UNIVERSITY OF HONG KONG

DOCTORAL THESIS

Efficient and Robust Machine Learning Systems for Mobile Edge Computing

Author:
Zekai SUN

Supervisor:
Prof. Heming CUI

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

School of Computing and Data Science

June 18, 2026

Abstract of thesis entitled

Efficient and Robust Machine Learning Systems for Mobile Edge Computing

Submitted by

Zekai SUN

for the degree of Doctor of Philosophy

at The University of Hong Kong

in June, 2026

Mobile Edge Computing (MEC) is becoming a key substrate for machine learning (ML) applications that sense, decide, and act near the physical world. By placing computation close to mobile devices, robots, and sensors, MEC shortens the path between data generation and intelligent response. This capability is important for robotic IoT, autonomous navigation, mobile perception, and latency-sensitive services, where useful intelligence requires online adaptation, timely inference, and practical deployment. However, cloud ML systems cannot be transplanted directly to MEC because their execution paths become coupled with unstable wireless links, mobile energy limits, and heterogeneous edge hardware.

The central problem studied in this thesis is a *granularity mismatch*. Conventional ML systems inherit data-center execution units: whole models for gradient synchronization, whole layers for collaborative inference, and per-operator remote procedure calls for transparent offloading. These software boundaries are misaligned with wireless fluctuation, intra-request overlap opportunities, and repeated ML runtime sequences. The mismatch appears as training stragglers, inference transmission serialization, and deployment round-trip amplification. This thesis addresses it through *granularity-aware execution*: selecting execution granularity according to workload structure, wireless dynamics, and deployment constraints.

To support stronger intelligence, this thesis presents ROG, a row-granulated distributed training system for robotic IoT. ROG observes that robotic wireless bandwidth can change while gradients are still being transmitted, making whole-model synchronization too coarse. It changes synchronization granularity from whole models to parameter rows through Row Synchronous Parallel and an Adaptive Transmission Protocol. ROG improves training accuracy by 4.9%–6.5% under the same time budget and reduces energy consumption by 20.4%–50.7% for the same target accuracy.

To support faster feedback, this thesis presents LOPInfer, a local-operator parallel inference system for MEC service workloads. LOPInfer observes that layer-wise partitioning serializes device computation, wireless transmission, and server computation

within each request, although many DNN operators expose finer local dependencies. It changes scheduling granularity from whole layers to local operations, enabling computation and communication to overlap within one inference. LOPInfer reduces per-inference latency by up to 50% and device energy by up to 75% while preserving model semantics.

To support practical deployment, this thesis presents RRTO, a record/replay transparent offloading system for MEC inference. RRTO observes that ML inference repeatedly executes a stable operator sequence, so forwarding each low-level operator through a separate RPC is unnecessarily reactive. It changes transparent control granularity from per-operator RPCs to per-inference operator-sequence replay. RRTO reconstructs the steady-state sequence through a two-stage Operator Sequence Search and replays each inference as one remote execution, reducing latency and energy by up to 98% without source-code modification.

Together, ROG, LOPInfer, and RRTO show that MEC intelligence should be designed around the execution unit at which a system synchronizes, schedules, or controls ML work. ROG lowers synchronization from whole models to parameter rows, LOPInfer lowers scheduling from whole layers to local operations, and RRTO raises transparent control from per-operator RPCs to sequence-level replay. Across training, inference, and deployment, these systems instantiate *granularity-aware execution* for adaptive, responsive, and deployable edge ML. All three systems have been open-sourced.

(500 words)

Efficient and Robust Machine Learning Systems for Mobile Edge Computing

by

Zekai SUN
Ph.D HKU

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy

at

University of Hong Kong
June, 2026

Declaration

I, Zekai SUN, declare that this thesis titled, "Efficient and Robust Machine Learning Systems for Mobile Edge Computing", which is submitted in fulfillment of the requirements for the Degree of Doctor of Philosophy, represents my own work except where due acknowledgement have been made. I further declared that it has not been previously included in a thesis, dissertation, or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

Signed: Zekai Sun

Date: June 18, 2026

For Mama and Papa

Acknowledgements

I would like to express my sincere gratitude to my supervisor, collaborators, labmates, friends, and family for their guidance, support, and encouragement throughout my doctoral study. This section will be further personalized before final submission.

Zekai SUN
University of Hong Kong
June 18, 2026

List of Publications

JOURNALS:

[1] **Zekai Sun**, Xiuxian Guan, Zheng Lin, Yuhao Qing, Haoze Song, Zihan Fang, Zhe Chen, Fangming Liu, Heming Cui, Wei Ni, and Jun Luo, "RRTO: A High-Performance Transparent Offloading System for Model Inference in Mobile Edge Computing," *IEEE Transactions on Mobile Computing*, minor revision.

[2] **Zekai Sun**, Xiuxian Guan, Zheng Lin, Zihan Fang, Xiangming Cai, Zhe Chen, Fangming Liu, Heming Cui, Jie Xiong, Wei Ni, and Jun Luo, "LOPInfer: A High-Performance Local-Operator Parallel Inference System for Service Workloads in Mobile Edge Computing," *IEEE Transactions on Services Computing*, major revision.

[3] Zongyuan Zhang, Tianyang Duan, Zheng Lin, Dong Huang, Zihan Fang, **Zekai Sun**, Ling Xiong, Hongbin Liang, Heming Cui, and Yong Cui, "State-Aware Perturbation Optimization for Robust Deep Reinforcement Learning," *IEEE Transactions on Mobile Computing*, 2025.

[4] Junming Wang, Xiuxian Guan, **Zekai Sun**, Tianxiang Shen, Dong Huang, Fangming Liu, and Heming Cui, "OMEGA: Efficient Occlusion-Aware Navigation for Air-Ground Robots in Dynamic Environments Via State Space Model," *IEEE Robotics and Automation Letters*, 2024.

[5] Junming Wang, **Zekai Sun**, Xiuxian Guan, Tianxiang Shen, Dong Huang, Zongyuan Zhang, Tianyang Duan, Fangming Liu, and Heming Cui, "HE-Nav: A High-Performance and Efficient Navigation System for Aerial-Ground Robots in Cluttered Environments," *IEEE Robotics and Automation Letters*, 2024.

[6] Shengliang Deng, Xiuxian Guan, **Zekai Sun**, Shixiong Zhao, Tianxiang Shen, Xusheng Chen, Tianyang Duan, Yuexuan Wang, Jia Pan, Yanjun Wu, Libo Zhang, and Heming Cui, "COORP: Satisfying Low-Latency and High-Throughput Requirements of Wireless Network for Coordinated Robotic Learning," *IEEE Internet of Things Journal*, 2022.

CONFERENCES:

[1] Xiuxian Guan*, **Zekai Sun***, Shengliang Deng, Xusheng Chen, Shixiong Zhao, Zongyuan

Zhang, Tianyang Duan, Yuexian Wang, Chenshu Wu, Yong Cui, Libo Zhang, Yanjun Wu, Rui Wang, and Heming Cui, "ROG: A High Performance and Robust Distributed Training System for Robotic IoT," in *ACM/IEEE International Symposium on Microarchitecture*, 2022. ACM Results Reproduced Badge. *Equal contribution.

[2] Zekai Sun, Xiuxian Guan, Junming Wang, Fangming Liu, and Heming Cui, "New Problems in Distributed Inference for DNN Models on Robotic IoT," in *Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating algorithms for Distributed systems*, 2024.

[3] Haoze Song, Xusheng Chen, Ruijie Gong, **Zekai Sun**, Tianxiang Shen, Cheng Li, Hao Feng, Sen Wang, and Heming Cui, "Perseus: Achieving Strong Consistency and High Data Freshness for Scalable Geo-distributed HTAP," in *ACM Conference on Management of Data*, 2026.

[4] Zongyuan Zhang, Tianyang Duan, Zheng Lin, Dong Huang, Zihan Fang, **Zekai Sun**, Ling Xiong, Hongbin Liang, Heming Cui, Yong Cui, and Yue Gao, "Robust Deep Reinforcement Learning in Robotics via Adaptive Gradient-Masked Adversarial Attacks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2025.

[5] Guichao Zhu, Lintian Lei, Yuhao Qing, Yichao Fu, Fanxin Li, Dong Huang, **Zekai Sun**, and Heming Cui, "FOLDMOE: Efficient Long Sequence MoE Training via Attention-MoE Pipelining," in *Annual Meeting of the Association for Computational Linguistics*, 2025.

[6] Zongyuan Zhang, Tianyang Duan, **Zekai Sun**, Xiuxian Guan, Junming Wang, Hongbin Liang, Yong Cui, and Heming Cui, "Prediction-based Hierarchical Reinforcement Learning for Robot Soccer," in *IEEE/CIC International Conference on Communications in China*, 2024.

[7] Junming Wang, **Zekai Sun**, Xiuxian Guan, Tianxiang Shen, Zongyuan Zhang, Tianyang Duan, Dong Huang, Shixiong Zhao, and Heming Cui, "AGRNav: Efficient and Energy-Saving Autonomous Navigation for Air-Ground Robots in Occlusion-Prone Environments," in *IEEE International Conference on Robotics and Automation*, 2024.

[8] Xiuxian Guan, **Zekai Sun**, Shengliang Deng, Shixiong Zhao, Tianxiang Shen, Tsz On Li, Yuexuan Wang, Rui Wang, and Heming Cui, "MVSAS: Semantic-Aware Scheduling for Low Latency and High Precision in Wireless Multi-View Applications," in *IEEE International Conference on Parallel and Distributed Systems*, 2021.

[9] Xiuxian Guan, Yawei Li, Yuexuan Wang, **Zekai Sun**, Shengliang Deng, and Heming Cui, "CTDMA: Color-aware TDMA Network System For Low latency and High Throughput in Dense D2D Wireless Network," in *IEEE International Conference on Mobile Ad Hoc and Sensor Systems*, 2020.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
ATP	**A**daptive **T**ransmission **P**rotocol
BSP	**B**ulk **S**ynchronous **P**arallel
CRIMP	**C**oordinated **R**obotic **I**mplicit **M**apping and **P**ositioning
CRUDA	**C**oordinated **R**obotic **U**nsupervised **D**omain **A**daptation
DNN	**D**eep **N**eural **N**etwork
DP	**D**ata **P**arallelism
GPU	**G**raphics **P**rocessing **U**nit
IoT	**I**nternet **o**f **T**hings
IOS	**I**nference **O**perator **S**equence
LOP	**L**ocal-**O**perator **P**arallelism
MEC	**M**obile **E**dge **C**omputing
ML	**M**achine **L**earning
MTA	**M**inimum **T**ransmission **A**mount
RPC	**R**emote **P**rocedure **C**all
RSP	**R**ow **S**ynchronous **P**arallel
RTT	**R**ound-**T**rip **T**ime
SGD	**S**tochastic **G**radient **D**escent
SSP	**S**tale **S**ynchronous **P**arallel
TP	**T**ensor **P**arallelism

List of Symbols

Global notations

w	model parameters	—
g	gradients computed by a worker	—
$\bar{g}$	averaged gradients maintained by a parameter server	—
η	learning rate	—
t	staleness threshold	iterations
N	total number of training iterations	—
P	number of workers in distributed training	—
M	number of model rows or model parts	—

ROG

g_i'	accumulated gradients of row i	—
v_i^r	latest training iteration on worker r that updates row i	iteration
V	version storage for row-level staleness control	—
S	staleness threshold used in convergence analysis	iterations
MTA	minimum percentage of rows transmitted before reaching staleness threshold	—
t_{MTA}	transmission time budget associated with MTA	s

LOPInfer

b	available bandwidth used by scheduling decisions	bit/s
τ	time budget for operation-level scheduling	s
X^b	schedule plan under bandwidth b	—
x_M^b	sub-operation placement on the mobile device under bandwidth b	—
x_R^b	sub-operation placement on the remote server under bandwidth b	—

RRTO

R	minimum repeat count used in operator sequence search	—
IOS	reconstructed inference operator sequence	—

Chapter 1

Introduction

MACHINE learning is increasingly deployed in systems that sense, decide, and act close to the physical world. Mobile Edge Computing (MEC) provides a natural substrate for such systems by placing cloud-computing capabilities at access networks and nearby edge servers rather than only in remote clouds, an architecture advocated by industry standardization and extensively studied by the research community [53, 91, 2, 90]. This proximity shortens the control loop between data generation and intelligent response, but it also changes the systems problem. MEC ML systems must execute across unstable wireless links, battery-limited devices, heterogeneous accelerators, and existing application software. This thesis studies how MEC can support ML systems that are adaptive through online training, responsive through low-latency inference, and deployable through transparent offloading. The central challenge is a common *granularity mismatch*: conventional ML systems inherit execution units from data-center settings, but those units do not match MEC wireless dynamics, workload structure, or deployment constraints. The thesis addresses this challenge through *granularity-aware execution*, a systems principle that selects the synchronization, scheduling, or control unit according to both ML workload structure and MEC execution conditions.

1.1 Motivation: Machine Learning for MEC Intelligence

Mobile Edge Computing moves computation, storage, and networking resources from remote cloud data centers to the proximity of mobile devices, robots, and sensors. This architectural shift is important because many intelligent applications are no longer offline analytics jobs whose results can arrive later. A mobile robot must react to obstacles while it is moving; a perception service must process camera input before the scene changes; and an IoT deployment must coordinate distributed sensors without sending every decision through a distant cloud. The importance of MEC is not only an assumption of this thesis. ETSI identified Mobile Edge Computing as a key technology toward 5G because it enables ultra-low latency, high bandwidth, and context-aware services at the edge of the mobile network [53]. Industry-oriented MEC architecture work further positions MEC as a practical way to support IoT services by moving cloud capabilities into the radio access network, where applications can exploit local context and avoid

unnecessary wide-area round trips [124]. From the network-architecture perspective, MEC has also been surveyed as an emerging 5G edge-cloud architecture whose orchestration, service placement, and resource-management problems are central to future mobile networks [140].

Academic surveys provide the same message from several complementary directions. Communication-focused MEC studies emphasize that placing computation at the edge can reduce mobile-service latency, relieve backhaul pressure, and support mobility-aware resource allocation [91]. Broader MEC surveys identify nearby edge resources as a foundation for latency-sensitive mobile and IoT applications, and computation-offloading surveys frame the edge as a response to the persistent gap between resource-constrained devices and compute-intensive workloads [2, 90, 81]. More generally, edge-computing research argues that moving computation closer to data sources is necessary for applications that need low latency, location awareness, mobility support, and bandwidth efficiency [131]. These independent lines of evidence—standardization, industrial architecture, 5G network orchestration, and academic surveys—make MEC a well-recognized substrate for intelligent mobile and IoT systems rather than merely a convenient deployment choice.

However, proximity alone does not make MEC intelligence practical. An edge server can provide more computation than a mobile device, but the device still communicates over wireless links whose bandwidth changes with distance, occlusion, contention, and mobility. The device also has limited battery energy and often runs applications built on existing ML frameworks, runtime libraries, and closed-source components. These conditions make MEC different from both cloud-only and device-only computing. In the cloud, ML systems can assume stable networks and controlled software stacks; on the device, they can avoid communication but are limited by local resources. MEC lies between these extremes: it offers nearby acceleration, but every training update, inference request, and offloading decision must cross a volatile device–edge boundary.

This substrate becomes especially important when it supports the full lifecycle of ML. First, MEC systems need **online training** to improve intelligence after deployment. Robots and mobile services operate in open environments where lighting, weather, background, user behavior, and sensor placement change over time, so models trained offline may lose accuracy and must adapt using field data [86, 6, 136, 173]. Second, MEC systems need **low-latency inference** to provide timely feedback. Perception, navigation, and user-facing services often run with batch size one and tight response-time budgets; even when an edge server has abundant GPU capacity, the device still pays wireless transmission delay and energy. Third, MEC systems need **transparent deployment** so that existing applications and ML frameworks can use nearby acceleration without invasive source-code or model-graph rewrites. These requirements define the scope of this thesis: online training makes MEC intelligence stronger, low-latency inference makes it responsive, and transparent deployment makes it usable.

Robotic IoT exposes these requirements in one concrete MEC scenario. Robots are mobile and energy-constrained, yet they generate rich sensory data and require timely decisions. They communicate through wireless links while nearby edge servers provide stronger computation than the robot itself. The resulting systems tension is persistent: a robot should learn from local experience, react to new inputs immediately, and offload computation with minimal engineering burden, but the execution path is coupled with unstable communication and limited battery energy. Therefore, MEC is not merely a placement choice between device and server. It is a systems setting in which training synchronization, inference scheduling, and offloading control must be redesigned around wireless dynamics, mobile energy, and real software stacks. This motivates the thesis-level question: how can MEC support efficient, robust, and deployable ML across training, inference, and deployment?

1.2 Challenges: Granularity Mismatch in MEC Intelligence

Directly migrating data-center ML systems to MEC is difficult because the inherited execution unit is often the wrong unit for the edge environment. In this thesis, *granularity mismatch* denotes a mismatch between the unit at which an ML system synchronizes, schedules, or controls execution and the unit at which MEC conditions actually change, expose parallelism, or charge overhead. Conventional systems often coordinate execution at data-center-friendly units: a whole model or iteration in BSP/SSP and parameter-server training [45, 49, 76], a whole DNN layer in collaborative inference [64, 51, 80, 23], or one intercepted runtime operator in transparent remote execution [39]. These units are convenient because they match existing software abstractions. In MEC, however, wireless bandwidth may change within one model update, one inference request may contain overlap opportunities hidden inside a layer, and every low-level remote call may pay a wireless round trip. The result is a common pattern: the system executes at one granularity, while the bottleneck occurs at another.

- **Training challenge: whole-model synchronization is too coarse for wireless online training.** At the training layer, the *granularity mismatch* is between whole-model synchronization and sub-second wireless fluctuation. Online training is necessary because robots and mobile services must improve models using newly collected field data, but distributed training is communication-sensitive. Bulk Synchronous Parallel [45] and Stale Synchronous Parallel [49] coordinate workers at model or iteration granularity. This granularity is too coarse for robotic IoT: our measurements show that wireless bandwidth fluctuates by 20% every 0.4 s and by 40% every 1.2 s, comparable to the time needed to transmit one model update. A communication plan that was reasonable at the beginning of an iteration can become invalid before the gradients finish transmitting, leaving fast workers stalled at synchronization barriers. Network-aware scheduling and gradient compression [20, 52, 138, 83] reduce communication cost, but they still treat

the whole model as the synchronization unit; in our measurement, stall time still accounts for 45.2% of each iteration.

- **Inference challenge: whole-layer partitioning is too coarse for single-request feedback.** At the inference layer, the *granularity mismatch* is between whole-layer partitioning and the fine-grained overlap structure inside one inference request. Low-latency inference is necessary because MEC services must return fresh perception or control results on battery-limited devices. Device-only inference avoids communication but may be slow and energy-hungry; server-only inference uses stronger GPUs but depends on uplink transfer; and layer-wise collaborative inference [64, 51, 80, 23] serializes execution because the device computes a prefix, transmits one intermediate tensor, and waits for the server to compute the suffix. Data parallelism, tensor parallelism, and pipeline parallelism also do not solve this single-request problem: data parallelism needs large batches, tensor parallelism requires frequent all-reduce over bandwidth-limited wireless links, and pipeline parallelism improves throughput across requests rather than latency of one request. Under realistic MEC conditions, transmission accounts for about 50% of end-to-end latency and about 40% of device energy, so leaving transmission on the critical path directly weakens responsiveness.

- **Deployment challenge: per-operator RPC control is too fine for transparent acceleration.** At the deployment layer, the *granularity mismatch* is between per-operator RPC control and the repeated operator-sequence structure of ML inference. Transparent deployment is necessary because many MEC applications cannot afford intrusive changes to application code, model graphs, or framework internals. Non-transparent offloading can expose high-level model structure, but it increases engineering cost and may not support closed-source libraries, runtime-generated model structures, or dynamically optimized kernels [117, 33, 32, 168, 99]. Transparent offloading [39] preserves compatibility by intercepting low-level runtime calls, but it forwards control at the granularity of individual operators or runtime functions. A single inference can trigger 5,895 RPCs over 522 operators in our measurement. Under MEC wireless links, each RPC pays at least one round trip, so accumulated control overhead dominates latency and energy and can cause up to 95% slowdown relative to non-transparent systems.

These challenges show that MEC intelligence cannot be achieved by only choosing where a model runs. A system must also choose the execution unit at which it coordinates work. Training needs a synchronization unit that can react within wireless fluctuation, inference needs a scheduling unit that exposes overlap inside one request, and deployment needs a control unit that preserves transparency without paying thousands of round trips. This leads to the solution principle of this thesis: *granularity-aware execution*.

1.3 Solutions: Granularity-Aware Execution for MEC Intelligence

This thesis solves these challenges by building MEC ML systems around *granularity-aware execution*. The principle is to reselect the unit at which an MEC ML system synchronizes, schedules, or controls execution so that the unit matches workload structure, wireless dynamics, and deployment constraints. It is not a single optimization technique. It is a systems design rule: do not inherit whole models, whole layers, or per-operator RPCs merely because they are convenient software boundaries; instead, expose the unit that preserves correctness while reducing wireless waiting, enabling useful overlap, or amortizing control round trips. The three systems in this thesis instantiate this rule for training, inference, and deployment.

- **ROG: row-granulated distributed training for robotic IoT.** ROG resolves the training mismatch by changing the execution granularity from whole-model synchronization to parameter-row synchronization (Chapter 3). The insight is that rows provide the right middle ground: they are fine enough to react to transient bandwidth drops during one iteration, yet coarse enough to keep metadata, scheduling, and convergence control practical. ROG introduces Row Synchronous Parallel, which applies bounded staleness at row granularity and preserves the convergence guarantee of Stale Synchronous Parallel, and an Adaptive Transmission Protocol that schedules row transmission according to staleness, gradient importance, and real-time bandwidth. Under the same time budget, ROG improves training accuracy by 4.9%–6.5% and reduces energy consumption by 20.4%–50.7% for the same target accuracy, compared with BSP, SSP, and a state-of-the-art network-aware baseline.

- **LOPInfer: local-operator parallel inference for MEC service workloads.** LOPInfer resolves the inference mismatch by changing the execution granularity from whole-layer scheduling to local-operation scheduling (Chapter 4). The insight is that many DNN operators are local: parts of their outputs depend only on corresponding parts of their inputs. LOPInfer decomposes eligible operators into independent local operations, tracks producer-consumer dependencies to preserve model semantics, and schedules these operations across the device and edge server with a constrained optimization that respects MEC transmission and compute limits. This exposes intra-layer and cross-layer overlap between computation and communication within a single inference, reducing per-inference latency by up to 50% and device energy consumption by up to 75% compared with state-of-the-art layer-partitioning baselines.

- **RRTO: record/replay transparent offloading for MEC inference.** RRTO resolves the deployment mismatch by changing the execution granularity from per-operator RPC control to per-inference operator-sequence replay (Chapter 5). The insight

is that ML inference follows a stable operator sequence across requests, so reactive per-operator forwarding is unnecessary after warm-up. RRTO records low-level runtime behavior, reconstructs the steady-state operator sequence through a two-stage Operator Sequence Search that reduces search complexity from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$, and replays each inference as one proactive remote execution. RRTO preserves transparency while reducing per-inference latency and energy by up to 98% over the state-of-the-art transparent baseline and approaching non-transparent offloading.

Together, these systems make the same argument across the MEC intelligence lifecycle. ROG makes online training robust by lowering synchronization to rows, LOPInfer makes inference responsive by lowering scheduling to local operations, and RRTO makes deployment practical by raising transparent control to sequence-level replay. The common contribution is not one isolated optimization, but a granularity-aware methodology for building MEC ML systems that remain useful under real wireless and software constraints.

Adoption. All three systems have been open-sourced. ROG was published at MICRO 2022 and received the ACM Results Reproduced Badge; LOPInfer and RRTO are under review at major venues in services computing and mobile computing. The systems have been evaluated on commercial mobile robots and GPU servers under indoor and outdoor MEC networks. The LOPInfer design also builds on the authors' APPLIED 2024 workshop paper, which first quantified the mismatch between data-center inference parallelism and robotic IoT service workloads.

1.4 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the background and related work, with emphasis on the MEC wireless characteristics shared by training, inference, and deployment. Chapter 3 presents ROG, a row-granulated distributed training system for robotic IoT. Chapter 4 presents LOPInfer, a local-operator parallel inference system for MEC service workloads. Chapter 5 presents RRTO, a record/replay transparent offloading system for MEC inference. Finally, Chapter 6 concludes the thesis by summarizing the granularity-aware execution principle, stating limitations, and discussing future work.

Chapter 2

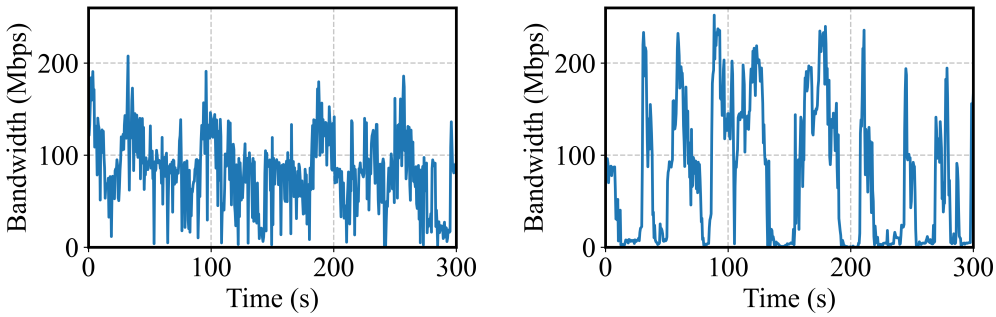
Background and Related Work

CHAPTER 1 framed the thesis around a common *granularity mismatch*: MEC ML systems inherit execution units that were convenient in data centers but ill-suited to wireless, mobile, and deployment-constrained edge environments. This chapter provides the technical background behind that claim and positions the three systems against prior work. It first introduces the wireless and systems characteristics shared by training, inference, and deployment. It then reviews the background and related work for ROG, LOPInfer, and RRTO, emphasizing where prior systems improve MEC execution and where they leave the execution granularity unchanged.

The unifying message is that MEC changes the cost model of ML execution. In data centers, training and inference typically run over stable wired networks, homogeneous accelerators, and controlled software stacks. In MEC, the critical path includes wireless transmission, mobile hardware constraints, battery energy, and compatibility with application software. As a result, the execution units inherited from conventional systems—whole models for gradient synchronization, whole layers for collaborative inference, and individual operators for transparent remote control—no longer match the dynamics of the environment or the structure of the workload. The remainder of this chapter develops the common wireless background (§2.1) and then shows how prior work addresses, partially addresses, or leaves open the mismatch at the training, inference, and deployment layers.

2.1 Wireless and Systems Background for Mobile Edge Computing

MEC places compute resources near mobile devices and sensors, usually at access networks or nearby edge servers [2]. This proximity reduces wide-area delay and enables mobile applications to use nearby accelerators, making MEC attractive for robotic IoT, mobile perception, and latency-sensitive services. However, the edge path is not simply a shorter cloud path. Unlike data-center fabrics, MEC networks often rely on local wireless links whose throughput changes over time. For ML workloads, this means that communication cost is not a static number that can be measured once and used



(a) Indoors **(b) Outdoors**

Figure 2.1: Wireless transmission instability of TCP between a robot and the base station in MEC networks. The experiment samples throughput every 0.1 s while robots move in indoor and outdoor environments. The large temporal variation explains why training synchronization, collaborative inference, and transparent offloading cannot rely on a fixed network cost during one ML execution.

throughout execution; it can change while a model update, activation tensor, or runtime call is still on the critical path.

Wireless bandwidth in MEC is both limited and unstable. Although modern Wi-Fi and cellular technologies provide high peak throughput, mobile devices and robots may not fully exploit this capacity because of radio hardware, antenna placement, contention, and energy constraints. More importantly, real throughput varies with mobility, communication distance, physical occlusion, channel contention, and environmental reflection [84, 93, 113, 37, 127, 122]. For robotic IoT, these factors are normal operating conditions rather than rare failures: the device moves, the environment changes, and the network path evolves with the task.

Fig. 2.1 shows wireless instability measured in a MEC robot surveillance experiment. Four-wheeled robots navigated through a lab and a campus garden at speeds of 5–40 cm/s. Using iperf [57], the experiment measured real-time wireless bandwidth between the robot and a base station over TCP [146] at 0.1-second intervals for five minutes. The average bandwidth was 93 Mbps indoors and 73 Mbps outdoors, while the outdoor measurements exhibited higher fluctuation and occasional near-zero drops due to obstacles and reduced signal reflections. The key point is not only that the average bandwidth is lower outdoors; it is that the bandwidth changes at a time scale comparable to ML communication steps.

This fluctuation affects each layer of the thesis in a different way. In distributed training, a worker that starts communicating under good bandwidth can become a straggler before all gradients are transmitted. In collaborative inference, activation transmission can occupy a large fraction of the latency and energy budget of one request. In transparent offloading, every remote procedure call pays wireless round-trip time, so thousands of small calls amplify the effect of RTT and jitter. Therefore, the

same wireless trace can create three different bottlenecks depending on the execution granularity used by the ML system.

Energy further strengthens this coupling between communication and execution. Mobile devices and robots cannot treat waiting as free. Even when a device waits for synchronization, an activation transfer, or a remote inference result, CPU, GPU, memory, and networking components still consume non-negligible power because of static and leakage power [101, 67]. Consequently, reducing communication-induced waiting improves both latency and energy efficiency. This is why the thesis evaluates not only speed but also energy: in MEC, performance and battery lifetime are two views of the same execution path.

2.2 Background and Related Work for ROG: Distributed Training in Robotic IoT

2.2.1 Online Training for Robotic IoT

Robotic applications increasingly depend on ML models for object recognition, control, mapping, localization, and environmental perception. These models may face distribution shifts after deployment. A model trained in one lighting condition, weather condition, or physical environment may perform poorly in another. Online training and adaptation can improve field intelligence by learning from newly collected sensory data, including unsupervised domain adaptation and implicit mapping from online observations [86, 6, 136, 173].

Distributed training is a natural way to use the data and computation of multiple robots. In the parameter-server architecture, each worker keeps a model replica, computes gradients on local data, pushes gradients to the server, and pulls updated parameters or gradients before continuing [77, 76]. In robotic IoT, however, the objective is not only final accuracy after an unlimited training run. The system must reach useful accuracy within a wall-clock and energy budget while robots remain mobile and connected through unstable wireless links. This makes straggler mitigation a first-order systems problem rather than a secondary optimization.

2.2.2 Synchronization Models and Straggler Mitigation

Bulk Synchronous Parallel (BSP) synchronizes all workers at every iteration [45]. It provides strong consistency, but it is vulnerable to stragglers. In a data center, stragglers are often associated with slow machines or shared-cluster interference. In robotic IoT, a straggler can be caused by a transient bandwidth drop during gradient transmission. When one worker takes much longer to transmit gradients, all other workers stall at the synchronization barrier, consuming time and energy without making progress.

Stale Synchronous Parallel (SSP) allows fast workers to proceed with bounded staleness [49]. It reduces the stall of BSP, but it introduces a tradeoff between throughput and statistical efficiency. A small staleness bound remains sensitive to wireless stragglers, while a large bound can harm convergence. Network-aware variants adjust staleness according to estimated network states [20, 52], and gradient compression reduces the amount of data to transmit [138, 83]. These techniques are useful, but they still schedule communication at model or iteration granularity.

Unfortunately, leaving the granularity unchanged limits how much these systems can respond to wireless dynamics. A whole-model transmission can last longer than the time scale over which the wireless link changes, so a decision made before communication may be invalid by the time it matters. In contrast, ROG changes the synchronization unit itself: it applies bounded staleness and transmission scheduling at row granularity. This gives the system enough flexibility to react during synchronization while keeping metadata and convergence control practical.

2.3 Background and Related Work for LOPInfer: Collaborative Inference in MEC

2.3.1 Inference Paradigms for MEC and Robotic IoT

MEC inference can be executed locally, remotely, or collaboratively. Local inference avoids network transmission, but it consumes device computation and may be slow on mobile hardware. Server-only inference uses GPU-equipped edge servers, but it depends on uplink bandwidth and may transmit large inputs. Collaborative inference partitions a DNN between the device and the server, usually by selecting a layer boundary and transmitting the intermediate activation [64, 51, 80, 23]. This paradigm is attractive because it can exploit both local computation and edge acceleration, but its effectiveness depends on whether the chosen partition exposes enough overlap and avoids excessive transmission.

Prior work has also examined whether data-center inference parallelism can be used in robotic IoT. Data parallelism is unsuitable for single-request robotic inference because real-time applications usually use batch size one, leaving no complete mini-batches to split. Tensor parallelism partitions layer tensors across devices, but it requires frequent all-reduce communication. In bandwidth-limited robotic IoT, this synchronization can dominate inference time and energy. Pipeline parallelism can increase throughput across multiple requests, but its layer partitioning still serializes computation and transmission within one request. The ApPLIED 2024 precursor to LOPInfer quantified these problems and showed that data-center parallelism does not directly translate to robotic IoT inference.

Layer-wise partitioning remains the most common collaborative inference approach because DNN layers provide convenient graph boundaries and some intermediate activations are smaller than raw inputs [51]. Existing systems optimize partition points according to model structure, device capability, network bandwidth, latency constraints, and energy constraints [64, 80, 23]. These methods are effective for choosing where to transmit, but they do not remove the stop-and-wait structure of one inference: the device computes a prefix, transmits an activation tensor, and waits for the server to compute the suffix.

ApPLIED 2024 precursor. The ApPLIED 2024 workshop paper measured why datacenter parallelism is ill-suited to robotic IoT inference. It showed that, at batch size one, data parallelism degenerates to single-device execution; tensor parallelism inflates latency and energy by orders of magnitude because of wireless all-reduce; and pipeline parallelism improves throughput across requests rather than single-request latency. It also measured that wireless transmission can account for up to 69% of inference time even under optimized layer partitioning. These observations seed the design of LOPInfer (Chapter 4), which lowers the scheduling unit below a whole layer to expose intra-request compute–communication overlap, and they are reused as part of the LOPInfer motivation (§4.2.4).

2.3.2 Transmission Bottleneck and Local-Operator Parallelism

The stop-and-wait structure creates both latency and energy bottlenecks. Prior analysis on robotic IoT shows that even with optimized layer partitioning, communication can take a large fraction of inference time. LOPInfer further shows that, in MEC service workloads, transmission accounts for about 50% of end-to-end latency and about 40% of device energy under layer-wise partitioning. The energy cost is high because the mobile device cannot enter a deep sleep state while waiting for the remote result; it must keep enough components active to continue execution once the result arrives.

This bottleneck motivates a finer scheduling unit. Many DNN operators are local operators, meaning that parts of their output depend only on corresponding partial inputs. Examples include element-wise activations and convolution over local tensor regions. If these operators are decomposed into local operations, downstream computation can begin as soon as the required partial inputs arrive, instead of waiting for an entire layer output. The useful execution unit is therefore not always the layer boundary exposed by the model graph, but the local dependency unit inside the operator.

Unfortunately, layer-partitioning and energy-aware offloading systems still treat a layer output as the transferable unit, so they can choose a better boundary but cannot remove the stop-and-wait behavior inside one request. In contrast, LOPInfer schedules local operations under device computation, server computation, and wireless transmission constraints. It preserves model semantics by tracking producer-consumer dependencies, while enabling intra-layer and cross-layer overlap of computation and communication. This makes LOPInfer complementary to earlier systems: it attacks the

transmission bottleneck caused by scheduling granularity itself.

2.4 Background and Related Work for RRTO: Transparent Inference Offloading

2.4.1 Non-Transparent and Transparent Offloading

Inference offloading uses MEC accelerators to reduce the burden on mobile devices. Non-transparent systems typically modify application code, model graphs, or framework execution to place computation on a GPU server. Such systems can be efficient because they expose high-level model structure to the offloading runtime. However, they increase deployment cost and may not support closed-source applications, framework-specific optimizations, or runtime-generated execution structures [117, 33, 32, 168, 99]. This deployment cost is important in MEC because edge acceleration is useful only if existing applications can adopt it without being rewritten for every device, framework, or edge server.

Transparent offloading avoids these modifications. It intercepts low-level runtime or system calls generated by ML frameworks and redirects them to a remote server through RPCs [39]. This approach is attractive because existing applications can use remote acceleration without source-code changes. The cost of transparency is that the offloading layer sees low-level calls rather than explicit model-level tasks. As a result, it may issue one remote call for each intercepted operator or runtime function, even when those calls belong to a repeated inference sequence.

2.4.2 RPC Granularity and Record/Replay Execution

The main performance problem of transparent offloading is RPC granularity. A DNN inference request can invoke hundreds of framework operators and many more backend calls. If each operation becomes a separate RPC, every call adds wireless round-trip delay to the critical path. Under MEC bandwidth and RTT conditions, this repeated control overhead can dominate the latency and energy of inference. The bottleneck is therefore not only the amount of data transmitted, but also the number of sequential remote interactions.

