RRTO: A High-Performance Transparent Offloading System for Model Inference in Mobile Edge Computing

Zekai Sun, Xiuxian Guan, Zheng Lin, Yuhao Qing, Haoze Song, Zihan Fang, Zhe Chen, *Member, IEEE,* Fangming Liu, *Senior Member, IEEE,* Heming Cui, *Member, IEEE*, Wei Ni, *Fellow, IEEE*, Jun Luo, *Fellow, IEEE*

***Abstract*—Deploying Machine Learning (ML) applications on resource-constrained mobile devices remains challenging due to limited computational resources and poor platform compatibility. While Mobile Edge Computing (MEC) offers an offloading-based inference paradigm using GPU servers, existing approaches can be divided into non-transparent and transparent methods, with the former necessitating source-code modifications. Non-transparent offloading achieves high performance but requires intrusive code changes, limiting compatibility with diverse applications. Transparent offloading, in contrast, offers wide compatibility but introduces two challenges: performance degradation under MEC's low and fluctuating bandwidth, where per-operator remote procedure calls (RPCs) inflate end-to-end latency and energy consumption; and black-box logic reconstruction, which requires recovering the operator sequence solely from low-level runtime logs without hints from upper-layer frameworks. To address these challenges, we propose RRTO, the first high-performance transparent offloading system tailored for MEC inference that exploits the static operator sequence in ML models to eliminate repetitive RPCs. RRTO applies record-and-replay to coalesce reactive per-operator RPCs into a proactive one-shot RPC per inference, and introduces a two-stage Operator Sequence Search to reconstruct the operator sequence accurately from raw logs. Evaluation demonstrates that RRTO reduces per-inference latency and energy consumption by up to 98% relative to state-of-the-art transparent methods while achieving performance comparable to non-transparent approaches, without requiring any source-code modification.**

***Index Terms*—Computation Offloading, model inference, mobile edge computing, distributed system and network**

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

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

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

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

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

W. Ni is with Data61, CSIRO, Marsfield, NSW 2122, Australia, and the School of Computing Science and Engineering, and the University of New South Wales, Kennington, NSW 2052, Australia (e-mail: wei.ni@ieee.org).

J. Luo is with the School of Computer Engineering, Nanyang Technological University, Singapore (e-mail: junluo@ntu.edu.sg).

(Corresponding author: Zheng Lin and Heming Cui)

I. Introduction

Machine learning (ML) has become fundamental to a wide range of mobile applications, from intelligent wearable sensors [1] and autonomous vehicles [2] to industrial IoT systems [3]. These applications are built upon advances in object detection [4], robotic control [5], and environmental perception [6], 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 [7] 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. I) 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. II-A). 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 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 [10] and CUDA-X libraries [11]) and runtime-optimized model structures (e.g., Just-In-Time compilation [12]) where code modifications are often infeasible or prohibitive. While alternatives like CUDA Graph [8] and TorchScript [13] have attempted to mitigate the

Method	Category	Code Modification	Application Diversity	Platform Compatibility	High Performance
Device-only Inference	Device-only	N/A	✓	✗	✗
Native Offloading	Non-transparent	High	✗	✓	✓
CUDA Graph [8]	Non-transparent	Low	✗	✗	✓
Cricket [9]	Transparent	N/A	✓	✓	✗
RRTO (Ours)	Transparent	N/A	✓	✓	✓

TABLE I: Comparison of representative inference methods in MEC.

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. II-B).

Transparent offloading methods [9] 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 [14]), which are typically forwarded to backend system functions (e.g., *aten::add* and *aten::conv2d* within CUDA's "cudaLaunchKernel" [15]). 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. II-C).

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. II-C2) 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 [4]), which in turn yield thousands of RPC invocations per inference (e.g., 5,895 in [4]), 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 [16], so communication delay from one-by-one handling can dominate end-to-end inference time (up to 95% in [9]). 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. III-C1). 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 paper 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.

