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*

arXiv:2507.21739v1 [cs.NI] 29 Jul 2025

***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 offloading-based inference paradigm using GPU servers, existing approaches are divided into non-transparent and transparent methods, with the latter necessitating modifications to the source code. Non-transparent offloading achieves high performance but requires intrusive code modification, limiting compatibility with diverse applications. Transparent offloading, in contrast, offers wide compatibility but introduces significant transmission delays due to per-operator remote procedure calls (RPCs). To overcome this limitation, we propose RRTO, the first high-performance transparent offloading system tailored for MEC inference. RRTO introduces a record/replay mechanism that leverages the static operator sequence in ML models to eliminate repetitive RPCs. To reliably identify this sequence, RRTO integrates a novel Operator Sequence Search algorithm that detects repeated patterns, filters initialization noise, and accelerates matching via a two-level strategy. Evaluation demonstrates that RRTO achieves substantial reductions of up to 98% in both per-inference latency and energy consumption compared to state-of-the-art transparent methods and yields results comparable to non-transparent approaches, all without necessitating any source code modification.**

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

I. Introduction

Machine learning (ML) has become fundamental to a wide range of mobile applications, from intelligent wearable sensors [1], [2] and autonomous vehicles [3]–[5] to industrial IoT systems [6]–[15]. These applications are built upon advances in object detection [16]–[18], robotic control [19]–[23], and environmental perception [24], 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 [25]–[28], including restricted computational power and battery life.

Recent studies [29] have proposed mobile edge computing (MEC) as a promising solution for high-performance inference by leveraging GPU servers (e.g., edge devices with powerful GPUs) at the edge of the radio access network. 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 times faster inference and 49% lower energy consumption compared than device-only inference. However, this inherent requirement for source code modification poses a critical limitation, which not only significantly increases engineering overhead but also severely curtails application diversity (i.e., support various upper-layer applications). Specifically, it impedes integration with closed-source software (e.g., TensorRT [32] and CUDA-X libraries [33]) and runtime-optimized model structures (e.g., Just-In-Time compilation [34]) where code modifications are often infeasible or prohibitive. While alternatives like CUDA Graph [30] and TorchScript [35] have attempted to mitigate the extent of these modifications via model-level packaging, they remain intrinsically non-transparent and suffer from limited

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).

Method	Category	Code Modification	Application Diversity	Platform Compatibility	High Performance
Device-only Inference	Device-only	N/A	✓	✗	✗
Native Offloading	Non-transparent	High	✗	✓	✓
CUDA Graph [30]	Non-transparent	Low	✗	✗	✓
Cricket [31]	Transparent	N/A	✓	✓	✗
RRTO (Ours)	Transparent	N/A	✓	✓	✓

TABLE I: Comparison of representative inference methods in MEC.

platform compatibility due to their framework-specific nature (see Sec. II-B).

Transparent offloading methods [31] enable model inference to be offloaded to GPU servers without code modification, offering convenience at the cost of reduced performance. During inference, ML models execute a sequence of operators (e.g., *torch.nn.functional.add()*, *torch.nn.functional.conv2d()* in PyTorch [36]), which are typically forwarded to backend system functions (e.g., *aten::add* and *aten::conv2d* within CUDA's "cudaLaunchKernel" [37]). Transparent offloading intercepts these calls and redirects their execution to GPU servers via Remote Procedure Calls (RPC), avoiding source code changes (see Sec. II-C). However, since each operator triggers a separate RPC, the accumulated Round-Trip Time (RTT) introduces significant transmission overhead in MEC networks. This is because ML models often contain hundreds of operators (e.g., 522 in [17]) and invoke thousands of RPCs during inference (e.g., 5895 in [17]); in wireless networks commonly used for mobile devices, each RTT can take several milliseconds [38], making transmission delay dominate inference time (up to 95% for [31]).

To address these challenges, one idea is to reduce transmission delay in transparent offloading by shifting from a reactive to a preactive approach. Existing transparent offloading methods trigger RPCs only after operators are invoked during inference, leading to sequential scheduling and communication delays. This reactive behavior stems from their general-purpose design for remote GPU usage, where operator patterns in upper-layer applications are often unpredictable, making proactive scheduling infeasible. In contrast, ML inference typically follows a fixed and predictable operator sequence due to the static computation graph (see Sec. III-B1). This enables a preactive offloading mechanism that uses operator tracing and record/replay to anticipate operator sequences. By directly replaying recorded operators on the GPU server, the preactive approach eliminates per-operator RPC communication during the offloading process, significantly reduces transmission delay, and brings the performance benefits of non-transparent offloading back to transparent offloading.

Integrating the record/replay mechanism into transparent offloading systems presents three key challenges in accurately identifying the correct operator sequence for each inference. First, transparent offloading systems intercept function calls from upper-layer applications to GPU devices at the system layer, where RRTO can only access low-level log records of the operators called without any direct hints from upper-layer applications, making it difficult to determine which inference task each operator belongs to. Second, operator RPCs generated during model loading and initial inference may differ from those in the regular inference loop, introducing variability that complicates sequence identification, where even small mismatches can compromise correctness. Third, large operator traces, often containing tens of thousands of entries, create a vast search space that hinders timely identification of the correct sequence.

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 automatically records the operators invoked during the first inferences using traditional RPCs as in traditional transparent offloading systems and replays the identified fixed-order operator sequence in subsequent inferences once the specific operator sequence for each inference is identified. To accurately identify this sequence, RRTO introduces the *Operator Sequence Search* algorithm. First, the algorithm leverages the observation that the target sequence is repeatedly embedded in the complete log during multiple inferences, ensuring task-level association of each operator. Second, the algorithm verifies completeness by checking whether the repeated sequence aligns with the entire log, while ignoring inconsistencies caused by model loading or initialization. Third, a three-level fast match algorithm is applied in the Operator Sequence Search algorithm to efficiently reduce the search space and accelerate 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 propose a record/replay mechanism for transparent offloading system that eliminates per-operator RPC communication during offloading, significantly reducing transmission delay.
- We design the Operator Sequence Search algorithm that accurately reconstructs operator sequences to support the record/replay mechanism.
- 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.

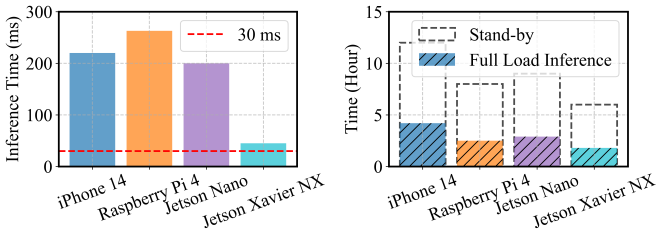
The rest of the paper is organized as follows. Sec. II motivates the design of RRTO by revealing the challenges in current MEC networks. Sec. III presents the system design of RRTO. Sec. IV introduces the system implementation, followed by performance evaluation in Sec. V. Related works and technical 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 [36], TensorFlow [39], CUDA [37], and OpenCL [40]) and mobile devices are equipped with various hardware accelerators (e.g., GPU [41], FPGA [42], and SoC [43]), 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.