Coarsening remote execution can reduce this overhead, but doing so transparently is difficult. The system must reconstruct task-level execution from low-level traces without relying on application annotations. It must distinguish model initialization from steady-state inference, handle repeated requests, preserve dependency order, and ensure that replayed execution consumes the correct inputs and produces the correct outputs. A wrong sequence can silently break correctness, while an overly conservative sequence falls back to per-operator RPCs and loses the benefit of transparent acceleration.

Unfortunately, prior transparent systems preserve compatibility by exposing exactly the wrong control granularity for MEC: one low-level RPC at a time. RRTO addresses this problem with a record/replay design. It records runtime behavior, identifies the steady-state operator sequence through a two-stage Operator Sequence Search, and replays the sequence proactively as one remote execution. Compared with prior transparent offloading systems, RRTO preserves compatibility while changing the remote-control unit from per-operator RPCs to sequence-level replay.

2.5 Synthesis and Scope

The related work above shows that the three thesis systems solve different MEC problems through the same systems lens. ROG reduces the synchronization unit from models to rows so distributed training can react to transient wireless fluctuation. LOPInfer reduces the inference scheduling unit from layers to local operations so one request can overlap computation and transmission. RRTO raises the transparent remote-control unit from per-operator RPCs to sequence-level replay so deployment remains compatible without paying repeated wireless round trips.

Together, these changes instantiate *granularity-aware execution*: they choose the execution unit that matches the workload and wireless behavior at each layer instead of inheriting a convenient software boundary from data-center systems. The scope of the thesis is systems support for existing ML workloads rather than new model architectures or learning objectives. Within this scope, the goal is to make training, inference, and deployment practical under MEC constraints: unstable wireless links, limited battery energy, heterogeneous mobile devices, and real software stacks. This chapter therefore provides the bridge from the thesis-level framing in Chapter 1 to the three concrete systems in Chapters 3–5.

Chapter 3

ROG: Row-Granulated Distributed Training for Robotic IoT

CHAPTER 2 showed that MEC wireless links fluctuate at a time scale that directly interferes with ML communication. This chapter studies the first lifecycle requirement of MEC intelligence: online training for stronger deployed models. At this layer, the *granularity mismatch* is between whole-model synchronization and sub-second wireless fluctuation. Robots collect field data that can improve perception, mapping, and adaptation, but their distributed training traffic crosses unstable wireless links. Conventional Bulk Synchronous Parallel and Stale Synchronous Parallel synchronize at whole-model granularity. As a result, a bandwidth drop during one gradient transmission can invalidate the communication schedule formed at the beginning of the iteration, causing fast workers to stall and wasting battery energy.

ROG resolves this mismatch by changing the synchronization granularity from whole models to parameter rows. This is the training-layer instance of *granularity-aware execution*: the system chooses a synchronization unit that matches both the row structure of model parameters and the time scale of robotic wireless fluctuation. Row Synchronous Parallel (RSP) extends the bounded-staleness convergence guarantee of Stale Synchronous Parallel to individual rows across workers, while the Adaptive Transmission Protocol (ATP) schedules row transmission according to staleness, gradient importance, and real-time bandwidth. This granularity is fine enough to react within a single iteration and coarse enough to keep scheduling and convergence control practical. ROG has been published at MICRO 2022 with the ACM Results Reproduced Badge, and its full open-source release is available at `https://github.com/hku-systems/ROG`.

The remainder of this chapter reuses the original paper text. Section 3.1 motivates ROG in robotic IoT and summarizes its contributions. Section 3.2 describes robotic wireless characteristics and closely related synchronization models. Section 3.3 gives the system overview. Section 3.4 presents RSP and ATP together with the convergence

analysis. Sections 3.5 and 3.6 describe the implementation and evaluation under indoor and outdoor robotic MEC networks. Section 3.7 summarizes the chapter.

3.1 Introduction

This section instantiates the thesis-level *granularity mismatch* at the training layer. In robotic IoT, online training must adapt models with field data while the wireless bandwidth between robots fluctuates at the same time scale as gradient synchronization. The core question is therefore not only how to reduce communication volume, but also how to choose a synchronization granularity that can react to wireless changes without weakening convergence.

Critical robotic tasks such as rescue [119] and disaster response [165] are more prevalently leveraging machine learning models (e.g., objective recognition models [68] or action control models [41, 159]) deployed over a team of mobile robots. These models typically require real-time training to adapt pre-trained parameters to changing environments [155, 158] (e.g., from sunny to foggy), but it is often expensive for the robots to access a cloud data center for model training due to the lack of stable internet access. Therefore, such training for critical robotic tasks is often distributedly deployed among a team of robots over the robotic IoT networks [111, 19].

Such distributed training typically adopts the parameter server paradigm [77, 76]: each device keeps a copy of the model and computes the model's parameter updates (gradients) on its own data iteratively; between iterations, a synchronization barrier (BSP [45]) is inserted, where each device pauses its computation, pushes the computed gradients of the whole model to and pulls the averaged gradients from a parameter server (located on one of the devices) over wireless networks. The process of training iterates until the shared model converges (i.e., reaches a desired accuracy).

To ensure high performance of the critical robotic tasks with the typically limited computation power and battery energy on robots, we identify that such distributed training should meet the following requirements (**3Rs**):

- Robust (**R1**): The critical robotic tasks are often confronted with complex environments (e.g., crowds, damaged areas). The performance of the distributed training should be resilient to these environments for the robots to adapt to various changing environments and fulfill the critical tasks.
- High training throughput (i.e., the number of training iterations in unit time) (**R2**): Given a tight time budget, high training throughput is crucial for high training accuracy to better adapt to the changing environments.
- High statistical efficiency (i.e., the training accuracy gain per training iteration) (**R3**): With higher statistical efficiency, the training model can reach higher accuracy with the same number of training iterations.

3Rs are important for the training model to reach high accuracy given a tight training time budget, while preserving battery energy to reach a desired accuracy.

Unfortunately, although such distributed training with BSP empirically achieves high statistical efficiency (**R3**) [49], the instability of real-world robotic IoT networks hinders it from meeting **R1** and **R2** by causing the ***straggler effect***: the transmission of gradients from some devices (i.e., stragglers) can be dramatically delayed (e.g., transmission time prolonged from 1.43s to 12.9s recorded in an unstable environment, see Sec. 3.2.2) by sharp bandwidth degradation due to movement of the devices [94, 113], occlusion from obstacles [127, 36], etc.; the devices that finish transmission (i.e., non-stragglers) have to stall until the delayed gradients from stragglers are transmitted in severely downgraded bandwidth, prolonging training iterations (violating **R1** and **R2**) and wasting energy stalling.

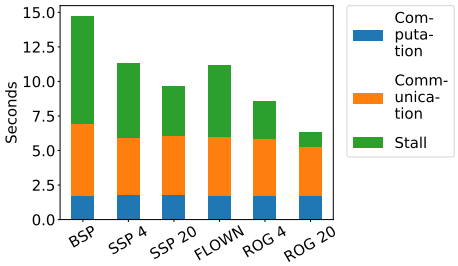
Although mainly designed for datacenter networks, Stale Synchronous Parallel (SSP) [49, 27] has the potential to mitigate such straggler effect. SSP allows non-stragglers to continue computing without the latest gradients from stragglers and only stall when the gradients from stragglers fall behind (stale) for a preset number of iterations (staleness threshold).

However, when coping with the instability of real-world robotic IoT networks, **R2** and **R3** are contradictory in SSP: high statistical efficiency requires a small staleness threshold [49], while high training throughput requires a large staleness threshold. In our evaluation (Fig. 3.1), SSP with a small threshold (4) achieved similar statistical efficiency as BSP but suffered severe straggler effect (stall time on average takes up 44.1% of the duration of a training iteration); a larger threshold (20) slightly reduced the stall time to 42.5% of a training iteration, at the cost of lower statistical efficiency.

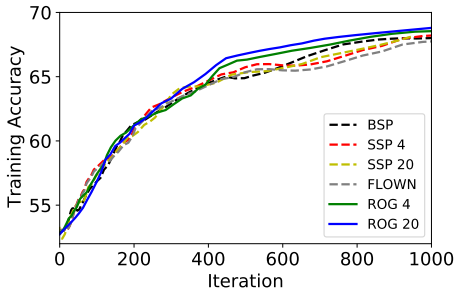
Recent studies [20, 52] extend SSP by dynamically assigning (scheduling) the staleness threshold to simultaneously fulfill **R2** and **R3**: higher staleness threshold for devices that are estimated to have low bandwidth and less contribution to training accuracy; smaller threshold for the opposite. However, they are designed for datacenter networks and wired edge networks and cannot fulfill **R1**, because the random and rapid nature (see Sec. 3.2.2) of bandwidth degradation in wireless networks can transform the non-stragglers estimated during scheduling into stragglers during the actual transmission, making the scheduling mismatch with the actual bandwidth. In our evaluation (Fig. 3.1), such methods still suffered the straggler effect, which caused stall time to on average take up 45.2% of a training iteration, violating **R1**.

The key reason for the problem of the above methods is that they are synchronizing the model gradients on the granularity of a whole model, whose transmission time is typically coarser (longer) than the granularity (or frequency) of bandwidth fluctuation in real-world robotic IoT networks. From the view of robustness (**R1**) and training throughput (**R2**), the scheduling based on the granularity of a whole model can not adapt to the real-time fluctuation of bandwidth and will be frequently invalidated,

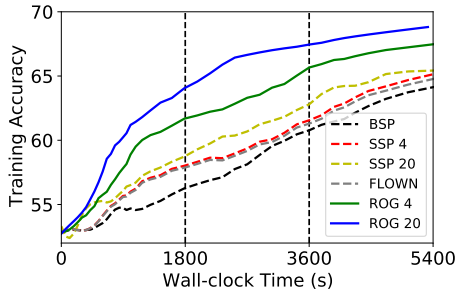
causing more stall and reduced training throughput. From the view of statistical efficiency (**R3**), the scheduling treats all computed gradients from a device as a whole and neglects that gradients from a device have different contributions to training accuracy (e.g., gradients with small absolute values contribute little). Thus, it is a must to break up the gradient transmission and schedule the transmission of the gradients with a finer granularity.



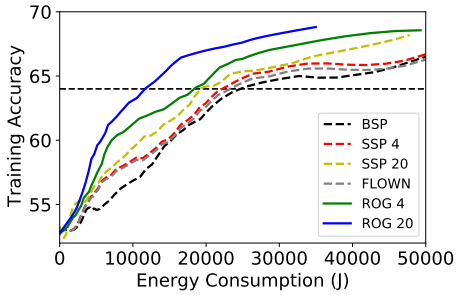
(a) Average time composition of a training iteration.



(b) Statistical efficiency.



(c) Training accuracy against wall-clock time.



(d) Energy consumption against training accuracy.

Figure 3.1: Comparison between ROG and the baselines on the unsupervised domain adaptation application paradigm in the outdoor environments.

In this chapter, we present ROG, a ROw-Granulated, high-performance and robust wireless distributed training system optimized for real-world robotic IoT networks. We choose the granularity of rows after comparing three typical levels of granularity to break up the parameters of an ML model: layers (matrixes), rows (matrix rows), and elements (individual parameters). Specifically, element granularity requires indexing each element of the whole model for management, taking up data volume comparable to the whole model (high management cost); layer granularity is large in size and is still comparable with the granularity of bandwidth fluctuation (low transmission flexibility). Row granularity best trades off between management cost and transmission flexibility and enables that whenever bandwidth fluctuation happens, ROG can in real-time adapt to it by adjusting the scheduling of rows to be transmitted, at a negligible cost of transmitting only one row in degraded bandwidth.

The design of ROG is confronted with two major challenges. The first one is how to guarantee convergence in ROG when synchronizing gradients on row granularity.

We propose Row Synchronous Parallel (RSP) that breaks up and enforces the staleness control of SSP to each row of a model across different devices and different rows within a device. RSP guarantees convergence by confining the divergence of rows within the staleness threshold and thus confining the divergence of the whole training model on different devices. We formally prove that RSP achieves the same convergence guarantee as SSP (see Sec. 3.4.3).

The second challenge is under RSP, how to properly schedule the transmission of each row from different devices to fulfill **3Rs**. ROG adaptively aligns the transmission time of each device by speculatively transmitting each row with a novel *Adaptive Transmission Protocol* (ATP). In a training iteration, ATP monitors the transmission time taken by the transmitted rows and in real-time updates the scheduling of the pending rows to be transmitted, to ensure that all devices roughly spend equal time transmitting gradients under random and sharp bandwidth fluctuation (**R1**), avoiding the straggler effect (**R2**). ATP further prioritizes the transmission of different rows based on their staled versions and contribution to model convergence (e.g., the absolute values of the gradients), reducing stall and accelerating convergence (**R3**).

We implemented ROG in PyTorch [116] and evaluated ROG on a team of mobile robots under two representative real-world online training application paradigms (unsupervised domain adaptation and implicit mapping and positioning, see 3.2). We compared ROG with BSP [45], SSP [49] and a SOTA dynamic threshold method [20] (referred to as FLOWN) under different real-world robotic IoT networks environments (namely indoor with moderate instability and outdoor with more severe instability). We also minimized the communication volume with gradient compression [138] (the compressed gradients were only sized at 2.1 MByte and 0.75 MByte in the two paradigms) to conduct the tightest comparison between ROG and the baselines. Evaluation shows that:

- ROG achieves high accuracy. ROG achieved a 4.9% ~6.5% accuracy gain over the baselines after training for 60 minutes, due to 25.2%~80.4% higher training throughput and non-degraded statistical efficiency under outdoor and indoor environments.
- ROG is energy-efficient. With the above advantage of training throughput and statistical efficiency, ROG reduced battery energy consumption by 20.4%~50.7% compared with the baselines when the training model reached a same high accuracy,
- ROG is scalable. When increasing the number of robots involved or increasing the batch size of training, ROG still achieved a 3.0%~5.3% accuracy gain and a 30.3% ~55.1% energy consumption reduction over the baselines.
- ROG is easy to use. It took only tens of lines of code to apply ROG to existing ML applications.

Our main contribution is RSP, a new row-granulated synchronization model and ATP, a fine-grained scheduling strategy optimized for distributed training over real-world robotic IoT networks. ROG fulfills **3Rs**: while conducting row-granulated staleness control to guarantee convergence with RSP, ATP schedules the transmission of each row adaptively to the fluctuating bandwidth (**R1**), so as to avoid the straggler effect (**R2**) and make gradients with more contribution to training accuracy be transmitted first (**R3**). We envision that ROG will nurture diverse ML applications deployed on mobile robots in the field, such as robot rescue [119], disaster response [165], and robot surveillance [150, 170], making them fast and energy-efficiently adapt to changing environments under an extremely unstable local wireless network without being affected by the straggler effect. ROG's code is released on `https://github.com/hku-systems/ROG`.

In the rest of this chapter, Sec. 3.2 introduces the background, Sec. 3.3 gives an overview of ROG, Sec. 3.4 presents the detailed design of ROG, Sec. 3.6 evaluates ROG, and Sec. 3.7 summarizes the chapter.

3.2 Background

3.2.1 Online training on Robotic IoT

While machine learning methods heavily rely on labeled training datasets (supervised training), it is costly to label datasets in every possible environment. As a result, various unsupervised training algorithms are developed to learn knowledge from unlabeled data. For example, adversarial unsupervised domain adaptation methods [86, 6] typically adapt a pretrained model to a new environment by training it with both shifted (noised) unlabeled data from the new environment and labels predicted with generative methods. With such methods, robots can adapt their pretrained models to new environments after training with online collected data to retain high accuracy of the models. As another example, implicit mapping and positioning [136, 173] construct a machine learning model representation of a 3D dense map by training the model with online collected unlabeled image sequences. We envision that the prosperity of these unsupervised training algorithms is making online training on online collected data on robots feasible and practical.

3.2.2 Characteristics of Robotic IoT Networks

In real-world robotic IoT applications (Fig. 3.2), devices typically need to move around for rescue, search, etc. Although wireless networks suffice for high mobility, the occlusion of obstacles and the change of distances among devices cause *instability* in the bandwidth capacity: sharp bandwidth fluctuation with random duration happens frequently and randomly. This causes divergence in gradient transmission time from different robots and the straggler effect.

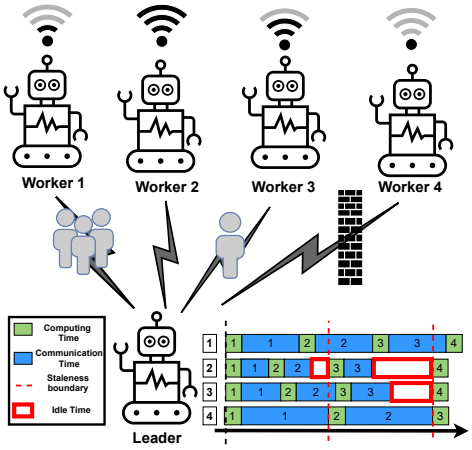
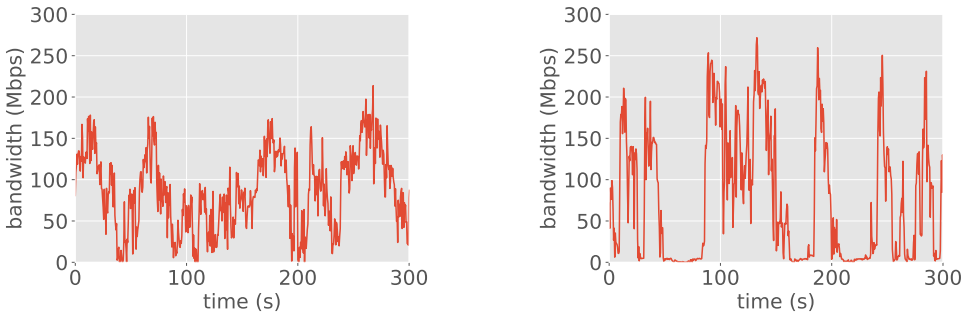


Figure 3.2: The instability of real-world robotic IoT networks.

To demonstrate the instability, we set up a robot surveillance task: two four-wheel robots navigate around several given points at 5~40cm/s speed in our lab (indoors) and campus garden (outdoors). The hardware and wireless network settings are as described in Sec. 3.6. We believe our setup represents the state-of-the-art (SOTA) computation and communication capabilities of robotic IoT devices.

We saturated the wireless network connection with iperf [58] and recorded the average bandwidth capacity between these two robots every 0.1s for 5 minutes, shown in Fig. 3.3. Both indoor and outdoor records show frequent and sharp bandwidth fluctuation. Statistically, on average a 20% fluctuation of bandwidth capacity happened every 0.4s, and a 40% fluctuation typically happened every 1.2s. Such times are comparable to the time of transmitting compressed gradients recorded with ideal wireless networks (e.g., 1.47s), causing high variability of transmission time. Besides, the outdoors bandwidth more frequently dropped to extremely low values around 0Mbit/s, exhibiting higher instability than indoors. The reason is the outdoor open area lacks walls to reflect wireless signals. When there are obstacles (e.g., trees) between communicating robots, fewer signals could be received in the outdoor area than the indoor.

Comparison with Datacenter Networks and Edge Networks. Compared with robotic IoT networks, datacenter networks (for distributed training in datacenter) and edge networks (for federated learning) are wired and often exhibit much lower bandwidth fluctuation. In datacenter networks, bandwidth fluctuation is typically caused by congestion on intermediate switches, and could be mitigated by scheduling traffic on switches [31]. In edge networks, bandwidth fluctuation is often caused by the variation of overall traffic volume, and typically happens at the scale of hours [110]. Existing methods target these two types of networks, and are not designed for handling instability in robotic IoT networks.



(a) Indoors. **(b)** Outdoors.

Figure 3.3: The instability of robotic IoT networks. A 40% fluctuation of bandwidth typically happens every 1.2s, comparable to the time of transmitting compressed model gradients.

3.2.3 Impact of Straggler Effect on Power Consumption

People may think a stalling robot can be consuming little energy. However, we recorded the energy consumption when a robot is stalling due to the straggler effect and found that a stalling robot still consumed almost one third of the energy consumption when the robot was computing (see Sec. 3.6). That is because the device cannot be put into low power sleep mode even when stalling, as it has to wait for messages from the parameter server and promptly continue working when stragglers catch up, and chips like CPU, GPU, and memory consume non-negligible power even when not computing, due to the static power consumption rooted in transistors' leakage current [101]. Consequently, besides damaging training throughput, stall caused by the straggler effect also has a major impact on the power consumption of the training process.

3.2.4 Related Work

BSP, SSP and their Variants. Bulk Synchronous Parallel (BSP) methods [49] enforces synchronization between each iteration. Therefore, it could easily get blocked by stragglers. To mitigate the straggler effect in datacenter networks while guaranteeing model convergence, SSP is usually adopted [49] in practice. By loosening the synchronization barrier, SSP allows fast workers to continue their iterations when the updates from slow workers are staled until the staled version reaches a *staleness threshold*. With a small staleness threshold, SSP ensures that all gradients extracted from each device's dataset equally contribute to the SGD convergence (same as BSP), which is widely reported to be necessary for an SGD process to achieve high statistical efficiency and high final accuracy [22, 10, 66, 8]. However, a small threshold cannot contain the instability in real-world robotic IoT networks while a large threshold sacrifices high statistical efficiency.

Inheriting SSP's more flexible synchronization model compared with BSP, subsequent studies (including federated learning) [132, 20, 52] extensively explored the scheduling of synchronization among workers according to network conditions and

the contribution to training accuracy. Scheduling strategies work well in datacenter networks and edge networks with slow and moderate bandwidth fluctuation. However, these scheduling strategies are not robust to the rapid and random bandwidth fluctuation in robotic IoT networks, because their model-granulated scheduling and transmission are often coarser-grained than the transient instability of real-world robotic IoT networks.

Gradient Compression. Gradient compression greatly reduces the communication traffic volume and is indeed essential for practical distributed training over wireless networks. Some lossy gradient compression methods [83] (information is lost during compression and cannot be recovered) achieve up to 0.1% compression rate (i.e., size after compression divided by original size), but they cannot provide convergence guarantee [83]. This chapter only considers lossless compression methods (e.g., the lost information during compression is compensated with error compensation [138]) which have a typical compression rate of around 3% [138].

Even with gradient compression, communication still takes a major time portion in distributed training on robotic IoT devices for two reasons. First, the devices typically share the same wireless channel, incurring traffic volume proportional to the number of devices involved in the distributed training process. Second, with the rapid advancement of SOTA robotic IoT devices [59], the computation time on each device is also decreasing. As a result, the communication time is typically comparable to the computation time.

Consequently, even with gradient compression, the straggler effect, which severely prolongs the communication time, still has a major impact on the distributed training process. In our experiments, a Jetson Xavier NX [109] device out of a four-device team computed gradients in 2.18s and ideally needed to wait for 1.47s upon the synchronization barrier in BSP (four devices push and pull the compressed gradients sized 2.1MByte, summing up to 134.4Mbit), which is comparable to (equal to 67.4% of) the computation time. Meanwhile, the straggler effect in the above indoor scenario caused each device to on average stall for 2.23 s in each iteration, equal to 102.2% of the computation time, severely degrading the training throughput.

3.3 Overview

3.3.1 Workflow

Fig. 3.4 presents the workflow of ROG and compares it with BSP and SSP in unstable networks. The random and sharp wireless bandwidth fluctuation causes the transmission time of each model among devices in BSP and SSP to diverge and causes the straggler effect. Since the transmission time of each row among the devices also diverge in ROG, to avoid the straggler effect, the main idea of ROG is to align the transmission time among all devices in real-time by dynamically and adaptively scheduling the

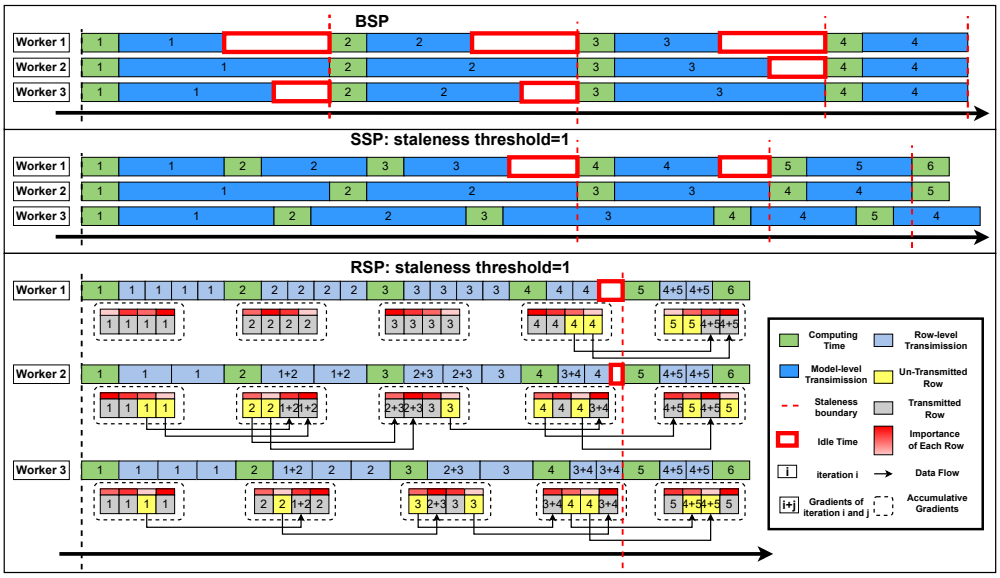


Figure 3.4: Workflow of ROG. The training model in RSP is divided into four rows and each row is synchronized in one row-level transmission. The bandwidth on these three devices is identical at the same time point in the three cases and the bandwidth is interfered by distance, occlusion, etc..

transmission of rows, as shown in Fig. 3.4. The design of ROG tackles three problems: how to properly break up the gradient synchronization granularity, how to guarantee convergence, and how to schedule the gradient transmission in real-time.

The choice of granularity. Out of three possible granularity choices: elements, rows and layers, we choose rows to best tradeoff between the management overhead and flexibility in transmission (duration of transmission of the smallest unit). While ROG is adaptively transmitting the smallest units, it causes a management overhead that we need to at least maintain a list of indexes of all the smallest managed units on the whole model and transmit the list during every model synchronization, so that the adaptively transmitted units can be correctly indexed to its position on the model.

On the one hand, element granularity will apparently cause an index list as large as the number of elements of the whole model; as an integer (an index) and a floating-point number (an element) typically take up the same amount of data volume when being transmitted (i.e., the default int32 and float32 encoding configuration of PyTorch [116]), the transmission data volume will be doubled during every model synchronization. On the other hand, layer granularity typically causes a small index list (e.g., 226 layers in the model [74] with 16.95M elements we evaluated for the first application paradigm); however, a layer of a model can be still large at size (e.g., the largest layer of the aforementioned model has 1.18M elements) and bandwidth degradation during its transmission will still evidently prolong the training iteration. Overall, as a row of a model is neither too small (33307 rows on the aforementioned model that take up data volume only sized 0.24% of the model size for indexing in our evaluation) nor too big (a row typically contains several to hundreds of elements), row granularity is

the best choice for ROG.

Row Stale Parallel (RSP). Since not all rows are synchronized in an iteration in ROG, gradients of different rows of the training model on a device can have different versions. Uncontrolled version differences could slow down the training and even fail to guarantee convergence. We find that breaking up and applying the staleness control of SSP to each row of the training model would confine the divergence of the same row across different devices and thus confine the divergence of the training model across different devices, which is key to the convergence of the distributed training process [49]. Consequently, we design RSP that adopts a two-level row-granulated staleness control: for the same row on the training model across different workers, the staled version should be within a preset staleness threshold; for different rows within the same worker, the staled version should also be within the same staleness threshold. Workers are forced to wait when these two requirements are not met, as shown in Fig. 3.4. In this way, RSP provably achieves the same level of convergence guarantee as SSP (see Sec. 3.4.3).

Adaptive Transmission Protocol (ATP). To align the transmission time among all devices, for a straggler in an iteration, ATP controls it to transmit MTA (minimum transmission amount, an empirical lower bound of the number of rows to be transmitted by stragglers to avoid stall) of the total rows and reports its transmission time of MTA (MTA time) to other devices. A non-straggler then keeps transmitting rows for MTA time (or all of their rows if the transmission finishes before MTA time), so that the transmission time among stragglers and non-stragglers is balanced and the straggler effect is avoided. Among rows within a device, ATP maintains the importance (depth of the red color in Fig. 3.4) of each row based on its possibility to cause stall (e.g., the staled version) and the contribution of gradients of each row to training accuracy (e.g., the absolute value of the gradients); the rows with the highest importance will be transmitted first to minimize stall time and optimize statistical efficiency.

Technically, besides the management overhead, smaller granularity also brings extra transmission overhead. To ensure that non-stragglers keep transmitting rows only for MTA time, a straightforward approach is inserting judgement about whether MTA time is reached between the transmission of each two successive rows. However, such an approach is infeasible in ROG because empirically the transmission time of a row is comparable to the time cost of the inserted judgement, leading to severe underutilization of the bandwidth capacity. Instead, we co-design ATP with the underlying transmission protocol and enable *speculative transmission*: the device continuously transmits rows in the priority determined by their importance without inserting judgement and discards the ongoing transmitting row once the MTA time is reached (see Sec. 3.4). In this way, the transmission overhead is reduced to possibly discarding the last row transmitted if its transmission is incomplete, which is also negligible.

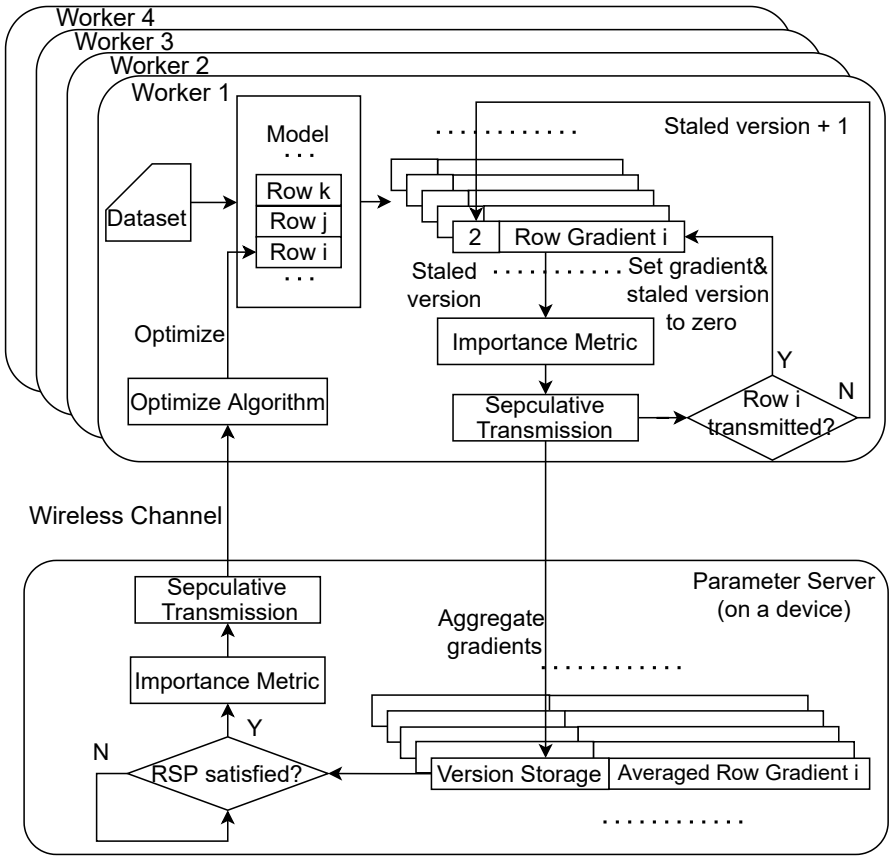


Figure 3.5: Architecture of ROG. Note that on the parameter server, similar to the worker side, ROG checks whether a row is transmitted and manages its accumulated gradients and the version storage accordingly. We leave out this part of the figure for simplicity.

3.3.2 Architecture of ROG

Fig. 3.5 shows the architecture of ROG. On each worker and the parameter server, ROG divides the parameters of the shared model into rows and maintains the gradients and staled version of each row individually. In an iteration, each worker computes gradients of the model based on its own share of the dataset, and Importance Metric sorts the order of transmission of each row according to the row's staled version and the average absolute values of the gradients. Speculative Transmission keeps transmitting for the aforementioned MTA time on non-straggler or transmits MTA on stragglers, balancing the transmission time of each worker. ROG increases the staled version of un-transmitted rows by one and set the staled version and gradients of transmitted rows to zero, so that only gradients of un-transmitted rows will be accumulated.

The parameter server aggregates and averages the received gradients, and updates Version Storage of the corresponding rows. If the requirements of RSP are met, ROG will similarly determine the importance of the rows in Importance Metric and transmit the most important rows' gradients in Speculative Transmission for MTA time or transmit MTA; otherwise, idle time (stall) will be inserted until RSP is met. Note that ROG maintains a copy of the gradients for each worker, because the importance of each row

can differ for different workers and thus different rows can be transmitted for different workers. If ROG sends the gradients of a row to a specific worker, ROG will only set the gradients on the copy for this worker to zero and the gradient copies for other workers are not affected.

On reception of gradients of certain rows, the worker optimizes the parameters of these rows with the received gradients. It is worth noting that, since the produced gradients of each worker will either be accumulated at the worker side or the parameter server side and eventually be sent to each worker, the model on each worker will be optimized with exactly the same gradients. Thus convergence of the shared model will not be affected.

3.4 Detailed Design

3.4.1 Algorithms of ROG

Here we present how ROG integrates RSP and ATP to achieve finer-grained staleness control and adaptive scheduling. The local worker part is given in Algo. 1 and the parameter server part is given in Algo. 2. Details of ATP are mainly described in Algo. 3 and Algo. 4 in the next subsection.

Algorithm 1: Local Worker

Function *LocalWorker_ROG()*: `// On workers`
  Data: w: local model parameters; η: learning rate; N: total training iterations; *iters*: training iterations that each row is pushed; $g\prime$: accumulated gradients; t: staleness threshold
  for *each iteration n:* 1...N **do**
    $g \leftarrow$ Training(w);
    $g\prime \leftarrow g\prime + g$
    iters $\leftarrow$ PushGradients($g\prime$, n, *iters*);
    PullAveragedGradients(w, *eta*);
Function *PushGradients($g\prime$, n, iters, t)*:
  `// Worker mode of ImportanceMetric`
  ImportanceMetric($g\prime$, *iters*, $'worker'$);
  Transmitted $\leftarrow$ SpeculativeTransmission($g\prime$, n, t);
  for *each row i in Transmitted* **do**
    $g\prime_i \leftarrow 0$;
    $iters_i \leftarrow n$;
  end
Function *PullAveragedGradients(w, eta)*:
  $\bar{g} \leftarrow$ RecvGradients()
  for *each* $\bar{g}_i$ *received from server* **do**
    $w_i \leftarrow w_i - \eta\bar{g}_i$;
  end

On the worker side in Algo. 1, when gradients are computed in a training iteration, they are added to the accumulated gradients (line 2, 3) and we then transmit the accumulated gradients to the parameter server in PushGradients(). PushGradients() sorts

Algorithm 2: Parameter Server

Function *ParameterServer_ROG()*:
  Data: $\bar{g}^r$: averaged gradient for worker r; $\bar{g}_i^r$: i-th row of $\bar{g}^r$; v_i^r: the latest training iteration on worker r that updates row i; V: $\{v_i^r\}$; num: the number of workers; P: rows' priority; t: staleness threshold
  upon *receive gradients $g\prime$ from worker r* **do**
    `// Worker r push gradients`
    $g\prime, n \leftarrow$ RecvGradients(r)
    for *each row i in $g\prime$* **do**
      $v_i^r \leftarrow n$;
      for *each worker s* **do**
        $\bar{g}_i^s \leftarrow \bar{g}_i^s + \frac{g\prime_i}{num}$;
    for *each row i in $g\prime$* **do**
      `//` v_i^r `triggers the finer-grained threshold`
      while $v_i^r - min(V) \geq t$ **do**
        wait for other worker to update $\bar{g}_i$
    `// Worker r pull gradients`
    `// Server mode of ImportanceMetric`
    ImportanceMetric($\bar{g}$, V, $'server'$);
    Transmitted $\leftarrow$ SpeculativeTransmission($\bar{g}$, t);
    for *each row i in Transmitted* **do**
      $\bar{g}_i^r \leftarrow 0$;

the transmission order of the accumulated gradients of each row and then speculatively transmits these rows such that the transmission time among different workers is balanced in SpeculativeTransmission() (line 7 to 8). SpeculativeTransmission() also reports the latest training iteration that produced these gradients to the parameter server for it to maintain its Version Storage. Accumulated gradients of the transmitted rows will be assigned to zero and their latest training iteration that is pushed to the parameter server is recorded in line 9 to 11. In PullAveragedGradients(), pulled averaged gradients of certain rows would be used to update the parameters of the corresponding rows in line 15 to 16.