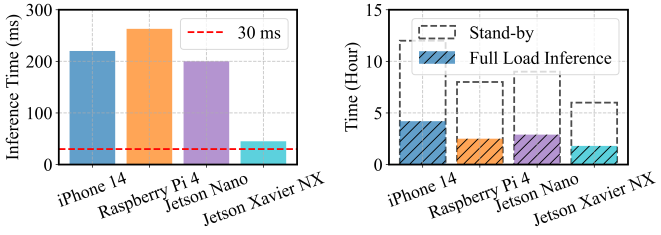
The rest of the paper is organized as follows. Sec. II motivates the design of RRTO by revealing the challenges in current MEC networks. Sec. III presents the system design of RRTO. Sec. IV introduces the system implementation, followed by performance evaluation in Sec. V. Related works and technical limitations are discussed in Sec. VI. Finally, conclusions are presented in Sec. VII.

II. Background

A. 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 [14], TensorFlow [17], CUDA [15], and OpenCL [18]) and mobile devices are equipped with various hardware accelerators (e.g., GPU [19], FPGA [20], and SoC [21]), 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. 1a, the inference latency on various mobile devices exceeds the 30 ms threshold required for smooth video fluency [22] (indicated by the red dotted line). Fig. 1b shows that frequent inference reduces device standby time to 20–40% of their normal duration, degrading user experience and device usability.



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

Fig. 1: The performance of VGG-16 under device-only inference across different mobile devices [23]–[26].

B. 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 [10] and CUDA-X library [11]) 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 [12] and dynamic conventional kernel selection with different numbers/sizes based on task-specific requirements [27]. 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 [8] and TorchScript [13], 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 [28] and multiple inference scheduling [3], [29], [30]), unlike native 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. VI).

C. Transparent Offloading

1) 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. 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 [15].
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 [9] 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,

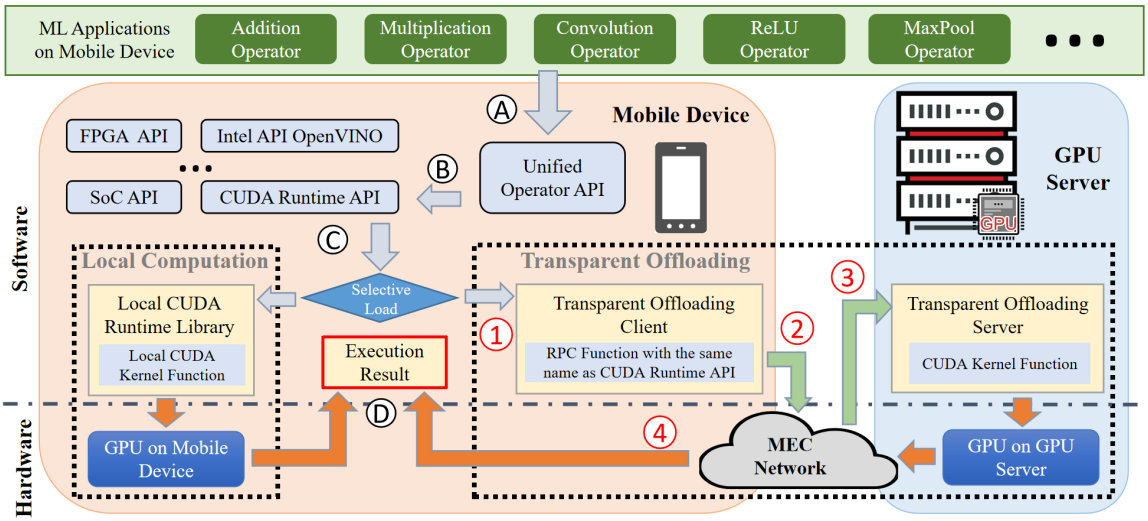


Fig. 2: Workflow of Transparent Offloading System for Model Inference in MEC.

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. 2) proceeds as follows:

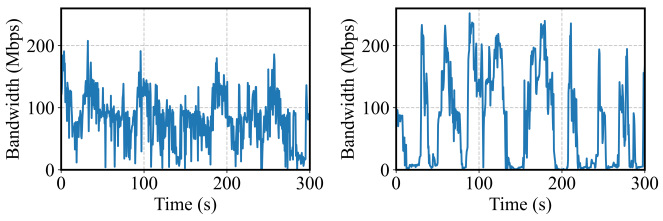
(1) The modified dynamic link library ensures that each operator prioritizes calling the RPC functions that share names with the CUDA runtime API, intercepting all CUDA kernel function calls.
(2) The transparent offloading client transmits the called CUDA runtime API and required parameters to the GPU 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. 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.

2) 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 [31]).