In addition, the performance of device-only inference is constrained by the limited computational power of processors 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 [44] (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 [45]–[48].

B. Non-Transparent Offloading

Non-transparent offloading strategies relocate model computations to GPU servers by mandating application source code modifications, a requirement that inherently increases engineering complexity and restricts 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 or prohibitive. High-performance libraries (e.g., TensorRT [32] and CUDA-X library [33]) 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 [34] and dynamic conventional kernel selection with different numbers/sizes based on task-specific requirements [49]. These adaptive methods change model structure at runtime to improve speed and accuracy. 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 [30] and TorchScript [35],s reduce the amount of 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 greater constraints on mobile software development that already plague device-only deployment today.

To further optimize inference latency and energy efficiency, diverse scheduling strategies have been incorporated into non-transparent offloading systems. Unlike native offloading, which typically relocates the entire model to GPU servers, these advanced methods employ fine-grained scheduling at various stages of model inference (e.g., layer partitioning [50] and multiple inference scheduling [7], [51], [52]). 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 (device-only inference), the complete, top-down 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 ML model's structure to complete the entire computation process.
- B Each operator accesses the appropriate function library through a unified operator API tailored to the type of the device running the application; for instance, on NVIDIA GPUs, this is the CUDA runtime library [37].
- C The operating system loads the local CUDA runtime library by default, and executes the relevant CUDA kernel functions on the mobile device's GPU.
- D The local CUDA runtime library returns the execution results (e.g., 'cudaSuccess' from "cudaLaunchKernel" and computation result from "cudaMemcpyDtoH") to the upper-layer application.

Transparent offloading methods [31] typically employ a strategy of rewriting dynamic link libraries by defining functions that share names with those in the CUDA runtime API and prioritizing the loading of these custom libraries by setting the *LD_PRELOAD* environment variable. This setup causes the dynamic linker to redirect calls from the original library functions to the custom library functions with the same name, effectively intercepting library functions. Subsequently, the custom library packages the function calls along with the required parameters and data, and sends them to the GPU server via RPC. It also modifies GPU memory management and the launching of CUDA kernel functions to ensure that RPCs are executed correctly on the GPU server. By this means, these methods achieve transparent offloading by intercepting kernel functions one-by-one at the system layer using similarly named functions in the dynamic link library and offloading their execution to the GPU server.

Compared to using the GPU on the mobile device, changes in the system call process primarily occur in step C. The

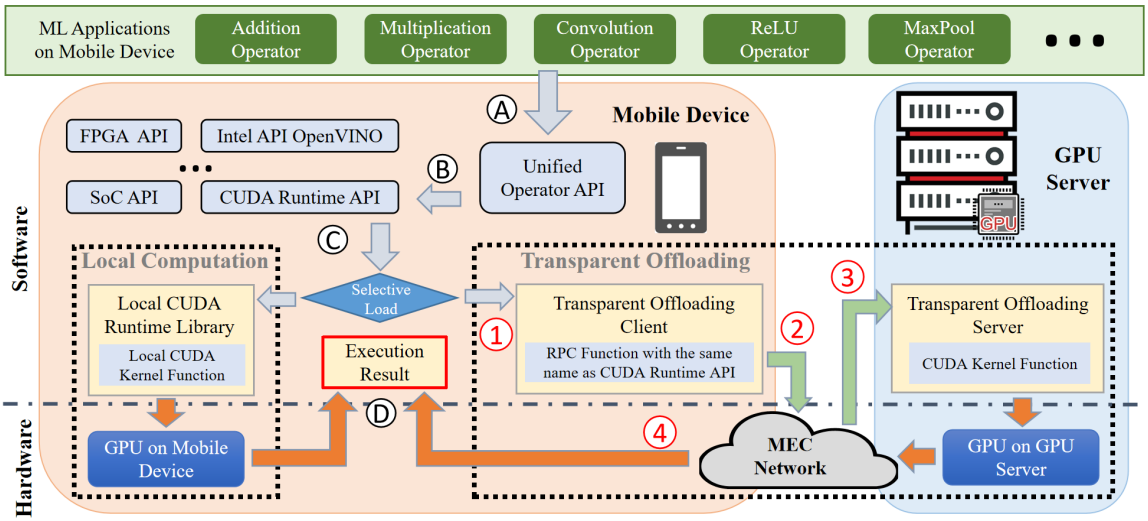


Fig. 2: Workflow of Transparent Offloading System for Model Inference in MEC.

detailed steps of the transparent offloading process (depicted in the right part in Fig. 2) are as follows:

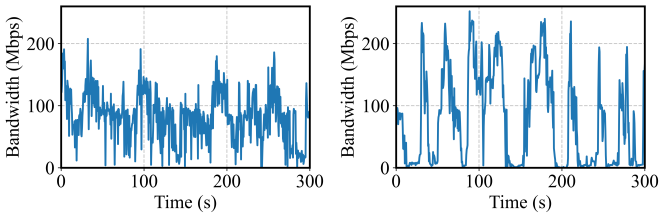
(1) The dynamic link library is modified such that each operator prioritizes calling the RPC functions that have the same names as those in the CUDA runtime API, enabling the effective identification and interception of all CUDA kernel function calls.
(2) The transparent offloading client transmits the called CUDA runtime API and the required parameters to the GPU server via the MEC network using RPC.
(3) The transparent offloading server on the GPU server launches the corresponding CUDA kernel functions and completes the computation.
(4) The execution results are sent back to the client. The transparent offloading system then returns these results to the upper-layer function calls.

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 equipped with high-speed technologies (e.g., 200–800 Gbps for InfiniBand [53]).

The inherent bandwidth capacity of wireless networks is inherently constrained by both theoretical limits and practical implementation factors in MEC. While Wi-Fi 6 can achieve a peak bandwidth of 1.2 Gbps per stream [54], mobile devices lack the necessary hardware to fully leverage this capacity [55]. The actual available bandwidth varies significantly due to factors such as device mobility [56], signal obstruction [57], and channel contention [58].

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 [59], we measured real-time wireless bandwidth capacity between the robot and a base station over TCP [60] 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



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

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

measurements exhibiting higher fluctuations and occasional near-zero drops due to obstacles and reduced signal reflections. This indicates that transparent offloading is hard to sustain in real-world wireless networks, as the RTT for operator-level RPCs in traditional transparent offloading systems is usually higher than in data center networks.