On the parameter server side in Algo. 2, upon reception of gradients of certain rows from a worker r, ROG on the parameter server side first finds the corresponding rows on the shared model and then accumulates the averaged gradients as shown in line 2 to 6. From line 7 to line 9, we only consider the situation that the times (training iterations) that gradients of row i ($\bar{g}_i$) are updated by worker r (v_i^r) should not be ahead of the updated times of any rows by any workers ($min(V)$) more than the threshold. That's because when the threshold is triggered, we only need to stall the non-stragglers and wait for stragglers to catch up to satisfy RSP. In line 10 to 13, ROG determines the transmission priority of rows, speculatively transmits these rows, and manages the accumulated gradients of transmitted rows similar to the worker side.

3.4.2 Adaptive Transmission Protocol

Algorithm 3: Importance Metric

Function *ImportanceMetric*:
  Data: $g\prime$: gradients of all rows; *iters*: training iterations that each row is updated; *mode*: worker or parameter server mode
  $importance \leftarrow \{\}$
  for *each row i in* $g\prime$ **do**
    if *mode=='worker'* **then**
      $j \leftarrow f_1 \times \text{mean}(\text{abs}(g_i\prime)) + f_2 \times (\max(iter) - iter_i)$
    else
      $j \leftarrow f_1 \times \text{mean}(\text{abs}(g_i\prime)) + f_2 \times (iter_i - \min(iter))$
    end
    $importance$.append(j)
  Sort($g\prime$, $importance$)

Algorithm 4: Speculative Transmission

Function *SpeculativeTransmission*:
  Data: $g\prime$: sorted gradients of all rows; n: current training iteration training; t: staleness threshold
  $MTA \leftarrow \text{MTATable}(t) \times \text{len}(g\prime)$
  $t_{MTA} \leftarrow \text{GetMTATime}()$
  $Transmitted \leftarrow \text{SendWithTimeout}(g\prime, t_{MTA})$
  if *len(Transmitted)* $< MTA$ **then**
    Send($g\prime$[len(*Transmitted*): MTA])
    $Transmitted \leftarrow MTA$
  end
  UpdateMTATime()
  return *Transmitted*

The ATP protocol consists primarily of two functions: *ImportanceMetric* (Algo. 3) that prioritizes the transmission of different rows, and *SpeculativeTransmission* (Algo. 4) that records and aligns the gradient transmission time among different workers.

Importance Metric in Algo. 3 shows our scheme to prioritize the gradient rows. Notably, we treat workers and the parameter server differently (line 3 to 6). Besides absolute values of gradients ($mean(abs(g_i\prime))$), since the staleness threshold is triggered and handled at the parameter server side, workers pushing gradients to the parameter server need to especially give priority (bigger j) to the staled rows, so as to reduce the possibility to trigger the staleness threshold and cause stall. Thus we add a term $max(iter) - iter_i$ to the importance of each row on workers to estimate the number of iterations that the row has not been pushed (staled) to the parameter server (line 4, f_1 and f_2 are empirical coefficients). On the contrary, pulling gradients from the parameter server will not affect the triggering of the staleness threshold, and thus we give extra priority to fresher rows (estimated with $iter_i - min(iter)$ in line 6) that typically

have higher contribution to training accuracy. These rows are then sorted in descending order according to their assigned importance and will be transmitted in the sorted order.

After sorting these rows, Speculative Transmission (Algo. 4) retrieves the scheduled transmission time (t_{MTA}) and enforces the transmission time limit by setting the timeout of the ongoing transmission to t_{MTA} (line 3). Upon timeout, the ongoing transmission will be immediately stopped and the transmitted gradients will be recorded in *Transmitted*. If at least P percent of rows are transmitted each time, at most $(1-P)^s$ percent of the row will remain un-transmitted after s steps, because all rows are transmitted and updated independently of each other. In order to ensure all rows are transmitted before triggering the threshold, there should be $(1-P)^{S-1} < P$, where the staleness threshold is S. We set a minimum transmission amount (MTA), which the percentage of rows per transmission cannot be lower than, to be the solution to the above inequality in Table 3.1. If the amount of transmitted rows has not reached MTA, Speculative Transmission would go on transmitting the remaining rows of MTA in line 4 to 7. The possible cost of such speculative transmission is that the transmission of the last row transmitted could be incomplete and needs to be discarded, which is a negligible cost thanks to the small size of a row.

Threshold	2	3	4	5	6	7	8
MTA	0.5	0.38	0.32	0.28	0.25	0.22	0.2

Table 3.1: MTA values under different thresholds

3.4.3 Proof of guaranteed convergence

Following the convention in [49], we refer to x as the "system state", and the operation $x \leftarrow x + u$ as "writing an update", where u is a "model update". We define $D(x||x\prime) = \frac{1}{2}\|x - x\prime\|^2$ and assume that P workers write model updates to the parameter server independently. Let u_t be the tth update written by workers and $\tilde{x}_t$ be the tth model parameters read by workers, and we divide the whole model parameters into M parts by row, which means $u_t = [u_t^1, u_t^2, ..., u_t^M]^T$ where u_t^i is the ith row of u_t and $\tilde{x}_t^i$ is the ith row of $\tilde{x}_t$.

This chapter focuses on SGD [13] and proves convergence of each row and, further, convergence of the entire model. Since ROG either synchronizes or aggregates each row of parameter updates, no parameter update in a row is lost; thus the final convergence of the whole training model can provably have the same convergence guarantee as SSP.

Theorem 1 (SGD under RSP) *Suppose we want to find the minimizer $x^\star$ of a convex function $f(x) = \sum_{t=1}^{T} f_t(x)$, via gradient descent on one component ∇f_t at a time. We assume the components f_t are also convex. Let $u_t = -\eta_t \nabla f_t(\tilde{x}_t)$, where $\eta_t = \frac{\sigma}{\sqrt{t}}$ with $\sigma = \frac{F}{L\sqrt{2(S+1)P}}$ for certain constants F, L, and $S_{max} = MAX_{i=1,2,...,M}(S_i)$. Then, assuming that $||\nabla f_t(x)|| \leq L$ for all t (i.e. f_t are L-Lipschitz), and that $MAX_{x,x\prime \in x} D(x||x\prime) \leq F^2$ (the optimization problem*

has bounded diameter), we claim that $R[x] = \sum_{t=1}^{T}(f_t(\tilde{x}_t) - f(x^\star)) \leq o(T)$*, which implies* $E_t[f_t(\tilde{x}_t) - f_t(x^\star)] \to 0$ *and thus convergence.*

Proof. We define $\tilde{g}_t = \nabla f_t(\tilde{x}_t) = [\tilde{g}_t^1, \tilde{g}_t^2, ..., \tilde{g}_t^M]^T$ and there is

$$\begin{aligned} R[x] &= \sum_{t=1}^{T}(f_t(\tilde{x}_t) - f(x^\star)) \leq \sum_{t=1}^{T}\langle \nabla f_t(\tilde{x}_t), \tilde{x}_t - x^\star \rangle \\ &\quad \text{(the property of convex functions)} \\ &= \sum_{t=1}^{T}\langle \tilde{g}_t, \tilde{x}_t - x^\star \rangle = \sum_{t=1}^{T}\sum_{i=1}^{M}\left\langle \tilde{g}_t^i, \tilde{x}_t^i - x^{i,\star} \right\rangle \\ &\quad \text{(the property of inner product)} \\ &= \sum_{i=1}^{M}\sum_{t=1}^{T}\left\langle \tilde{g}_t^i, \tilde{x}_t^i - x^{i,\star} \right\rangle \quad \text{(since } i \text{ and } t \text{ are independent)} \end{aligned} \tag{3.1}$$

Then, we are going to prove the convergence of each row by finding the upper boundary of $\sum_{t=1}^{T}\left\langle \tilde{g}_t^i, \tilde{x}_t^i - x^{i,\star} \right\rangle$ for each row. Thanks to SSP's pioneering work [49], we can learn the following Lemma 1.

Lemma 1 *For P workers with staleness threshold* $S\prime$*, we assume that* $||\nabla f\prime_t(x)|| \leq L\prime$ *and* $max_{x^i, x\prime^i \in X^i} D(x^i||x\prime^i) \leq F\prime^2$*. If we set the initial step size* $\sigma = \frac{F}{L\sqrt{2\kappa}}$*, where* $\kappa = (s+1)P$ *and assume T is large enough that* $\frac{1}{2\kappa} + \frac{\kappa}{\sqrt{T}} \leq 1$*, then*

$$\begin{aligned} R[X] &\leq F\prime L\prime\sqrt{2\kappa T}\left[3 + \frac{1}{2\kappa} + \frac{\kappa}{\sqrt{T}}\right] \\ &\leq 4F\prime L\prime\sqrt{2(S\prime+1)PT} \end{aligned} \tag{3.2}$$

Because the parameters of each row are independent of each other, RSP keeps the same staleness threshold control for each row as SSP does. In this way, RSP breaks the granularity of the synchronization model from the whole model to rows and still achieves the same convergence guarantee for any single row as SSP from Lemma 1, which means each RSP's row still satisfies Inequality 3.2:

$$\begin{aligned} R[X^i] &= \sum_{t=1}^{T}\left\langle \tilde{g}_t^i, \tilde{x}_t^i - x^{i,\star} \right\rangle \leq 4F\prime L\prime\sqrt{2(S_i+1)PT} \\ &\leq 4F\prime L\prime\sqrt{2(S_{max}+1)PT} \quad (S_{max} = MAX_{i=1,2,...,M}(S_i)) \end{aligned} \tag{3.3}$$

Based on Inequality 3.1 and Inequality 3.3, we further have

$$\begin{aligned} R[X] &\leq \sum_{i=1}^{M}\sum_{t=1}^{T}\left\langle \tilde{g}_t^i, \tilde{x}_t^i - x^{i,\star} \right\rangle \leq M * MAX(R[X^i]) \\ &= M * 4F\prime L\prime\sqrt{2(S_{max}+1)PT} \end{aligned} \tag{3.4}$$

As $x = [x^1, x^2, ..., x^M]^T$ and the parameters of each row are independent of each other, there is $\|x - x\prime\|^2 = \sum_{i=1}^{M} \left\|x^i - x\prime^i\right\|^2$, and we can further have

$$D(x||x\prime) = \frac{1}{2}\|x - x\prime\|^2 = \frac{1}{2}\sum_{i=1}^{M}\left\|x^i - x\prime^i\right\|^2 = \sum_{i=1}^{M} D(x^i||x\prime^i) \tag{3.5}$$

Since $D(x^i||x\prime^i)$ is independent of each other with various i, in order to maximize $D(x||x\prime)$, all $D(x^i||x\prime^i)$ need to be maximized, which means

$$MAX(D(x||x\prime)) = \sum_{i=1}^{M} MAX(D(x^i||x\prime^i)) = M * MAX(D(x^i||x\prime^i)) \tag{3.6}$$

In other words, there is $F^2 = M * F\prime^2$ and $F\prime = \frac{F}{\sqrt{M}}$can be obtained after deformation. In the same way, we can have $L\prime = \frac{L}{\sqrt{M}}$.

Returning to the proof of Theorem 1, we substitute $F\prime$,$L\prime$ into Inequality 3.4 and

$$\begin{aligned} R[X] &\leq M * 4F\prime L\prime\sqrt{2(S_{max}+1)PT} \\ &= M * 4\frac{F}{\sqrt{M}}\frac{L}{\sqrt{M}}\sqrt{2(S_{max}+1)PT} \\ &= 4FL\sqrt{2(S_{max}+1)PT} \leq o(T) \end{aligned} \tag{3.7}$$

Until now, we have completed the proof of Theorem 1, which means $\tilde{x}_t$ converges in expectation to the minimizer $x^\star$.

3.5 Implementation

ROG is implemented as an optimizer in PyTorch [116] with nearly 1200 lines of code. ROG exposes similar APIs as existing PyTorch optimizers (`torch.optim` [148]), so that it can be integrated by simply replacing the application's original optimizer with ROG's optimizer. Under the hood, ROG launches a parameter server on one of the devices to keep track of the training process among all the devices. On each device, ROG transparently inspects the underlying tensors storing parameters of the model and tracks each row's versions. Each time `optimizer.step()` is called, ROG updates the parameters and row versions of the local model, and synchronizes with the parameter server according to the ATP protocol.

During the synchronization process, the communicated gradients are compressed before transmission and decompressed after reception using the lossless one-bit compression algorithm described in [138] which typically reduces the transmission volume to 3.2% of the uncompressed counterpart. Our implemented compression process is as follows: the gradients are first compressed to one-bit tensors as defined in [138] on GPU (or CPU on devices without a GPU) and these tensors are then serialized and

moved to CPU using cupy.packbits() [28] (or numpy.packbits() [104] on devices without a GPU) for transmission. The decompression process is exactly the reversion of the compression process. For the underlying optimizer, we implemented the block-wise distributed SGD-momentum algorithm in [138] and integrated it with [85] which supports staleness and local updates in SGD-momentum algorithms without damaging convergence.

In Speculative Transmission of ATP, we implemented SendWithTimeout() that enforces a time limit for the transmission and discards the ongoing transmission if the time limit is reached. The enforcement of the time limit is simply accomplished with *socket*: setting a timeout with $socket.settimeout()$ and then transmitting with $socket.sendall()$. One issue is that once the ongoing transmission is discarded, it is difficult for the receiver to be aware of the ending of the transmission and the discarded transmission can bring many fragments of incomplete information into the buffer of the receiver. To cope with it, we wrap such transmission with several unique bytes at both the beginning and the ending of the transmission, so that the receiver can be aware of the start and the ending of the transmission once it retrieves these unique bytes and the fragments are skipped.

3.6 Evaluation

Testbed. The evaluation was performed on five devices consisting of three four-wheel robots and two laptops. Each robot was equipped with an NVIDIA Jetson Xavier NX [145] and each laptop was equipped with an Intel Core i7-8565U CPU@1.80GHz CPU and a 940mx GPU (weaker than the NVIDIA Jetson Xavier NX in computation power). One laptop was chosen as the parameter server and directly connected to all other devices by enabling an IEEE802.11ac [56] hotspot over 80MHz channel at 5GHz frequency. Since the heterogeneity in computation power among the devices is out of our scope, we adopted dynamic batching in [149] to make all the involved devices spend equal time computing gradients in each iteration (see Table 3.2).

Experiment Scenarios. Our first evaluated online training application paradigm is referred to as coordinated robotic unsupervised domain adaptation (CRUDA): a team of robots are recognizing the images of objectives captured by their cameras with a shared objective recognition model, and the recognition accuracy is accidentally degraded by environmental noises (domain shift) such as fog. Such a paradigm is fundamental and representative [162, 161, 50] in typical robotic application scenarios including search, rescue, surveillance, and field exploration. In these scenarios, powerful datacenter servers are typically unavailable due to the lack of internet access in damaged buildings and outdoor fields. To recover the recognition accuracy as soon as possible, the team of robots adapt the shared model to the noises via wireless distributed training on collected noised images. We assume the dataset with noise for online adaptation can be

generated following the unsupervised domain adaptation methods[86, 6], which typically train the model on generated adversarial (noised) examples for adaptation; we generated these adversarial examples by adding noise to the original datasets for simplicity.

The second paradigm is referred to as coordinated robotic implicit mapping and positioning (CRIMP): a team of robots continuously collect images of their surroundings through their cameras; meanwhile, the team of robots cooperatively constructs a shared implicit map (a machine learning model representing the 3D map of an area) over the collected images and positions themselves in the shared map. This is also an important task for robotics, as it provides not only 3D reconstruction of the area of interest, but also the real-time positioning information of the robots which is essential for many robotic tasks such as navigation and exploration. The major metric we use for CRIMP is the trajectory error (i.e., the error between the ground-truth positions of the robots and their predicted positions).

Experiment Environments. We setup two real-world environments for our evaluation, namely *indoors* and *outdoors*. In the *indoors* scenario, robots move around in our laboratory with desks and separators interfering with wireless signals. In the *outdoors* scenario, robots move around in our campus garden with trees and bushes interfering with wireless signals and this scenario imposes higher level of instability as discussed in Sec. 3.2.2.

Baselines. We compared ROG with BSP [45], SSP [49] and the framework proposed in [20] (referred to as FLOWN). FLOWN is one of the most SOTA scheduling-based methods specified for distributed training over unstable wireless networks.

Datasets. For CRUDA, we use the well-studied Fed-CIFAR100 [144, 26] as the image dataset of objectives, with 50000 samples (100 types and each has 500 images) for training and 10000 samples (100 images for each type) for testing. This dataset is also plausible for simulating the real-world unbalanced data distribution by partitioning the images into 500 shards using the Pachinko Allocation Method [78] and we equally divided the dataset into four parts without overlap, each for one of the workers. We follow the methods of DeepTest [147], a DNN testing framework, to add noises to the Fed-CIFAR100 dataset to simulate fog and brightness changes. For CRIMP, we use a short sequence of 500 continuous images captured inside an apartment from the ScanNet dataset [29] and separate the sequence into several continuous sequences for each robot. One of the images is fixed and shared among all the robots as the shared starting point of mapping and positioning.

Models. We choose different model sizes for the two applications respectively to evaluate how ROG performs under different communication data volumes. For CRUDA, we choose ConvMLP [74] as the objective recognition model as it achieves both lightweight (total gradients are sized 65 MB before compression and 2.1MB after compression) and high recognition accuracy (89.13%) on the Fed-CIFAR100 dataset. Our added noise

leads to a lowered recognition accuracy (52.88%) of ConvMLP and we need to online train the ConvMLP model to recover its accuracy. For CRIMP, we choose nice-slam [174], one of the SOTA implicit mapping and positioning methods. The nice-slam model we used is sized 24.2MB before compression and 0.76MB after compression, which is much smaller than the size of ConvMLP.

The default training configuration of ConvMLP [74] and statistics are listed in Table 3.2 and we used the default training configuration of the demo of nice-slam [173]. Note that we include time cost for compression and decompression in the computation time.

batch size (robot)	batch size (laptop)	learning rate	compress + decompress time cost
24	16	1e-6	0.42s to 0.51s

Table 3.2: Default Setup

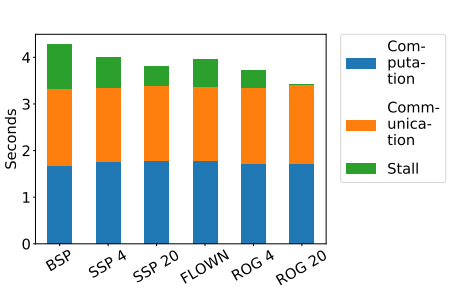
The evaluation questions are as follows:

- RQ1: How does ROG benefit real-world robotic applications compared to baseline systems in terms of training accuracy and power consumption by fulfilling **3Rs**?
- RQ2: How does ROG handle the unstable wireless networks?
- RQ3: How sensitive is ROG to different batch sizes, different numbers of devices, and different thresholds?
- RQ4: What are the limitations and potentials of ROG?

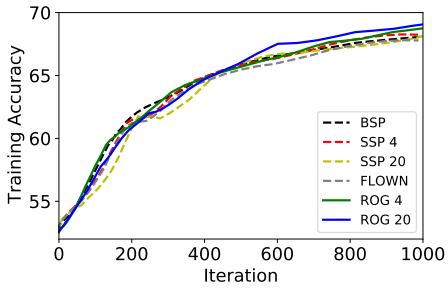
3.6.1 End-to-end Performance

We first compare the training accuracy/trajectory error of the two applications and energy consumption of their training processes over time under different training systems. The training accuracy of CRUDA and trajectory error of CRIMP were both obtained by checkpointing and validating the training model on each worker every 50 training iterations and then averaging the validated accuracy/trajectory error among the workers. We measured and averaged the energy consumption of the whole development board including CPU, GPU, memory and wireless card on all robots with jtop [12], a well-recognized monitoring tool for NVIDIA Jetson boards. Since jtop only reports transient power consumption, we approximated the energy consumption of each run by recording the power consumption at 10 Hz and calculating the total power consumption with numerical integration.

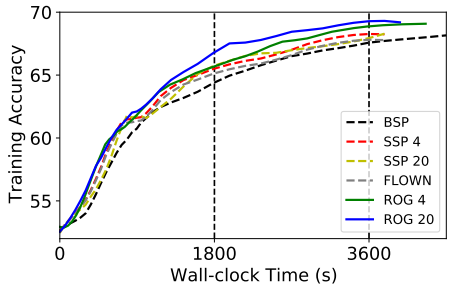
CRUDA. Our evaluation results of CRUDA in Fig. 3.1 and Fig. 3.6 show that ROG achieved both high training accuracy and high energy efficiency. When training for 30 minutes, ROG achieved 3.3%~6.8% higher accuracy than the baselines in *outdoors* in Fig. 3.1(c), and up to 1.8% accuracy gain in *indoors* in Fig. 3.6(c) due to reduced instability. When training for 60 minutes, ROG achieved 4.9%~6.5% higher accuracy



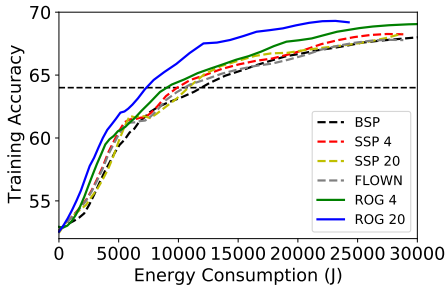
(a) Average time composition of a training iteration.



(b) Statistical efficiency.



(c) Training accuracy against wall-clock time.



(d) Energy consumption against training accuracy.

Figure 3.6: Comparison between ROG and the baselines with CRUDA in *indoors*.

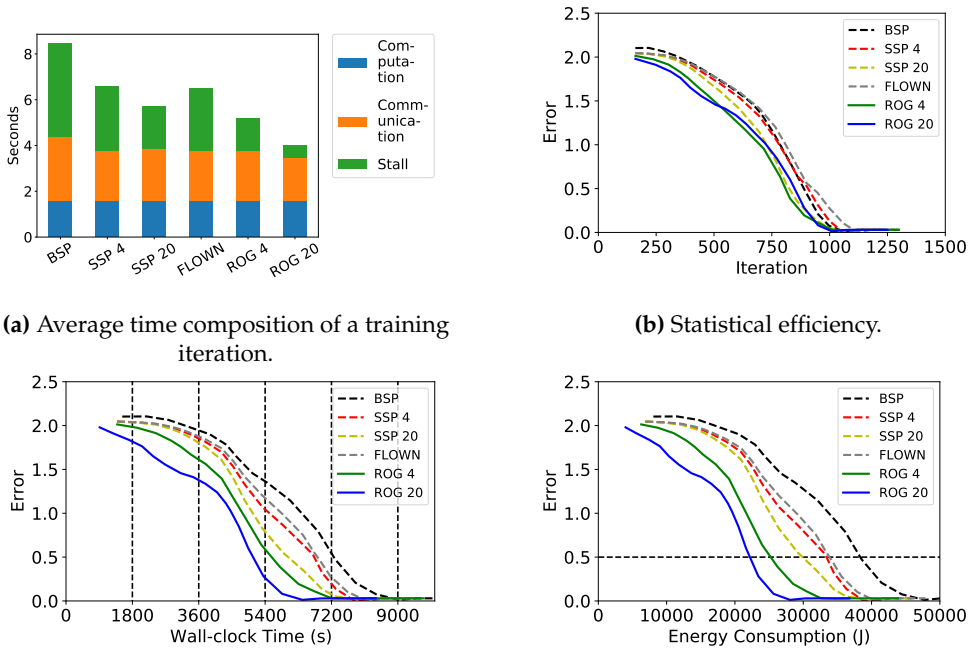
than the baselines in *outdoors*. Such accuracy gains have been reported as critical in real-world robotic applications [162, 135]. In terms of energy consumption, when the training model reached an accuracy of 64.0%, ROG saved 20.4%~50.7% of the battery energy in *outdoors* (Fig. 3.1(d)). The reduction of energy consumption was up to 41.3% in *indoors* still due to reduced instability.

The key reason for ROG's high performance is its mitigation of stall time (high training throughput, **R2**) without sacrificing statistical efficiency (high statistical efficiency, **R3**) in various environments with different levels of instability (robustness, **R1**). Fig. 3.1(a) and Fig. 3.6(a) show the recorded average time composition of a training iteration where a shorter total time duration of a training iteration implies higher training throughput. In a training iteration while all systems took almost the same time computing gradients and communicating the compressed gradients, BSP, SSP-4, SSP-20 and FLOWN suffered at least 4.8s stall in *outdoors* and 0.4s stall in *indoors*. ROG reduced the stall time by 49.1% to 86.5% in *outdoors* and by 42.4% to 97.6% in *indoors*. ROG achieved less stall time reduction in *indoors* because the wireless networks are less unstable in *indoors*. By breaking down the whole model into rows and transmitting at the row granularity, ROG prevents transiently degraded bandwidth from blocking the overall training process.

Fig. 3.1(b) and Fig. 3.6(b) show that ROG achieved similar statistical efficiency as BSP. To avoid being blocked by degraded bandwidth during synchronization, it is inevitable to reduce the transmission traffic volume and postpone the synchronization of

some rows, which causes staleness in un-transmitted gradients. To minimize its impact on statistical efficiency, ROG's ATP identifies rows with large gradients and prioritizes them. Therefore, even if stragglers transmit fewer gradients than non-stragglers, important changes to the model are always synced, resulting in a comparable statistical efficiency as BSP.

To further understand the energy consumption statistics, we identify three major states, namely *computation*, *communication*, and *stall* of a system during training, and measure the power consumption of each state. We obtained the power consumption of different states in Table 3.3 by matching power consumption records with the training system status log. There was minor (below 5%) difference across all the evaluated systems, since all the systems do not change how computation and stall states behave, while the overhead of scheduling during communication is negligible. The stall state power was nearly 30% of the computation state power, since chips like CPU, GPU, and memory consume non-negligible power even when not computing (i.e., in stall state) due to the static power consumption rooted in transistors' leakage current [101]. Note that communication and stall have similar power consumption, this may be due to the relatively low wireless transmission rate (compared to high-speed datacenter networks), involving little energy-consuming operations. Since ROG reduced stall time, the corresponding power consumed during stall was reduced accordingly, accounting for ROG's high energy efficiency.



(a) Average time composition of a training iteration.

(b) Statistical efficiency.

(c) Training accuracy against wall-clock time.

(d) Energy consumption against training accuracy.

Figure 3.7: Comparison between ROG and the baselines with CRIMP in *outdoors*.

CRIMP. We mainly evaluated CRIMP in *outdoors*, as shown in Fig. 3.7. As shown in

	computation	communication	stall
Power (W)	13.35	4.25	4.04

Table 3.3: Power (Watt) in different states.

Fig. 3.7(c) and Fig. 3.7(d), ROG also achieved similar high training accuracy (less trajectory error) and high energy efficiency in CRIMP. When training for 30 minutes, ROG reduced 6%~13% trajectory error compared with the baselines in Fig. 3.7(c). After training for 60 minutes, the reduction of trajectory error of ROG over the baselines increased to 16%~30%. Also, the energy consumption reduction of ROG over the baselines is outstanding: 32%~41% less energy to reach the trajectory error of 0.5 in Fig. 3.7(d). Note that while the model size of CRIMP amounts to only one third of CRUDA and its average communication time is reduced, the straggler effect of CRIMP is still severe with stall time taking up 60% of the communication time in BSP in Fig. 3.7(a). That is because with a smaller model, the computation time in a training iteration is also reduced and communication remains the bottleneck of the training process.

3.6.2 Micro-Event Analysis

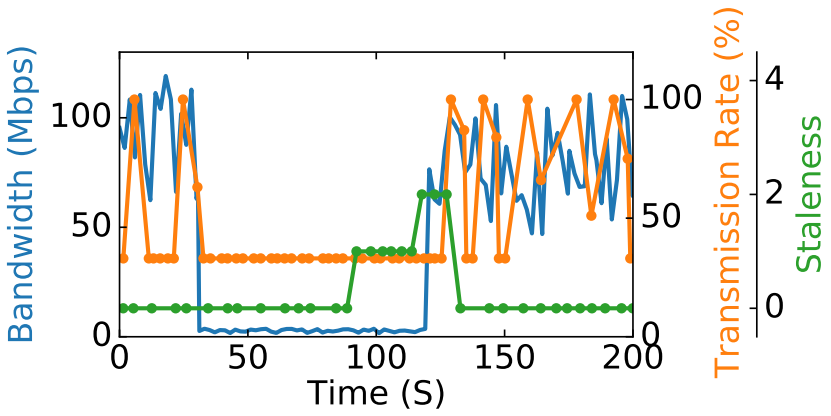


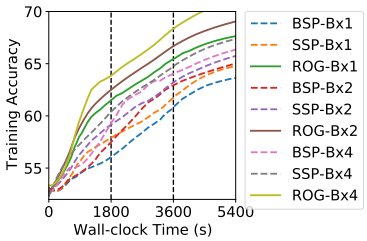
Figure 3.8: Real-time bandwidth and the percentage of rows transmitted by ROG

To further understand the performance gain of ROG, we recorded the real-time bandwidth and how ROG responded to it by adjusting the percentage of rows to be transmitted out of all rows in each iteration (referred to as transmission rate) on one robot, as shown in Fig. 3.8. How many training iterations that this robot fell behind the fastest worker is also recorded (referred to as staleness). Since proactive methods (e.g., measuring with `iperf`) would affect the application traffic and bandwidth, we passively measured the real-time bandwidth with the expected throughput reported by `iw` [40]. Note that `iw`'s output is an estimation of the physical layer bitrate which deviates from the actual bandwidth the application could exploit, we normalize the output with its average.

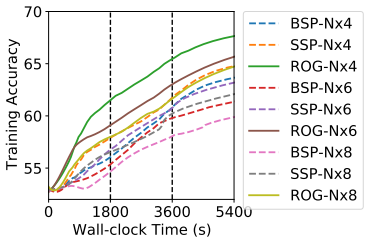
When bandwidth was fluctuating in the former part of Fig. 3.8, ROG responded immediately and adjusted the transmission rate on a robot accordingly. This aligned the transmission time between this robot and the fastest, avoiding possible straggler

effect and the staleness was in a low level (0 to 1). In this way, ROG prevented a robot from straggling and stalling the training process. When bandwidth degraded to an extremely low level and lasted for a long time in the middle part of Fig. 3.8, it was impossible to perform even minimal necessary synchronization under such conditions, and no system could keep in sync. Thus staleness slowly accumulated on this robot. When bandwidth recovered in the latter part of Fig. 3.8, this robot caught up quickly (staleness decreased) because it was allowed to transmit partial of its rows.

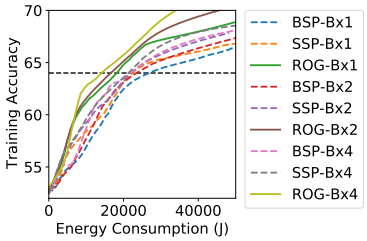
3.6.3 Sensitivity Studies



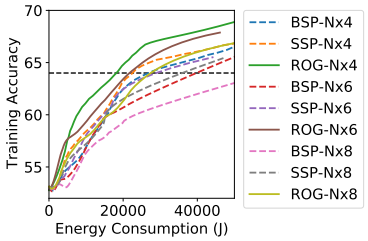
(a) Accuracy with various batch sizes.



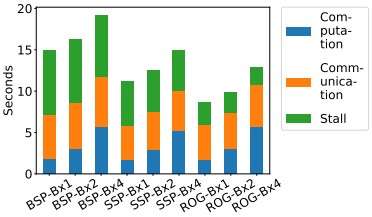
(b) Accuracy with various numbers of workers.



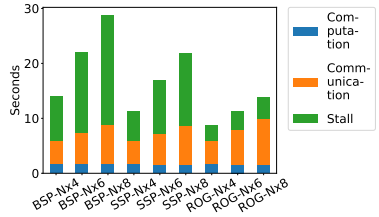
(c) Energy consumption with various batch sizes.



(d) Energy consumption with various numbers of workers.



(e) Average time composition with various batch sizes.



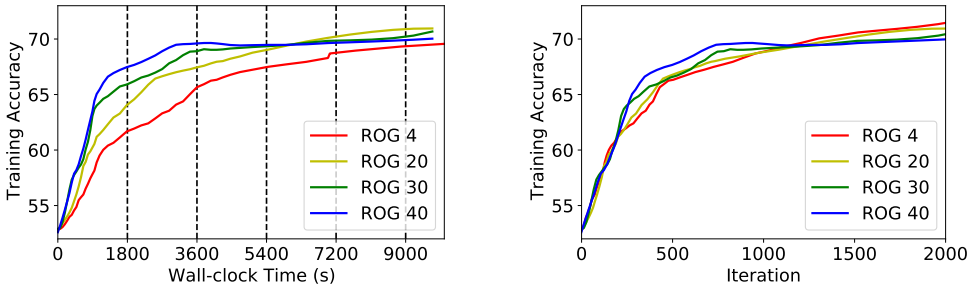
(f) Average time composition with various numbers of workers.

Figure 3.9: Sensitivity Studies about different batch sizes (left column) and worker numbers (right column)

batch size. We varied the batch size (x2, x4) of training in CRUDA in *outdoors* to examine how ROG performs with different ratios of computation and communication, as shown in the left column of Fig. 3.9. As FLOWN typically achieved performance between SSP and BSP, we omit it in the following sections for simplicity. When the batch size increased, the computation time proportionally increased and thus communication time would take a smaller portion in a training iteration. In this case, the straggler effect

will be less severe (stall time decreased in the baselines) and ROG's potential gain over the baselines is limited. When training for 30 minutes, ROG achieved 5.3% accuracy gain over the baselines and 30.3% energy consumption reduction when training accuracy reached 64% in the doubled batch size case; When the batch size was increased to four times, ROG achieved 3.5% accuracy gain over the baselines and 33.7% energy consumption reduction when training accuracy reached 64%.

Number of workers. Increasing the number of training workers (4, 6, 8 workers) in CRUDA caused more severe straggler effect, as shown in the right column of Fig. 3.9. First, as the workers all share a same wireless channel, varied number of workers involved will incur traffic volume proportional to the number of workers, causing communication time to take a larger portion in a training iteration. Second, the contention for wireless channel among workers is an extra source of instability, deteriorating the straggler effect. In this case in Fig. 3.9, when training for an hour, ROG achieved 3.0% accuracy gain over the baselines and 48.1% energy consumption reduction when training accuracy reached 64.2% in the 6 workers case; When the number of workers was increased to 8, ROG achieved 3.7% accuracy gain over the baselines and 55.1% energy consumption reduction when training accuracy reached 64%.



(a) Training accuracy against wall-clock time. **(b)** Statistical efficiency.

Figure 3.10: Sensitivity Studies about different thresholds

Threshold. We empirically evaluated ROG's performance under a wider variety of staleness thresholds with the default training configuration with CRUDA, as shown in Fig. 3.10. From Fig. 3.10 we can learn that there is a tradeoff between training speed and final training accuracy when using different thresholds in ROG: while a large threshold (30 or 40) brings higher training throughput and potentially even higher statistical efficiency in the early stage of training, a too large threshold will degrade the statistical efficiency in the late stage of training and lead to slightly degraded final training accuracy (similarly reported by SSP with different staleness thresholds [49]). The reason could be although ROG limits divergence of training models on different workers and guarantees convergence, a large threshold inevitably leads to larger level of divergence among the models and leads to suboptimal final training accuracy. Depending on whether the training task requires extra fast training speed or high training quality, there would be an optimal threshold selection for it and we leave automatic finding the optimal threshold as future work.

3.6.4 Lessons learned

Wireless distributed training over robots still faces many challenges. During the implementation and evaluation of ROG, we find that wireless distributed training over robots is challenging both systematically and algorithmically. Systematically, unlike GPU clusters equipped with fast interconnects such as InfiniBand [125], robots lack fast and stable network connection between each other for model synchronization and lack enough power for long-term training, which are partially mitigated by ROG. Algorithmically, the collected data of different robots are typically non-IID (e.g., different robots are surveilling and capturing data from different parts of an area), while there is not yet a well-recognized method for distributed training over non-IID datasets.

Generalizability of ROG. Due to limitations of our hardware, we did not evaluate ROG's performance on a wider variety of wireless networks such as 5G [129] or WiMAX [4], but only the most common and easily accessible Wi-Fi networks on robots under different environments. However, while these various wireless networks differ in throughput and communication range, decay of wireless signals due to varying distance or occlusion is still a common issue among them. The resulting throughput fluctuation will still cause the straggler effect in these wireless networks, where ROG will be beneficial. Overall, ROG is optimized for distributed training over any wireless LAN with frequent bandwidth fluctuation and we leave the evaluation of ROG over a wider variety of wireless networks as future work.

