Mobile Edge-Cloud Computing,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 4, pp. 1548–1564, 2024.

[44] C. Tang, Y. Ding, S. Xiao, Z. Huang, and H. Wu, “Collaborative Service Caching, Task Offloading, and Resource Allocation in Caching-Assisted Mobile Edge Computing,” *IEEE Trans. Serv. Comput.*, vol. 18, no. 4, pp. 1966–1981, 2025.

[45] J. Cai, W. Liu, Z. Huang, and F. R. Yu, “Task Decomposition and Hierarchical Scheduling for Collaborative Cloud-Edge-End Computing,” *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 4368–4382, 2024.

[46] L. Wang and K.-J. Yoon, “Knowledge Distillation and Student-Teacher Learning for Visual Intelligence: A Review and New Outlooks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, pp. 3048–3068, Jun. 2021.

[47] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, “BottleFit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing,” in *Proc. WoWMoM*, Jun. 2022, pp. 337–346.

[48] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness,” in *Proc. NeurIPS*, Dec. 2022, pp. 16 344–16 359.

Zekai Sun is a Ph.D. student at the University of Hong Kong. His research interests include distributed systems, edge computing, and systems for machine learning.

Xiuxian Guan is an HKU–SUSTech joint Ph.D. student in Computer Science (HKU) and Electrical and Electronic Engineering (SUSTech). His research interests include distributed systems, edge computing, and systems for machine learning.

Zheng Lin is a Ph.D. student at the University of Hong Kong. His research interests include wireless networking, edge intelligence, and distributed machine learning.

Zihan Fang is a Ph.D. student in Computer Science at the City University of Hong Kong. Her research interests include vehicular networks and distributed machine learning.

Xiangming Cai is with the School of Physics and Information Engineering, Fuzhou University. His research interests include wireless communications and underwater acoustic communications.

Zhe Chen is an Assistant Professor at the School of Computer Science, Fudan University. His research interests include computer networking and mobile systems.

Fangming Liu is a Full Professor at Huazhong University of Science and Technology. His research interests include cloud and edge computing, data center and green computing, SDN/NFV/5G, and applied machine learning/AI.

Heming Cui is an Associate Professor in Computer Science at the University of Hong Kong. His research interests include operating systems, programming languages, distributed systems, and cloud computing, with a focus on software reliability and security.

Jie Xiong is an Associate Professor in the College of Computing and Data Science at Nanyang Technological University. His research interests include wireless sensing, mobile computing, and smart health.

Wei Ni [Fellow, IEEE] is a Professor at the School of Engineering, Edith Cowan University, and a Conjoint Professor at the University of New South Wales. His research interests include machine learning, online learning, stochastic optimization, and their applications to system efficiency and integrity.

Jun Luo [Fellow, IEEE] is a Professor at the School of Computer Science and Engineering, Nanyang Technological University. His research interests include mobile and pervasive computing, wireless networking, machine learning and computer vision, security and privacy, and applied operations research.