While the CPU and GPU typically employ an asynchronous paradigm (CUDA kernels are enqueued to the GPU at the inference start, allowing the GPU to process them independently while the CPU awaits overall completion), this paradigm faces severe challenges in MEC due to limited network bandwidth, prolonging the RTT for each remote CUDA kernel launch via RPC. This prolonged RTT, often several milliseconds per RPC [38] (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 [61]. Consequently, the cumulative overhead from frequent RPCs for individual kernel dispatches emerges as a critical performance bottleneck in traditional transparent offloading systems, negating many benefits of the CPU-GPU asynchronous paradigm in constrained network environments.

3) RPC Optimization

Remote Procedure Call (RPC) [62] is a fundamental communication protocol enabling processes to request services from remote computers over a network. While common strategies like Caching [63] (storing results of previous RPC calls), Batching [64] (aggregating multiple RPC calls into a single request), and Asynchronous RPC [65] (allowing non-blocking execution so the client can perform other tasks while awaiting the server's response) aim to enhance RPC performance, they prove largely ineffective in reducing the substantial communication costs of transparent offloading systems during model inference. Specifically, Caching fails because each inference typically processes unique input, necessitating fresh operator computations. While effective in reducing the number of network requests via aggregation, Batching has a critical drawback: It does not eliminate per-operator RPCs and, more importantly, must rely on latency-inducing timeouts for batch formation. This reliance is necessary because the total number of operations is unknown *a priori*, a constraint that our operator search algorithm overcomes. Furthermore, Asynchronous RPC methods compromise the correctness of inference results because they cannot guarantee the 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. Overview

This section introduces RRTO, a high-performance transparent offloading system meticulously designed for model inference in MEC, featuring a novel record/replay mechanism. The overall architecture of RRTO is depicted in Fig. 4. Unlike conventional transparent offloading systems (Fig. 2), RRTO adeptly integrates this record/replay mechanism into the core components of the transparent offloading framework while maintaining transparency to upper-layer applications. This architectural choice ensures that RRTO enables offloading without necessitating any modifications to the application's source code. The operational logic for both client and server components is further elaborated on pseudocode in Sec. III-C2.

Within the context of a single inference application, RRTO employs its specialized record/replay mechanism to effectively differentiate operators belonging to distinct inference tasks. For the first few inference executions, RRTO enters the recording phase (①). In this mode, it adheres to the execution pattern of traditional transparent offloading systems, where each operator's execution is individually offloaded to the GPU server via RPCs, as illustrated by the green lines in Fig. 4.

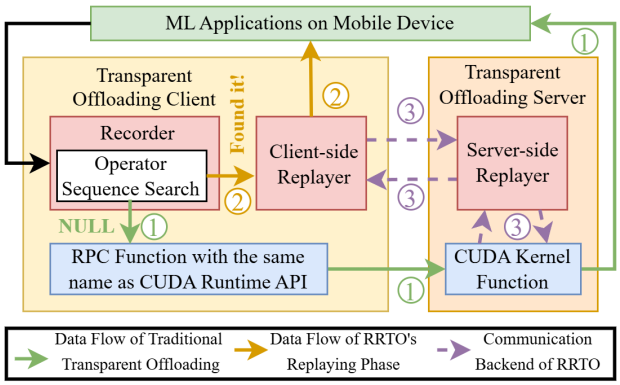


Fig. 4: Architecture of RRTO, with key components highlighted in red boxes.

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-B2).

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 strategically deployed on both the client and server, as illustrated by the orange lines in Fig. 4. The client-side replayer assumes a pivotal role to ensure RRTO's transparency and high-performance simultaneously: It intelligently furnishes the upper-layer applications with the execution results from previously recorded RPC calls (mainly 'cudaSuccess' from "cudaLaunchKernel"), a mechanism analogous to the RPC caching techniques employed in existing optimization methods (see Sec. II-C3). This approach enables the offloading client seamlessly invokes system functions for subsequent operators, creating the illusion of local execution, until it reaches the end operator. Upon encountering this operator, it pauses to await the actual inference output, which is computed and transmitted back 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 the transparency and system integrity while eliminating per-operator RPC communication overhead during the replaying phase.

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 innovative approach enables RRTO to achieve a communication overhead closely approximating that of non-transparent offloading systems. Specifically, it transmits only the raw input and final output data, and circumvents per-operator RPC communication during the offloading process in the replaying phase, as visualized by the purple dotted lines in Fig. 4.

When multiple ML applications run concurrently on a

mobile device, RRTO distinguishes operators from different applications by leveraging the inherent capability of existing transparent offloading systems to separate RPC functions originating from distinct ML applications. This capability ensures that shared remote GPUs can be utilized across multiple applications and that inference results are accurately returned to their respective sources, thereby improving performance for each application. However, addressing performance issues arising from resource or network contention among concurrent tasks requires multiple inference scheduling strategies (see Sec. VI), and consequently, the integration of these strategies into RRTO to further enhance its performance under such conditions remains a key objective for future work.

B. Recording Phase Design

1) ML Models with Fixed-Order Operators

ML models can be broadly classified into static and dynamic activation types, based on whether their sequence of activated computational operators and corresponding data flow paths remain invariant or adapt to the input during a single inference pass.

Static Activation Models (SAMs) execute a predetermined, input-invariant sequence of computational operators. Thus, for any given input, the specific operations and their order are constant. These include: i) Multilayer Perceptrons (MLPs) and Convolutional Neural Networks (CNNs) [17]: These utilize fixed architectures where layers consistently perform predefined mathematical operations (e.g., matrix multiplication and convolution) regardless of input values. ii) Traditional ML models (e.g., Linear Regression [66], Support Vector Machines [67], and K-Nearest Neighbors [68]): These also operate statically, their core inference logic applying a fixed set of calculations or comparisons predictable before runtime. iii) Transformer encoders [69], Recurrent Neural Networks (RNNs) on fixed-length sequences [70], and full Transformers on fixed-length tasks [71]: These exhibit static behavior when processing sequences padded or truncated to a uniform, predefined length, thereby fixing the number of recurrent unrollings or activated self-attention blocks. Characterized by fixed structures and an absence of input-dependent execution paths, these models exhibit computationally regular processes. This predictability makes them the primary target for our RRTO system and the baseline methods considered in this work.