Finer granularity brings extra management overhead and transmission overhead. While Finer granularity in ROG enables more flexibility in scheduling to adapt to the instability of real-world robotic IoT networks, it also brings extra management overhead (e.g., index) and transmission overhead in the wireless distributed training process. Although we minimized these overheads by choosing a balanced granularity of rows and enabling speculative transmission, such overhead cannot be eliminated and potentially limits the performance gain of ROG over the baselines.

Future work. We would like to apply and evaluate ROG in a wider variety of online learning robotic tasks and environments in the future. Also, it is of interest to explore further improvements of ROG such as pipelining communication and computation on a robot in the training process as described in [79] or even decoupling communication and computation. We believe such investigation will enable even faster and more robust wireless distributed training in real-world Robotic IoT Networks.

3.7 Conclusion

This chapter has presented ROG, a row-granulated distributed training system optimized for robotic IoT networks. By breaking model synchronization into rows and adaptively scheduling the transmission of each row, ROG balances transmission time among workers under unstable wireless bandwidth and prevents stragglers from stalling

the whole training process. In this way, ROG improves training throughput while preserving statistical efficiency, achieving higher accuracy and better energy efficiency in distributed training. These results show that row-level granularity is an effective execution unit for online model adaptation in mobile and robotic edge environments.

Chapter 4

LOPInfer: Local-Operator Parallel Inference for MEC Service Workloads

CHAPTER 3 addressed how robots can improve deployed models through online training under unstable wireless links. This chapter turns to the second lifecycle requirement of MEC intelligence: low-latency inference for timely feedback. At this layer, the *granularity mismatch* is between whole-layer partitioning and the fine-grained overlap structure inside one inference request. MEC services often run with batch size one on battery-limited devices, so the objective is not only high throughput but also low per-request latency and energy. Conventional layer-wise partitioning schedules a DNN at whole-layer granularity: the device computes a prefix, transmits an intermediate activation as one unit, and waits for the server to compute the suffix. This stop-and-wait structure leaves wireless transmission on the critical path. Data-center parallelism techniques are also ill-suited to MEC service workloads because data parallelism needs large batches, tensor parallelism requires frequent all-reduce collectives, and pipeline parallelism improves throughput rather than single-request latency.

LOPInfer resolves this mismatch by changing the scheduling granularity from whole layers to local operations. This is the inference-layer instance of *granularity-aware execution*: the system chooses a scheduling unit that exposes producer-consumer structure inside one request while respecting wireless transmission and device-server computation limits. LOPInfer observes that many DNN operators are *local*: parts of their outputs depend only on corresponding parts of their inputs, so their computation can be decomposed into independent sub-operations. Local Operation Parallelism (LOP) tracks producer-consumer dependencies among these sub-operations to preserve model semantics, and the Local Operation Scheduling Strategy (LOSS) formulates device-server placement as a constrained optimization under MEC bandwidth and compute constraints. Together, they enable intra- and cross-layer overlap between computation and transmission within a single inference. The design also extends the authors' ApPLIED 2024 workshop paper on distributed inference for DNN models

on robotic IoT, which first quantified the mismatch between data-center parallelism and MEC service workloads; that analysis is reused as part of the motivation in Section 4.2.4. LOPInfer is released at `https://github.com/hku-systems/LOPInfer`.

The remainder of this chapter reuses the original paper text. Section 4.1 motivates LOPInfer and summarizes its contributions. Section 4.2 describes MEC service workloads, the limits of device-only, server-only, and layer-wise collaborative inference, and why conventional parallel computing is ill-suited to MEC. Section 4.3 gives the system overview. Section 4.4 presents LOP and LOSS, including the scheduling formulation and adaptive control under wireless fluctuation. Sections 4.5 and 4.6 describe the implementation and evaluation on commercial robots and GPU servers under indoor and outdoor MEC networks. Section 4.7 discusses related work and limitations, and Section 4.8 summarizes the chapter.

4.1 Introduction

This section instantiates the thesis-level *granularity mismatch* at the inference layer. MEC inference must return each request quickly and energy-efficiently, yet layer-wise collaborative inference treats an entire layer output as the scheduling and transmission unit. The core question is therefore how to expose a finer unit that preserves DNN semantics while allowing computation and communication to overlap within one request.

Mobile edge computing (MEC) underpins real-time intelligent services (e.g., autonomous navigation [172], robot perception and control [96, 153], and IoT systems [157]) 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.

Conventional MEC inference follows three paradigms (Fig. 4.1): device-only, server-only, and collaborative inference via layer-wise partitioning [80, 51, 64, 23, 24]. 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 [51], offloading features rather than the input can substantially reduce transmission payloads, thereby improving end-to-end latency and energy efficiency (Sec. 4.2.5).

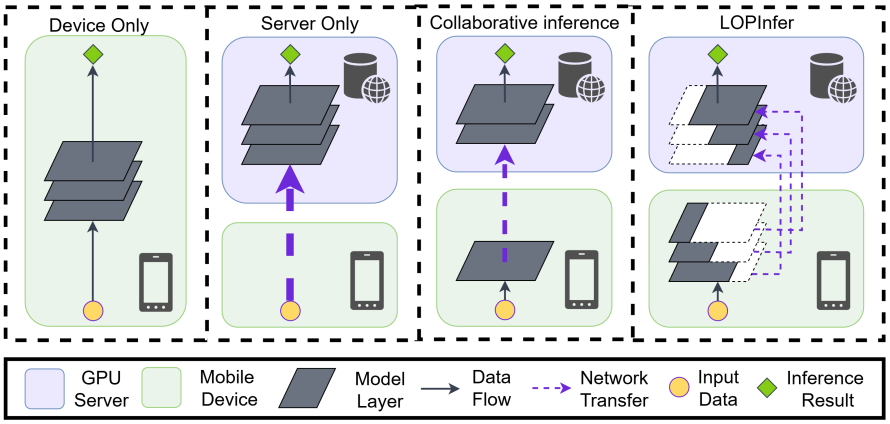


Figure 4.1: Comparison between conventional MEC inference paradigms with LOPInfer. Inference results computed on the GPU server must be transmitted back to the mobile device for application utilization.

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. 4.2.2). 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. 4.2.3). 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. 4.2.3). 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 [44] 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) [67]; 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 chapter, 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. 4.2.4), 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 chapter 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`.

4.2 Background and Motivation

4.2.1 MEC Service Workloads

We target real-time intelligent MEC services (e.g., autonomous navigation [172], robot perception and control [96, 153], and IoT systems [157]) 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.

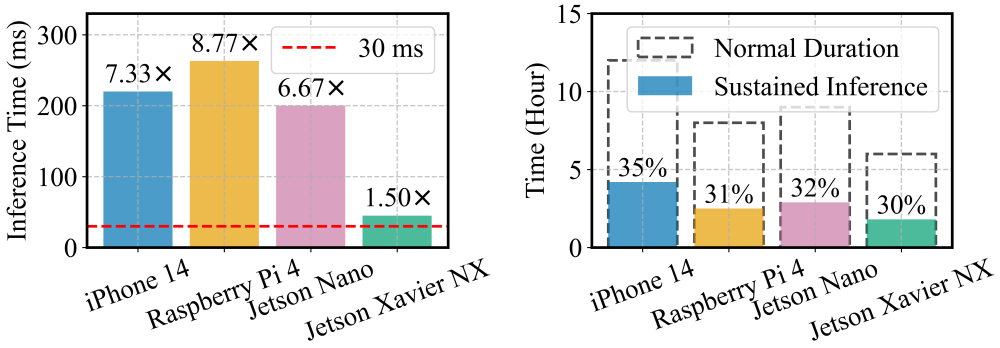
4.2.2 Device-Only Inference

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 in Fig. 4.2(a), the average per-request latency exceeds the 30 ms threshold for smooth video fluency [46] (red dotted line). Fig. 4.2(b) further shows that sustained inference reduces device standby time to 30–35% of normal, degrading user experience and undermining overall device practicality.

4.2.3 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. 4.3, on the same testbed as Sec. 4.5 (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 [105] 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



(a) Average Inference Latency **(b)** Battery Life

Figure 4.2: The performance of VGG-16 under device-only inference across different mobile devices [97, 106, 102].

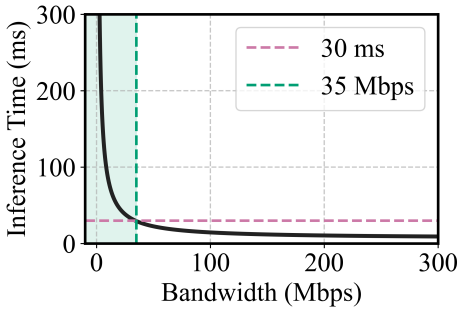


Figure 4.3: The inference time of VGG-16 under server-only inference across different network bandwidth.

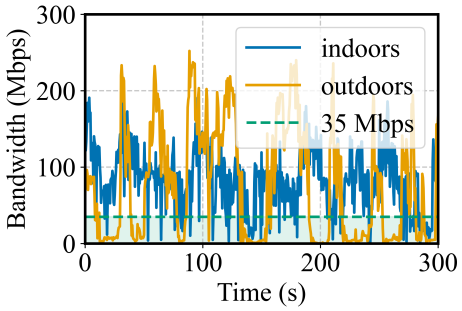


Figure 4.4: The wireless transmission instability of TCP between our robot and the base station in MEC networks.

6 can reach a peak of 1.2 Gbps per stream [84], commodity mobile hardware cannot fully utilize this capacity [166]; the actually available throughput further varies with device mobility [93], signal obstruction [37], and channel contention [122].

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 [57] 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. 4.4, 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.

4.2.4 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

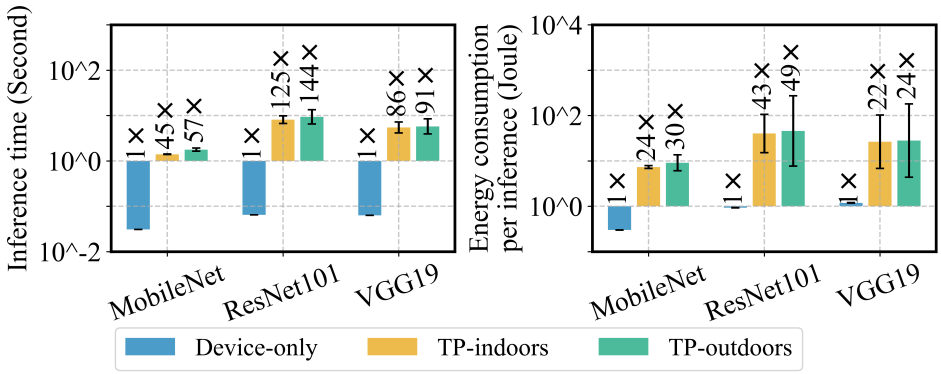


Figure 4.5: The performance of TP for different models. The cross marker (×) denotes the mean value.

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 [98], a state-of-the-art TP method, on our testbed (Sec. 4.5) with batch size 1. As shown in Fig. 4.5, TP's synchronization overhead inflates per-request end-to-end latency by 45×-144× and on-device energy by 22×-49× relative to device-only inference. Although recent work reduces TP communication in data centers (e.g., joint spatial/temporal partitioning [152]), 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 [44]. SPINN [71] combines layer partitioning with early exits to reduce activation traffic and average cost, at the expense of accuracy and with

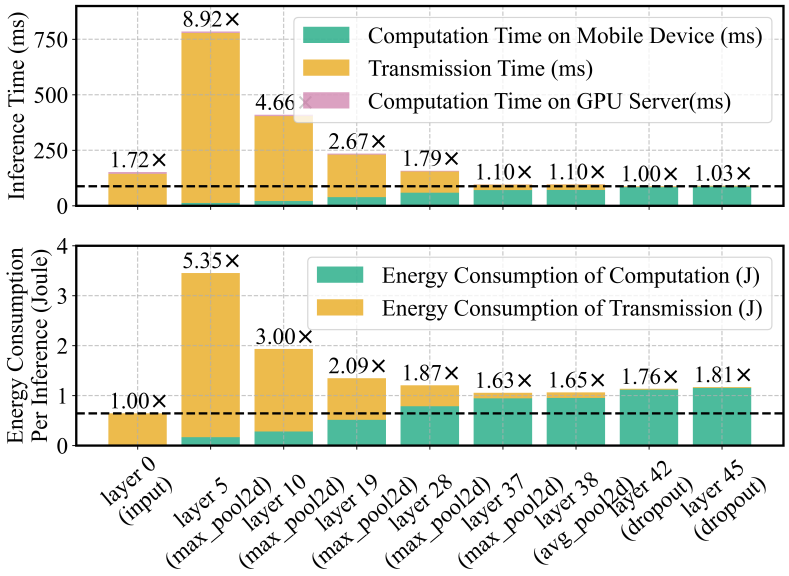


Figure 4.6: The performance of different layer partitioning strategies for VGG-19 under 35 Mbps in our experiments. The X-axis represents various partitioning points, where 'layer i' denotes that all layers up to and including the i_{th} layer are executed on the robot, while the remaining layers are offloaded to the GPU server. Note that transmission time depends on network bandwidth.

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 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.

4.2.5 Existing Collaborative Inference

Existing collaborative inference methods [80, 51, 64, 23, 24] primarily use layer partitioning to trade off per-request latency and energy (Fig. 4.6). Device-only and server-only are the two extremes, corresponding to splitting after the final layer or before the first; in Fig. 4.6, "layer 0" denotes server-only with all layers on the GPU server. Prior work typically targets either faster inference [51, 64, 80] or lower energy under deadlines [23, 24]. 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 [169]. Designed for IoT collectives, they are ill-suited to MEC: given the large device–edge performance gap, allocations often degenerate to server-only inference. [11] 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. 4.4.3), respects MEC constraints, and sustains high parallelism—capabilities absent from existing fine-grained approaches in this setting.

4.3 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.

4.3.1 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

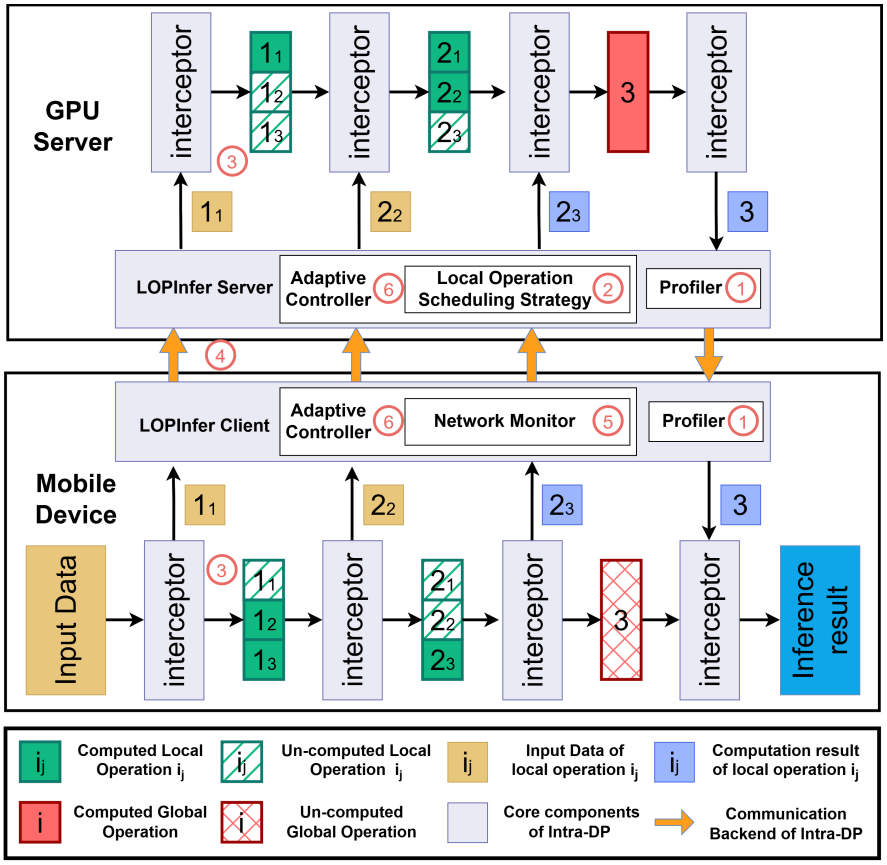


Figure 4.7: Overview and inference workflow of LOPInfer. Local operation i_j represents the j-th local operation within the i-th operator.

(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.

4.3.2 Overall Workflow

As shown in Fig. 4.7, 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. 4.4.3) 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. 4.4.1, ③) for per-operator tensor partitioning and reconstruction to enable partial-tensor execution, while Local-Operation Parallelism (Sec. 4.4.2, ④) pipelines local operations with dependency-preserving scheduling. A lightweight network monitor (⑤) continuously estimates network conditions using standard tools [167]; an adaptive controller (Sec. 4.4.4, 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. 4.4.1). 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. 4.4.3) 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.

4.4 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. 4.4.1) and establish the conditions under which operator-level parallel execution is provably correct. Building on this taxonomy, Sec. 4.4.2 introduces LOP, a parallel computing technique that guarantees inference correctness at the granularity of local operations. To orchestrate these parallel local operations, Sec. 4.4.3 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. 4.4.4 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.

4.4.1 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. 4.7 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 $\text{op} : (X, W) \to Y$ 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 $y[i] = g_i(X[R(\{i\})], W)$ 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 [169].

- **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 [54]	428 / 3	RegNet [164]	231 / 3
ResNet [142]	341 / 3	VGGNet [133]	73 / 5
ResNeXt [163]	342 / 3	ConvNeXt [160]	340 / 6

Table 4.1: 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 [96, 153]), benefit from LOPInfer via fine-grained operator parallelism. Table 4.1 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.

4.4.2 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. 4.7, 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.

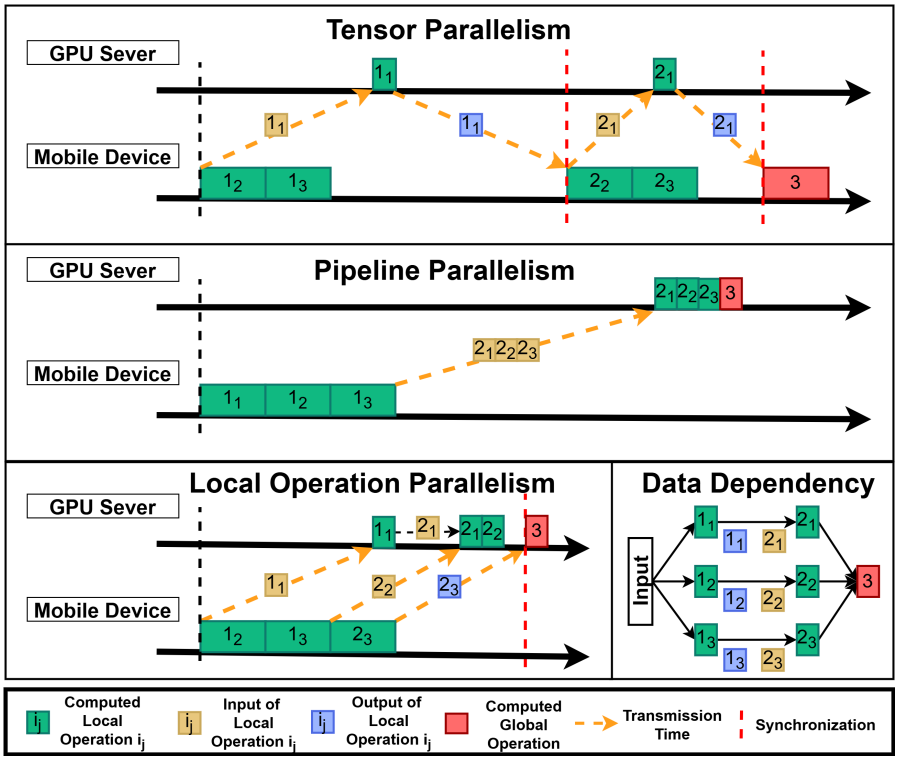


Figure 4.8: Workflow of TP, PP and LOP. In the three cases above, each local operator executes three operations with identical computation times on both the mobile device and the GPU server, along with the corresponding transmission time, while the lower right corner illustrates the data dependencies between operations.

Under these invariants, the parallel execution is functionally equivalent to the original inference by preserving correctness for both local and global operators.

Fig. 4.8 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 '$\mathrm{LO}i_j$'. 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 '$\mathrm{LO}1_2$', '$\mathrm{LO}1_3$', and '$\mathrm{LO}2_3$' on the mobile device, it concurrently transmits the inputs required by upcoming local operations '$\mathrm{LO}1_1$'), 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 '$\mathrm{LO}2_1$' on the GPU server to directly consume the output of '$\mathrm{LO}1_1$' resident there), eliminating avoidable transfers. Unlike TP, which incurs frequent all-reduce across tensor shards, LOP restricts synchronization to global operators only, 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 [67], shorter idle periods reduce energy per inference even with an active wireless network interface cards.

4.4.3 Local Operation Scheduling Strategy

This section explains how LOPInfer schedules computation and transmission for local operations to achieve fast, energy-efficient inference (Fig. 4.7, ②). LOSS leverages the fact that wireless bandwidth fluctuates on seconds-scale [115], 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.

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.

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.

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

$$\min T = s_M(v_{\text{out}}), \tag{4.1}$$

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 [23]. Exploration of such trade-offs is left to future work.

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 ($x_M(v_{\text{in}}) = input$ and $x_R(v_{\text{in}}) = \emptyset$) and the final result is consumed on the mobile device ($x_M(v_{\text{out}}) = output$ 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:

$$s_M(v) = \begin{cases} s_M(u) + C_M(u), \\ \quad \text{if } x_M(v) - child(x_M(u)) = \emptyset, \\ \max\big(s_R(u) + C_R(u) + T_{\text{RM}}(u, v), \\ \quad\quad s_M(u) + C_M(u)\big), \\ \quad \text{if } x_M(v) - child(x_M(u)) \neq \emptyset. \end{cases} \tag{4.2}$$

Here, 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 $x_M(v) -$ child$(x_M(u))$. When $x_M(v) -$ 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 [11] 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 [51], 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.

Solution Algorithm

Algorithm 5: LOSS heuristic solver (Offline Schedule)

Input: bandwidth b; time budget τ; 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.

X_{DO}^b, X_{SO}^b, $X_{LP}^b \leftarrow$ Baselines(model,b)
$X^b \leftarrow$ Best(X_{DO}^b, X_{SO}^b, X_{LP}^b); Beam $\leftarrow \{X^b\}$
for all local operator v in topological order **do**
Beam $\leftarrow$ ExpandBeam(Beam, v, b)
end for
$X^b \leftarrow$ Best(Beam); $t \leftarrow$ Now(); count $\leftarrow 0$
while Now()$-t < \tau$ and count $< K$ **do**
$X^b \leftarrow$ LNSRefine(X^b, b)
count $\leftarrow$ count $+1$
end while
$X^b \leftarrow$ Best(X^b, X_{DO}^b, X_{SO}^b, X_{LP}^b);
return X^b

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. 5) to plan quickly under a given bandwidth while satisfying the constraints in Sec. 4.4.3. 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 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. 4.4.3 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. 4.4.3) 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 [118] typically requires a large evaluation budget (population size × generations) and additional decoding/repair, inflating planning time. RL [60] entails substantial training cost and confronts a extremely vast operation-level state–action space; enforcing 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.

4.4.4 Adaptive Control Mechanism

The adaptive control mechanism in LOPInfer runs on both mobile device and GPU server (Fig. 4.7, 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 [167] (Fig. 4.7, 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.

Algorithm 6: 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.

$b \leftarrow$ TESTBANDWIDTH(); `Input(0)` $\leftarrow$ `input`
for all layer u in model **do**
  if $Input(u) \neq \emptyset$ and $x_M^b(u) \neq \emptyset$ **then**
    `output(u)` $\leftarrow$ COMPUTE(`Input(u)`, $x_M^b(u)$)
  end if
  for all edge $e = (u, v) \in E$ **do**
    if $(x_M^b(v) -$ `child`$(x_M^b(u)) \neq \emptyset$ **then**
      `Input(v)` $\leftarrow$ COMBINE(`Output(u)`,`Receive()`)
    end if
    if $x_R^b(v) -$ `child`$(x_R^b(u)) \neq \emptyset$ **then**
      SEND(`Output(u)`, $x_R^b(v) -$ `child`$(x_R^b(u))$)
    end if
  end for
end for
return *ret* $\leftarrow$ `Output(t)`

Algorithm 7: LOPInfer server at runtime stage

$b \leftarrow$ RECEIVE(); `Input(0)` $\leftarrow \emptyset$
for all layer u in model **do**
  if $Input(u) \neq \emptyset$ and $x_R^b(u) \neq \emptyset$ **then**
    `output(u)` $\leftarrow$ COMPUTE(`Input(u)`, $x_R^b(u)$)
  end if
  for all edge $e = (u, v) \in E$ **do**
    if $x_M^b(v) -$ `child`$(x_M^b(u)) \neq \emptyset$ **then**
      SEND(`Output(u)`, $x_M^b(v) -$ `child`$(x_M^b(u))$)
    end if
    if $(x_R^b(v) -$ `child`$(x_R^b(u)) \neq \emptyset$ **then**
      `Input(v)` $\leftarrow$ COMBINE(`Output(u)`,`Receive()`)
    end if
  end for
end for

The adaptive control procedure is realized by Alg. 6 (client) and Alg. 7 (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 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. 6 and Alg. 7). 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. 6).

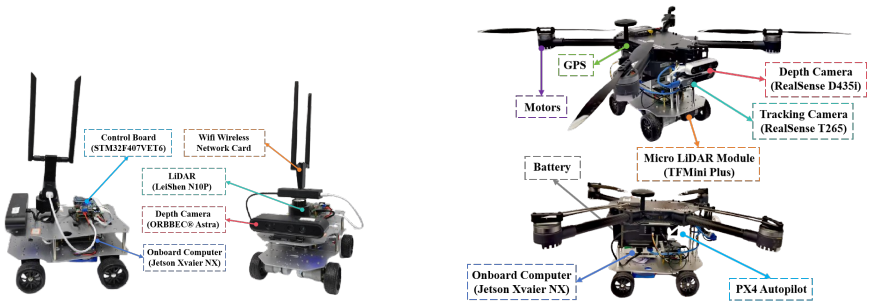
4.5 Implementation

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

4.5.1 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. 4.9(a)) and an air–ground hybrid robot (Fig. 4.9(b)). Each platform integrates an NVIDIA Jetson Xavier NX (8 GB) [106] 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. 4.9. The GPU server is a desktop with an Intel i7-7700K CPU



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

Figure 4.9: The detailed composition of the robot platforms.

	inference	communication	standby
Energy (Watt)	13.35	4.25	4.04

Table 4.2: Power draw (Watt) of our robot in different states.

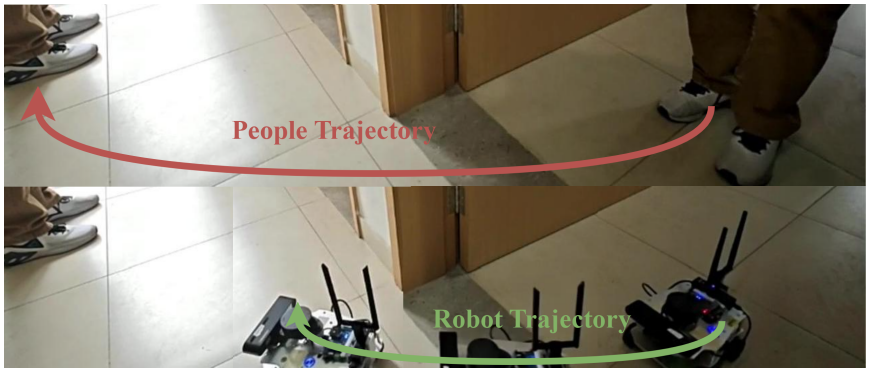


Figure 4.10: Kapao [96], a real-time people-tracking application on our four-wheeled robot with a CNN-based keypoint detection model.

and an NVIDIA GeForce RTX 3080 GPU, connected to the robots over Wi-Fi 6 on a 5 GHz, 80 MHz channel.

Table 4.2 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 [106]. 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.

4.5.2 Experiment Setup

Task. We evaluate two real-world robotic workloads on our physical robot platforms: KAPAO for people tracking [96] (Fig. 4.10) and AGRNav for autonomous navigation [153] (Fig. 4.11). Low per-request (per-frame) inference latency is critical for closed-loop operation; as illustrated in Figs. 4.10 and 4.11, faster inference shortens perception–action delay and mitigates target loss and localization error. For evaluation, we run KAPAO

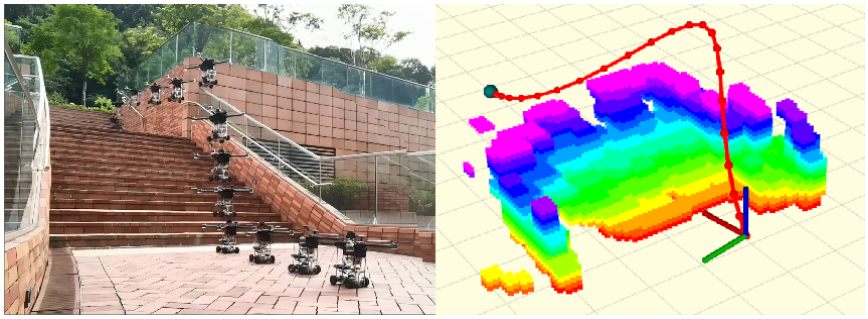


Figure 4.11: AGRNav [153], a navigation application on our air-ground robot with a CNN-based 3D semantic scene completion model.

on CrowdPose [75] and AGRNav on SemanticKITTI [9] using our testbed. To assess generality beyond robotics workloads, we also benchmark widely used MEC models (VGGNet [133], ConvNeXt [160], ResNet [142], and DenseNet [54]), using their Torchvision implementations [92] on CIFAR-100 [69]. 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. 4.4 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** [80]: 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** [23]: 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. 4.2.4, DP is inapplicable because the batch size is 1. TP is also unsuitable: on our hardware, its end-to-end latency (Fig. 4.5) 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 chapter.

4.6 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.

4.6.1 Superiority of LOPInfer

Inference Time. Fig. 4.12 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. 4.4). The improvement on DenseNet is less pronounced; we analyze model-structure sensitivity in Sec. 4.6.3. A micro-event study (Sec. 4.6.2) further explains LOPInfer's efficiency.

Energy Consumption. Fig. 4.13 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.

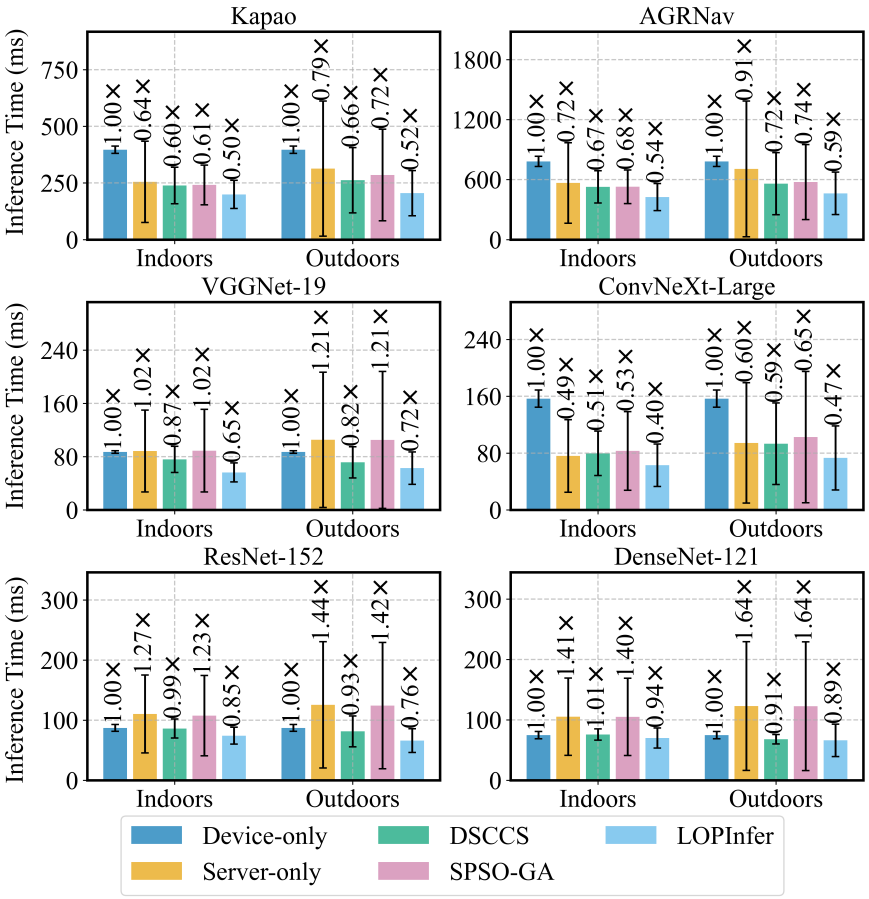


Figure 4.12: Inference time for different models across various environments and systems. The cross marker (×) denotes the mean value.

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, Table 4.2 shows the robot sustains roughly 95% of its active power even when idle, dominated by CPU/GPU/memory static leakage [67]; the wireless NIC adds only 0.21 W during transmission versus 13.35 W during computation.

Fig. 4.14 reports device energy per inference across systems and scenarios. Consistent with the power profiles in Fig. 4.13, 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 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. 4.4.3) is

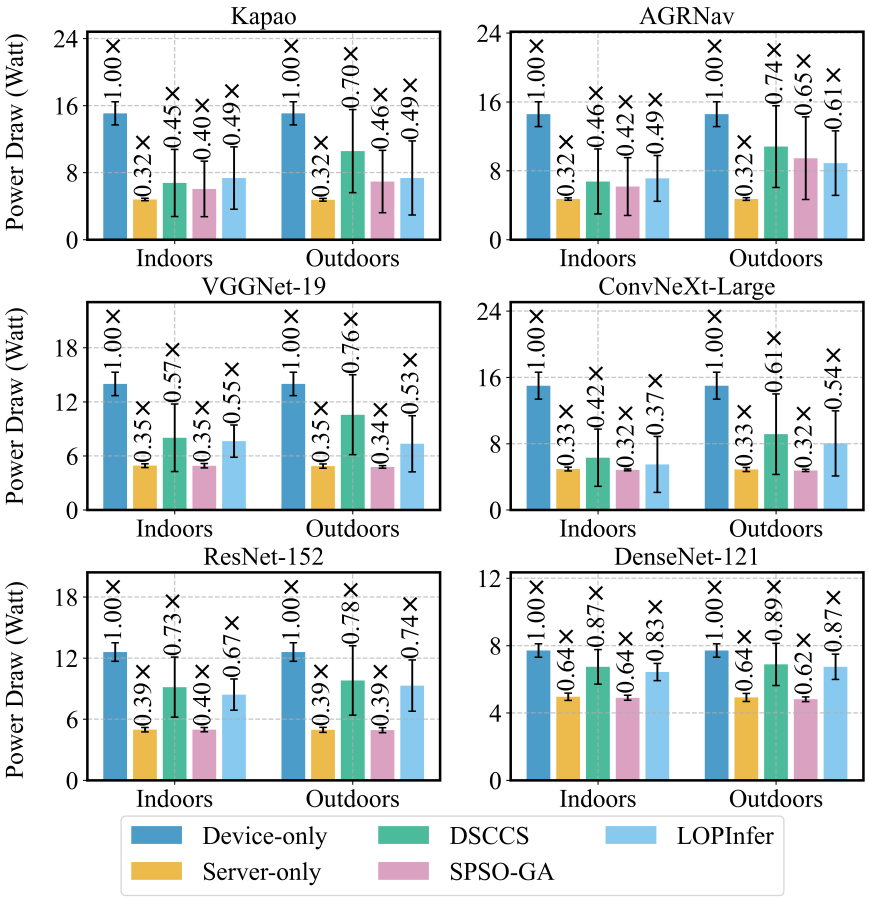


Figure 4.13: Power draw for different models across various environments and systems. The cross marker (×) denotes the mean value.

left to future work.

4.6.2 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. 4.15). 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.