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 [32], mobile devices lack the necessary hardware to fully leverage this capacity [33]. The actual available bandwidth varies significantly due to factors such as device mobility [34], signal obstruction [35], and channel contention [36].



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

Fig. 3: The wireless transmission instability of TCP between our robot and the base station in MEC networks.

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 [37], we measured real-time wireless bandwidth capacity between the robot and a base station over TCP [38] at 0.1-second intervals for five minutes. As shown in Fig. 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 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 [16] (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 [39]. 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.

3) RPC Optimization

Remote Procedure Call (RPC) [40] is a fundamental communication protocol enabling processes to request services from remote computers over a network. Common strategies like Caching [41] (storing previous RPC results), Batching [42] (aggregating multiple RPCs into one request), and Asynchronous RPC [43] (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. III).

III. System Design

A. 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

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

In MEC networks, T_{RTT} is on the order of milliseconds and can be volatile (Sec. II-C2); 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

$$T_{\text{RRTO}} \;=\; \sum_{i=1}^{N} T_{\text{comp}}^{(i)} \;+\; T_{\text{input}} \;+\; T_{\text{output}} \;+\; T_{\text{RTT}}, \tag{2}$$

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:

$$\forall\, op_i \in \mathcal{S}^\star, \quad \text{Input}(op_i) \;\subseteq\; \Big(\bigcup_{j<i} \text{Output}(op_j)\Big) \cup \text{Input}_{\text{global}}. \tag{3}$$

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 (2), subject to the data dependency constraint (3) and the framework/runtime semantics (e.g., determinism and side-effect freedom).

B. 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. 4 summarizes the architecture and the integration of record/replay into the

workflow of traditional transparent offloading system (Fig. 2); no application source-code changes are required. Client and server workflows are detailed in Sec. III-D2.

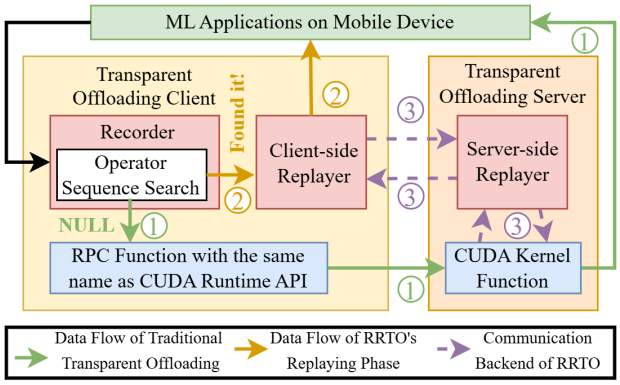


Fig. 4: Architecture of RRTO, with key components highlighted in red boxes.

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. 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. III-C3).

Once the recorder successfully identifies the complete inference operator sequence, RRTO transitions to the replaying phase (②). During this phase, subsequent inferences are executed by replaying the pre-identified operator sequence through replayer components deployed on both the client and server (orange lines in Fig. 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. II-C3). 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. 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 a promising future direction, potentially leveraging the same provenance information used for single-pipeline validation to separate coexisting pipelines via dependency-based demultiplexing (see Sec. VI).

C. Recording Phase Design

1) 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 [4], whose layers perform fixed operations (e.g., matrix multiplications, convolutions) independent of input values; ii) traditional ML models (e.g., Linear Regression [44], SVMs [45], KNN [46]), whose inference applies a fixed set of computations or comparisons predetermined before runtime; and iii) transformer encoders [47], RNNs on fixed-length sequences [48], and full Transformers on fixed-length tasks [49], 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 [50], where the number of decoding steps depends on the generated length; ii) Mixture-of-Experts (MoE) models [51], where a gating mechanism activates an input-dependent sparse subset of experts; iii) RNNs on variable-length sequences without padding [52], where the number of cell activations follows sequence length; iv) decision trees and ensembles [53], whose root-to-leaf comparison sequence is input-specific; and v) GNNs [54] 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 [4]) 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 [50], [51], 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 [55], such techniques are beyond this paper's scope.

2) Objectives and Challenges

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.

3) 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 [56] 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. 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. 5e).