Conversely, Dynamic Activation Models (DAMs) adapt their computational path or the number of activated operators based on the specific characteristics of the input data. These include: i) Transformers (e.g., GPT-like models for generation) [72]: These exhibit dynamic behavior, particularly in auto-regressive sequence generation, where the total number of decoding steps (and thus operator activations) adapts to the runtime-determined length of the generated sequence. ii) Mixture of Experts (MoE) models [73]: These are inherently dynamic, as a gating mechanism activates an input-dependent sparse subset of "expert" sub-networks (operator groups) for processing. iii) RNNs on variable-length sequences [74]: These operate dynamically (without padding), as the number of recurrent cell activations directly matches the input's sequence length, making the operator count input-dependent. iv) Decision Trees and their ensembles [75]: These define dynamic inference paths, where the sequence of activated comparison operators from root to leaf is directly dictated by input features, creating input-specific computational routes. v) Graph Neural Networks (GNNs) [76]: These exhibit dynamic behavior, particularly when operating on graphs of varying sizes/structures or using input-dependent neighborhood sampling. Consequently, key operational aspects adapt dynamically to each input graph or query (e.g., the number of processed nodes, the effective layer depth, and the scope of neighbor aggregation including its associated operators). While this dynamic nature enhances model adaptability and expressiveness, it also complicates runtime profiling (e.g., execution time and I/O data sizes), leading to fewer optimization systems designed specifically for them.

To conclude, SAMs (e.g., MLPs and CNNs for computer vision [17]) are widely adopted for mobile applications. These applications necessitate high-performance inference (low-latency and energy-efficient) but typically operate on resource-constrained mobile devices, making offloading-based inference over MEC networks essential for achieving such performance. In contrast, DAMs, which demand significant computing resources and less stringent real-time inference requirements [72], [73], are suited for data center deployment.

Our system, RRTO, employs a record/replay mechanism optimized for the consistent operator sequences inherent in SAMs. When RRTO encounters a DAM where the operator sequence changes, its replayer component detects this inconsistency. In such instances, RRTO temporarily disables its specific optimizations, reverts to a standard transparent offloading workflow, and attempts to re-establish a stable execution sequence. Given the predominance of SAMs in mobile applications, these fallback events are expected to be infrequent, thereby preserving RRTO's performance advantages. Optimizing inference for DAMs with their varying operator sequences remains an open challenge for offloading systems in general. While some research explores solutions, such as predicting future layer activations [77], these advanced techniques are beyond the scope of this paper, which concentrates on SAM optimization.

2) Operator Sequence Search

A core design decision in RRTO is to achieve full transparency by operating without any hints from the application or the underlying ML framework. One might consider a simpler approach that relies on explicit markers inserted into a framework's source code (e.g., in PyTorch) to identify inference boundaries. However, such a method perpetuates the same platform compatibility issues that already plague device-only deployment across diverse frameworks and hardware. It fails to solve the fundamental deployment problem for the heterogeneous MEC ecosystem. Consequently, our system must autonomously discover the correct sequence of operators corresponding to an inference. This hint-free approach, while technically more demanding, is a prerequisite for creating a truly universal and easy-to-deploy offloading solution.

The performance of RRTO hinges on its ability to accurately identify the correct inference operator sequence, since even a

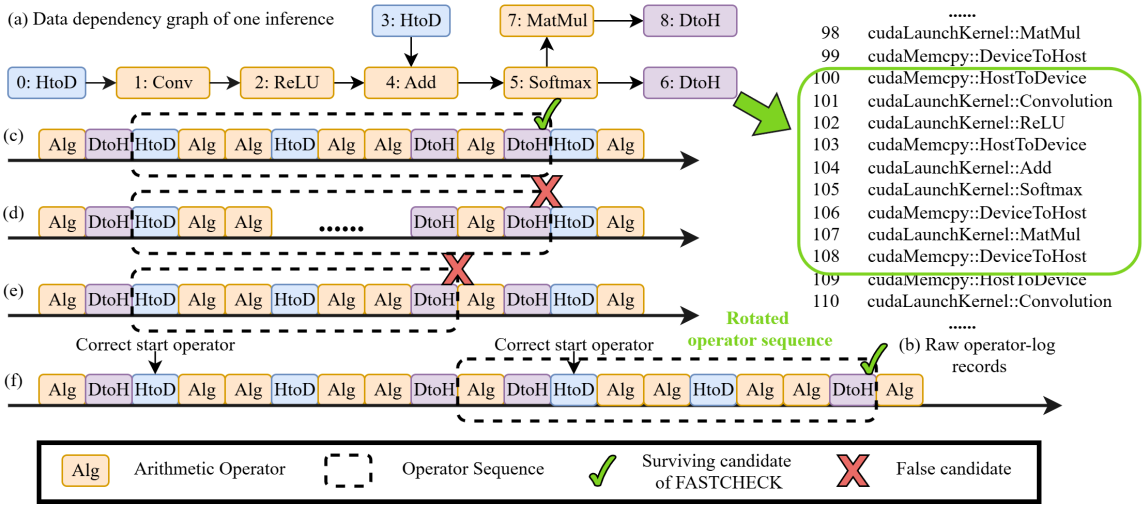


Fig. 5: Illustration of Operator Sequence Search.

single missing or extra operator disrupts the end-to-end data flow and produces incorrect results. To achieve this under realistic conditions, Operator Sequence Search must meet three fundamental challenges: i) Transparency: RRTO cannot rely on any high-level framework hooks or metadata and must infer the sequence solely from raw operator-log records. ii) Initialization variability: real ML models typically emit extra or altered operators during model loading and the very first inference; these one-time initialization artifacts must not be mistaken for the regular inference pattern. iii) Large trace size: real-world operator traces can contain tens or even hundreds of thousands of entries, so any brute-force search over all candidate subsequences would be prohibitively expensive. The pseudo code for the algorithm of operator sequence search is outlined in **Alg. 1**, and Fig. 5 gives an example of how these challenges are addressed in our algorithm.

Algorithm 1 **Operator_Sequence_Search**

Input: `Logs`: list of OperatorInfo entries from the first N inferences; R: minimum number of repeats for check
Output: inference operator sequence `IOS` or `NULL`

function OPERATORSEQUENCESEARCH(Logs, R)
  $S \leftarrow \{v.index \mid v.func = \text{"HtoD"}, v \in \texttt{Logs}\}$
  $T \leftarrow \{v.index \mid v.func = \text{"DtoH"}, v \in \texttt{Logs}\}$
  if $S = \emptyset$ or $T = \emptyset$ **then return** NULL
  end if
  Tags $\leftarrow \{\text{Tag}(v) \mid v \in \texttt{Logs}\}$
  $end \leftarrow \text{MAX}(T)$
  $starts \leftarrow S \cup \{i+1 \mid i \in T\}$
  for each $j \in starts$ **and** $j \leq end$ **do**
    $\ell \leftarrow end - j$
    if FASTCHECK(Tags, j, ℓ, R) **then**
      for each $k \in S$ **and** $j - \ell \leq k \leq j$ **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

To address these challenges, our algorithm exploits three key observations. ① A static ML model executes the same operator sequence for each inference, yielding a highly regular pattern in the logs. ② The real-time requirements of MEC enforces processing inference immediately, which begins with a host-to-device memory copy of the raw input ("cudaMemcpyHtoD", short for "HtoD") and ends with a device-to-host memory copy of the result ("cudaMemcpyDtoH", short for "DtoH"). By treating these copies as special memory transfer operations and grouping any following synchronization calls with them (e.g., "cudaStreamSynchronize"), we obtain reliable boundary markers. ③ The correct inference sequence must satisfy all data-dependency constraints: every operator's inputs must originate either from the raw input, from a prior operator's output (share the same memory address), or from the model's parameters.