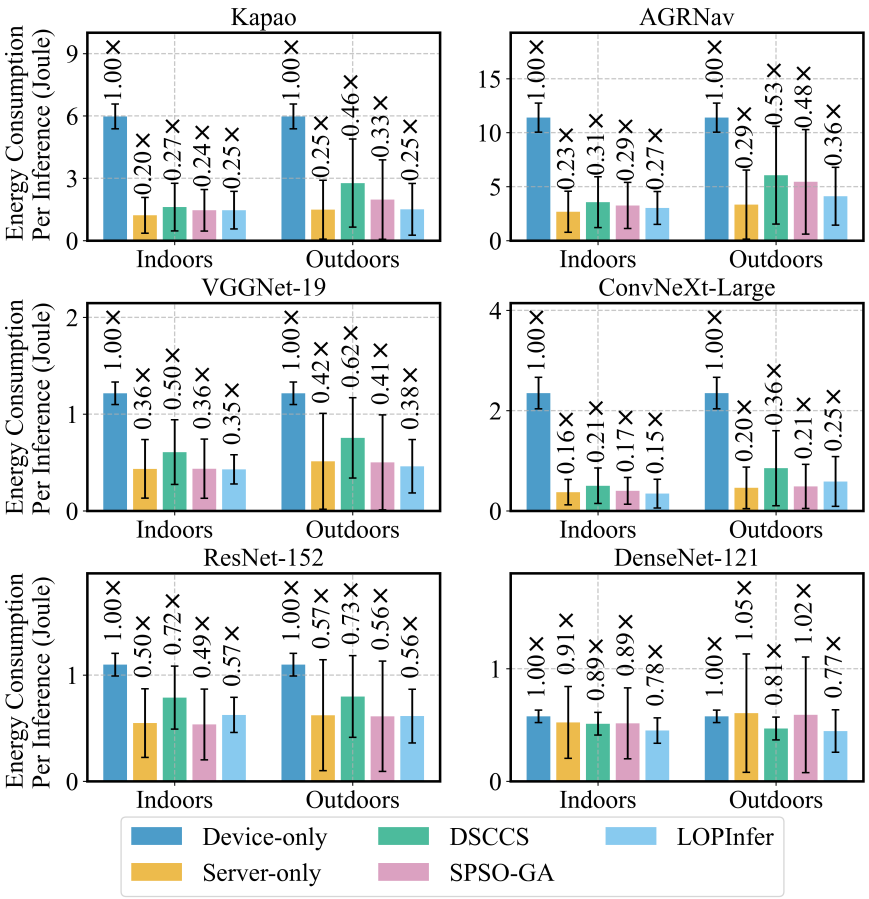


Figure 4.14: Energy consumption per inference request for different models across various environments and systems. The cross marker (×) denotes the mean value.

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. 4.16 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 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. 4.15), 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

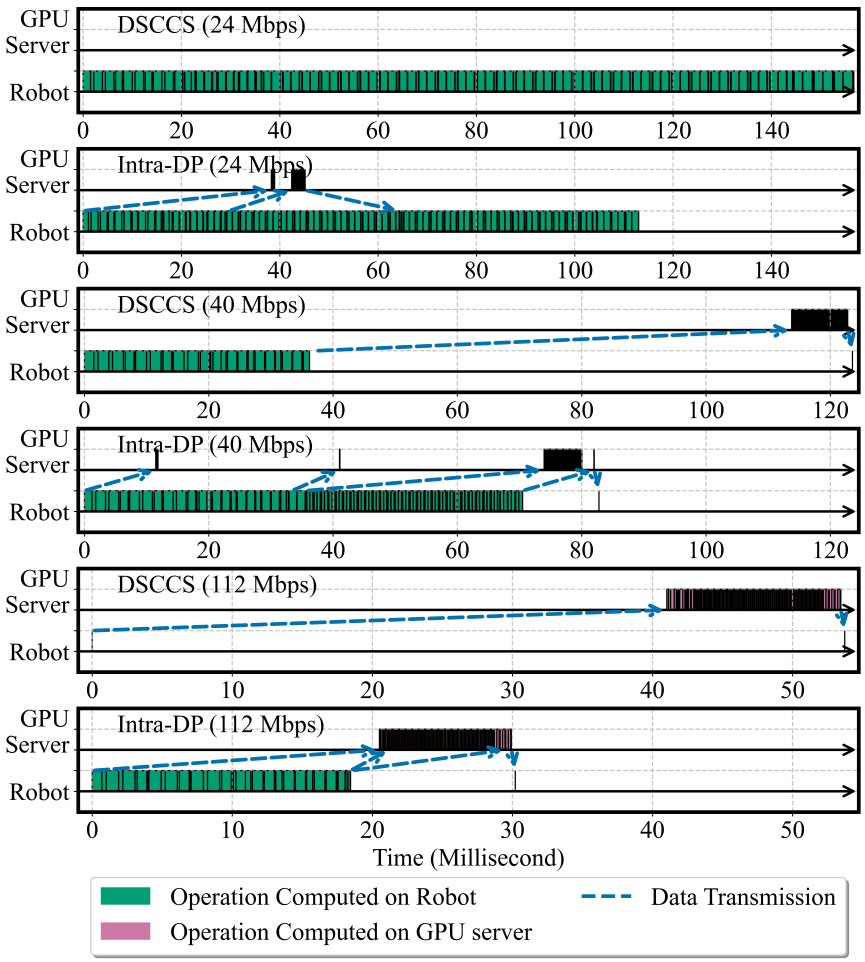


Figure 4.15: Snapshots of schedule plan of LOPInfer and baseline during runtime under various network bandwidth.

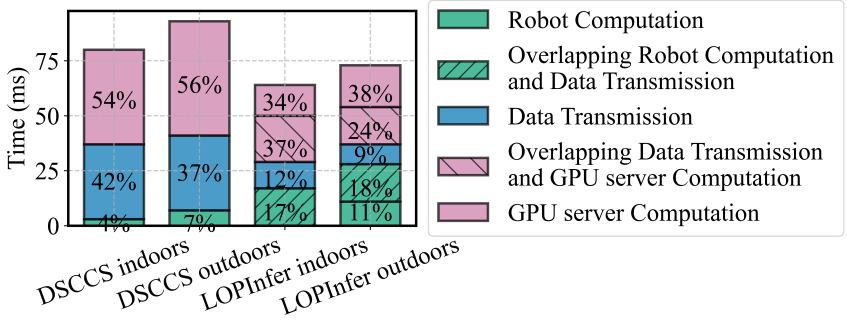


Figure 4.16: Breakdown of each phase of the inference process.

under the fixed bandwidth; Table 4.3 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. 4.4.3. 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 4.3: 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

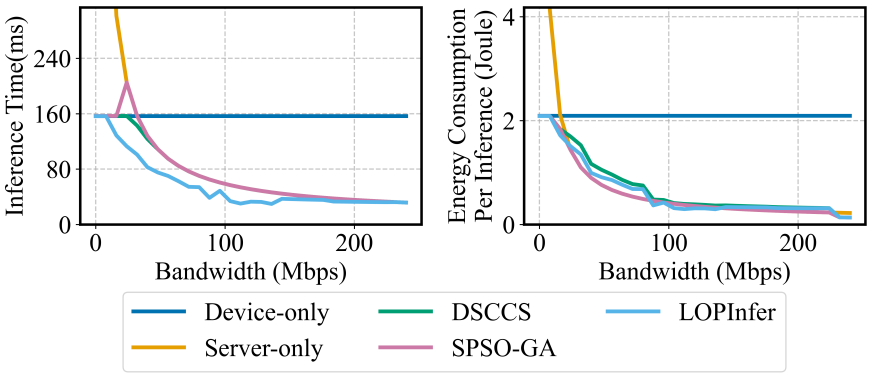


Figure 4.17: Performance comparison of LOPInfer and baselines under different network bandwidth conditions.

4.6.3 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. 4.17, 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. 4.12 and 4.14 are smaller than in Fig. 4.17 because runtime bandwidth estimation introduces noise and temporal mismatch that can occasionally select suboptimal buckets.

Model Structure. As shown in Fig. 4.18, 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.

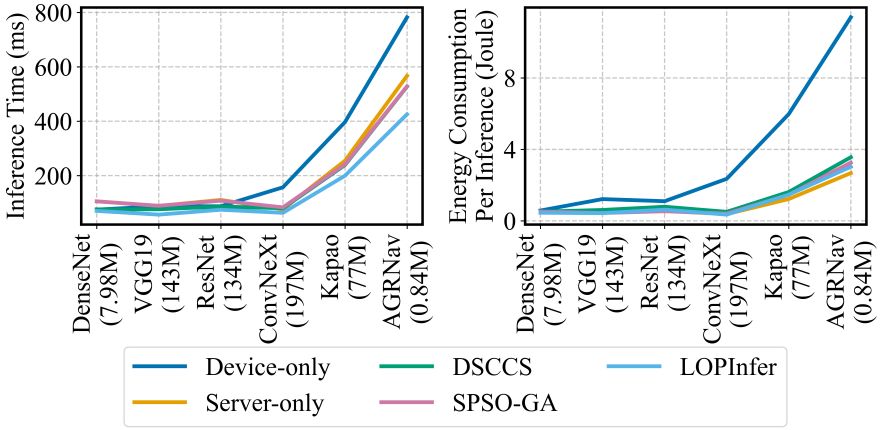


Figure 4.18: Performance comparison of LOPInfer and baselines with varying model structures.

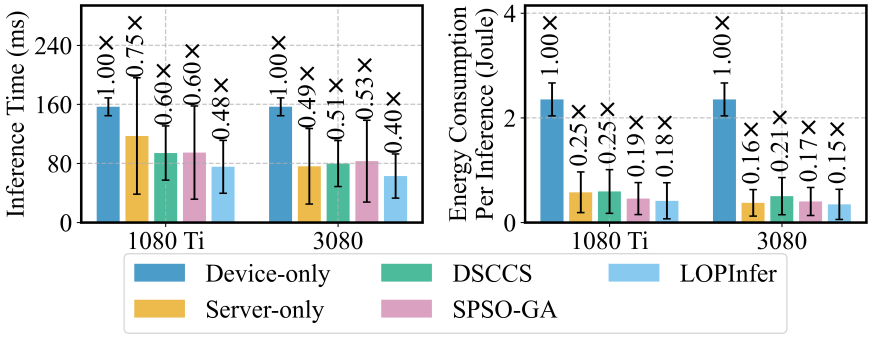


Figure 4.19: Performance comparison of LOPInfer and baselines on GPU servers with varying computational power.

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. 4.19). 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. 4.2(a)) is far more capable than typical smartphones, LOPInfer should yield even greater benefits on more resource-constrained mobile devices.

4.7 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 [93, 37, 122]. 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 [103], increasing the relative benefit of LOPInfer. Finally, consistent with our sensitivity analysis (Sec. 4.6.3), LOPInfer's gains grow with model compute intensity and server capability, yielding larger improvements over baselines.

Inference Request Scheduling. Prior work [137, 141, 15] 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 [154]. Hybrid offloading [95] 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 [71] 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 [30]. In particular, in vision transformers, patch-wise computations (e.g., patch embedding/projection) are also block-local operations under our definition.

4.8 Conclusion

This chapter has 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 compared with baselines, showing that local-operation granularity is an effective execution unit for real-time MEC inference.

Chapter 5

RRTO: Record/Replay Transparent Offloading for MEC Inference

CHAPTER 4 improved per-request inference by changing the scheduling unit inside a DNN. This chapter studies the third lifecycle requirement of MEC intelligence: practical deployment for existing applications. At this layer, the *granularity mismatch* is between per-operator RPC control and the repeated operator-sequence structure of ML inference. MEC acceleration is useful only if it can be adopted by applications, frameworks, and runtime libraries without extensive rewriting. Non-transparent offloading can achieve good performance because it exposes model-level structure, but it requires intrusive changes and may not support closed-source components, just-in-time compiled models, or dynamically selected kernels. Transparent offloading avoids these changes by intercepting low-level runtime calls, but it exposes a per-operator remote-control unit: a single inference can trigger thousands of RPCs (e.g., 5,895 for 522 operators), each paying a wireless round trip. Under MEC bandwidth and RTT conditions, this accumulated control cost dominates latency and energy and can slow transparent inference down by up to 95% relative to non-transparent baselines.

RRTO resolves this mismatch by changing transparent control from per-operator RPCs to per-inference operator-sequence replay. This is the deployment-layer instance of *granularity-aware execution*: the system chooses a control unit that matches the repeated structure of inference while preserving the compatibility of transparent offloading. RRTO observes that ML inference follows a stable operator sequence across requests, so reactive per-operator forwarding is unnecessary after warm-up. It records low-level runtime behavior, reconstructs the steady-state sequence through a two-stage Operator Sequence Search that reduces search complexity from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$, and replays each subsequent inference as one proactive remote execution. The system shifts communication complexity from linear $\mathcal{O}(N)$ (one RPC per operator) to constant $\mathcal{O}(1)$ (one RPC per inference) without requiring source-code modification. RRTO is released at `https://github.com/hku-systems/RRTO`.

The remainder of this chapter reuses the original paper text. Section 5.1 motivates RRTO in the context of MEC deployment and summarizes its contributions. Section 5.2 describes device-only inference, non-transparent offloading, transparent offloading, and the RPC-optimization limitations that make per-operator forwarding ineffective over wireless MEC links. Section 5.3 presents the design of RRTO, including the architecture, the static-operator-sequence observation, and the two-stage Operator Sequence Search. Sections 5.4 and 5.5 describe the implementation and evaluation on real robots under indoor and outdoor MEC networks. Section 5.6 discusses related work and limitations, and Section 5.7 summarizes the chapter.

5.1 Introduction

This section instantiates the thesis-level *granularity mismatch* at the deployment layer. MEC offloading must remain compatible with existing applications and frameworks, but transparent offloading exposes a very fine per-operator RPC unit that amplifies wireless round trips. The core question is therefore how to preserve transparency while raising the control granularity to the level of a complete inference.

Machine learning (ML) has become fundamental to a wide range of mobile applications, from intelligent wearable sensors [89] and autonomous vehicles [126] to industrial IoT systems [171]. These applications are built upon advances in object detection [96], robotic control [153], and environmental perception [16], all of which require high-performance (low-latency and energy-efficient) inference. Deploying ML models on real-world mobile devices (e.g., smartphones, robots, and IoT devices) faces two main challenges: platform compatibility, which requires supporting diverse software frameworks and hardware accelerators on mobile devices; and limited on-device computing resources, including restricted computational power and battery life.

Recent studies [2] have proposed mobile edge computing (MEC) as a promising solution for high-performance inference by leveraging GPU-equipped edge servers (GPU servers). Conventional MEC approaches (illustrated in Tab. 5.1) fall into three categories: device-only inference, non-transparent offloading, and transparent offloading, based on whether source code modification is required for enabling offloading. Device-only inference demands significant engineering effort due to poor platform compatibility and cannot deliver high performance because of limited on-device resources (see Sec. 5.2.1). As a result, many ML applications are turning to offloading-based inference over MEC networks, which offers high performance and broad compatibility by leveraging GPU servers.

Non-transparent offloading strategies enhance model inference performance by relocating model computations to GPU servers by modifying applications' source code. For instance, our experiments demonstrate that native offloading, where the entire

Method	Category	Code Modification	Application Diversity	Platform Compatibility	High Performance
Device-only Inference	Device-only	N/A	✓	×	×
Native Offloading	Non-transparent	High	×	✓	✓
CUDA Graph [117]	Non-transparent	Low	×	×	✓
Cricket [39]	Transparent	N/A	✓	✓	×
RRTO (Ours)	Transparent	N/A	✓	✓	✓

Table 5.1: Comparison of representative inference methods in MEC.

model is hosted remotely, achieves up to 3.7× faster inference and 49% lower energy consumption compared than device-only inference. However, this inherent requirement for source code modification poses a critical limitation, which not only significantly increases engineering overhead but also severely curtails application diversity (i.e., support various upper-layer applications). Specifically, it impedes integration with closed-source software (e.g., TensorRT [32] and CUDA-X libraries [168]) and runtime-optimized model structures (e.g., Just-In-Time compilation [99]) where code modifications are often infeasible or prohibitive. While alternatives like CUDA Graph [117] and TorchScript [33] have attempted to mitigate the extent of these modifications via model-level packaging, they remain intrinsically non-transparent and suffer from limited platform compatibility due to their framework-specific nature (see Sec. 5.2.2).

Transparent offloading methods [39] offload model inference to GPU servers without requiring any code modification to applications or frameworks. At runtime, ML models issue a sequence of operator invocations (e.g., `torch.nn.functional.add()` and `torch.nn.functional.conv2d()` in PyTorch [112]), which are typically forwarded to backend system functions (e.g., `aten::add` and `aten::conv2d` within CUDA's `cudaLaunchKernel` [65]). Transparent offloading intercepts these calls from upper-layer applications at the system layer and redirects their execution to GPU servers via Remote Procedure Calls (RPCs), thereby avoiding source-code changes (see Sec. 5.2.3).

However, existing transparent offloading methods face two main challenges in MEC. First, they perform poorly over the low and highly variable bandwidth of mobile wireless links (Sec. 5.2.3) because operator-level RPCs serialize communication and make the accumulated round-trip time (RTT) the dominant cost. ML models typically consist of hundreds of operators (e.g., 522 in [96]), which in turn yield thousands of RPC invocations per inference (e.g., 5,895 in [96]), since each operator triggers one or more separate RPCs; on wireless networks used by mobile devices, each RTT is typically on the order of several milliseconds [128], so communication delay from one-by-one handling can dominate end-to-end inference time (up to 95% in [39]). Thus, the key performance lever in MEC is not merely faster links but fewer round trips: per-operator RPCs should be replaced by coarse-grained, batched, or persistent execution to amortize RTTs.

Second, realizing coarse-grained execution in a transparent manner hinges on reconstructing stable, task-level operator sequences from black-box runtime traces: transparent offloading has access only to low-level operator logs with no application-level hints, leading to three sub-challenges:

(i) demultiplexing, as attributing each operator invocation to its originating inference task is non-trivial;

(ii) non-stationarity, since operator RPCs during model loading and the initial inference differ from the steady-state inference loop, where even small sequence mismatches can violate correctness;

(iii) scale, because traces with tens of thousands of entries induce a large combinatorial search space that impedes timely sequence recovery.

Absent reliable sequence reconstruction, systems cannot safely apply batching, fusion, or persistent execution, and thus often revert to per-operator RPCs, reproducing the performance degradation observed above under low and variable bandwidth.

To tackle the above challenges, we propose **RRTO**, a **T**ransparent **O**ffloading system optimized for model inference in MEC with a novel **R**ecord/**R**eplay mechanism: RRTO records operator invocations during the first inferences using traditional RPCs as in traditional transparent offloading systems to reconstruct a fixed-order task-level operator sequence; Once stabilized, subsequent inferences are executed by replaying this operator sequence on GPU server via a single per-inference control RPC, which eliminates per-operator communication. First, to sustain high performance in MEC networks, RRTO replaces reactive per-operator RPCs with proactive one-shot control enabled by record/replay. Existing transparent systems issue RPCs only after operators are invoked, which is a reactive byproduct of general-purpose remote GPU usage where application-level operator patterns are unpredictable. In contrast, ML inference typically follows a static computation graph with a fixed operator order (Sec. 5.3.3). By anticipating the operator sequence through tracing and replaying it on the server, RRTO amortizes RTT to one round trip per inference via a proactive approach, substantially reducing communication delay and approaching the performance of non-transparent offloading while preserving application transparency.

Second, to recover the operator sequence from black-box runtime logs, RRTO introduces a two-stage *Operator Sequence Search* that first identifies candidate sequences and then verifies inter-operator data dependencies to reconstruct the steady-state operator sequence solely from raw logs. This search procedure

(i) exploits the fact that the target sequence is repeatedly embedded in the complete log across multiple inferences, which establishes task-level attribution for each operator;

(ii) checks completeness by aligning the repeated sequence with the full log while tolerating transient inconsistencies due to model loading or initialization; and

(iii) adopts a two-stage matching strategy that efficiently reduces the search space and accelerates sequence identification.

The key contributions of this chapter are summarized as follows:

- To the best of our knowledge, RRTO is the first high-performance transparent offloading system tailored for model inference in MEC. The code is released at `https://github.com/hku-systems/RRTO`.
- We design a proactive record/replay mechanism that converts reactive per-operator RPCs into a one-shot control per inference. This design shifts the communication complexity from linear $\mathcal{O}(N)$ (per-operator RPCs) to constant $\mathcal{O}(1)$ (one-shot replay), thereby eliminating per-operator communication and substantially reducing transmission delay.
- We cast transparent MEC offloading as a zero-hint black-box logic reconstruction problem and develop a two-stage Operator Sequence Search. The search algorithm recovers the operator sequence solely from raw runtime logs while tolerating initialization variability, and reduces search complexity from a brute-force $\mathcal{O}(L^2)$ to linear time $\mathcal{O}(L)$.
- We empirically evaluate RRTO with extensive experiments. The results demonstrate that RRTO outperforms state-of-the-art baselines in MEC without requiring any source code modifications.

The rest of the paper is organized as follows. Sec. 5.2 motivates the design of RRTO by revealing the challenges in current MEC networks. Sec. 5.3 presents the system design of RRTO. Sec. 5.4 introduces the system implementation, followed by performance evaluation in Sec. 5.5. Related works and technical limitations are discussed in Sec. 5.6. Finally, conclusions are presented in Sec. 5.7.

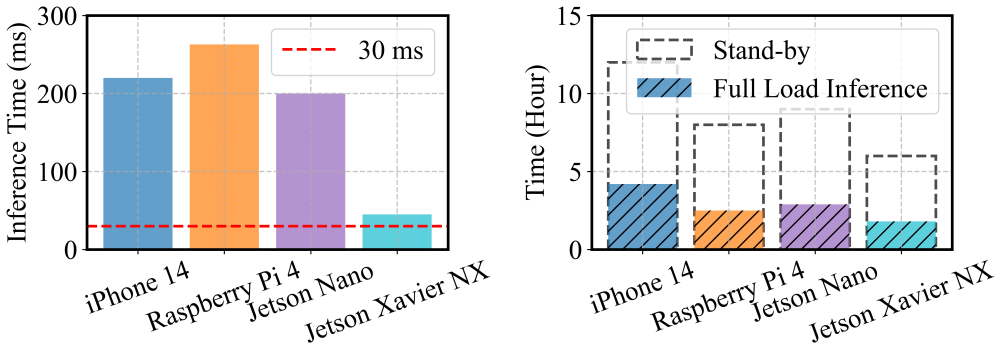
5.2 Background

5.2.1 Device-only Inference

Device-only inference, which runs models directly on mobile devices, faces two major limitations: poor platform compatibility and restricted performance. Mobile applications are built on diverse software frameworks (e.g., PyTorch [112], TensorFlow [1], CUDA [65], and OpenCL [100]) and mobile devices are equipped with various hardware accelerators (e.g., GPU [61], FPGA [114], and SoC [70]), making deployment on real-world devices labor-intensive and error-prone. Engineers must adapt each model to specific hardware and software configurations, sacrificing portability and increasing cost.

Moreover, the performance of device-only inference is constrained by the limited computational power and battery capacity. As shown in Fig. 5.1(a), the inference latency on various mobile devices exceeds the 30 ms threshold required for smooth video

fluency [46] (indicated by the red dotted line). Fig. 5.1(b) shows that frequent inference reduces device standby time to 20–40% of their normal duration, degrading user expe-



(a) Inference Time **(b) Battery Life**

Figure 5.1: The performance of VGG-16 under device-only inference across different mobile devices [97, 120, 106, 102].

5.2.2 Non-Transparent Offloading

Non-transparent offloading strategies relocate model computations to GPU servers by mandating application source code modifications, increasing engineering complexity and restricting application diversity. This non-transparency severely limits their use, particularly with closed-source software and runtime-adaptive model structures where such code alterations are often infeasible. High-performance libraries (e.g., TensorRT [32] and CUDA-X library [168]) are widely adopted in mobile applications but do not expose source code, making them incompatible with non-transparent approaches. Many real-world applications also employ runtime optimization techniques, including Just-In-Time compilation [99] and dynamic conventional kernel selection with different numbers/sizes based on task-specific requirements [5]. These adaptive methods change model structure at runtime to improve speed and accuracy, and non-transparent offloading cannot accommodate such closed-source or dynamic environments. In contrast, transparent offloading intercepts system calls during runtime, enabling support for both closed-source libraries and runtime-adaptive model structures without requiring code modification.

Alternative methods, such as CUDA Graph [117] and TorchScript [33], reduce the amount of required code modification by packaging models for GPU-server deployment. However, these approaches are still non-transparent and cannot support applications involving runtime variability or proprietary libraries. Further, they reduce platform compatibility by depending on specific software frameworks, which imposes additional constraints on mobile software development beyond those already faced by device-only deployment.

To further optimize latency and energy efficiency, non-transparent offloading systems incorporate fine-grained scheduling strategies at various inference stages (e.g., layer partitioning [64] and multiple inference scheduling [81, 43, 171]), unlike native

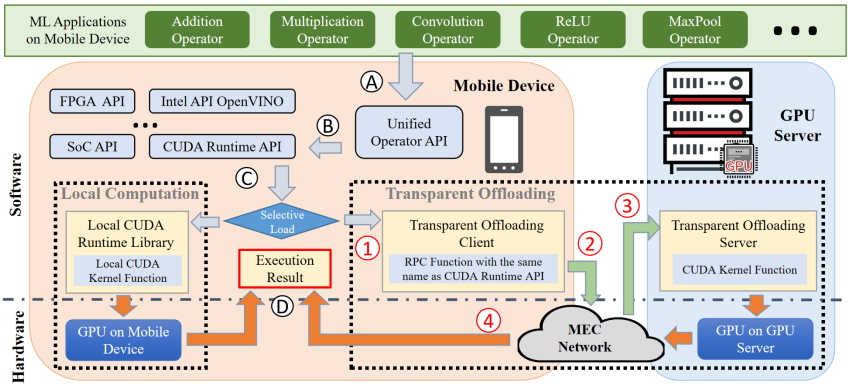


Figure 5.2: Workflow of Transparent Offloading System for Model Inference in MEC.

offloading that relocates the entire model to GPU servers. The proven effectiveness of these scheduling optimizations within non-transparent frameworks forms a basis for their adaptation to RRTO, with the goal of further enhancing its performance (as detailed in Sec. 5.6).

5.2.3 Transparent Offloading

Existing framework

When a mobile application utilizes the GPU on the mobile device for inference, the system call flow for an individual operator is as follows (illustrated in the left part of Fig. 5.2):

A The ML application sequentially invokes the corresponding operators based on the model's structure to complete the computation process.

B Each operator accesses the appropriate function library through a unified operator API tailored to the device; for instance, on NVIDIA GPUs, this is the CUDA runtime library [65].

C The operating system loads the local CUDA runtime library by default and executes the relevant CUDA kernel functions on the device's GPU.

D The local CUDA runtime library returns execution results (e.g., 'cudaSuccess' from "cudaLaunchKernel" and computation results from "cudaMemcpyDtoH") to the upper-layer application.

Transparent offloading methods [39] typically rewrite dynamic link libraries by defining functions that share names with the CUDA runtime API and prioritizing their loading via the *LD_PRELOAD* environment variable. This causes the dynamic linker to redirect calls from the original library functions to the custom ones, effectively intercepting them. The custom library then packages function calls with their parameters and data, sends them to the GPU server via RPC, and modifies GPU memory management and kernel launching to ensure correct server-side execution. By this means, these

methods achieve transparent offloading by intercepting kernel functions one-by-one at the system layer and offloading their execution to the GPU server.

Compared to using the GPU on the mobile device, changes primarily occur in step C. The transparent offloading process (depicted in the right part of Fig. 5.2) proceeds as follows:

(1) The modified dynamic link library ensures that each operator prioritizes calling the RPC functions that share names with the CUDA runtime API, intercepting all CUDA kernel function calls.

(2) The transparent offloading client transmits the called CUDA runtime API and required parameters to the GPU server via the MEC network using RPC.

(3) The transparent offloading server launches the corresponding CUDA kernel functions and completes the computation.

(4) The execution results are returned to the client, which then returns them to the upper-layer function calls.

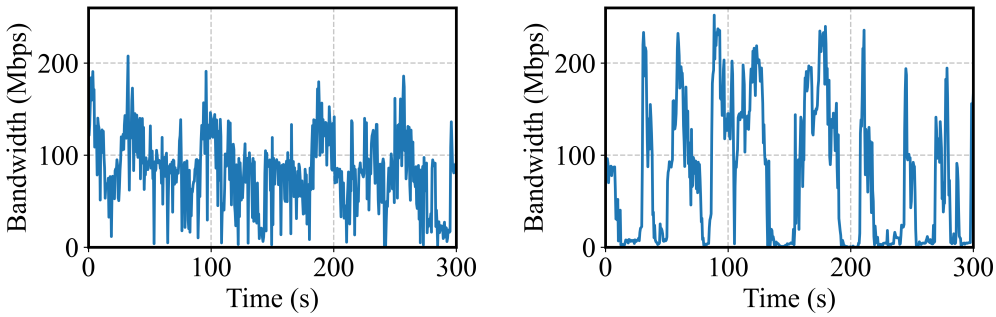
Runtime homogeneity is a fundamental assumption in interception-based transparent offloading: to forward an intercepted client call, the system must map it one-to-one to an equivalent server-side runtime API (Fig. 5.2, step B). When the client and server expose different frameworks (e.g., OpenCL vs. CUDA) or the client relies on a CPU library without kernel-launch semantics, such a correspondence is absent. Consequently, existing interception-based transparent offloading systems, including RRTO, require both sides to export the same GPU runtime (e.g., CUDA on both), and, to the best of our knowledge, do not support cross-framework offloading.

Challenges in MEC Networks

In real-world scenarios, mobile devices primarily rely on wireless networks, offering high mobility but limited bandwidth compared to data center networks (e.g., 200–800 Gbps for InfiniBand [105]).

The bandwidth capacity of wireless networks is constrained by both theoretical limits and practical factors in MEC. While Wi-Fi 6 can achieve a peak bandwidth of 1.2 Gbps per stream [84], mobile devices lack the necessary hardware to fully leverage this capacity [166]. The actual available bandwidth varies significantly due to factors such as device mobility [93], signal obstruction [37], and channel contention [122].

To examine wireless instability in MEC scenarios, we conducted a robot surveillance experiment where four-wheeled robots navigated through a lab (indoors) and a campus garden (outdoors) at speeds of 5–40 cm/s. Using iperf [57], we measured real-time wireless bandwidth capacity between the robot and a base station over TCP [146] at 0.1-second intervals for five minutes. As shown in Fig. 5.3, the average bandwidth was 93 Mbps indoors and 73 Mbps outdoors, with outdoor measurements exhibiting higher fluctuations and occasional near-zero drops due to obstacles and reduced signal



(a) Indoors **(b) Outdoors**

Figure 5.3: The wireless transmission instability of TCP between our robot and the base station in MEC networks.

reflections. Under these conditions, transparent offloading issues frequent per-operator RPCs that incur disproportionate protocol overhead and sensitivity to RTT and jitter, reducing throughput and inflating tail latency.

While the CPU–GPU asynchronous paradigm (CUDA kernels are enqueued at inference start, allowing the GPU to process them independently while the CPU awaits completion) works well locally, it faces severe challenges in MEC due to limited bandwidth, prolonging the RTT for each remote kernel launch via RPC. This prolonged RTT, often several milliseconds per RPC [128] (and compounded by potentially much longer transfer times for inference inputs/outputs depending on application requirements), starkly contrasts with mere microseconds needed for a local kernel launch command and tens of microseconds to milliseconds for its execution on a GPU server [107]. Consequently, cumulative RPC overhead for individual kernel dispatches becomes a critical bottleneck in traditional transparent offloading, negating the benefits of the asynchronous paradigm in constrained network environments.

RPC Optimization

Remote Procedure Call (RPC) [35] is a fundamental communication protocol enabling processes to request services from remote computers over a network. Common strategies like Caching [134] (storing previous RPC results), Batching [72] (aggregating multiple RPCs into one request), and Asynchronous RPC [42] (non-blocking execution) prove largely ineffective for reducing the communication costs of transparent offloading during model inference. Specifically, Caching fails because each inference typically processes unique input, necessitating fresh operator computations. Batching reduces the number of network requests via aggregation but does not eliminate per-operator RPCs and, more importantly, must rely on latency-inducing timeouts for batch formation, since the total number of operations is unknown *a priori*, a constraint that our operator search algorithm overcomes. Furthermore, Asynchronous RPC compromises the correctness of inference results because it cannot guarantee sequential execution of

GPU operations on the server. This sequential integrity is vital not only for launching CUDA kernels ("cudaLaunchKernel") in the correct order but for "cudaMemcpyHtoD" and "cudaMemcpyDtoH" operations to manage input and output data accurately. These memory transfer operations, often involving larger data volumes and thus longer transmission times than kernel launches, are particularly prone to misordering under asynchronous execution, leading to corrupted data. In contrast, RRTO significantly reduces communication costs by eliminating most operator-level RPCs, representing a specialized co-design of RPC optimization and transparent offloading tailored for model inference (see Sec. 5.3).

5.3 System Design

5.3.1 Problem Formulation

We model a single inference of an ML application as an ordered operator sequence $\mathcal{S} = (op_1, \ldots, op_N)$, where $N = |\mathcal{S}|$ denotes the number of operators. For each op_i, let $\text{Input}(op_i)$ and $\text{Output}(op_i)$ denote the sets of input and output tensors, respectively (with associated buffer pointers and sizes).

Latency in traditional transparent offloading. Conventional transparent offloading (e.g., Cricket) remotely launches each operator via an RPC. Let $T_{\text{comp}}^{(i)}$ be the server-side compute time of op_i, $T_{\text{trans}}^{(i)}$ the data transfer time for its arguments, and T_{RTT} the network round-trip time per RPC. The total inference latency is

$$T_{\text{trad}} = \sum_{i=1}^{N} \left(T_{\text{comp}}^{(i)} + T_{\text{trans}}^{(i)} + T_{\text{RTT}} \right). \tag{5.1}$$

In MEC networks, T_{RTT} is on the order of milliseconds and can be volatile (Sec. 5.2.3); consequently, the $N \times T_{\text{RTT}}$ term often dominates when N is large.

RRTO objective. RRTO eliminates cumulative per-operator round-trip delays by constructing a verifiable, dependency-preserving execution sequence $\mathcal{S}^\star$ that the server replays within a single RPC transaction. Under identical server hardware, the aggregate computation time remains $\sum_{i=1}^{N} T_{\text{comp}}^{(i)}$, so the end-to-end latency becomes

$$T_{\text{RRTO}} = \sum_{i=1}^{N} T_{\text{comp}}^{(i)} + T_{\text{input}} + T_{\text{output}} + T_{\text{RTT}}, \tag{5.2}$$

where T_{input} and T_{output} are the one-shot transmission times for the end-to-end input and output; and T_{RTT} is the single RTT for this replay RPC. Consequently, the communication complexity (the number of RPC invocations) reduces from $\mathcal{O}(N)$ (per-operator RPCs) to $\mathcal{O}(1)$ (one-shot replay), which removes per-operator control-plane exchanges and shortens the end-to-end transmission delay.

Constraint (Data Dependency). The replay sequence must preserve data dependencies:

$$\forall\, op_i \in \mathcal{S}^\star, \quad \text{Input}(op_i) \;\subseteq\; \Big(\bigcup_{j<i} \text{Output}(op_j)\Big) \;\cup\; \text{Input}_{\text{global}}. \tag{5.3}$$

Here, $\text{Input}_{\text{global}}$ collects tensors available prior to replay, including the end-to-end inference input and resident model parameters. This condition ensures that replaying $\mathcal{S}^\star$ is mathematically equivalent to the original per-operator execution.

Problem statement. Given operator trace logs from the first few inferences, find a sequence $\mathcal{S}^\star$ that minimizes T_{RRTO} in (5.2), subject to the data dependency constraint (5.3) and the framework/runtime semantics (e.g., determinism and side-effect freedom).

5.3.2 System Overview

This section presents RRTO, a high-performance transparent offloading system for model inference in MEC, featuring a novel record/replay mechanism: it records the operators invoked during initial runs and replays a fixed-order sequence in subsequent inferences. RRTO addresses two challenges: performance degradation under MEC's low/fluctuating bandwidth and reconstruction of black-box inference logic. To mitigate network bottlenecks, RRTO replaces reactive per-operator RPCs with a proactive per-inference control RPC, eliminating operator-level communication and reducing cumulative RTT and energy consumption while preserving transparency. From raw logs, it recovers the steady-state sequence via a two-stage search that

(i) leverages repeated cross-inference embeddings for task-level association,

(ii) enforces data-dependency constraints while tolerating initialization variability, and

(iii) prunes the search space through staged matching.

Fig. 5.4 summarizes the architecture and the integration of record/replay into the workflow of traditional transparent offloading system (Fig. 5.2); no application source-code changes are required. Client and server workflows are detailed in Sec. 5.3.4.

Within the context of a single inference application, RRTO employs its record/replay mechanism to differentiate operators belonging to distinct inference tasks. For the first few inference executions, RRTO enters the recording phase (①). In this mode, each operator's execution is individually offloaded to the GPU server via RPCs, following the execution pattern of traditional transparent offloading systems (green lines in Fig. 5.4). When RRTO intercepts CUDA kernel function calls originating from upper-layer ML applications, its recorder component logs these invoked functions, including their parameters and return values. Concurrently, it performs an operator sequence search to precisely identify the sequence of inference operators (see Sec. 5.3.3).