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 1, 2, and 3, and Fig. 5 illustrates the overall idea. Algorithms 1–3 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

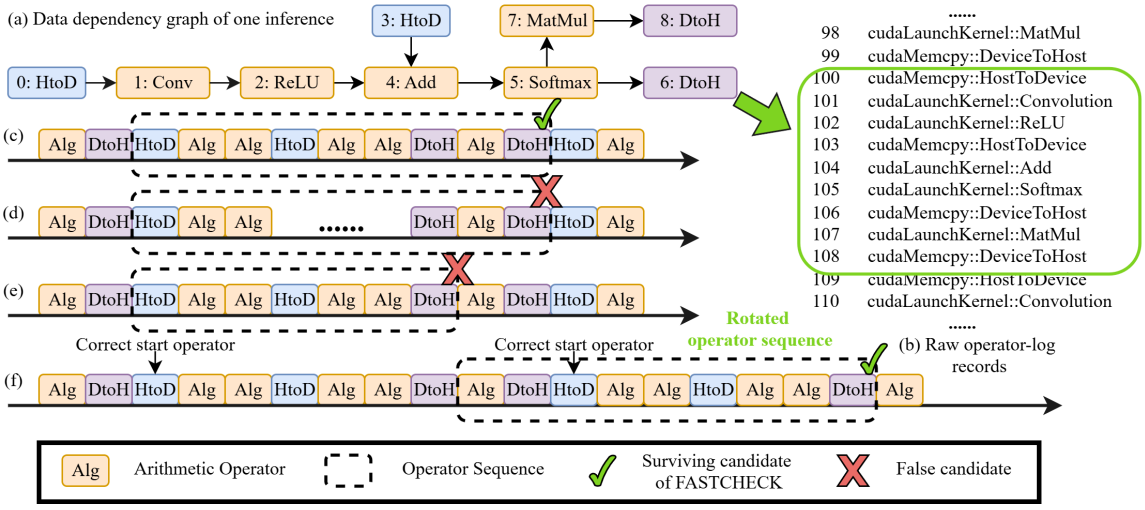


Fig. 5: Illustration of Operator Sequence Search. The start of a sequence need not be an HtoD; our search also considers the first operator after any DtoH and realigns rotated sequences during FULLCHECK.

Algorithm 1 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`);
  $e \leftarrow \max(T)$;
  $C \leftarrow S \cup \{\, t+1 \mid t \in T$ **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 2 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 $p \ge 0$ **do**
    if SLICEEQ(`Tags`, s, p, ℓ) = `False` **then**
      break;
    end if
    $cnt \leftarrow cnt + 1$;
    $p \leftarrow p - \ell$;
  end while
  return $(cnt \ge R)$;
end function

count, respectively. Algorithm 2 details FASTCHECK, which quickly filters unlikely candidates by comparing tag slices. Algorithm 3 details FULLCHECK, which performs exact verification at the recorded-entry level and additionally checks data dependencies. Here, SLICEEQ$(\texttt{Tags}, i, j, \ell)$ 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. 5c); if it lies within a rotated sequence, the candidate instead starts at the operator immediately following a previous end operator (Fig. 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 3). It realigns rotated candidates to the true `HtoD`/`DtoH` boundaries (observation ②; Fig. 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

Algorithm 3 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 ← s + ℓ − 1;
    if e ∉ T then
        return False;
    end if
    if CheckDataDependency(Logs, s, ℓ) = False then
        return False;
    end if
    cnt ← 0;
    p ← s;
    while p ≥ 0 do
        match ← True;
        for t ← 0 to ℓ − 1 do
            if EntryEq(Logs[s+t], Logs[p+t]) = False then
                match ← False;
                break;
            end if
        end for
        if match = False then
            break;
        end if
        cnt ← cnt + 1;
        p ← p − ℓ;
    end while
    return (cnt ≥ R);
end function
```

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 1). 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.

4) 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 1 (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 [57] and uncertainty-quantification practice for deep learning [58]. 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.

5) 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.

D. Replaying Phase Design

1) 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. 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. V), including lower GPU utilization, longer inference times, and increased energy consumption.

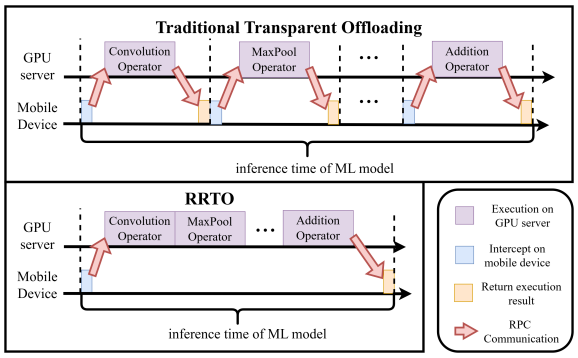


Fig. 6: Workflow of RRTO during the replaying phase.

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. 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.

2) 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. 4** and the server component in **Alg. 5**. Algorithms 4, 5, and 6 summarize the client-side control flow, the server-side replay routine, and replay-time argument reconstruction. Since Algorithm 1 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.

In **Alg. 4**, 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 4 **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 ← OPERATORSEQUENCE-
   SEARCH(Logs, R);
28:            end if
29:        end if
30:    end if
31:    RETURNRESULT(r);
32: end while
```