Unfortunately, relying solely on the maximum-repeated-substring algorithm [78] based on observation ① fails to segment the operator sequence correctly, because when the sequence repeats continuously in the log, the algorithm merges multiple consecutive iterations into a single maximal substring (Fig. 5d). Likewise, using only the memory-copy markers of observation ② falls short when multiple "HtoD" or "DtoH" events occur within a single inference, leaving ambiguous which copy corresponds to the true beginning or end (Fig. 5e).

Based on these observations, we introduce a robust two-stage matching strategy. This strategy is engineered to systematically reduce the search complexity, first by rapidly identifying plausible candidates and subsequently by performing detailed verification on this narrowed selection, thereby accelerating the overall identification process.

In the first stage, `FASTCHECK` scans the log to identify candidate start and end operators based on memory-copy boundaries of observation ② (*Lines 2 and 3 in Alg. 1*) and

Algorithm 2 FastCheck_and_FullCheck

```
1: function FastCheck(Tags,start, length, R)
2:     count ← 0, pos ← start
3:     while pos ≥ 0 and Tags[start:start+length] = Tags[pos:pos+length] do
4:         count ← count + 1
5:         pos ← pos - length
6:     end while
7:     return count ≥ R
8: end function
9: function FullCheck(Logs, start, length, R,T)
10:    end ← start+length-1
11:    if end ∉ T or CheckDataDependency(Logs, start, length) = False then
12:        return False
13:    end if
14:    count ← 0, pos ← start
15:    while pos ≥ 0 do
16:        if for all 0 ≤ t < length, Logs[start+t] ≡ Logs[pos+t] then
17:            count ← count + 1
18:            pos ← pos - length
19:        else
20:            break
21:        end if
22:    end while
23:    return count ≥ R
24: end function
```

then locates possible operator sequences by detecting repeated sequences based on observation ① (*Line 3 in Alg. 2*). Each candidate ends at the current last end operator (*Line 7 in Alg. 1*); if that operator truly marks the inference end, the candidate begins at the matching start operator (Fig. 5c), whereas if it lies within a rotated sequence, the candidate instead begins at the next occurrence of an end operator (Fig. 5f) (*Line 8 in Alg. 1*). FASTCHECK then linearizes the log into a compact string of operator categories ("HtoD", "DtoH", arithmetic, etc.) that it can count in linear time how often the candidate substring reappears in the latter portion of the log (*Line 6 in Alg. 1*) and thereby confirm sustained repetition despite initialization variability (*Line 3 in Alg. 2*). By focusing solely on category tags and recurrence, FASTCHECK prunes the vast majority of false candidates caused by one-time initialization artifacts or spurious memory-copy boundaries.

Once FASTCHECK has identified the surviving candidates, operator sequence search advances to the second stage, FULLCHECK, which verifies each candidate exhaustively (**Alg. 2**). Because a FASTCHECK candidate may be a cyclic rotation of the true sequence, FullCheck first realigns it to the correct start and end markers using the host-to-device and device-to-host boundaries from observation ② (Fig. 5f, *Line 12 in Alg. 1*). It then checks that every operator in the sequence satisfies the required data dependencies according to observation ③ (*Line 11 in Alg. 2*) that inputs must come from raw input, a prior operator's output at the same address, or model parameters. Finally, it conducts a full one-to-one, record-level comparison across the entire log to confirm that the sequence repeats exactly as predicted by observation one (*Line 16 in Alg. 2*). Although this final validation is the most time-consuming step, it is applied only to a small set of high-confidence candidates.

In this way, the operator sequence search algorithm addresses the three challenges, and locates the correct operator sequence efficiently by keeping transparency through pure log analysis, discounting one-time initialization variability via repeated-sequence detection, and taming large traces with a two-stage FastCheck+FullCheck strategy. This design enables RRTO to extract the precise inference operator sequence without any framework support, even in demanding real-world deployments.

C. 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 the per-operator RPC communication required by existing transparent offloading methods 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 system performance (see Sec. V), including lower GPU utilization on servers, longer inference times, and increased energy consumption per inference.

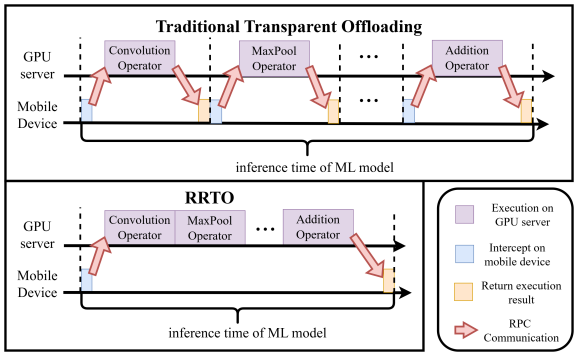


Fig. 6: Workflow of RRTO during the replaying phase.

To address the communication costs in the transparent offloading systems, RRTO introduces an automatic recording and replay mechanism. Given that ML models in mobile applications often exhibit static and predictable sequences of operator invocation during inference, RRTO records the operators called during the initial inferences and replays this recorded sequence for subsequent inferences. As illustrated in Fig. 6, during the replay phase, RRTO only offloads the start operator in the sequence via RPC as done in traditional transparent offloading systems, streamlining the start of the inference task. Subsequent operators are directly invoked on the GPU server side, instead of being invoked by RPCs from the mobile device side as in traditional systems. Consequently, RRTO executes all required operators within the inference operator sequence in one shot at the beginning of each 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. The transparent offloading process is outlined in two parts: the client component is detailed in **Alg. 3**, and the server component in **Alg. 4**.