Once the recorder successfully identifies the complete inference operator sequence, RRTO transitions to the replaying phase (②). During this phase, subsequent inferences

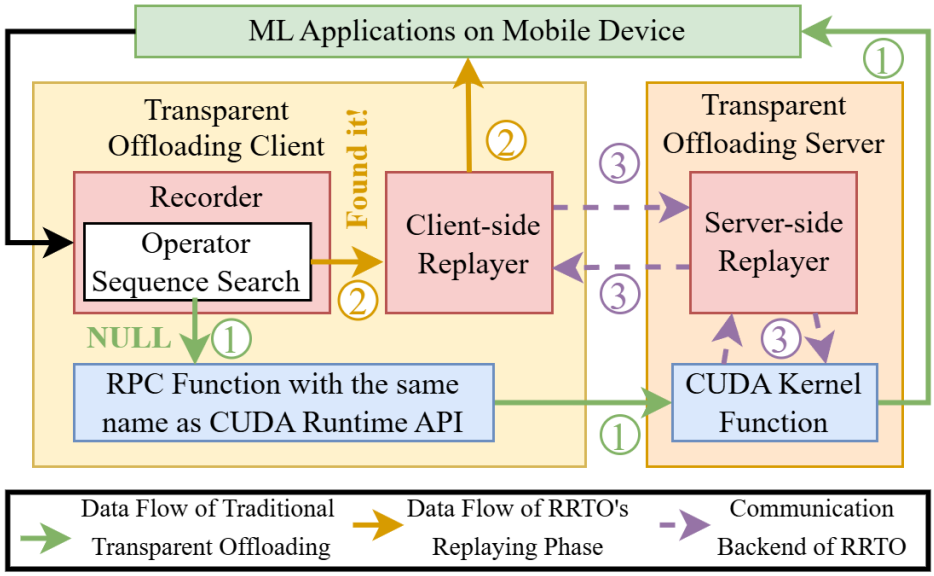


Figure 5.4: Architecture of RRTO, with key components highlighted in red boxes.

are executed by replaying the pre-identified operator sequence through replayer components deployed on both the client and server (orange lines in Fig. 5.4). The client-side replayer ensures both transparency and high performance: it furnishes upper-layer applications with execution results from previously recorded RPC calls (mainly 'cudaSuccess' from "cudaLaunchKernel"), a mechanism analogous to RPC caching (see Sec. 5.2.3). This enables the offloading client to seamlessly invoke system functions for subsequent operators, creating the illusion of local execution, until reaching the end operator, at which point it awaits the actual inference output from the server-side replayer. Concurrently with this primary replay operation, the client-side replayer monitors for any deviations or failures in the predicted operator sequence and adeptly detects the initiation of new inference tasks. In this way, RRTO maintains transparency and system integrity while eliminating per-operator RPC overhead during replay.

Simultaneously, the server-side replayer efficiently performs the operator computations on the GPU server (③). It replays the identified sequence and transmits only the final inference result back to the client. This enables RRTO to achieve communication overhead closely approximating that of non-transparent offloading systems: it transmits only the raw input and final output, circumventing per-operator RPC communication during the replaying phase (purple dotted lines in Fig. 5.4).

When multiple ML applications run concurrently on a mobile device, RRTO inherits the task-level isolation that existing transparent offloading systems already maintain for correct remote GPU service. Because such systems must preserve application boundaries, RRTO naturally records and replays each application within its own RPC context, preventing operators from different applications from being mixed into one replay sequence. Our current design assumes one steady operator stream per application and does not target arbitrary intra-application interleavings of multiple models, which

are uncommon in our target mobile workloads. Extending RRTO to distinguish multiple model pipelines within one shared application is promising future work, potentially leveraging the same provenance information used for single-pipeline validation to separate coexisting pipelines via dependency-based demultiplexing (see Sec. 5.6).

5.3.3 Recording Phase Design

Targeted Models

Static/Dynamic Activation Models. We classify ML models into static-activation models (SAMs) and dynamic-activation models (DAMs) according to whether the sequence of activated computational operators and the associated dataflow remain input-invariant within a single inference.

SAMs execute a predetermined, input-invariant operator sequence; for any input, the operations and their order are fixed. Representative SAMs include:

(i) MLPs and CNNs [96], whose layers perform fixed operations (e.g., matrix multiplications, convolutions) independent of input values;

(ii) traditional ML models (e.g., Linear Regression [55], SVMs [151], KNN [73]), whose inference applies a fixed set of computations or comparisons predetermined before runtime; and

(iii) transformer encoders [34], RNNs on fixed-length sequences [7], and full Transformers on fixed-length tasks [38], where uniform padding/truncation fixes the number of unrollings or activated self-attention blocks.

Their fixed structure yields regular, predictable computation, making SAMs the primary target for RRTO and the offloading systems in this work.

DAMs adapt their computational path or the number of activated operators to input characteristics. Examples include:

(i) autoregressive Transformers for generation [14], where the number of decoding steps depends on the generated length;

(ii) Mixture-of-Experts (MoE) models [130], where a gating mechanism activates an input-dependent sparse subset of experts;

(iii) RNNs on variable-length sequences without padding [139], where the number of cell activations follows sequence length;

(iv) decision trees and ensembles [21], whose root-to-leaf comparison sequence is input-specific; and

(v) GNNs [62] operating on graphs with varying sizes/structures or input-dependent neighborhood sampling, which changes the number of processed nodes, effective depth, and aggregation scope.

While DAMs improve adaptability and expressiveness, their input-dependent execution complicates runtime profiling (e.g., latency and I/O sizes), and fewer offloading systems are tailored to them.

Dynamic Kernel Selection. In practice, library-level kernel planning may vary across runs. We distinguish two cases by whether the runtime-API call sequence changes within a single inference (we consider the operator sequence at the runtime-API level, e.g., HtoD/DtoH/cudaLaunchKernel):

(i) No intra-inference change: during any single inference pass, each operator's kernel choice is fixed (though the chosen kernels may differ across operators or across different inferences due to autotuning), and the resulting runtime-call sequence is unchanged; we classify such models as SAMs.

(ii) Intra-inference change: if kernel planning alters the order or multiplicity of runtime-API calls during inference (e.g., enabling/disabling fusion, splitting a layer), the operator sequence becomes input-dependent; we classify the model as a DAM.

To conclude, SAMs (e.g., MLPs and CNNs for computer vision [96]) are widely adopted in mobile applications. These applications require high-performance inference (low latency and high energy efficiency) on resource-constrained mobile devices, making MEC-based offloading essential for achieving such performance. In contrast, DAMs, which demand significant computing resources and have less stringent real-time inference requirements [14, 130], are better suited to data-center deployment.

Accordingly, RRTO employs a record/replay mechanism optimized for the consistent operator sequences of SAMs. When RRTO encounters a DAM whose operator sequence changes, its replayer component detects this inconsistency. If an operator outside the core model pipeline is nevertheless invoked on every inference, it becomes part of the steady-state sequence and will be learned together with the rest of the pipeline. By contrast, if such an operator appears only occasionally in an ad hoc manner, the operator sequence is no longer fixed across inferences. This is no longer the SAM setting targeted by RRTO. Accordingly, RRTO treats such executions as outside its target scope: during the search phase, the occasional extra operator prevents the sequence from repeating identically over R consecutive inferences, so FULLCHECK will not accept an incorrect candidate; during the replay phase, the step-wise consistency check (MATCHREPLAYSTEP) detects the unexpected operator, aborts replay immediately, falls back to a standard transparent offloading workflow, and reruns the operator-sequence search only after enough new evidence has been collected. Given the predominance of SAMs in mobile applications, these fallback events are expected to be infrequent, thereby preserving the benefits of RRTO. Optimizing offloading for DAMs with varying operator sequences remains an open challenge for offloading systems in general; while some work predicts future layer activations [143], such techniques are beyond this chapter's scope. **I/O-Bound Models.** A core design restriction in RRTO is full

transparency: it operates without application- or framework-provided hints. Instrumenting frameworks (e.g., PyTorch) with explicit markers appears simpler but perpetuates the same platform compatibility issues that already plague device-only deployment and does not address heterogeneous MEC environments. Hence, RRTO must autonomously recover, from black-box runtime logs, the exact operator sequence that constitutes an inference. This hint-free design is technically harder but necessary for a universal, low-touch offloading solution.

RRTO's performance hinges on accurate sequence recovery: a single missing or spurious operator breaks end-to-end dataflow and yields incorrect outputs. Under realistic conditions, Operator Sequence Search must reconstruct stable, task-level sequences from low-level operator logs without application-level context, facing three sub-challenges:

(i) demultiplexing, since assigning each operator invocation to its originating inference task is non-trivial;

(ii) non-stationarity, because operator RPCs during model loading and warm-up differ from steady-state inference, so even minor mismatches violate correctness; and

(iii) scale, because traces with tens of thousands of events create a combinatorial search space that must be pruned for timely recovery.

Without reliable sequence reconstruction, systems cannot safely apply one-shot replay and must fall back to per-operator RPCs, reproducing the performance degradation under MEC's low and fluctuating bandwidth.

Operator Sequence Search

Key Observations. To address these sub-challenges, RRTO's Operator Sequence Search leverages three observations. ① For a SAM, each inference emits the same operator sequence, producing a regular trace in the logs. ② Real-time constraints in MEC enforce immediate inference execution; each invocation is bracketed by a host-to-device copy of the raw input ("cudaMemcpyHtoD", short for "HtoD") and a device-to-host copy of the result ("cudaMemcpyDtoH", short for "DtoH"). Treating these copies as special transfer operations and coalescing subsequent synchronizations (e.g., "cudaStreamSynchronize") yields reliable boundary markers. ③ A valid inference sequence must satisfy data dependencies: every operator's inputs originate from the raw input, a prior operator's output (same buffer address), or model parameters. Together, these observations provide regularity, boundaries, and correctness constraints for sequence recovery.

However, relying solely on observation ① via the maximum-repeated-substring algorithm [17] fails to segment the operator sequence correctly, because when the sequence repeats continuously, the algorithm merges multiple consecutive iterations into a single maximal substring (Fig. 5.5d). Similarly, using only observation ② becomes

ambiguous when multiple "HtoD" or "DtoH" events occur within a single inference, obscuring the true start or end (Fig. 5.5e).

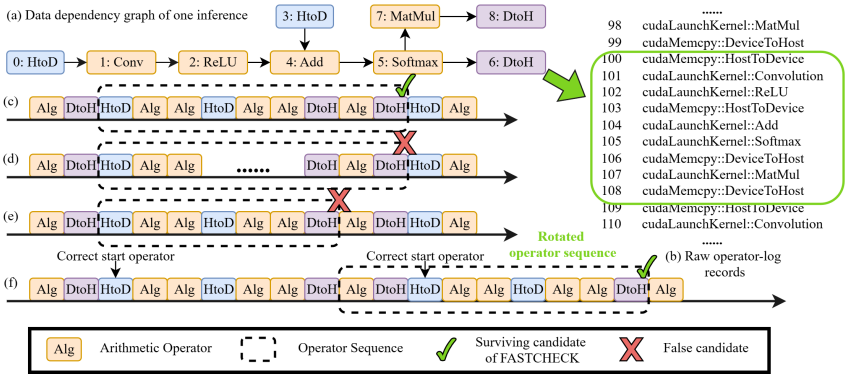


Figure 5.5: Illustration of Operator Sequence Search. The start of a sequence need not be an HtoD; our search also considers the first operator after any DtoH and realigns rotated sequences during FULLCHECK.

Algorithm 8: Operator Sequence Search

Input: `Logs`, a list of `OperatorInfo` entries collected from the first N inferences; R, the minimum repeat count
Output: inference operator sequence `IOS`, or `NULL`

function OperatorSequenceSearch(`Logs`, R)
  $S \leftarrow$ FINDINDICES(`Logs`, `'HtoD'`);
  $T \leftarrow$ FINDINDICES(`Logs`, `'DtoH'`);
  if $S = \emptyset$ **or** $T = \emptyset$ **then**
    return `NULL`;
  end if
  `Tags` $\leftarrow$ BUILDTAGS(`Logs`);
  $e \leftarrow \max(T)$;
  $C \leftarrow S \cup \{\, t+1 \mid t \in T \text{ and } t+1 \le e \,\}$;
  for each $s \in C$ **do**
    $\ell \leftarrow e - s$;
    if $\ell \le 0$ **then**
      continue;
    end if
    if FASTCHECK(`Tags`, s, ℓ, R) **then**
      $K \leftarrow \{\, k \in S \mid s - \ell \le k \le s \,\}$;
      for each $k \in K$ **do**
        if FULLCHECK(`Logs`, k, ℓ, R, T) **then**
          `IOS` $\leftarrow$ `Logs`$[k \ldots k+\ell-1]$;
          return `IOS`;
        end if
      end for
    end if
  end for
  return `NULL`;
end function

Algorithm 9: FastCheck

Input: `Tags`; start index s; candidate length ℓ; minimum repeat count R
Output: Boolean (whether the candidate passes FASTCHECK)

function FastCheck(`Tags`, s, ℓ, R)
  $cnt \leftarrow 0$;
  $p \leftarrow s$;
  while $p \geq 0$ **do**
  if SLICEEQ(`Tags`, s, p, ℓ) = `False` **then**
  break;
  end if
  $cnt \leftarrow cnt + 1$;
  $p \leftarrow p - \ell$;
  end while
  return $(cnt \geq R)$;
end function

Algorithm 10: FullCheck

Input: `Logs`; start index s; candidate length ℓ; minimum repeat count R; set T of `DtoH` indices
Output: Boolean (whether the candidate passes FULLCHECK)

function FullCheck(`Logs`, s, ℓ, R, T)
  $e \leftarrow s + \ell - 1$;
  if $e \notin T$ **then**
  return `False`;
  end if
  if CHECKDATADEPENDENCY(`Logs`, s, ℓ) = `False` **then**
  return `False`;
  end if
  $cnt \leftarrow 0$;
  $p \leftarrow s$;
  while $p \geq 0$ **do**
  $match \leftarrow$ `True`;
  for $t \leftarrow 0$ to $\ell - 1$ **do**
  if ENTRYEQ(`Logs`$[s + t]$, `Logs`$[p + t]$) = `False` **then**
  $match \leftarrow$ `False`;
  break;
  end if
  end for
  if $match =$ `False` **then**
  break;
  end if
  $cnt \leftarrow cnt + 1$;
  $p \leftarrow p - \ell$;
  end while
  return $(cnt \geq R)$;
end function

Two-stage Matching. Building on the above observations, RRTO adopts a two-stage Operator Sequence Search that first generates high-confidence candidates and then verifies them exhaustively, thereby reducing search complexity. The pseudocode is given in Algorithms 8, 9, and 10, and Fig. 5.5 illustrates the overall idea. Algorithms 8–10 present the search procedure in a unified notation: each one specifies its input and output, uses concise symbols in place of long textual expressions, and relies on helper routines rather than ad hoc inline conditions. Throughout them, s, e, ℓ, p, and cnt denote the start index, end index, candidate length, current probe position, and repetition count, respectively. Algorithm 9 details FASTCHECK, which quickly filters unlikely candidates by comparing tag slices. Algorithm 10 details FULLCHECK, which performs exact verification at the recorded-entry level and additionally checks data dependencies. Here, SLICEEQ(`Tags`, i, j, ℓ) returns `True` when the two tag slices are identical, and ENTRYEQ(x, y) returns `True` when two recorded entries match in function type and

replay-relevant arguments/results.

In the first stage, FASTCHECK uses `HtoD`/`DtoH` transfers to propose candidate boundaries (observation ②) and exploits repetition to locate possible operator sequences (observation ①). Each candidate ends at the current last end operator. If that operator is the true inference end, the candidate starts at the matching start operator (Fig. 5.5c); if it lies within a rotated sequence, the candidate instead starts at the operator immediately following a previous end operator (Fig. 5.5f). To evaluate candidates efficiently, FASTCHECK linearizes the log into category tags (e.g., `HtoD`, `DtoH`, arithmetic) and counts repeated tag substrings in linear time using the SLICEEQ helper routine. This tag-level check confirms sustained repetition despite initialization variability and prunes spurious candidates caused by one-off initialization or noisy boundaries.

After FASTCHECK filters the candidate set, FULLCHECK verifies each remaining candidate exhaustively (Algorithm 10). It realigns rotated candidates to the true `HtoD`/`DtoH` boundaries (observation ②; Fig. 5.5f), enforces data-dependency constraints (observation ③), and performs an exact, record-level comparison across R repetitions. FULLCHECK scans each candidate entry by entry, terminating at the first mismatch. Entry-level equality is not tied to absolute input buffer addresses; instead, FULLCHECK checks role-aware provenance consistency: model parameters must map to persistent buffers, intermediate activations must originate from preceding operators in the same pass, and the end-to-end input may be rebound to a fresh buffer as long as it is consistently identified as the current pass input. The very first operator of each inference has no preceding output to reference, but its input identity is bootstrapped by the `HtoD` boundary itself: the buffer materialized by the boundary `HtoD` event is, by definition, the current pass input regardless of its numerical address, so frameworks such as PyTorch that reallocate the input buffer per inference still yield consistent provenance. This accommodates allocator-driven reallocation while rejecting incorrect dependency chains. Although this stage is the most expensive, it applies only to the few candidates surviving the first stage.

By design, this two-stage Operator Sequence Search addresses demultiplexing and non-stationarity through boundary alignment and dependency checks, while taming scale through aggressive candidate pruning. It therefore enables precise sequence extraction from logs without framework support, even under real-world MEC conditions.

Start boundary need not be HtoD. RRTO constructs the start-candidate set as the union of

(i) indices of `HtoD` transfers, which serve as helpful hints, and

(ii) the first operator immediately following each `DtoH`, which is the definitive end boundary of the previous inference

(Algorithm 8). This design handles input prefetching and overlapping streams: even if the next inference's `HtoD` is issued before the previous `DtoH`, the operator at $t+1$ remains a valid start candidate. Accordingly, FASTCHECK can detect repeated sequences beginning at $t+1$, and FULLCHECK then realigns such cyclically rotated candidates to the prefetched `HtoD` while validating data dependencies and end boundaries. Hence, `HtoD` is a useful anchor, but not a mandatory start boundary.

End boundary is DtoH by design. RRTO uses `DtoH` to delimit the end of an inference, because the output is materialized on the host only upon `DtoH`. When an application batches multiple results into a single `DtoH`, RRTO conservatively coalesces the batch into one inference instance.

Determinism vs. correctness. Given the same logs, Operator Sequence Search always returns the same candidate inference operator sequence, i.e., it is deterministic. Correctness, however, requires that the returned sequence match the ground-truth per-inference execution sequence; determinism alone does not imply correctness. Therefore, FULLCHECK enforces three invariants: boundary anchoring to `HtoD`/`DtoH` events, provenance-constrained data dependencies, and exact repetition over at least R consecutive inferences. Exact address equality is only a sufficient special case; even with fresh input buffers across inferences, FULLCHECK identifies the same logical input by checking consumption patterns within the operator chain. Requiring recurrence over R inferences together with these semantic checks makes the probability of accepting an incorrect candidate extremely small. Increasing R further reduces this probability, at the cost of additional warm-up inferences. For the SAMs targeted by RRTO, a small R (3 by default) suffices in practice; cases requiring larger R usually correspond to DAM-like behavior beyond our target scope.

Formal Time Complexity Analysis

We next formalize the search complexity of Operator Sequence Search through theorem-style statements, each followed by its proof. Let L be the length of the concatenated raw log over the first R inferences used for search, and let N be the operator count in one steady-state inference. A brute-force substring search that tries every start–end pair incurs $\mathcal{O}(L^2)$ time and is impractical on resource-constrained clients. RRTO's two-stage design (FASTCHECK+FULLCHECK) exploits (i) stable `HtoD`/`DtoH` markers and (ii) repeated-sequence detection to prune the search.

Theorem 2 (Search complexity and the role of R) *Let K be the number of candidates that survive* FASTCHECK. *Then the expected time complexity of Operator Sequence Search is*

$$\mathcal{O}(L) + \mathcal{O}(KN).$$

Furthermore, under the steady-state assumption for SAMs, if an incorrect candidate must satisfy FULLCHECK *over R repeated inferences before being accepted, then its survival probability decreases multiplicatively in R (and, whenever the per-trial failure probability is bounded below*

one, at least geometrically in R), whereas the additional warm-up cost increases only linearly in R.

Proof. FASTCHECK uses a single forward scan with constant-time tag comparisons and periodicity checks, hence $\mathcal{O}(L)$. FULLCHECK performs (i) a one-to-one record comparison to rule out rotated sequences and initialization artifacts and (ii) a dependency check that maps inputs to earlier outputs or model parameters; both are linear in N using hash-indexed provenance metadata. Summing over the K surviving candidates gives $\mathcal{O}(KN)$. For the role of R, an incorrect candidate must independently satisfy the boundary, provenance, and entry-level invariants on every one of the R repeated passes, so its survival probability decreases multiplicatively in R; whenever each per-trial failure probability is bounded below one, the cumulative survival probability admits an upper bound that shrinks at least geometrically in R, while the warm-up cost grows by exactly one inference per unit increase of R.

Rationale for the default $R = 3$. Theorem 1 already establishes that reliability grows multiplicatively with R while warm-up cost grows only linearly, so the remaining question is how small R can be in practice. The principle that a handful of repeated observations suffices to drive residual uncertainty below operationally meaningful levels is well established in modern ML systems, including multi-run consensus for large language models [156] and uncertainty-quantification practice for deep learning [3]. For SAMs, this small-R regime also has a concrete engineering lower bound: a SAM inference is preceded by a short, non-repeating initialization phase (one-off allocations, first-call autotuning, framework setup), after which the operator stream enters a strictly repeating steady state. Hence $R = 1$ cannot separate initialization artifacts from steady-state operators, and $R = 2$ still admits an initialization-plus-first-steady pair as a false match; $R = 3$ is the smallest choice that guarantees at least two of the matched repetitions fall in the post-initialization steady state, which matches what we consistently observe on our SAM workloads. We therefore adopt $R = 3$ as the default and expose it as a configurable parameter for deployments preferring larger margins; a workload-aware adaptive R driven by a finer-grained probabilistic model of the per-invariant failure distribution is left as future work.

Error Analysis

In RRTO, Positive means that FULLCHECK accepts a candidate; Negative means it rejects a candidate. True indicates that the candidate equals the ground truth; False indicates it differs. Thus:

(i) True Positive (TP): correct operator sequence accepted;

(ii) True Negative (TN): incorrect sequence rejected;

(iii) False Positive (FP): incorrect sequence accepted;

(iv) False Negative (FN): correct operator sequence rejected.

To make FP practically negligible for SAMs, FULLCHECK enforces three invariants that an unrelated sequence is highly unlikely to satisfy simultaneously:

- Boundary identification. Every accepted sequence must start at the earliest input-consuming operator in the same pass (either an HtoD event or the first operator after a DtoH that ingests the inference input) and must end at the DtoH transfer producing the final output.
- Provenance-constrained dependencies. For every operator in the candidate, inputs must originate from

  (i) the current inference input, identified by its role in the current pass rather than by simple address equality alone,

  (ii) outputs of preceding operators within the same pass, or

  (iii) model parameters stored in persistent parameter buffers;

  thus, reusing the exact same address is only a sufficient special case, whereas fresh but semantically equivalent input buffers can still be recognized through the same operator-to-operator usage relations. These dependencies are checked explicitly, preventing arbitrary repetitions or spurious memory copies from forming a valid pipeline; and
- Repeated equality across R inferences. The candidate must reappear contiguously and identically for at least R consecutive inferences. This requires matching operator API/function call, tensor shapes/sizes, and input/output provenance across R repetitions, which is unlikely unless the candidate is the true operator sequence.

Taken together, these invariants render FP negligible for SAMs in our setting.

Potential FN causes include:

(i) the operator sequence genuinely varies across inferences (DAMs),

(ii) logs are truncated or corrupted,

(iii) R is set too high so that R identical repetitions cannot be found within the collected logs, or

(iv) extremely volatile allocators that destroy stable provenance information across inferences.

RRTO targets SAMs with fixed sequences, and logs are collected per task without arbitrary intra-task interleaving, so these conditions do not hold in our target setting.

5.3.4 Replaying Phase Design

Workflow of Replaying Phase

In this section, we present the workflow of RRTO during the replaying phase and explain how it eliminates per-operator RPC communication via its record/replay mechanism. Fig. 5.6 illustrates this workflow and compares it with traditional transparent offloading systems during model inference in MEC networks. Traditional transparent offloading suffers from frequent operator-level RPC communication, leading to high communication overhead and degraded performance (see Sec. 5.5), including lower GPU utilization, longer inference times, and increased energy consumption.

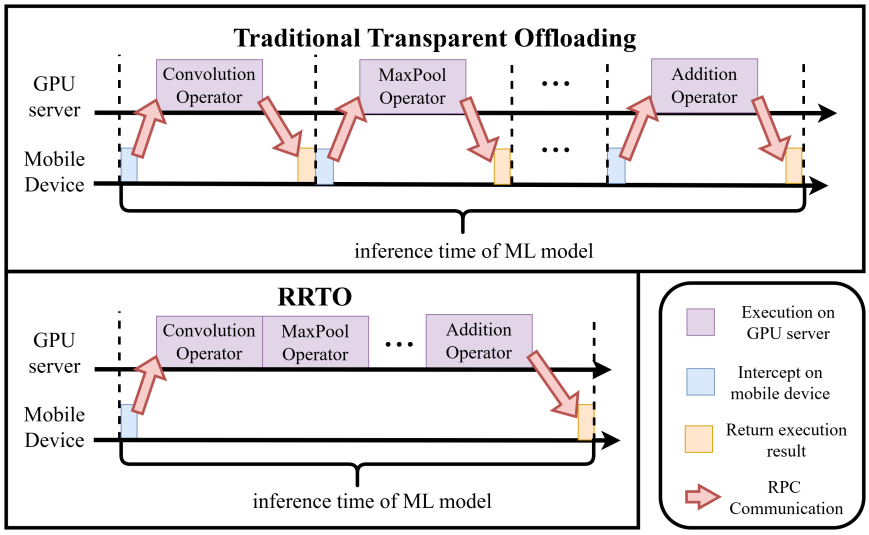


Figure 5.6: Workflow of RRTO during the replaying phase.

To address these communication costs, RRTO introduces an automatic recording and replay mechanism. Since ML models in mobile applications often exhibit static and predictable operator invocation sequences, RRTO records operators called during initial inferences and replays this sequence for subsequent ones. As illustrated in Fig. 5.6, during the replay phase, RRTO first verifies that the current invocation matches the recorded sequence boundary and then issues a guarded replay-start request carrying the first input payload. Subsequent operators are executed directly on the GPU server side, instead of being invoked by RPCs from the mobile device side as in traditional systems, while the client only virtualizes the expected return values and synchronization points. Consequently, RRTO executes all required operators within the inference operator sequence through one verified replay transaction per inference, eliminating the need for additional RPC communications for subsequent operators and significantly reducing communication costs.

Record/Replay Mechanism

Here we describe how RRTO implements its record/replay mechanism after the steady-state inference operator sequence has been identified. The client component is detailed

in **Alg. 11** and the server component in **Alg. 12**. Algorithms 11, 12, and 13 summarize the client-side control flow, the server-side replay routine, and replay-time argument reconstruction. Since Algorithm 8 returns an IOS aligned to an `HtoD` boundary, replay starts from the first input transfer of each inference. Throughout these algorithms, f, a, and r denote the intercepted CUDA function, its argument tuple, and the returned result; `Meta`, P, and `ArgsSeq` denote recorded argument templates, pointer map, and replay-ready argument sequence; MATCHREPLAYSTART and MATCHREPLAYSTEP are guarded predicates validating replay start and step-wise consistency. During recording, each CUDA API invocation is forwarded to the server and logged locally. Once a stable IOS is obtained, the client transfers only input payloads and waits for the final output, while intermediate operators are replayed per the recorded sequence.

Algorithm 11: RRTO_on_Client

Input: stream of intercepted CUDA API invocations (f, a);
Output: execution result r for the current invocation;
Parameter: minimum repeat count R; inferred operator sequence `IOS`; local log list `Logs`.

1: `IOS` ← `NULL`;
2: `Logs` ← [];
3: **while** `True` **do**
4: (f, a) ← GETNEXTCUDAINVOCATION();
5: **if** `IOS` = `NULL` **then**
6: SENDRPCTOSERVER(f, a);
7: r ← GETRPCEXECUTIONRESULT();
8: APPEND(`Logs`, (f, a, r));
9: `IOS` ← OPERATORSEQUENCESEARCH(`Logs`, R);
10: **else**
11: **if** MATCHREPLAYSTART(f, a, `IOS`) **then**
12: STARTRRTO(`IOS`, a);
13: **end if**
14: **if** f = **‘‘**`HtoD`**’’ then**
15: r ← SENDRPCTOSERVER(a);
16: **else if** f = **‘‘**`DtoH`**’’ then**
17: r ← WAITFORRRTORESULT();
18: **else**
19: **if** MATCHREPLAYSTEP(f, a, `IOS`) **then**
20: i ← FIND(`IOS`, f);
21: r ← `IOS`$[i]$[`ret`];
22: **else**
23: ABORTRRTO();
24: SENDRPCTOSERVER(f, a);
25: r ← GETRPCEXECUTIONRESULT();
26: APPEND(`Logs`, (f, a, r));
27: `IOS` ← OPERATORSEQUENCESEARCH(`Logs`, R);
28: **end if**
29: **end if**
30: **end if**
31: RETURNRESULT(r);
32: **end while**

In **Alg. 11**, the offloading client continuously serves the intercepted CUDA invocation stream. While `IOS` is `NULL`, RRTO remains in the recording phase (*Lines 4 to 8*): it forwards the current invocation (f, a) to the server, obtains the execution result r, appends (f, a, r) to `Logs`, and invokes Operator Sequence Search on the accumulated log, preserving traditional transparent offloading during warm-up while overlapping search with outstanding RPCs. Importantly, RRTO does not switch into replay mid-inference; it finishes recording the current inference and enables replay only at the next inference's first operator, ensuring replay always starts at a correct sequence boundary.

Once the inference operator sequence is identified, RRTO enters the replay phase on the mobile device (*Lines 10 to 28*). Replay starts conservatively: the client calls MATCHREPLAYSTART to jointly check function type, first-input transfer pattern, and boundary consistency before issuing STARTRRTO (*Line 12*). During replay, `HtoD` transfers carry the current input payload to the server (*Line 15*), `DtoH` waits for the final replay result (*Line 17*), and intermediate invocations return virtualized results only when MATCHREPLAYSTEP confirms step-wise alignment with `IOS` (typically a recorded status such as "cudaSuccess", *Lines 18 to 21*). On any mismatch, the client aborts replay immediately, falls back to normal RPC forwarding, and reruns Operator Sequence Search so that a new stable sequence can be learned safely (*Lines 22 to 26*). Finally, r is returned to the upper-layer application (*Line 28*).

Algorithm 12: RRTO_on_Server

Input: stream of client requests `task`, where each `task` is either `RPC` or `StartRRTO`;
Output: execution result r returned to the client when required;
Parameter: recorded metadata `Meta`; pointer map P; item size s; inference operator sequence `IOS`.

1: `IOS` ← `NULL`;
2: **while** `True` **do**
3: `task` ← GETNEXTCLIENTTASK();
4: **if** `task` = `RPC` **then**
5: (f, a) ← GETCLIENTINPUT();
6: r ← CUDARUNTIMELIBRARY(f, a);
7: SENDEXECUTIONRESULTBACK(r);
8: **else if** `task` = `StartRRTO` **then**
9: `IOS` ← GETCLIENTINPUT();
10: `payload0` ← GETCLIENTINPUT();
11: L_0 ← GETPAYLOADLEN(`payload0`);
12: `ArgsSeq` ← RRTOFIXARGS(`IOS`, `Meta`, P, L_0, s);
13: **for** $i \leftarrow 0$ to $|\texttt{IOS}| - 1$ **do**
14: f ← `IOS`$[i]$[`func`];
15: a ← `ArgsSeq`$[i]$;
16: **if** f = **'HtoD' then**
17: **if** $i = 0$ **then**
18: `payload` ← `payload0`;
19: **else**
20: `payload` ← GETCLIENTINPUT();
21: **end if**
22: a ← ATTACHPAYLOAD(a, `payload`);
23: **end if**
24: r ← CUDARUNTIMELIBRARY(f, a);
25: **if** f = **'DtoH' then**
26: SENDEXECUTIONRESULTBACK(r);
27: **end if**
28: **end for**
29: **end if**
30: **end while**

In **Alg. 12**, the RRTO offloading server continuously awaits tasks from the client, aligning its operational phase with that of the client to ensure synchronized processing. During the recording phase, the server processes RPC requests similarly to traditional transparent offloading systems (*Lines 3 to 5*). As the client on the mobile device initiates its replaying phase, the offloading server simultaneously begins its own (*Lines 7 to 21*), replaying the recorded inference operator sequence after receiving the verified start request with the first input payload. During this phase, RRTO configures each operator's parameters (*Line 13*, Alg. 13), supplies the current inference input in a role-consistent manner, and returns the final DtoH result once the sequence completes.

Hidden system complexity

Beyond the high-level replay control flow in Algorithms 11 and 12, replay-time arguments cannot always be reused verbatim because pointer values, lengths, shapes, and launch configurations may depend on the current input batch. RRTO therefore performs replay-time argument reconstruction in Algorithm 13. RRTO addresses several non-trivial systems challenges in RRTOFIXARGS (**Alg. 13**) to enable transparent, high-performance offloading:

(i) pointer swizzling that preserves dependency edges across distinct client–server address spaces;

(ii) client-side state virtualization that maintains CUDA stream semantics and return codes without local kernel execution; and

(iii) runtime support for batch-size variability.

These mechanisms jointly ensure correctness and performance in MEC deployments.

Algorithm 13: RRTOFIXARGS

Input: recorded operator sequence `IOS`; argument templates `Meta`; pointer map P; first `HtoD` payload length L_0; item size s;
Output: adjusted argument sequence `ArgsSeq` for replay.

1: **function** RRTOFIXARGS(`IOS`, `Meta`, P, L_0, s)
2: $b \leftarrow L_0/s$;
3: `ArgsSeq` $\leftarrow [\,]$;
4: **for** each op $\in$ `IOS` **do**
5: $a \leftarrow$ COPY(`Meta`[op]);
6: **for** each field x in a **do**
7: **if** ISPOINTER(x) **then**
8: $x \leftarrow P[x]$;
9: **else if** ISLENGTHFIELD(x) **then**
10: $x \leftarrow$ RECOMPUTELEN(op, b, `Meta`);
11: **else if** ISSHAPEORSTRIDEFIELD(x) **then**
12: $x \leftarrow$ RECOMPUTESHAPE(op, b, `Meta`);
13: **else if** ISLAUNCHFIELD(x) **then**
14: $x \leftarrow$ RECOMPUTELAUNCH(op, b, `Meta`);
15: **end if**
16: **end for**
17: APPEND(`ArgsSeq`, a);
18: **end for**
19: **return** `ArgsSeq`;
20: **end function**

Pointer swizzling and dependency preservation. Client buffers are allocated via framework memory pools, so client virtual addresses can vary across inferences, whereas server-side device buffers are managed independently. Correct replay must preserve data-dependency edges regardless of payload or batch sizes. During recording, RRTO builds a directed dependency graph from pointer identities (tensors and intermediate buffers) and synchronization boundaries (e.g., cudaStreamSynchronize). During replay, the server reconstructs the client-server pointer mapping and invokes RRTOFIXARGS (*Lines 5-20 in Alg. 13*) to specialize size-related arguments (tensor shapes/strides, buffer lengths, and grid/block dimensions) while preserving pointer identities and dependency edges. This ensures correct tensor materialization and kernel launches without per-operator RPC.

	inference	communication	standby
Energy (Watt)	13.35	4.25	4.04

Table 5.2: Power draw (Watt) of our robot in different states.

State Virtualization on the Client. Upper-layer frameworks expect synchronous return values, consistent stream/event ordering, and deterministic CUDA error codes. The client replayer (*Lines 20 in Alg. 11*) virtualizes the CUDA runtime state, synthesizing return codes (e.g., cudaSuccess), mirroring recorded stream/event ordering, and deferring externally observable effects to recorded synchronization points, preserving framework semantics without source modification while kernels execute remotely.

Batch-size variability support. RRTO supports variable batch sizes without re-recording, provided the operator sequence remains stable. During recording, it derives data dependencies from pointer identities and synchronization boundaries, which are batch-size agnostic; only payload lengths change. At replay start, the server infers the effective batch size from the first HtoD transfer (Cricket exposes its payload length) and invokes RRTOFIXARGS (*Line 2 in Alg. 13*) to specialize subsequent size-related arguments (tensor shapes/strides, buffer lengths, and grid/block dimensions) while preserving pointer identities and dependency edges. This enables correct replay across different batch sizes.