In **Alg. 5**, 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. 6), supplies the current inference input in a role-consistent manner, and returns the final DtoH result once the sequence completes.

3) Hidden system complexity

Beyond the high-level replay control flow in Algorithms 4 and 5, 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 6. RRTO addresses several non-trivial systems challenges in RRTOFIXARGS (**Alg. 6**) to enable transparent, high-performance offloading: i) pointer swizzling that preserves

Algorithm 5 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 ← 0 to |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
```

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.

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. 6*) 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.

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. 4*) virtualizes the CUDA runtime state, synthesizing return codes (e.g., cudaSuccess),

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

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. 6*) 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.

IV. IMPLEMENTATION

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

A. Implementing RRTO

Software. We implemented RRTO within Cricket's codebase [9], 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 [40], 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. 7), equipped with a Jetson Xavier NX [25] 8G onboard computer that is capable of CUDA-acclerated ML model inference. The system runs Ubuntu 20.04

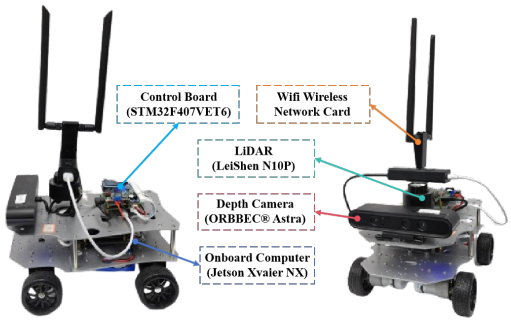


Fig. 7: The detailed composition of the robot platform.

	inference	communication	standby
Energy (Watt)	13.35	4.25	4.04

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

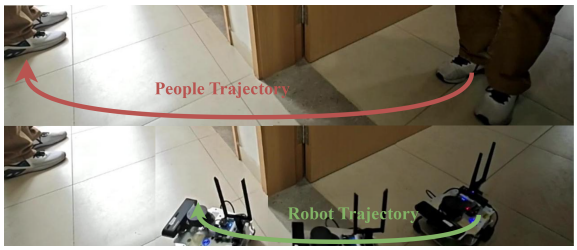


Fig. 8: Kapao [4], a real-time people-tracking application on our four-wheeled robot with a CNN-based human keypoint detection model.

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. 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.

Tab. II 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 [25]. 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.

B. Experiment Setup

Task. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig. 8 and Fig. 9. For evaluation, we run KAPAO on CrowdPose dataset [59] 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 [60] on CIFAR-100

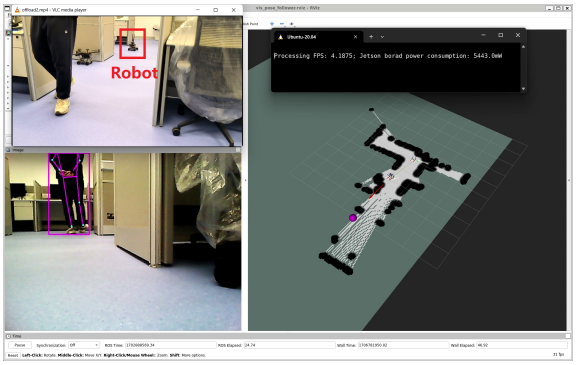


Fig. 9: A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner provides a third-angle observation of the entire experimental process.

dataset [61]: i) ResNet [62] and ConvNext [63] for object classification; ii) FCN [64] and DeepLabv3 [65] for semantic segmentation; iii) FasterRCNN [66] and RetainNet [67] 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. 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** [9]: A state-of-the-art transparent offloading system designed for remote GPU usage.

While advanced scheduling optimizations (e.g., layer partitioning [28] and multiple inference scheduling [3], [29], [30]) 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. VI.

V. 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.

A. Superiority of RRTO

Our evaluation results of our robotic application, KAPAO, as presented in Fig. 10, demonstrate that RRTO achieved performance comparable to non-transparent offloading system (NNTO) in both inference time and energy consumption.

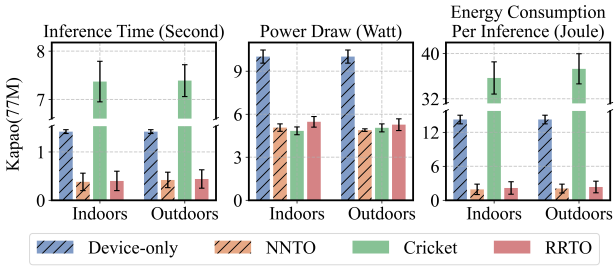


Fig. 10: Performance of Kapao in different environments with various systems.

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. V-B. 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.

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. 10 do not correspond to those in Tab. II. 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.