Algorithm 3 RRTO_on_Client

```
   Input: CUDA API called by the corresponding operator func and the required parameters args
   Output: The execution result ret
   Parameter: inference operator sequence IOS
 1: IOS ← ∅, Logs ← ∅
 2: while True do
 3:     if IOS is NULL then
 4:         // recorder
 5:         SendRPCtoServer(func, args)
 6:         IOS ← OperatorSequenceSearch(Logs)
 7:         ret ←GetRPCExecutionResult()
 8:         Logs ← Logs ∪ {(func, args, ret)}
 9:     else
10:         // replayer on mobile device
11:         if func is IOS[0]["func"] then
12:             StartRRTO(IOS)
13:         end if
14:         if func is "HtoD" then
15:             ret ← SendRPCtoServer(args)
16:         else if func is "DtoH" then
17:             ret ← WaitingForRRTO()
18:         else
19:             idx ← find(IOS, func)
20:             ret ← IOS[idx]["ret"]
21:         end if
22:     end if
23:     ReturnResult(ret)
24: end while
```

In **Alg. 3** on the offloading client side, RRTO takes a CUDA kernel function called by an operator and the requisite parameters as input and keep serving each operator. Initially, the algorithm checks whether the recorder has already identified the inference operator sequence to determine if the current phase should be recording or replaying. If the sequence has not been established, RRTO enters the recording phase (*Lines 4 to 8*), which involve sending an RPC to the server (*Line 5*), performing the operator sequence search (*Line 6*), and capturing and recording the RPC execution result (*Lines 7 to 8*), adhering to the execution pattern of traditional transparent offloading systems. To enhance system efficiency, RRTO overlaps the operator sequence search with the RPC execution, allowing the algorithm to complete while the client awaits the RPC result. In this context, $func$ denotes a specific CUDA API call (e.g., "cudaLaunchKernel", "cudaMemcpyHtoD", and "cudaMemcpyDtoH"). The term $args$ represents the required parameters for that call (e.g., CUDA kernel configurations and memory addresses of parameters). Finally, ret captures the execution result, which is typically a status code like "cudaSuccess" from "cudaLaunchKernel".

If the inference operator sequence is already identified, RRTO transitions to the replaying phase on the mobile device (*Lines 10 to 20*). This phase starts by initiating RRTO for a new inference at the start operator (*Line 12*), then returns the execution results of previous RPC calls for intermediate operators within the sequence (mainly 'cudaSuccess' from "cudaLaunchKernel", *Lines 19 to 20*), synchronizes inference input at "cudaMemcpyHtoD" (*Line 15*) and inference output at "cudaMemcpyDtoH" (*Line 17*). Finally, it returns the execution result of each function to the upper-layer applications (*Line 23*).

Algorithm 4 RRTO_on_Server

```
   Input: client task task
   Parameter: inference operator sequence IOS, the execution result ret
 1: IOS ← ∅
 2: while True do
 3:     if task is SendRPCtoServer then
 4:         func, args ← GetClientInput()
 5:         ret ← CUDARuntimeLibrary(func, args)
 6:     else if task is StartRRTO then
 7:         // replayer on server
 8:         IOS ← GetClientInput()
 9:         for all Op ∈ IOS do
10:             func ← Op["func''], args ← Op["args'']
11:             if func is "HtoD" then
12:                 input ← GetClientInput()
13:                 args ← RRTOFixArgs(args, input)
14:             end if
15:             ret ← CUDARuntimeLibrary(func, args)
16:             if func is "DtoH" then
17:                 SendExecutionResultBack(ret)
18:             end if
19:         end for
20:     end if
21: end while
```

In **Alg. 4**, 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 17*), effectively replaying the execution of the recorded inference operator sequence. During this phase, RRTO configures each operator's parameters (*Line 13*), typically supplying the current inference input data or its memory addresses, to ensure accurate and efficient execution.

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 [31], 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 [62], a transport-independent RPC library for Linux. We integrated RRTO's recorder and replayer into the corresponding RPC

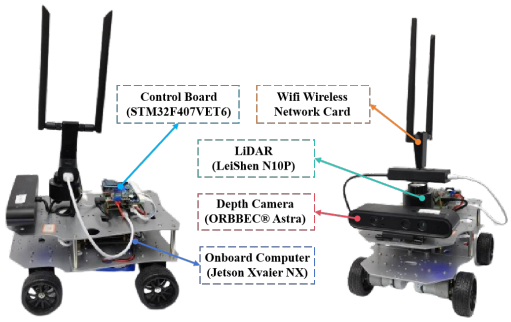


Fig. 7: The detailed composition of the robot platform.

	inference	communication	standby
Energy (Watt)	13.35	4.25	4.04

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

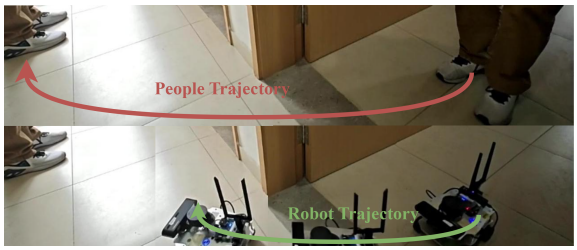


Fig. 8: Kapao [17], a real-time people-tracking application on our four-wheeled robot with a CNN-based human keypoint detection model.

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 [47] 8G onboard computer that is capable of CUDA-acclerated ML model inference. The system runs Ubuntu 20.04 with ROS Noetic and uses a dual-band USB network adapter (MediaTek MT76x2U) for wireless communication. Detailed hardware and sensor configurations are shown in Fig. 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 [47]. 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.

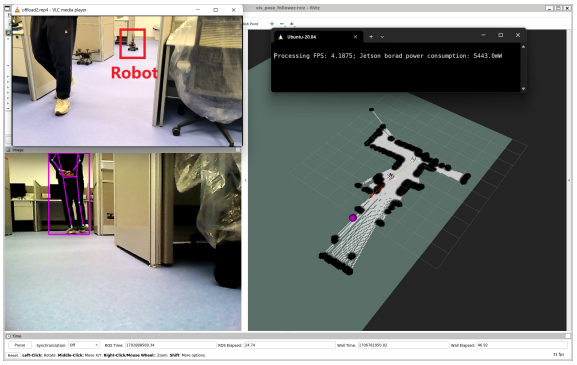


Fig. 9: A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner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

B. Experiment Setup

Task. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig. 8 and Fig. 9. 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 [79]: i) ResNet [80] and ConvNext [81] for object classification; ii) FCN [82] and DeepLabv3 [83] for semantic segmentation; iii) Faster-RCNN [84] and RetainNet [85] 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** [31]: A state-of-the-art transparent offloading system designed for remote GPU usage.

While advanced scheduling optimizations (e.g., layer partitioning [50] and multiple inference scheduling [7], [51], [52]) 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.