5.4 Implementation

In this section, we first elaborate on the implementation of RRTO, and then introduce the experiment setup.

5.4.1 Implementing RRTO

Software. We implemented RRTO within Cricket's codebase [39], a transparent offloading system that provides a virtualization layer for CUDA applications, enabling remote execution without the need for source code modifications and recompilation of applications. RRTO employs the same RPC library for communication operations as Cricket: Libtirpc [35], a transport-independent RPC library for Linux. We integrated RRTO's recorder and replayer into the corresponding RPC functions in Cricket, allowing for seamless integration and efficient operation of the record/replay mechanism.

Hardware. The evaluation was conducted on a customized four-wheeled robot (Fig. 5.7), equipped with a Jetson Xavier NX [106] 8G onboard computer that is capable of CUDA-acclerated ML model inference. The system runs Ubuntu 20.04 with ROS Noetic and uses a dual-band USB network adapter (MediaTek MT76x2U) for wireless communication. Detailed hardware and sensor configurations are shown in Fig. 5.7. The GPU server is a PC equipped with an Intel(R) i5 12400f CPU @ 4.40 GHz and an NVIDIA GeForce GTX 2080 Ti 11 GB GPU, connected to our robot via Wi-Fi 6 over a 160 MHz channel at 5 GHz frequency.

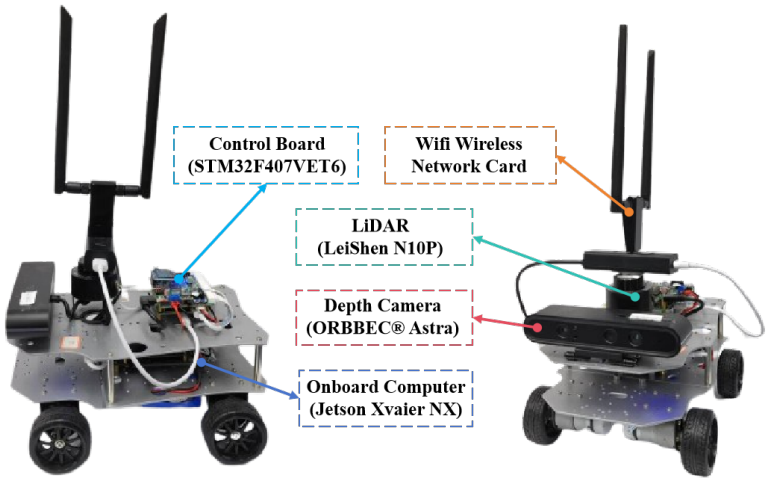


Figure 5.7: The detailed composition of the robot platform.

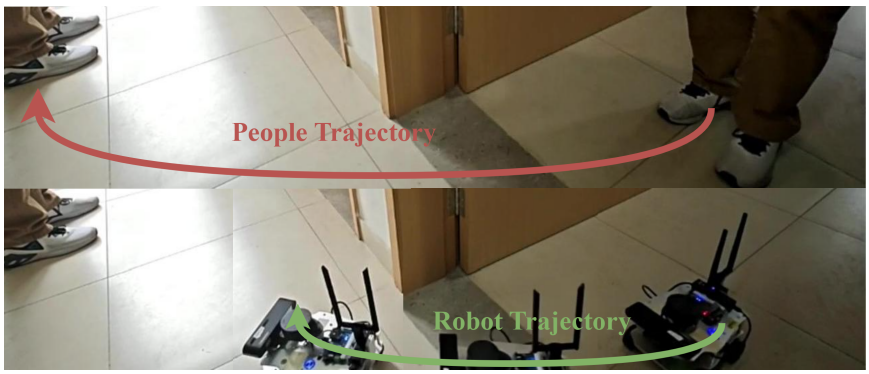


Figure 5.8: Kapao [96], a real-time people-tracking application on our four-wheeled robot with a CNN-based human keypoint detection model.

Tab. 5.2 summarizes the robots' on-board energy consumption (excluding motor power) in different states: inference (full GPU utilization, including CPU/GPU power), communication (communication with the GPU server, including wireless network card energy consumption), standby (no tasks to execute). Each Jetson Xavier NX is powered by a 21.6 Wh battery, sustaining up to 1.6 hours of continuous model inference. We continuously log the robot's instantaneous on-board power draw (in Watts) at 1-second intervals, utilizing the back-end power consumption and performance monitoring methodology from [106]. Subsequently, the energy consumption per inference (in Joules) is precisely calculated by integrating this power draw profile over the exact duration of each inference, determined by its start and end timestamps.

5.4.2 Experiment Setup

Task. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig. 5.8 and Fig. 5.9. For evaluation, we run KAPAO on CrowdPose dataset [75]

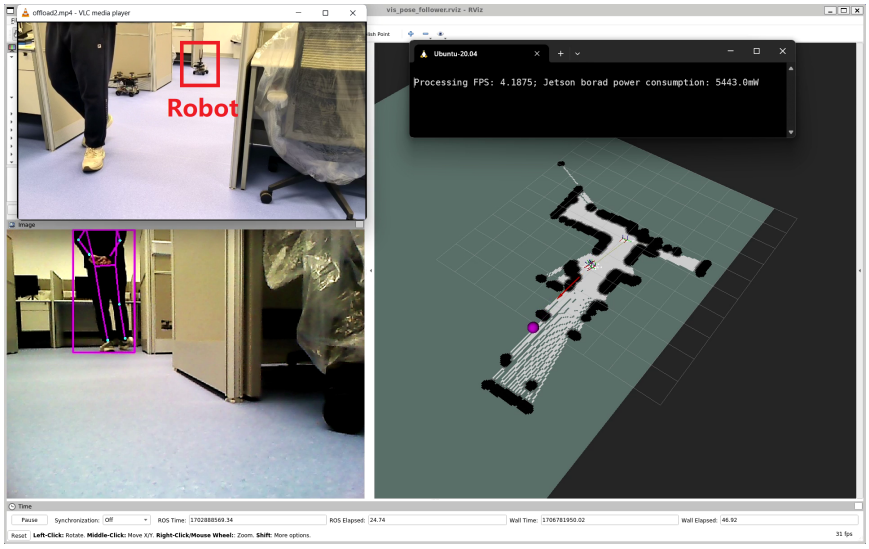


Figure 5.9: A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner provides a third-angle observation of the entire experimental process.

using our testbed. To demonstrate the generalization ability of RRTO, we also evaluated several representative models from three categories of real-world mobile applications with their implementations available in Torchvision [92] on CIFAR-100 dataset [69]:

(i) ResNet [48] and ConvNext [87] for object classification;

(ii) FCN [88] and DeepLabv3 [18] for semantic segmentation;

(iii) FasterRCNN [121] and RetainNet [82] for object detection.

Emulation Environments. We evaluated two real-world environments: indoors (robots move in our laboratory with desks and separators interfering with wireless signals) and outdoors (robots move in our campus garden with trees and bushes interfering with wireless signals, resulting in lower bandwidth). The corresponding bandwidths between the robot and the GPU server in indoors and outdoors scenarios are shown in Fig. 5.3.

Baselines. To comprehensively evaluate the performance of RRTO, we conducted comparative experiments against several baseline approaches:

- **Device-only inference ("Device-only")**: A conventional setup where the entire model is deployed and executed on the robot.
- **Native non-transparent offloading ("NNTO")**: A non-transparent offloading method that deploys the entire model on a GPU server by modifying the source code. We evaluate NNTO independently to demonstrate the benefits of offloading in inference latency and energy efficiency, and we use it as the theoretical upper bound

for offloading approaches because it incurs minimal system transmission overhead by transmitting only inference inputs and outputs. This comparison underscores the communication time savings and performance gains achieved by RRTO.

- **Cricket** [39]: A state-of-the-art transparent offloading system designed for remote GPU usage.

While advanced scheduling optimizations (e.g., layer partitioning [64] and multiple inference scheduling [81, 43, 171]) are adaptable to RRTO, their performance benefits are orthogonal to our core contributions. Consequently, to isolate the impact of our innovations and ensure a fair comparison against existing non-transparent offloading (NNTO) methods, we have excluded these optimizations from our current implementation and evaluation. Their integration into RRTO remains a promising direction for future work, as discussed in Sec. 5.6.

5.5 Evaluation

In this section, we evaluate the performance of RRTO from four aspects:

(i) comparison with baseline systems on inference latency and energy consumption in real-world mobile application;

(ii) sensitivity of RRTO in various MEC scenarios;

(iii) analysis of RRTO's record/replay mechanism;

(iv) performance evaluation of RRTO on large-scale models common to fundamental mobile applications.

5.5.1 Superiority of RRTO

Our evaluation results of our robotic application, KAPAO, as presented in Fig. 5.10, demonstrate that RRTO achieved performance comparable to non-transparent offloading system (NNTO) in both inference time and energy consumption.

In terms of inference time, RRTO reduced inference time by an average of 72% compared to local computation and 95% compared to Cricket in the indoors scenario; the reductions in the outdoors scenario were 69% and 94%, respectively. Device-only, which processes the entire model on the robot without any data transmission to a GPU server, exhibited consistent performance in both indoor and outdoor scenarios. In contrast, the substantial communication costs associated with Cricket's frequent RPCs from its transparent offloading mechanism considerably slowed its inference times, a point that will be elaborated on in Sec. 5.5.2. By implementing its innovative record/replay mechanism, RRTO effectively minimized these extensive communication costs, achieving inference times comparable to those of NNTO, with nearly identical communication expenses.

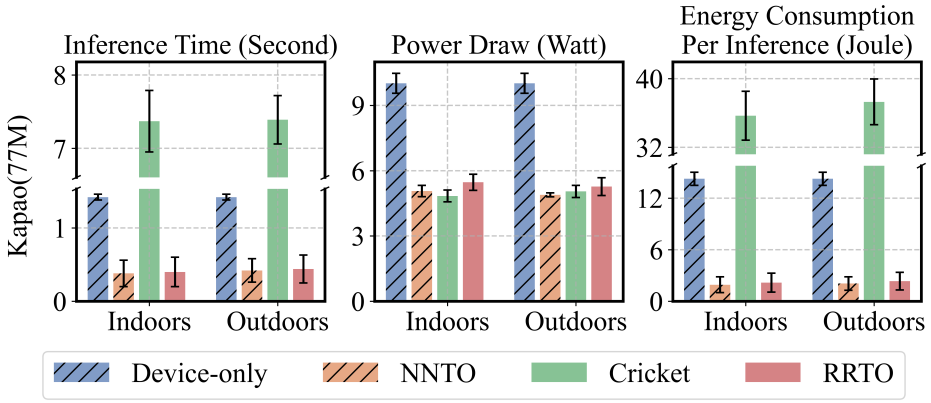


Figure 5.10: Performance of Kapao in different environments with various systems.

In terms of energy consumption, RRTO achieved substantial reductions, decreasing energy usage per inference by an average of 85% compared to local computation and 94% compared to Cricket in indoors scenarios; outdoors reductions were 84% and 93%, respectively. Due to the intensive computational demands of the model, device-only inference incurred high power draw, whereas other systems that offload computations to a GPU server exhibited reduced energy usage. Although RRTO only reduced power draw by 45% compared to local computation and even showed a 13% increase in power draw compared to Cricket, its shorter inference times resulted in significantly lower energy consumption per inference. It is important to note that the average power draw values shown in Fig. 5.10 do not correspond to those in Tab. 5.2. This discrepancy arises because our application does not fully utilize the GPU capabilities of the Jetson Xavier NX, resulting in lower average energy consumption during local computation than during the inference stage. Furthermore, additional CPU computing tasks (e.g., robot control) cause the average energy consumption of all offloading systems to increase above levels observed during communication and standby phases.

The prolonged inference times observed in outdoor scenarios for all offloading systems can be attributed to the lower bandwidth available outdoors (see Sec. 5.2.3), which results in extended transmission times compared to indoor scenarios. Analyzing performance in both indoors and outdoors settings, we find that RRTO is robust across various MEC scenarios. This robustness stems from RRTO's ability to eliminate the frequent transmission requirements of RPCs in Cricket, thereby reducing communication overhead to levels comparable to NNTO. Moreover, when GPU servers reside in commercial cloud environments, network congestion and routing inefficiencies further restrict available bandwidth [103] and drive up transmission costs, making RRTO even more advantageous than Cricket.

5.5.2 Micro-Event Analysis

To gain a deeper insight into the performance improvements facilitated by RRTO, we analyzed the RPC function calls made by Cricket during various stages of KAPAO

CUDA Runtime API	Composition during loading model	Composition during initializing inference	Composition during steady-state inference loop
cudaGetDevice	46858 (82.32%)	4789 (80.12%)	4735 (80.32%)
cudaGetLastError	4244 (7.46%)	616 (10.31%)	607 (10.30%)
cudaLaunchKernel	2752 (4.83%)	523 (8.75%)	522 (8.85%)
cudaMalloc	65 (0.11%)	4 (0.07%)	0 (0.00%)
cudaStreamIsCapturing	68 (0.12%)	4 (0.07%)	0 (0.00%)
cudaStreamSynchronize	1118 (1.96%)	16 (0.27%)	11 (0.19%)
cudaMemcpyHtoD	1117 (1.96%)	7 (0.12%)	3 (0.05%)
cudaMemcpyDtoH	1 (0.002%)	9 (0.15%)	8 (0.14%)
cudaMemcpyDtoD	701 (1.23%)	9 (0.15%)	9 (0.15%)

Table 5.3: Composition of RPC function calls during different stages of KAPAO inference.

inference, illustrating the characteristics of traditional transparent offloading mechanisms. A detailed breakdown of these calls is provided in Tab. 5.3.

Comparing the different stages of function calls in Tab. 5.3, we can see that KAPAO undergoes an initialization stage of inference different from subsequent inference loops. This is because the working process of KAPAO [96] follows the default detection model in Yolo v5 [63]: the inference pipeline is first initialized by generating a mesh grid of a certain size that fits the input image size, which serves as the storage of intermediates; then in the following loop iterations through the inference pipeline the mesh grid is reused and the operator call sequence is fixed. RRTO records all involved operators during the first few inferences, not just the initial process, and ignores the different operator sequences from the initializing inference until the correct operator sequence is found.

In the loop inference detailed in Tab. 5.3, we observed that a significant portion, specifically 90.62%, of RPC function calls consisted of "cudaGetDevice" and "cudaGetLastError". These calls, generated by PyTorch [112] due to our application's reliance on this framework, are crucial for determining the data's location, facilitating computations across multiple GPUs and parallel tasks.

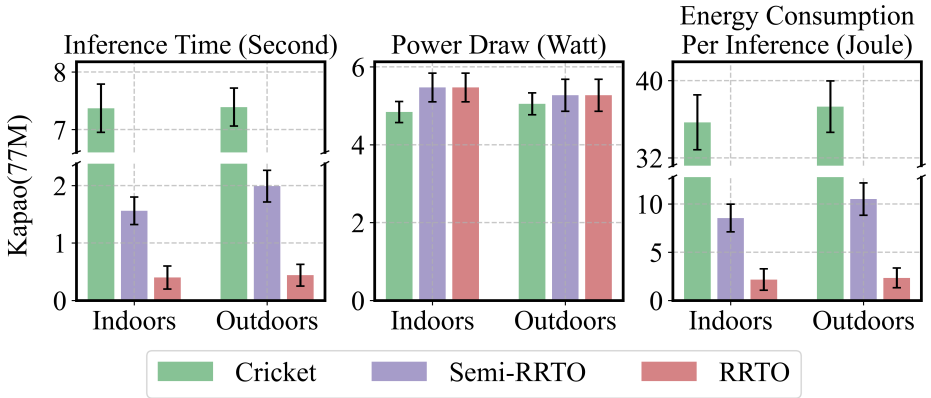


Figure 5.11: Semi-RRTO: only applying Caching [134] specifically to the RPCs of "cudaGetDevice" and "cudaGetLastError" in RRTO, effectively eliminating their transmission requirements.

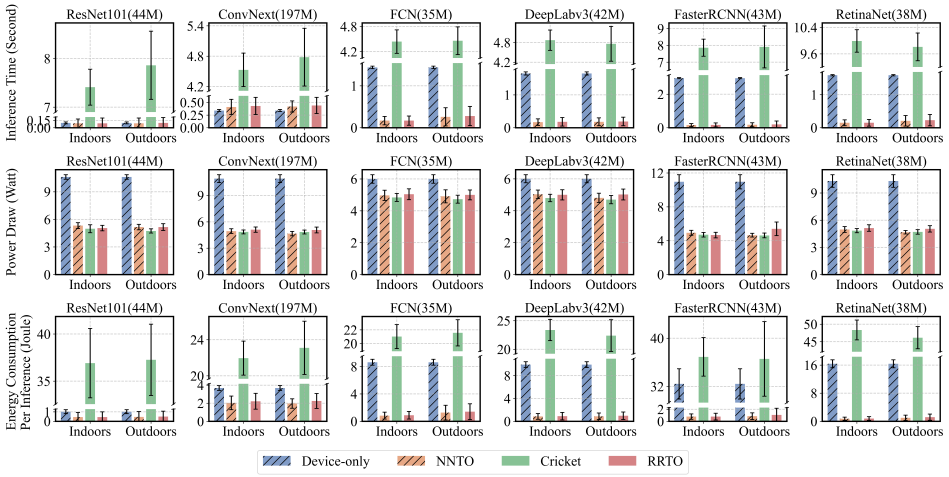


Figure 5.12: Performance of Torchvision models in different environments with various systems.

Despite restricting PyTorch to use only a single GPU sequentially and employing Caching (referred to as "semi-RRTO" in Fig. 5.11) to reduce the transmission of RPCs, semi-RRTO achieved inference time comparable only to local computation in our experiments and did not reach the speeds observed with NNTO. This is evident from the fact that "cudaLaunchKernel" still represents 8.85% of total RPC function calls, which are essential for notifying the server about subsequent computing tasks like additional convolution or maxpool operations. While traditional RPC optimization methods wait for "cudaLaunchKernel" RPCs from the client to direct the server's subsequent computing tasks (operators), RRTO recorded these "cudaLaunchKernel" function calls and directly executed the subsequent computing tasks on the server, thereby eliminating the need for ongoing communication with the client.

Regarding the remaining RPC functions, namely "cudaMalloc", "cudaStreamIsCapturing", "cudaStreamSynchronize", and "cudaMemcpyDtoD", which collectively account for 0.34% of the total RPC calls, they primarily handle data transmission and synchronization within the GPU and can also be replayed by RRTO on the server. However, "cudaMemcpyHtoD" and "cudaMemcpyDtoH", which account for 0. 19% of total RPC calls, are used primarily for data transmission between the CPU and GPU. They are mainly used for the input and output of the ML model and cannot be replayed by RRTO.

To further illustrate the performance gains achieved by RRTO, we compared it with baseline systems in the number of RPC calls and the resulting average GPU utilization on the GPU server during the execution of [96], measured using pynvml [108] and presented in Tab. 5.4. Unlike RRTO and Cricket, NNTO bypassed RPC by directly synchronizing the input and output data of the ML model between the CPU and the GPU at the application layer, requiring modifications to the source code. Cricket, on the other hand, experienced higher communication costs, which contribute to lower GPU utilization on the GPU server. Although RRTO also managed "cudaMemcpyDtoH" and "cudaMemcpyHtoD" like Cricket, resulting in 11 RPCs per inference, the performance improvements offered by RRTO were clearly advantageous.

	NNTO	Cricket	RRTO
RPCs for each inference	NA	5895	11
Average GPU utilization on the GPU server	29.0%	1.1%	27.5%

Table 5.4: Comparison between RRTO and the baselines about numbers of RPC calls and average GPU utilization on GPU server.

5.5.3 Validation on A Wider Range of Models

Next, we conducted a comprehensive evaluation of RRTO and other baseline systems across a diverse set of models commonly used in mobile devices, varying in parameter counts, as detailed in Fig. 5.12. We selected the two most prevalent models for each of the three fundamental robotic tasks (object classification, semantic segmentation, and object detection) to assess the generalizability of RRTO's performance. Our findings confirmed that RRTO achieves a performance comparable to NNTO without requiring any modifications to the source code. Cricket sometimes exhibited slower indoors performance compared to outdoors due to its exceptionally prolonged inference times and unstable network fluctuations, as shown in Fig. 5.3. Although RRTO consistently outperforms in various models, performance gains are relatively smaller for models with fewer parameters. This observation can be attributed to the fact that models with larger computational demands benefit more significantly from the robust computing power of the GPU server. Additionally, the substantial energy consumption incurred by extensive computations on the robot suggests that models with a larger number of parameters are more suited for offloading, thus deriving greater benefits from offloading. It is also pertinent to note that our experimental robot (Fig. 5.1(a)) possesses considerably higher computational power than typical mobile devices (e.g., smartphones), suggesting RRTO's benefits could be even more pronounced on more resource-constrained mobile devices.

5.6 Related Work and Discussion

Fixed Calculation Logic. RRTO leverages the characteristic that operators in the inference of a DNN model often follow a fixed order, allowing its record/replay mechanism to support other computational tasks [25], not solely SAMs, provided they exhibit fixed computational logic. However, RRTO is unable to support tasks with unfixed computational logic, such as DAMs with changing operator sequences or tasks involving complex logic and branching, due to its inability to replay varying sequences. Moreover, tasks with complex logic and branching are generally more suited for CPU rather than GPU execution [123], and optimizing inference for DAMs with changing operator sequences remains a pervasive challenge across all offloading systems.

Layer partitioning. Layer partitioning techniques [64] distribute ML model layers across mobile devices and GPU servers to improve end-to-end inference speed and energy efficiency while tolerating transmission failures and bandwidth fluctuation; because the optimal strategy depends on model architecture and application-specific

trade-offs, the problem has drawn sustained attention in mobile applications. Historically, such partitioning has favored non-transparent offloading, which prioritizes peak performance and minimal transmission overhead and, unlike transparent systems, has direct access to the model architecture needed for effective partitioning. RRTO brings these methods into a transparent framework: it matches the performance of non-transparent systems and recovers the model architecture from data dependencies surfaced by its operator sequence search, enabling partitioning at the finer operator granularity without sacrificing transparency.

Multiple inference Scheduling. Building on layer partitioning for single requests, multiple-inference scheduling minimizes aggregate latency and energy across concurrent DNN requests using decision algorithms such as urgency-based prioritization [81], deep reinforcement learning control [43], and joint relay selection [171] to assign execution location and timing. These algorithms plug directly into RRTO: during replay they choose the location and start time of each operator per task, reproducing the high performance of non-transparent schedulers without any source-code modification. A broader extension, namely fully non-deterministic interleaving of multiple models within one shared application, would require online demultiplexing and replay arbitration beyond the current per-task logging design, and we regard it as important future work.

Model Compression. On mobile devices, quantization and model distillation are two common compression techniques: quantization [47] lowers the precision of weights and activations to cut computation costs, whereas distillation [154] trains a smaller student model to mimic a larger teacher with fewer resources. Offloading is orthogonal to compression: instead of trading accuracy for efficiency, it accelerates inference by scheduling computation across devices while preserving the original model accuracy.

Cross-framework Offloading. RRTO and prior interception-based transparent offloading systems replay intercepted runtime calls on a remote server and therefore require runtime homogeneity to preserve a one-to-one API mapping for control, memory, launch, synchronization, and error semantics. When frameworks differ (e.g., OpenCL vs. CUDA or CPU libraries without kernel-launch semantics), this correspondence vanishes. We thus restrict RRTO to homogeneous runtimes and regard robust cross-framework offloading as open work; achieving it would require a compatibility layer that semantically translates every call and data layout, tantamount to framework porting (e.g., OpenCL-to-CUDA), which entails substantial engineering and maintenance costs.

5.7 Conclusion

This chapter has proposed RRTO, a high-performance transparent offloading system optimized for ML model inference in MEC. RRTO alleviates the communication overhead of traditional transparent offloading systems by replacing frequent per-operator RPCs with a record/replay mechanism. This design enables RRTO to approach the

performance of non-transparent offloading without requiring modifications to application source code. Consequently, RRTO makes transparent MEC inference faster, more energy-efficient, and more practical for diverse ML-driven mobile applications in real-world environments.

Chapter 6

Conclusion and Future Work

6.1 Summary

THIS thesis has studied how Mobile Edge Computing (MEC) can support machine learning systems that are adaptive, responsive, and deployable in the physical world. The thesis is built around one recurring systems problem: conventional ML systems inherit execution units from data-center environments, but these units do not match the wireless dynamics, workload structure, or deployment constraints of MEC. This *granularity mismatch* appears as communication-induced stragglers in online training, stop-and-wait transmission in collaborative inference, and round-trip amplification in transparent deployment. The thesis addresses these bottlenecks through *granularity-aware execution*: selecting the synchronization, scheduling, or control unit at which an MEC ML system should operate.

ROG (Chapter 3) addresses online training in robotic IoT. The key observation is that robotic wireless bandwidth fluctuates on the same time scale as gradient synchronization, so whole-model synchronization can become stale while a model update is still being transmitted. ROG resolves this mismatch by lowering the synchronization unit from whole models to parameter rows. Row Synchronous Parallel extends bounded staleness from model replicas to individual rows, while the Adaptive Transmission Protocol schedules row transmission according to staleness, gradient importance, and real-time bandwidth. Under the same wall-clock budget, ROG improved training accuracy by 4.9%–6.5% and reduced energy consumption by 20.4%–50.7% for the same target accuracy, compared with BSP, SSP, and a state-of-the-art network-aware baseline.

LOPInfer (Chapter 4) addresses collaborative inference for MEC service workloads. The key observation is that layer-wise partitioning serializes device computation, wireless transmission, and server computation within one inference request, even though many operators expose finer producer-consumer structure. LOPInfer resolves this mismatch by lowering the scheduling unit from whole layers to local operations.

Local Operation Parallelism decomposes eligible operators while preserving producer-consumer dependencies, and the Local Operation Scheduling Strategy places these operations across the mobile device and edge server under compute and transmission constraints. This design enables intra-layer and cross-layer compute–communication overlap, reducing per-request latency by up to 50% and device energy by up to 75% compared with state-of-the-art layer-partitioning baselines.

RRTO (Chapter 5) addresses transparent deployment for MEC inference. The key observation is that transparent offloading preserves compatibility by forwarding low-level runtime calls, but ML inference repeatedly executes a stable operator sequence after warm-up. RRTO resolves this mismatch by raising the control unit from per-operator RPCs to per-inference operator-sequence replay. It records runtime behavior, reconstructs the steady-state sequence through a two-stage Operator Sequence Search that reduces search complexity from $\mathcal{O}(L^2)$ to $\mathcal{O}(L)$, and replays each subsequent inference as one proactive remote execution. RRTO therefore keeps the compatibility of transparent offloading while reducing per-inference latency and energy by up to 98% over the state-of-the-art transparent baseline.

Overall, the three systems form a coherent lifecycle for MEC intelligence. ROG makes deployed models stronger by enabling robust online training; LOPInfer makes model serving more responsive by exposing overlap within a single request; and RRTO makes adoption easier by preserving transparent deployment without paying per-operator round trips. Their common contribution is methodological: instead of treating MEC as a smaller cloud, they redesign the execution granularity at the point where wireless variation, mobile energy, and software compatibility enter the critical path. This methodology provides a practical way to build ML systems that remain useful outside controlled data-center environments.

6.2 Limitations

This thesis has shown that *granularity-aware execution* is an effective principle for building MEC ML systems, but the three systems are not intended to cover every model, runtime, or deployment environment. Their limitations are best understood by examining the workload regularities and system assumptions that each design exploits.

Limitation 1: Dependence on exploitable workload regularity. The first limitation is that each system changes execution granularity by exploiting a specific regularity in the target workload. ROG assumes parameter-server-style distributed training in which gradients can be partitioned into rows and bounded staleness remains a meaningful convergence control. This makes row-level synchronization effective against wireless stragglers, but it does not solve the algorithmic challenge that data collected by different robots is often non-IID, nor does it automatically design task-specific online-learning objectives for perception, mapping, or control. LOPInfer assumes that a model contains enough local operators whose outputs can be decomposed into independently

computable local operations. Its benefit is therefore smaller for models dominated by global operators, reductions, large activations with low compute-to-communication ratios, or modern primitives whose useful parallelism is not yet captured by the current local-operation abstraction. RRTO assumes that inference exposes a stable operator sequence after warm-up. This assumption holds for many static DNN inference workloads, but it is weaker for dynamic models, input-dependent control flow, complex branching logic, or arbitrary interleavings of multiple models within one application. These assumptions are not accidental weaknesses; they are the workload structures that make a more suitable execution granularity visible.

Limitation 2: Limited coverage of wireless technologies, platforms, and deployment dynamics. The second limitation is that the evaluation covers practical MEC environments but not the full space of wireless edge deployments. ROG and LOPInfer were evaluated mainly over commodity Wi-Fi links on robotic platforms because Wi-Fi is widely available in robotic IoT testbeds and already exhibits substantial bandwidth fluctuation. However, the papers do not exhaustively evaluate 5G, WiMAX, multi-radio operation, or multi-hop robot communication. RRTO demonstrates the cost of wireless round trips on real robotic MEC testbeds, but it does not study edge-server handoff, replay-state migration, intermittent connectivity, or offloading to other nearby robots and idle edge devices. These conditions can change the best execution granularity during a training iteration, an inference request, or an offloading session. For example, a row-level transmission plan may need to move high-priority rows to a more stable link, a local-operation schedule may become suboptimal after an uplink drop, and a replayed operator sequence may need to migrate when the nearest edge server changes. The thesis therefore establishes mechanisms for dominant device–edge paths under measured wireless variation, but it does not yet provide a complete mobility-management layer for heterogeneous MEC networks.

Limitation 3: Separation from higher-level learning, scheduling, and compatibility policies. The third limitation is that granularity-aware execution provides systems primitives rather than complete end-to-end policies. ROG lowers synchronization to rows, but it does not automatically choose the best staleness threshold, row grouping, or non-IID-aware learning objective. LOPInfer lowers scheduling to local operations, but its solver is heuristic and does not guarantee global optimality; request-level scheduling, quantization, early-exit inference, and other accuracy–efficiency trade-offs remain outside the current scheduling formulation. RRTO raises transparent control to sequence-level replay, but it assumes homogeneous runtime APIs so that intercepted calls can be replayed with matching semantics on the server. Robust cross-framework offloading across CUDA, OpenCL, CPU libraries, and framework-specific backends would require semantic translation of calls, memory layouts, synchronization behavior, and error handling. Similarly, fully non-deterministic interleaving of multiple models would require online demultiplexing and replay arbitration beyond the current per-task logging design. Thus, the systems in this thesis should be viewed as granularity-aware execution mechanisms that must be composed with learning objectives, model

compression, request scheduling, privacy, safety, and runtime-compatibility policies.

6.3 Future Work

The limitations above suggest several concrete directions for extending *granularity-aware execution* beyond the three systems in this thesis.

Future Direction 1: Extending granularity-aware execution to modern and dynamic model structures. A first direction is to generalize the workload structures that can expose useful execution granularity. For inference, this means extending LOPInfer beyond the current class of local operators to cover Transformer primitives, attention, normalization, and reduction-heavy operators. One promising approach is to replace full-tensor synchronization for global operators with lightweight exchange of sufficient statistics. For example, softmax can synchronize the global maximum and the sum of exponentials, allowing each side to complete normalization locally without transferring the full tensor. Similar operator-aware synchronization may apply to attention score aggregation, normalization, and other reduction patterns. For transparent deployment, the corresponding challenge is to extend RRTO from fixed operator sequences to dynamic or partially dynamic execution. This requires online sequence classification, dependency-aware trace demultiplexing, and replay validation that can distinguish benign variation from semantic divergence. For training, the same direction suggests non-IID-aware row grouping and staleness control, so that ROG can account not only for wireless fluctuation and gradient importance but also for how different robots' data distributions affect convergence. The broader goal is to make granularity-aware execution applicable to model structures whose useful execution units are not obvious from static layers, rows, or repeated operator streams alone.

Future Direction 2: Communication-computation co-design over heterogeneous wireless MEC networks. A second direction is to redesign the three execution granularities for networks that are heterogeneous, mobile, and intermittently connected. ROG already shows that row-level synchronization can mitigate wireless stragglers, but future systems could further pipeline or decouple communication and computation so that robots continue useful local work while selected rows are transmitted over unstable links. In multi-radio environments, high-priority rows could be assigned to stable links while less urgent rows are delayed, compressed, or routed through secondary connections. LOPInfer could similarly adapt local-operation schedules across Wi-Fi, 5G, and device-to-device links, treating global-operator synchronization as a network-aware barrier rather than a fixed model event. RRTO could migrate replay state across edge servers during handoff and decide when the cost of migration exceeds the benefit of remote acceleration. Evaluating these designs requires testbeds beyond a single device–edge path, including UAVs, legged robots, mobile phones, multi-robot fleets, and replayed mobility traces. The key research question is how to choose execution

granularity jointly with link selection, handoff, and energy management rather than treating the network as an external bandwidth number.

Future Direction 3: Joint policy layers above granularity-aware primitives. A third direction is to compose the execution primitives in this thesis with higher-level system policies. For online training, this includes automatic selection of row granularity, staleness threshold, and transmission priority based on convergence progress, bandwidth variation, and non-IID data skew. For collaborative inference, this includes improving the local-operation scheduler with stronger optimization methods, approximation guarantees, or exact solvers for smaller subproblems, while integrating request-level scheduling for concurrent inference workloads. It also includes joint optimization with model compression, quantization, and early-exit policies, where the system decides not only where computation runs but also what numerical precision or accuracy target is appropriate under a latency and energy budget. For transparent deployment, the corresponding policy layer must arbitrate among multiple replaying models, recover from sequence mispredictions, and eventually support cross-framework offloading through semantic compatibility layers. These policies should not replace granularity-aware execution; rather, they should use rows, local operations, and operator-sequence replay as controllable primitives. Such a stack would move MEC ML systems from isolated optimizations toward deployable platforms that can train, infer, and offload under shared resource, accuracy, and compatibility constraints.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *Proc. OSDI*. Nov. 2016, pp. 265–283. DOI: 10.5555/3026877.3026899.

[2] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey". In: *IEEE Internet Things J.* 5.1 (Feb. 2018), pp. 450–465. DOI: 10.1109/JIOT.2017.2750180.

[3] M. Abdar, F. Pourpanah, S. Hussain, D. Rezazadegan, L. Liu, M. Ghavamzadeh, P. Fieguth, X. Cao, A. Khosravi, U. R. Acharya, V. Makarenkov, and S. Nahavandi. "A Review of Uncertainty Quantification in Deep Learning: Techniques, Applications and Challenges". In: *Inf. Fusion* 76 (Dec. 2021), pp. 243–297. DOI: 10.1016/j.inffus.2021.05.008.

[4] S. A. Ahson and M. Ilyas. *WiMAX: Standards and Security*. In collab. with M. Ilyas and S. A. Ahson. The WiMAX handbook. Baton Rouge: CRC Press, 2008. ISBN: 978-1-4200-4523-9. DOI: 10.1201/9781315219912.

[5] N. Armanfard, J. P. Reilly, and M. Komeili. "Local Feature Selection for Data Classification". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 38.6 (2015), pp. 1217–1227.

[6] M. Awais, F. Zhou, H. Xu, L. Hong, P. Luo, S.-H. Bae, and Z. Li. "Adversarial Robustness for Unsupervised Domain Adaptation". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021 IEEE/CVF International Conference on Computer Vision (ICCV). ISSN: 2380-7504. 2021, pp. 8548–8557. DOI: 10.1109/ICCV48922.2021.00845.

[7] S. Bai, J. Z. Kolter, and V. Koltun. "An Empirical Evaluation of Generic Convolutional and Recurrent Networks for Sequence Modeling". In: *arXiv preprint arXiv:1803.01271* (2018).

[8] R. Barandela, R. M. Valdovinos, J. S. Sánchez, and F. J. Ferri. "The Imbalanced Training Sample Problem: Under or over Sampling?" In: *Structural, Syntactic, and Statistical Pattern Recognition*. Ed. by A. Fred, T. M. Caelli, R. P. W. Duin, A. C. Campilho, and D. de Ridder. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 806–814. ISBN: 978-3-540-27868-9. DOI: 10.1007/978-3-540-27868-9_88.

[9] 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. DOI: 10.1109/ICCV.2019.00939.

[10] S. Bhattacharya, V. Rajan, and H. Shrivastava. "ICU Mortality Prediction: A Classification Algorithm for Imbalanced Datasets". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (Feb. 12, 2017). Number: 1. ISSN: 2374-3468. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10721 (visited on 10/09/2021).

[11] 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*. June 2024, pp. 412–424. DOI: 10.1145/3643730.3658760.