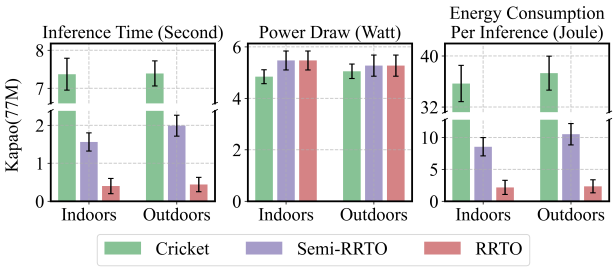


Fig. 11: Semi-RRTO: only applying Caching [41] specifically to the RPCs of "cudaGetDevice" and "cudaGetLastError" in RRTO, effectively eliminating their transmission requirements.

The prolonged inference times observed in outdoor scenarios for all offloading systems can be attributed to the lower bandwidth available outdoors (see Sec. II-C2), 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 [68] and drive up transmission costs, making RRTO even more advantageous than Cricket.

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

Comparing the different stages of function calls in Tab. III, we can see that KAPAO undergoes an initialization stage of inference different from subsequent inference loops. This is because the working process of KAPAO [4] follows the default detection model in Yolo v5 [69]: 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. III, we observed that a significant portion, specifically 90.62%, of RPC function calls consisted of "cudaGetDevice" and "cudaGetLastError". These calls, generated by PyTorch [14] 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.

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

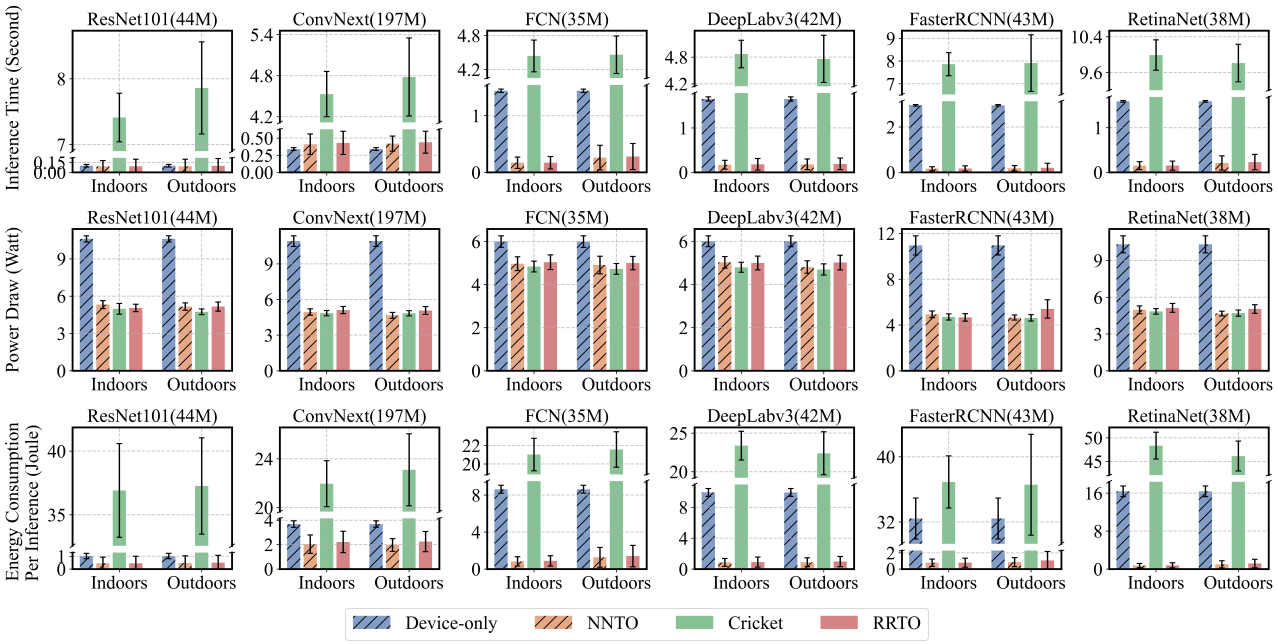


Fig. 12: Performance of Torchvision models in different environments with various systems.

Despite restricting PyTorch to use only a single GPU sequentially and employing Caching (referred to as "semi-RRTO" in Fig. 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 [4], measured using pynvml [70] and presented in Tab. IV. 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 IV: Comparison between RRTO and the baselines about numbers of RPC calls and average GPU utilization on GPU server.

C. 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. 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. 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. 1a) 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.

VI. 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 [71], 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 [72], and optimizing inference for DAMs with changing operator sequences remains a pervasive challenge across all offloading systems.

Layer partitioning. Layer partitioning techniques [28] 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 [29], deep reinforcement learning control [30], and joint relay selection [3] 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 [73] lowers the precision of weights and activations to cut computation costs, whereas distillation [74] 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.

VII. Conclusion

In this paper, we have proposed RRTO, a high-performance transparent offloading system optimized for ML model inference in MEC. RRTO effectively alleviates communication overhead, a common bottleneck in traditional transparent offloading systems stemming from frequent RPCs, by employing a novel record/replay mechanism. This innovative approach enables RRTO to achieve performance comparable to established non-transparent offloading methods, without necessitating any modifications to the application source code. Consequently, RRTO is poised to significantly advance the deployment of diverse, ML-driven mobile applications in real-world environments. By providing fast, energy-efficient, and seamlessly integrated inference capabilities, RRTO empowers mobile device systems to execute complex tasks with enhanced operational efficiency and effectiveness.

REFERENCES

[1] F. Luo, S. Khan, Y. Huang, and K. Wu, "Binarized Neural Network for Edge Intelligence of Sensor-Based Human Activity Recognition," *IEEE Trans. Mobile Comput.*, vol. 22, no. 3, pp. 1356–1368, Mar. 2023.