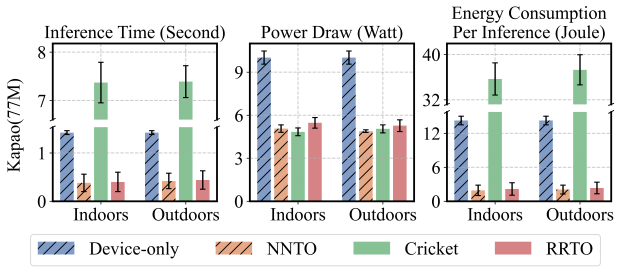


Fig. 10: Performance of Kapao in different environments with various systems.

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.

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 [86] 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 [17] follows the default detection model in Yolo v5 [87]: 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 [36] 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.

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

CUDA Runtime API	Composition during loading model	Composition during initializing inference	Composition during the following 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.

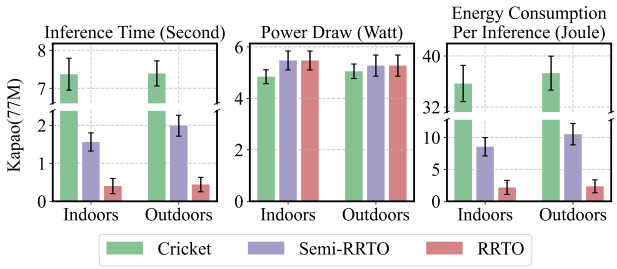


Fig. 11: Semi-RRTO: only applying Caching [63] specifically to the RPCs of "cudaGetDevice" and "cudaGetLastError" in RRTO, effectively eliminating their transmission requirements.

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 [17], measured using pynvml [88] 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 [89], 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 operator sequences. Moreover, tasks that

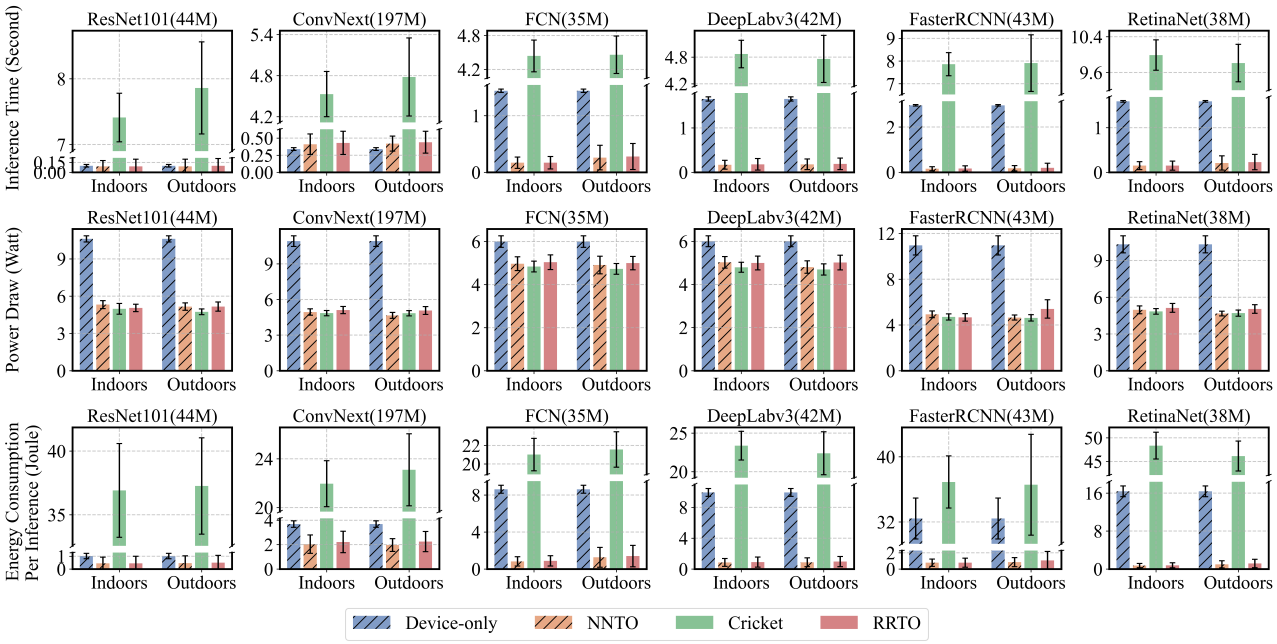


Fig. 12: Performance of Torchvision models in different environments with various systems.

entail complex logic and branching are generally more suited for CPU rather than GPU execution [90], and optimizing inference for DAMs with changing operator sequences remains a pervasive challenge across all offloading systems.

Layer partitioning. Layer partitioning techniques [50] optimize individual inference requests by distributing ML model layers across mobile devices and GPU servers, thereby enhancing end-to-end inference speed and energy efficiency while addressing the transmission failure and network bandwidth fluctuation of wireless links; their widespread adoption in mobile applications has consequently drawn significant research attention, as optimal strategies depend on factors such as model architecture and application-specific trade-offs between these performance metrics. Historically, such layer partitioning has predominantly favored non-transparent offloading systems, largely due to their prioritization of maximal end-to-end performance and minimal transmission overhead over implementation transparency, coupled with the inherent challenge for transparent offloading systems to acquire the model architecture information essential for effective partitioning. However, RRTO offers a path to integrate these established methods within a transparent framework by directly addressing these historical hurdles. Its capability to achieve performance comparable to non-transparent systems alleviates concerns regarding performance prioritization. Furthermore, RRTO overcomes the architectural information barrier by utilizing data dependencies within the inference operator sequence, identifiable through its operator sequence search, to discern the model architecture. This understanding subsequently allows for the adaptation of layer partitioning strategies to RRTO, refining scheduling granularity from the layer level to the more fine-grained operator level, all while maintaining transparency.

Multiple inference Scheduling. It schedules multiple DNN inference requests to optimize overall latency and energy consumption while leveraging existing layer partitioning approaches for individual executions. Various decision algorithms (e.g., urgency-based prioritization [51], deep reinforcement learning-based control [52], and joint optimization of relay selection [7]) are employed to determine execution locations and timing. Crucially, the decision algorithms inherent in such multiple inference scheduling frameworks can be seamlessly integrated into RRTO to determine the execution location and start time of operators within each inference task during its replay process, mirroring their established functionality in non-transparent systems. Consequently, by adopting these proven decision algorithms, RRTO can efficiently support multiple concurrent inference tasks, replicating the high performance already demonstrated by these scheduling methods in non-transparent offloading environments while avoiding the source code modification for each application.