[12] R. Bonghi. *Jetson stats*. original-date: 2018-11-24T19:42:07Z. Apr. 21, 2022. URL: https://github.com/rbonghi/jetson_stats (visited on 04/22/2022).

[13] L. Bottou. "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Ed. by Y. Lechevallier and G. Saporta. Heidelberg: Physica-Verlag HD, 2010, pp. 177–186. ISBN: 978-3-7908-2604-3. DOI: 10.1007/978-3-7908-2604-3_16.

[14] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. "Language Models Are Few-Shot Learners". In: *Proc. NeurIPS*. Vol. 33. 2020, pp. 1877–1901.

[15] J. Cai, W. Liu, Z. Huang, and F. R. Yu. "Task Decomposition and Hierarchical Scheduling for Collaborative Cloud-Edge-End Computing". In: *IEEE Trans. Serv. Comput.* 17.6 (2024), pp. 4368–4382. DOI: 10.1109/TSC.2024.3402169.

[16] A.-Q. Cao and R. de Charette. "MonoScene: Monocular 3D Semantic Scene Completion". In: *Proc. CVPR*. June 2022, pp. 3991–4001. DOI: 10.1109/CVPR52688.2022.00396.

[17] P. Charalampopoulos, T. Kociumaka, S. P. Pissis, and J. Radoszewski. "Faster Algorithms for Longest Common Substring". In: *arXiv preprint arXiv:2105.03106* (2021).

[18] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam. "Rethinking Atrous Convolution for Semantic Image Segmentation". In: *arXiv preprint arXiv:1706.05587* (June 2017).

[19] M. Chen, D. Gündüz, K. Huang, W. Saad, M. Bennis, A. V. Feljan, and H. V. Poor. "Distributed Learning in Wireless Networks: Recent Progress and Future Challenges". In: *arXiv:2104.02151 [cs, math]* (Apr. 2021). arXiv: 2104.02151. URL: http://arxiv.org/abs/2104.02151 (visited on 09/07/2021).

[20] M. Chen, Z. Yang, W. Saad, C. Yin, H. V. Poor, and S. Cui. "A Joint Learning and Communications Framework for Federated Learning Over Wireless Networks". In: *IEEE Transactions on Wireless Communications* 20.1 (Jan. 2021). Conference Name: IEEE Transactions on Wireless Communications, pp. 269–283. ISSN: 1558-2248. DOI: 10.1109/TWC.2020.3024629.

[21] T. Chen and C. Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proc. KDD*. 2016, pp. 785–794.

[22] X. Chen, L. Zhang, X. Wei, and X. Lu. "An effective method using clustering-based adaptive decomposition and editing-based diversified oversamping for multi-class imbalanced datasets". In: *Applied Intelligence* 51.4 (Apr. 1, 2021), pp. 1918–1933. ISSN: 1573-7497. DOI: 10.1007/s10489-020-01883-1. URL: https://doi.org/10.1007/s10489-020-01883-1 (visited on 10/09/2021).

[23] 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". In: *IEEE Trans. Parallel Distrib. Syst.* 33.3 (Mar. 2022), pp. 683–697. DOI: 10.1109/TPDS.2021.3097417.

[24] 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". In: *IEEE Trans. Serv. Comput.* 18.4 (2025), pp. 2388–2402. DOI: 10.1109/TSC.2025.3577451.

[25] R. Chowdhury and D. Subramani. "Optimal Path Planning of Autonomous Marine Vehicles in Stochastic Dynamic Ocean Flows Using a GPU-Accelerated Algorithm". In: *IEEE J. Ocean. Eng.* 47.4 (Oct. 2022), pp. 864–879. DOI: 10.1109/JOE.2022.3145429.

[26] *CIFAR-10 and CIFAR-100 datasets*. URL: https://www.cs.toronto.edu/~kriz/cifar.html (visited on 07/25/2022).

[27] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. "Exploiting Bounded Staleness to Speed Up Big Data Analytics". In: 2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14). 2014, pp. 37–48. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/cui (visited on 10/09/2021).

[28] *CuPy: NumPy & SciPy for GPU*. URL: https://cupy.dev/ (visited on 07/02/2022).

[29] A. Dai, A. X. Chang, M. Savva, M. Halber, T. Funkhouser, and M. Nießner. "ScanNet: Richly-annotated 3D Reconstructions of Indoor Scenes". In: *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*. 2017.

[30] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Proc. NeurIPS*. Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Dec. 2022, pp. 16344–16359.

[31] *Datacenter Traffic Control: Understanding Techniques and Tradeoffs | IEEE Journals & Magazine | IEEE Xplore*. URL: https://ieeexplore.ieee.org/abstract/document/8207422 (visited on 03/21/2022).

[32] P. Davoodi, C. Gwon, G. Lai, and T. Morris. "TensorRT Inference with TensorFlow". In: *Proc. GPU Technol. Conf. (GTC)*. Mar. 2019.

[33] Z. DeVito. "TorchScript: Optimized Execution of PyTorch Programs". In: *Retrieved January* (2022).

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding". In: *arXiv preprint arXiv:1810.04805* (Oct. 2018).

[35] S. Dickson. *libtirpc: Transport Independent RPC library*. https://git.linux-nfs.org/?p=steved/libtirpc.git. 2024.

[36] M. Ding, P. Wang, D. Lopez-Perez, G. Mao, and Z. Lin. "Performance Impact of LoS and NLoS Transmissions in Dense Cellular Networks". In: *IEEE Transactions on Wireless Communications* 15.3 (Mar. 2016), pp. 2365–2380. ISSN: 1536-1276. DOI: 10.1109/TWC.2015.2503391. arXiv: 1503.04251. URL: http://arxiv.org/abs/1503.04251 (visited on 03/10/2022).

[37] M. Ding, P. Wang, D. López-Pérez, G. Mao, and Z. Lin. "Performance Impact of LoS and NLoS Transmissions in Dense Cellular Networks". In: *IEEE Trans. Wireless Commun.* 15.3 (Mar. 2016), pp. 2365–2380. DOI: 10.1109/TWC.2015.2496143.

[38] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale". In: *Proc. ICLR*. 2020.

[39] N. Eiling, J. Baude, S. Lankes, and A. Monti. "Cricket: A Virtualization Layer for Distributed Execution of CUDA Applications with Checkpoint/Restart Support". In: *Concurrency Comput. Pract. Exp.* 34.14 (May 2022), e6474. DOI: 10.1002/cpe.6474.

[40] *en:users:documentation:iw [Linux Wireless]*. URL: https://wireless.wiki.kernel.org/en/users/documentation/iw (visited on 04/22/2022).

[41] M. Everett, Y. F. Chen, and J. P. How. "Collision Avoidance in Pedestrian-Rich Environments With Deep Reinforcement Learning". In: *IEEE Access* 9 (2021). Conference Name: IEEE Access, pp. 10357–10377. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3050338.

[42] S. Eyerman and I. Hur. "Efficient Asynchronous RPC Calls for Microservices: DeathStarBench Study". In: *arXiv preprint arXiv:2209.13265* (2022).

[43] Z. Fang, T. Yu, O. J. Mengshoel, and R. K. Gupta. "QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks". In: *Proc. CIKM*. Nov. 2017, pp. 2067–2070. DOI: 10.1145/3132847.3133123.

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

[45] A. V. Gerbessiotis and L. G. Valiant. "Direct bulk-synchronous parallel algorithms". In: *Algorithm Theory — SWAT '92*. Ed. by O. Nurmi and E. Ukkonen. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1992, pp. 1–18. ISBN: 978-3-540-47275-9. DOI: 10.1007/3-540-55706-7_1.

[46] 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*. May 2023, pp. 222–235. DOI: 10.1109/IoTDI57440.2023.00025.

[47] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen. "VecQ: Minimal Loss DNN Model Compression with Vectorized Weight Quantization". In: *IEEE Trans. Comput.* 70.5 (May 2021), pp. 696–710. DOI: 10.1109/TC.2020.2999683.

[48] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *arXiv preprint arXiv:1512.03385* (Dec. 2015).

[49] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server". In: *Advances in Neural Information Processing Systems*. Vol. 26. Curran Associates, Inc., 2013. URL: https://proceedings.neurips.cc/paper/2013/hash/b7bb35b9c6ca2aee2df08cf09d7016c2-Abstract.html (visited on 10/09/2021).

[50] H.-K. Hsu, C.-H. Yao, Y.-H. Tsai, W.-C. Hung, H.-Y. Tseng, M. Singh, and M.-H. Yang. "Progressive Domain Adaptation for Object Detection". In: 2020, pp. 749–757. URL: https://openaccess.thecvf.com/content_WACV_2020/html/Hsu_Progressive_Domain_Adaptation_for_Object_Detection_WACV_2020_paper.html (visited on 09/05/2021).

[51] 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. DOI: 10.1109/INFOCOM.2019.8737434.

[52] H. Hu, D. Wang, and C. Wu. "Distributed Machine Learning through Heterogeneous Edge Systems". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.5 (Apr. 3, 2020), pp. 7179–7186. ISSN: 2374-3468, 2159-5399. DOI: 10.1609/aaai.v34i05.6207. URL: https://aaai.org/ojs/index.php/AAAI/article/view/6207 (visited on 10/08/2021).

[53] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. *Mobile Edge Computing—A Key Technology Towards 5G*. ETSI White Paper 11. European Telecommunications Standards Institute, Sept. 2015. URL: https://www.etsi.org/images/files/ETSIWhitePapers/etsi_wp11_mec_a_key_technology_towards_5g.pdf.

[54] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger. "Densely Connected Convolutional Networks". In: *arXiv preprint arXiv:1608.06993* (Aug. 2016).

[55] Z. Huang and Y. Li. "Interpretable and Accurate Fine-Grained Recognition via Region Grouping". In: *Proc. CVPR*. 2020, pp. 8662–8672.

[56] *IEEE 802.11ac-2013*. In: *Wikipedia*. Page Version ID: 1076948038. Mar. 13, 2022. URL: https://en.wikipedia.org/w/index.php?title=IEEE_802.11ac-2013&oldid=1076948038 (visited on 03/22/2022).

[57] *iPerf - Download iPerf3 and Original iPerf Pre-Compiled Binaries*. https://iperf.fr/iperf-download.php. 2024.

[58] *iPerf - Download iPerf3 and original iPerf pre-compiled binaries*. URL: https://iperf.fr/iperf-download.php (visited on 02/02/2022).

[59] *Jetson Modules*. Oct. 2020. URL: https://developer.nvidia.com/embedded/jetson-modules (visited on 02/06/2022).

[60] J. Ji, K. Zhu, and L. Cai. "Trajectory and Communication Design for Cache-Enabled UAVs in Cellular Networks: A Deep Reinforcement Learning Approach". In: *IEEE Trans. Mobile Comput.* 22.10 (Oct. 2023), pp. 6190–6204. DOI: 10.1109/TMC.2022.3188661.

[61] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang. "CoDL: Efficient CPU-GPU Co-Execution for Deep Learning Inference on Mobile Devices". In: *Proc. MobiSys*. June 2022, pp. 209–221. DOI: 10.1145/3498361.3538931.

[62] B. Jiang, Z. Zhang, D. Lin, J. Tang, and B. Luo. "Semi-Supervised Learning with Graph Learning-Convolutional Networks". In: *Proc. CVPR*. 2019, pp. 11313–11320.

[63] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, K. Michael, T. Xie, J. Fang, imyhxy, Lorna, Z. Yifu, C. Wong, A. V, D. Montes, Z. Wang, C. Fati, J. Nadar, Laughing, UnglvKitDe, V. Sonck, tkianai, yxNONG, P. Skalski, A. Hogan, D. Nair, M. Strobel, and M. Jain. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. Version v7.0. Nov. 2022. DOI: 10.5281/zenodo.7347926.

[64] 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. DOI: 10.1145/3037697.3037698.

[65] S. Kato. "Implementing Open-Source CUDA Runtime". In: *Proc. of the 54the Programming Symposium*. Hakone, Japan, Jan. 2013, pp. 111–118.

[66] H. Kaur, H. S. Pannu, and A. K. Malhi. "A Systematic Review on Imbalanced Data Challenges in Machine Learning: Applications and Solutions". In: *ACM Computing Surveys* 52.4 (Aug. 30, 2019), 79:1–79:36. ISSN: 0360-0300. DOI: 10.1145/3343440. URL: https://doi.org/10.1145/3343440 (visited on 10/09/2021).

[67] 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". In: *Computer* 36.12 (Dec. 2003), pp. 68–75.

[68] N. Kourtellis, K. Katevas, and D. Perino. "FLaaS: Federated Learning as a Service". In: *Proceedings of the 1st Workshop on Distributed Machine Learning*. DistributedML'20. New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 7–13. ISBN: 978-1-4503-8182-6. DOI: 10.1145/3426745.3431337. URL: https://doi.org/10.1145/3426745.3431337 (visited on 09/20/2021).

[69] A. Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. University of Toronto, 2009.

[70] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. "DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices". In: *Proc. IPSN*. Apr. 2016, pp. 1–12. DOI: 10.1109/IPSN.2016.7460726.

[71] 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*. Sept. 2020. DOI: 10.1145/3372224.3419193.

[72] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou. "Dagger: Efficient and Fast RPCs in Cloud Microservices With Near-Memory Reconfigurable NICs". In: *Proc. ASPLOS*. ACM, 2021, pp. 36–51.

[73] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, et al. "Retrieval-Augmented Generation for

Knowledge-Intensive NLP Tasks". In: *Proc. NeurIPS*. Vol. 33. 2020, pp. 9459–9474.

[74] J. Li, A. Hassani, S. Walton, and H. Shi. "ConvMLP: Hierarchical Convolutional MLPs for Vision". In: *arXiv:2109.04454 [cs]* (Sept. 18, 2021). arXiv: 2109.04454. URL: http://arxiv.org/abs/2109.04454 (visited on 01/23/2022).

[75] 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*. June 2019, pp. 10855–10864. DOI: 10.1109/CVPR.2019.01112.

[76] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. "Communication Efficient Distributed Machine Learning with the Parameter Server". In: *Advances in Neural Information Processing Systems*. Vol. 27. Curran Associates, Inc., 2014. URL: https://proceedings.neurips.cc/paper/2014/hash/1ff1de774005f8da13f42943881c655f-Abstract.html (visited on 07/01/2022).

[77] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. "Scaling Distributed Machine Learning with the Parameter Server". In: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14). 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu (visited on 10/09/2021).

[78] W. Li and A. McCallum. "Pachinko allocation: DAG-structured mixture models of topic correlations". In: *Proceedings of the 23rd international conference on Machine learning - ICML '06*. Pittsburgh, Pennsylvania: ACM Press, 2006, pp. 577–584. ISBN: 978-1-59593-383-6. DOI: 10.1145/1143844.1143917. URL: http://portal.acm.org/citation.cfm?doid=1143844.1143917 (visited on 01/23/2022).

[79] Y. Li, M. Yu, S. Li, S. Avestimehr, N. S. Kim, and A. Schwing. "Pipe-SGD: A Decentralized Pipelined SGD Framework for Distributed Deep Net Training". In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/2c6a0bae0f071cbbf0bb3d5b11d90a82-Abstract.html (visited on 07/01/2022).

[80] 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". In: *IEEE Trans. Cloud Comput.* (May 2023).

[81] L. Lin, X. Liao, H. Jin, and P. Li. "Computation Offloading Toward Edge Computing". In: *Proc. IEEE* 107.8 (Aug. 2019), pp. 1584–1607. DOI: 10.1109/JPROC.2019.2922285.

[82] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. "Focal Loss for Dense Object Detection". In: *arXiv preprint arXiv:1708.02002* (Aug. 2017).

[83] Y. Lin, S. Han, H. Mao, Y. Wang, and W. J. Dally. "DEEP GRADIENT COMPRESSION: REDUCING THE COMMUNICATION BANDWIDTH FOR DISTRIBUTED TRAINING". In: (2018), p. 14.

[84] 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. 1. Mar. 2023, pp. 1–25. DOI: 10.1145/3579340.

[85] W. Liu, L. Chen, Y. Chen, and W. Zhang. "Accelerating Federated Learning via Momentum Gradient Descent". In: *IEEE Transactions on Parallel and Distributed Systems* 31.8 (2020). Conference Name: IEEE Transactions on Parallel and Distributed Systems, pp. 1754–1766. ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.2975189.

[86] X. Liu, Z. Guo, S. Li, F. Xing, J. You, C.-C. J. Kuo, G. El Fakhri, and J. Woo. "Adversarial Unsupervised Domain Adaptation with Conditional and Label Shift: Infer, Align and Iterate". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. 2021 IEEE/CVF International Conference on Computer Vision (ICCV). ISSN: 2380-7504. 2021, pp. 10347–10356. DOI: 10.1109/ICCV48922.2021.01020.

[87] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie. "A ConvNet for the 2020s". In: *arXiv preprint arXiv:2201.03545* (Jan. 2022).

[88] J. Long, E. Shelhamer, and T. Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *arXiv preprint arXiv:1411.4038* (Nov. 2015).

[89] F. Luo, S. Khan, Y. Huang, and K. Wu. "Binarized Neural Network for Edge Intelligence of Sensor-Based Human Activity Recognition". In: *IEEE Trans. Mobile Comput.* 22.3 (Mar. 2023), pp. 1356–1368. DOI: 10.1109/TMC.2021.3054403.

[90] P. Mach and Z. Becvar. "Mobile Edge Computing: A Survey on Architecture and Computation Offloading". In: *IEEE Commun. Surv. Tutor.* 19.3 (Aug. 2017), pp. 1628–1656. DOI: 10.1109/COMST.2017.2682318.

[91] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective". In: *IEEE Commun. Surv. Tutor.* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201.

[92] S. Marcel and Y. Rodriguez. "Torchvision the Machine-Vision Package of Torch". In: *Proc. ACM Multimedia*. Oct. 2010, pp. 1485–1488. DOI: 10.1145/1873951.1874254.

[93] A. Masiukiewicz. "Throughput Comparison between The New HEW 802.11 ax Standard and 802.11 n/ac Standards in Selected Distance Windows". In: *Int. J. Electron. Telecommun.* 65.1 (2019), pp. 79–84.

[94] A. Masiukiewicz. "Throughput comparison between the new HEW 802.11ax standard and 802.11n/ac standards in selected distance windows". In: *International Journal of Electronics and Telecommunications* 65.1 (Feb. 16, 2019). Number: 1, pp. 79–84. ISSN: 2300-1933. URL: http://www.ijet.pl/index.php/ijet/article/view/10.24425-ijet.2019.126286 (visited on 03/10/2022).

[95] 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*. June 2022, pp. 337–346. DOI: 10.1109/WoWMoM54355.2022.00032.

[96] 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*. Oct. 2022, pp. 37–54.

[97] *MLPerf Mobile Benchmarks*. https://mlcommons.org/working-groups/benchmarks/mobile/. 2023.

[98] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Di Francesco. "Distributed Inference Acceleration with Adaptive DNN Partitioning and Offloading". In: *Proc. INFOCOM*. July 2020, pp. 854–863. DOI: 10.1109/INFOCOM41043.2020.9155465.

[99] M. Mounesan, X. Zhang, and S. Debroy. "Infer-EDGE: Dynamic DNN Inference Optimization in 'Just-in-Time' Edge-AI Implementations". In: *arXiv preprint arXiv:2501.18842* (2025).

[100] A. Munshi. "The OpenCL Specification". In: *Proc. IEEE Hot Chips 21 Symp. (HCS)*. Aug. 2009.

[101] Nam Sung Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, Jie S. Hu, M. Irwin, M. Kandemir, and V. Narayanan. "Leakage current: Moore's law meets static power". In: *Computer* 36.12 (Dec. 2003), pp. 68–75. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1250885. URL: http://ieeexplore.ieee.org/document/1250885/ (visited on 04/22/2022).

[102] Z. Ning, M. Vandersteegen, K. Van Beeck, T. Goedemé, and P. Vandewalle. "Power Consumption Benchmark for Embedded AI Inference". In: *Proc. Int. Conf. Appl. Comput. WWW/Internet (AC)*. Jan. 2024, pp. 3–10. DOI: 10.33965/ac2024_202401L001.

[103] M. Noormohammadpour and C. S. Raghavendra. "Datacenter Traffic Control: Understanding Techniques and Tradeoffs". In: *IEEE Commun. Surv. Tutor.* 20.2 (June 2018), pp. 1492–1525. DOI: 10.1109/COMST.2017.2782753.

[104] *NumPy*. URL: https://numpy.org/ (visited on 07/02/2022).

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

[106] NVIDIA. *Jetson Xavier NX Series: The World's Smallest AI Supercomputer*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx-developer-kit/. 2024.

[107] NVIDIA. *NVIDIA Nsight Compute Documentation*. https://docs.nvidia.com/nsight-compute/index.html. 2024.

[108] NVIDIA. *pynvml: Python Bindings for the NVIDIA Management Library*. https://developer.nvidia.com/management-library-nvml/. 2024.

[109] *NVIDIA Jetson Xavier NX GPU Specs*. URL: https://www.techpowerup.com/gpu-specs/jetson-xavier-nx-gpu.c3642 (visited on 02/07/2022).

[110] *On the shoulders of giants: recent changes in Internet traffic*. The Cloudflare Blog. Mar. 17, 2020. URL: http://blog.cloudflare.com/on-the-shoulders-of-giants-recent-changes-in-internet-traffic/ (visited on 03/21/2022).

[111] J. Park, S. Samarakoon, A. Elgabli, J. Kim, M. Bennis, S.-L. Kim, and M. Debbah. "Communication-Efficient and Distributed Learning Over Wireless Networks: Principles and Applications". In: *Proceedings of the IEEE* 109.5 (May 2021). Conference Name: Proceedings of the IEEE, pp. 796–819. ISSN: 1558-2256. DOI: 10.1109/JPROC.2021.3055679.

[112] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M.

Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *arXiv preprint arXiv:1912.01703* (Dec. 2019).

[113] Y. Pei, M. W. Mutka, and N. Xi. "Connectivity and bandwidth-aware real-time exploration in mobile robot networks". In: *Wireless Communications and Mobile Computing* 13.9 (2013). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/wcm.1145, pp. 847–863. ISSN: 1530-8677. DOI: 10.1002/wcm.1145. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcm.1145 (visited on 03/10/2022).

[114] B. Plancher, S. M. Neuman, T. Bourgeat, S. Kuindersma, S. Devadas, and V. J. Reddi. "Accelerating Robot Dynamics Gradients on a CPU, GPU, and FPGA". In: *IEEE Robot. Autom. Lett.* 6.2 (Apr. 2021), pp. 2335–2342. DOI: 10.1109/LRA.2021.3057845.

[115] R. Poorzare and A. C. Augé. "How Sufficient Is TCP When Deployed in 5G mmWave Networks over the Urban Deployment?" In: *IEEE Access* 9 (Mar. 2021), pp. 36342–36355. DOI: 10.1109/ACCESS.2021.3063623.

[116] *PyTorch*. 2021. URL: https://www.pytorch.org (visited on 10/09/2021).

[117] B. Qiao, M. A. Özkan, J. Teich, and F. Hannig. "The Best of Both Worlds: Combining CUDA Graph with an Image Processing DSL". In: *Proc. DAC*. 2020, pp. 1–6.

[118] A. K. Qin, V. L. Huang, and P. N. Suganthan. "Differential Evolution Algorithm with Strategy Adaptation for Global Numerical Optimization". In: *IEEE Trans. Evol. Comput.* 13.2 (Apr. 2008), pp. 398–417.

[119] J. P. Queralta, J. Taipalmaa, B. Can Pullinen, V. K. Sarker, T. Nguyen Gia, H. Tenhunen, M. Gabbouj, J. Raitoharju, and T. Westerlund. "Collaborative Multi-Robot Search and Rescue: Planning, Coordination, Perception, and Active Vision". In: *IEEE access* 8 (2020). Publisher: IEEE, pp. 191617–191643. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3030190.

[120] RaspberryPi. *Raspberry Pi*. https://www.raspberrypi.com/documentation/computers/configuration.html#power-consumption. 2022.

[121] S. Ren, K. He, R. Girshick, and J. Sun. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *arXiv preprint arXiv:1506.01497* (June 2015).

[122] 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". In: *IEEE Trans. Veh. Technol.* 67.12 (Dec. 2018), pp. 12095–12108.

[123] V. Rosenfeld, S. Breß, and V. Markl. "Query Processing on Heterogeneous CPU/GPU Systems". In: *ACM Comput. Surv.* 55.1 (Jan. 2023), pp. 1–38. DOI: 10.1145/3514272.

[124] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. "Mobile-Edge Computing Architecture: The Role of MEC in the Internet of Things". In: *IEEE Consumer Electronics Magazine* 5.4 (Oct. 2016), pp. 84–91. DOI: 10.1109/MCE.2016.2590118.

[125] sadmin. *InfiniBand - A low-latency, high-bandwidth interconnect*. InfiniBand Trade Association. URL: https://www.infinibandta.org/about-infiniband/ (visited on 07/01/2022).

[126] A. N. Saridena and A. Choromanska. "DNN Patching: Progressive Fixing and Augmenting the Functionalities of DNNs for Autonomous Vehicles". In: *IEEE Robot. Autom. Lett.* 7.2 (Apr. 2022), pp. 3257–3264. DOI: 10.1109/LRA.2022.3143598.

[127] N. I. Sarkar and O. Mussa. "The effect of people movement on Wi-Fi link throughput in indoor propagation environments". In: *IEEE 2013 Tencon - Spring*. IEEE 2013 Tencon - Spring. Apr. 2013, pp. 562–566. DOI: 10.1109/TENCONSpring.2013.6584508.

[128] M. Schneider, F. Haag, A. K. Khalil, and D. A. Breunig. "Evaluation of Communication Technologies for Distributed Industrial Control Systems: Concept and Evaluation of 5G and WiFi 6". In: *Procedia CIRP* 107 (2022), pp. 588–593.

[129] M. Shafi, A. F. Molisch, P. J. Smith, T. Haustein, P. Zhu, P. De Silva, F. Tufvesson, A. Benjebbour, and G. Wunder. "5G: A Tutorial Overview of Standards, Trials, Challenges, Deployment, and Practice". In: *IEEE Journal on Selected Areas in Communications* 35.6 (June 2017). Conference Name: IEEE Journal on Selected Areas in Communications, pp. 1201–1221. ISSN: 1558-0008. DOI: 10.1109/JSAC.2017.2692307.

[130] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer". In: *Proc. ICLR*. 2017.

[131] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges". In: *IEEE Internet Things J.* 3.5 (Oct. 2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[132] W. Shi, S. Zhou, and Z. Niu. "Device Scheduling with Fast Convergence for Wireless Federated Learning". In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. ISSN: 1938-1883. June 2020, pp. 1–6. DOI: 10.1109/ICC40277.2020.9149138.

[133] K. Simonyan and A. Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv preprint arXiv:1409.1556* (Sept. 2014).

[134] A. Singhvi, A. Akella, M. Anderson, R. Cauble, H. Deshmukh, D. Gibson, M. M. Martin, A. Strominger, T. F. Wenisch, and A. Vahdat. "Cliquemap: Productionizing an RMA-Based Distributed Caching System". In: *Proc. SIGCOMM*. ACM, 2021, pp. 93–105.

[135] S. Siva and H. Zhang. "Robot perceptual adaptation to environment changes for long-term human teammate following". In: *The International Journal of Robotics Research* (Jan. 2020). Publisher: SAGE Publications Ltd STM, p. 0278364919896625. ISSN: 0278-3649. DOI: 10.1177/0278364919896625. URL: https://doi.org/10.1177/0278364919896625 (visited on 09/05/2021).

[136] E. Sucar, S. Liu, J. Ortiz, and A. J. Davison. "iMAP: Implicit Mapping and Positioning in Real-Time". In: *2021 IEEE/CVF International Conference on Computer*

Vision (ICCV). 2021 IEEE/CVF International Conference on Computer Vision (ICCV). Montreal, QC, Canada: IEEE, Oct. 2021, pp. 6209–6218. ISBN: 978-1-66542-812-5. DOI: 10.1109/ICCV48922.2021.00617. URL: https://ieeexplore.ieee.org/document/9710431/ (visited on 06/15/2022).

[137] 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". In: *IEEE Trans. Serv. Comput.* 17.4 (2024), pp. 1548–1564. DOI: 10.1109/TSC.2024.3355937.

[138] J. Sun, T. Chen, G. Giannakis, and Z. Yang. "Communication-Efficient Distributed Learning via Lazily Aggregated Quantized Gradients". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019. URL: https://papers.nips.cc/paper/2019/hash/4e87337f366f72daa424dae11df0538c-Abstract.html (visited on 10/05/2021).

[139] I. Sutskever, O. Vinyals, and Q. V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Proc. NeurIPS*. Vol. 27. 2014.

[140] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration". In: *IEEE Commun. Surv. Tutor.* 19.3 (2017), pp. 1657–1681. DOI: 10.1109/COMST.2017.2705720.

[141] 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". In: *IEEE Trans. Serv. Comput.* 18.4 (2025), pp. 1966–1981. DOI: 10.1109/TSC.2025.3586093.

[142] S. Targ, D. Almeida, and K. Lyman. "ResNet in ResNet: Generalizing Residual Architectures". In: *arXiv preprint arXiv:1603.08029* (Mar. 2016).

[143] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. N. Paravecino. "Efficient Algorithms for Device Placement of DNN Graph Operators". In: *Proc. NeurIPS*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin. Vol. 33. Dec. 2020, pp. 15451–15463.

[144] *tff.simulation.datasets.cifar100.load_data | TensorFlow Federated*. 2022. URL: https://www.tensorflow.org/federated/api_docs/python/tff/simulation/datasets/cifar100/load_data (visited on 01/23/2022).

[145] *The World's Smallest AI Supercomputer*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/ (visited on 02/14/2022).

[146] Y. Tian, K. Xu, and N. Ansari. "TCP in Wireless Environments: Problems and Solutions". In: *IEEE Commun. Mag.* 43.3 (Mar. 2005), S27–S32. DOI: 10.1109/MCOM.2005.1404592.

[147] Y. Tian, K. Pei, S. Jana, and B. Ray. "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 303–314. ISBN: 9781450356381. DOI:

10.1145/3180155.3180220. URL: https://doi.org/10.1145/3180155.3180220.

[148] *torch.optim — PyTorch 1.11.0 documentation*. URL: https://pytorch.org/docs/stable/optim.html (visited on 03/24/2022).

[149] S. Tyagi and P. Sharma. "Taming Resource Heterogeneity In Distributed ML Training With Dynamic Batching". In: *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS). Aug. 2020, pp. 188–194. DOI: 10.1109/ACSOS49614.2020.00041.

[150] E. Vidal, J. D. Hernández, N. Palomeras, and M. Carreras. "Online Robotic Exploration for Autonomous Underwater Vehicles in Unstructured Environments". In: *2018 OCEANS - MTS/IEEE Kobe Techno-Oceans (OTO)*. 2018 OCEANS - MTS/IEEE Kobe Techno-Oceans (OTO). May 2018, pp. 1–4. DOI: 10.1109/OCEANSKOBE.2018.8559224.

[151] H. Wang, Y. Yang, and B. Liu. "GMC: Graph-Based Multi-View Clustering". In: *IEEE Trans. Knowledge Data Eng.* 32.6 (2019), pp. 1116–1129.

[152] 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. DOI: 10.1145/3620665.3640407.

[153] 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, pp. 4494–4501. DOI: 10.1109/ICRA48891.2024.10565321.

[154] L. Wang and K.-J. Yoon. "Knowledge Distillation and Student-Teacher Learning for Visual Intelligence: A Review and New Outlooks". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 44.6 (June 2022), pp. 3048–3068. DOI: 10.1109/TPAMI.2021.3055560.

[155] M. Wang and W. Deng. "Deep visual domain adaptation: A survey". In: *Neurocomputing* 312 (Oct. 2018), pp. 135–153. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2018.05.083. URL: https://www.sciencedirect.com/science/article/pii/S0925231218306684 (visited on 09/20/2021).

[156] X. Wang, J. Wei, D. Schuurmans, Q. V. Le, E. H. Chi, S. Narang, A. Chowdhery, and D. Zhou. "Self-Consistency Improves Chain of Thought Reasoning in Language Models". In: *Proc. ICLR*. May 2023.

[157] 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". In: *IEEE Trans. Serv. Comput.* 18.2 (2025), pp. 1039–1053. DOI: 10.1109/TSC.2025.3528346.

[158] G. Wilson and D. J. Cook. "A Survey of Unsupervised Deep Domain Adaptation". In: *ACM Transactions on Intelligent Systems and Technology* 11.5 (July 2020), 51:1–51:46. ISSN: 2157-6904. DOI: 10.1145/3400066. URL: https://doi.org/10.1145/3400066 (visited on 10/04/2021).

[159] J. Woo and N. Kim. "Collision avoidance for an unmanned surface vehicle using deep reinforcement learning". In: *Ocean Engineering* 199 (Mar. 2020), p. 107001. ISSN: 0029-8018. DOI: 10.1016/j.oceaneng.2020.107001. URL: https://www.sciencedirect.com/science/article/pii/S0029801820300792 (visited on 08/23/2021).

[160] 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*. June 2023, pp. 16133–16142. DOI: 10.1109/CVPR52729.2023.01551.

[161] M. Wulfmeier, A. Bewley, and I. Posner. "Addressing appearance change in outdoor robotics with adversarial domain adaptation". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). ISSN: 2153-0866. Sept. 2017, pp. 1551–1558. DOI: 10.1109/IROS.2017.8205961.

[162] M. Wulfmeier, A. Bewley, and I. Posner. "Incremental Adversarial Domain Adaptation for Continually Changing Environments". In: *2018 IEEE International Conference on Robotics and Automation (ICRA)*. ISSN: 2577-087X. May 2018, pp. 4489–4495. DOI: 10.1109/ICRA.2018.8460982.

[163] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He. "Aggregated Residual Transformations for Deep Neural Networks". In: *arXiv preprint arXiv:1611.05431* (Jan. 2017).

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

[165] K.-M. Yang, J.-B. Han, and K.-H. Seo. "A Multi-robot Control System based on ROS for Exploring Disaster Environment". In: *2019 7th International Conference on Control, Mechatronics and Automation (ICCMA)*. 2019 7th International Conference on Control, Mechatronics and Automation (ICCMA). Nov. 2019, pp. 168–173. DOI: 10.1109/ICCMA46720.2019.8988650.

[166] 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. DOI: 10.1145/3544216.3544250.

[167] J. Yao, S. S. Kanhere, and M. Hassan. "An Empirical Study of Bandwidth Predictability in Mobile Computing". In: *Proc. WiNTECH*. Sept. 2008, pp. 11–18. DOI: 10.1145/1410077.1410081.

[168] X. Yi. "A Study of Performance Programming of CPU, GPU accelerated Computers and SIMD Architecture". In: *arXiv preprint arXiv:2409.10661* (2024).

[169] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu. "DeepSlicing: Collaborative and Adaptive CNN Inference with Low Latency". In: *IEEE Trans. Parallel Distrib. Syst.* 32.9 (Sept. 2021), pp. 2175–2187. DOI: 10.1109/TPDS.2021.3058532.

[170] Y. Zhang, D. Shi, Y. Wu, Y. Zhang, L. Wang, and F. She. "Networked Multi-robot Collaboration in Cooperative–Competitive Scenarios Under Communication Interference". en. In: *Collaborative Computing: Networking, Applications and Worksharing*. Ed. by H. Gao, X. Wang, M. Iqbal, Y. Yin, J. Yin, and N. Gu. Vol. 349.

Cham: Springer International Publishing, 2021, pp. 601–619. ISBN: 978-3-030-67536-3 978-3-030-67537-0. DOI: 10.1007/978-3-030-67537-0_36. (Visited on 03/30/2021).

[171] Z. Zhao, R. Zhao, J. Xia, X. Lei, D. Li, C. Yuen, and L. Fan. "A Novel Framework of Three-Hierarchical Offloading Optimization for MEC in Industrial IoT Networks". In: *IEEE Trans. Ind. Informat.* 16.8 (2019), pp. 5424–5434.

[172] 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". In: *IEEE Trans. Serv. Comput.* 18.2 (2025), pp. 1097–1109. DOI: 10.1109/TSC.2025.3536319.

[173] Z. Zhu, S. Peng, V. Larsson, W. Xu, H. Bao, Z. Cui, M. R. Oswald, and M. Pollefeys. *NICE-SLAM: Neural Implicit Scalable Encoding for SLAM*. Number: arXiv:2112.12130. Apr. 21, 2022. DOI: 10.48550/arXiv.2112.12130. arXiv: 2112.12130[cs]. URL: http://arxiv.org/abs/2112.12130 (visited on 06/21/2022).

[174] Z. Zhu, S. Peng, V. Larsson, W. Xu, H. Bao, Z. Cui, M. R. Oswald, and M. Pollefeys. "NICE-SLAM: Neural Implicit Scalable Encoding for SLAM". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2022.