[2] A. N. Saridena and A. Choromanska, "DNN Patching: Progressive Fixing and Augmenting the Functionalities of DNNs for Autonomous Vehicles," *IEEE Robot. Autom. Lett.*, vol. 7, no. 2, pp. 3257–3264, Apr. 2022.

[3] 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," *IEEE Trans. Ind. Informat.*, vol. 16, no. 8, pp. 5424–5434, 2019.

[4] 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.

[5] 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.

[6] A.-Q. Cao and R. de Charette, "MonoScene: Monocular 3D Semantic Scene Completion," in *Proc. CVPR*, Jun. 2022, pp. 3991–4001.

[7] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile Edge Computing: A Survey," *IEEE Internet Things J.*, vol. 5, no. 1, pp. 450–465, Feb. 2018.

[8] 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.

[9] N. Eiling, J. Baude, S. Lankes, and A. Monti, "Cricket: A Virtualization Layer for Distributed Execution of CUDA Applications with Checkpoint/Restart Support," *Concurrency Comput. Pract. Exp.*, vol. 34, no. 14, p. e6474, May 2022.

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

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

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

[13] Z. DeVito, "TorchScript: Optimized Execution of PyTorch Programs," *Retrieved January*, 2022.

[14] 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," *arXiv preprint arXiv:1912.01703*, Dec. 2019.

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

[16] 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," *Procedia CIRP*, vol. 107, pp. 588–593, 2022.

[17] 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.

[18] A. Munshi, "The OpenCL Specification," in *Proc. IEEE Hot Chips 21 Symp. (HCS)*, Aug. 2009.

[19] 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*, Jun. 2022, pp. 209–221.

[20] 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," *IEEE Robot. Autom. Lett.*, vol. 6, no. 2, pp. 2335–2342, Apr. 2021.

[21] 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.

[22] 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.

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

[24] RaspberryPi, "Raspberry Pi," https://www.raspberrypi.com/documentation/computers/configuration.html\#power-consumption, 2022.

[25] 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.

[26] 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.

[27] N. Armanfard, J. P. Reilly, and M. Komeili, "Local Feature Selection for Data Classification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 6, pp. 1217–1227, 2015.

[28] 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.

[29] L. Lin, X. Liao, H. Jin, and P. Li, "Computation Offloading Toward Edge Computing," *Proc. IEEE*, vol. 107, no. 8, pp. 1584–1607, Aug. 2019.

[30] 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.

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

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

[33] 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.

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

[35] M. Ding, P. Wang, D. López-Pérez, G. Mao, and Z. Lin, "Performance Impact of LoS and NLoS Transmissions in Dense Cellular Networks," *IEEE Trans. Wireless Commun.*, vol. 15, no. 3, pp. 2365–2380, Mar. 2016.

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

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

[38] Y. Tian, K. Xu, and N. Ansari, "TCP in Wireless Environments: Problems and Solutions," *IEEE Commun. Mag.*, vol. 43, no. 3, pp. S27–S32, Mar. 2005.

[39] NVIDIA, "NVIDIA Nsight Compute Documentation," https://docs.nvidia.com/nsight-compute/index.html, 2024.

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

[41] 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.

[42] 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.

[43] S. Eyerman and I. Hur, "Efficient Asynchronous RPC Calls for Microservices: DeathStarBench Study," *arXiv preprint arXiv:2209.13265*, 2022.

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

[45] H. Wang, Y. Yang, and B. Liu, "GMC: Graph-Based Multi-View Clustering," *IEEE Trans. Knowledge Data Eng.*, vol. 32, no. 6, pp. 1116–1129, 2019.

[46] 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.

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

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

[49] 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.

[50] 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.

[51] 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.

[52] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proc. NeurIPS*, vol. 27, 2014.

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

[54] B. Jiang, Z. Zhang, D. Lin, J. Tang, and B. Luo, "Semi-Supervised Learning with Graph Learning-Convolutional Networks," in *Proc. CVPR*, 2019, pp. 11 313–11 320.

[55] 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*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, Dec. 2020, pp. 15 451–15 463.

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

[57] 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.

[58] 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," *Inf. Fusion*, vol. 76, pp. 243–297, Dec. 2021.

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

[60] S. Marcel and Y. Rodriguez, "Torchvision the Machine-Vision Package of Torch," in *Proc. ACM Multimedia*, Oct. 2010, pp. 1485–1488.

[61] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, Tech. Rep., 2009.

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

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

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

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

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

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

[68] M. Noormohammadpour and C. S. Raghavendra, "Datacenter Traffic Control: Understanding Techniques and Tradeoffs," *IEEE Commun. Surv. Tutor.*, vol. 20, no. 2, pp. 1492–1525, Jun. 2018.

[69] 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," Nov. 2022.

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

[71] R. Chowdhury and D. Subramani, "Optimal Path Planning of Autonomous Marine Vehicles in Stochastic Dynamic Ocean Flows Using a GPU-Accelerated Algorithm," *IEEE J. Ocean. Eng.*, vol. 47, no. 4, pp. 864–879, Oct. 2022.

[72] V. Rosenfeld, S. Breß, and V. Markl, "Query Processing on Heterogeneous CPU/GPU Systems," *ACM Comput. Surv.*, vol. 55, no. 1, pp. 1–38, Jan. 2023.

[73] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "VecQ: Minimal Loss DNN Model Compression with Vectorized Weight Quantization," *IEEE Trans. Comput.*, vol. 70, no. 5, pp. 696–710, May 2021.

[74] L. Wang and K.-J. Yoon, "Knowledge Distillation and Student-Teacher Learning for Visual Intelligence: A Review and New Outlooks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, pp. 3048–3068, Jun. 2022.

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

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

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

Yuhao Qing is a Ph.D. student in Computer Science at the University of Hong Kong. His research interests include machine learning systems and cloud computing.

Haoze Song is a Ph.D. student in Computer Science at the University of Hong Kong. His research interests include distributed computing, data management in cloud computing, and scalable real-time systems.

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

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

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

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

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

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