Model Compression. Quantization and model distillation are two most common model compression techniques for mobile devices. Quantization [91] reduces the precision of model weights and activations to lower computation costs, while model distillation [92] trains a smaller model to mimic a larger one with fewer resources. Unlike model compression techniques that trade accuracy for efficiency, offloading methods are orthogonal, achieving fast inference without compromising accuracy by effectively scheduling computational tasks.

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] Y. Tang, Z. Chen, A. Li, T. Zheng, Z. Lin, J. Xu, P. Lv, Z. Sun, and Y. Gao, "MERIT: Multimodal Wearable Vital Sign Waveform Monitoring," *arXiv preprint arXiv:2410.00392*, 2024.

[3] 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.

[4] Z. Lin, L. Wang, J. Ding, B. Tan, and S. Jin, "Channel Power Gain Estimation for Terahertz Vehicle-to-Infrastructure Networks," *IEEE Commun. Lett.*, vol. 27, no. 1, pp. 155–159, 2022.

[5] Z. Fang, Z. Lin, S. Hu, H. Cao, Y. Deng, X. Chen, and Y. Fang, "IC3M: In-Car Multimodal Multi-Object Monitoring for Abnormal Status of Both Driver and Passengers," *arXiv preprint arXiv:2410.02592*, 2024.

[6] Z. Lin, Z. Chen, Z. Fang, X. Chen, X. Wang, and Y. Gao, "Fedsn: A federated learning framework over heterogeneous leo satellite networks," *IEEE Transactions on Mobile Computing*, 2024.

[7] 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.

[8] J. Peng, J. Duan, Z. Lin, H. Yuan, Y. Gao, and Z. Chen, "SigChord: Sniffing Wide Non-Sparse Multiband Signals for Terrestrial and Non-Terrestrial Wireless Networks," *arXiv preprint arXiv:2504.06587*, 2025.

[9] H. Yuan, Z. Chen, Z. Lin, J. Peng, Y. Zhong, X. Hu, S. Xue, W. Li, and Y. Gao, "Constructing 4D Radio Map in LEO Satellite Networks with Limited Samples," *IEEE INFOCOM*, 2025.

[10] Z. Chen, C. Cai, T. Zheng, J. Luo, J. Xiong, and X. Wang, "RF-Based Human Activity Recognition Using Signal Adapted Convolutional Neural Network," *IEEE Trans. Mobile Comput.*, vol. 22, no. 1, pp. 487–499, 2021.

[11] Z. Lin, L. Wang, B. Tan, and X. Li, "Spatial-spectral terahertz networks," *IEEE Transactions on Wireless Communications*, vol. 21, no. 6, pp. 3881–3892, 2021.

[12] Y. Zhang, H. Chen, Z. Lin, Z. Chen, and J. Zhao, "Fedac: An adaptive clustered federated learning framework for heterogeneous data," *arXiv preprint arXiv:2403.16460*, 2024.

[13] H. Yuan, Z. Chen, Z. Lin, J. Peng, Z. Fang, Y. Zhong, Z. Song, and Y. Gao, "SatSense: Multi-Satellite Collaborative Framework for Spectrum Sensing," *IEEE Trans. Cogn. Commun. Netw.*, 2025.

[14] J. Peng, Z. Chen, Z. Lin, H. Yuan, Z. Fang, L. Bao, Z. Song, Y. Li, J. Ren, and Y. Gao, "SUMS: Sniffing Unknown Multiband Signals under Low Sampling Rates," *IEEE Trans. Mobile Comput.*, 2024.

[15] Z. Zhao, Z. Chen, Z. Lin, W. Zhu, K. Qiu, C. You, and Y. Gao, "LEO Satellite Networks Assisted Geo-Distributed Data Processing," *IEEE Wireless Commun. Lett.*, 2024.

[16] Z. Lin, W. Wei, Z. Chen, C.-T. Lam, X. Chen, Y. Gao, and J. Luo, "Hierarchical Split Federated Learning: Convergence Analysis and System Optimization," *IEEE Trans. Mobile Comput.*, 2025.

[17] 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.

[18] Z. Lin, G. Zhu, Y. Deng, X. Chen, Y. Gao, K. Huang, and Y. Fang, "Efficient Parallel Split Learning over Resource-Constrained Wireless Edge Networks," *IEEE Trans. Mobile Comput.*, vol. 23, no. 10, pp. 9224–9239, 2024.

[19] 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.

[20] T. Duan, Z. Zhang, Z. Lin, Y. Gao, L. Xiong, Y. Cui, H. Liang, X. Chen, H. Cui, and D. Huang, "Rethinking adversarial attacks in reinforcement learning from policy distribution perspective," in *ICASSP 2025-2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2025, pp. 1–5.

[21] Z. Lin, G. Qu, Q. Chen, X. Chen, Z. Chen, and K. Huang, "Pushing Large Language Models to the 6G Edge: Vision, Challenges, and Opportunities," *arXiv preprint arXiv:2309.16739*, 2023.

[22] Z. Zhang, T. Duan, Z. Lin, D. Huang, Z. Fang, Z. Sun, L. Xiong, H. Liang, H. Cui, and Y. Cui, "State-aware perturbation optimization for robust deep reinforcement learning," *arXiv preprint arXiv:2503.20613*, 2025.

[23] Z. Lin, Z. Chen, X. Chen, W. Ni, and Y. Gao, "HASFL: Heterogeneity-Aware Split Federated Learning over Edge Computing Systems," *arXiv preprint arXiv:2506.08426*, 2025.

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

[25] S. Lyu, Z. Lin, G. Qu, X. Chen, X. Huang, and P. Li, "Optimal resource allocation for u-shaped parallel split learning," in *2023 IEEE Globecom Workshops (GC Wkshps)*, 2023, pp. 197–202.

[26] Z. Lin, X. Hu, Y. Zhang, Z. Chen, Z. Fang, X. Chen, A. Li, P. Vepakomma, and Y. Gao, "SplitLoRA: A Split Parameter-Efficient Fine-Tuning Framework for Large Language Models," *arXiv preprint arXiv:2407.00952*, 2024.

[27] Z. Sun, X. Guan, Z. Lin, Z. Fang, X. Cai, Z. Chen, F. Liu, H. Cui, J. Xiong, W. Ni *et al.*, "Intra-dp: A high performance collaborative inference system for mobile edge computing," *arXiv preprint arXiv:2507.05829*, 2025.

[28] Z. Lin, G. Qu, W. Wei, X. Chen, and K. K. Leung, "Adaptsfl: Adaptive Split Federated Learning in Resource-Constrained Edge Networks," *IEEE Trans. Netw.*, 2024.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

[60] 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.

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

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

[63] 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.

[64] 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.

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

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

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

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

[76] 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.

[77] 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.

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

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

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

[81] 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.

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

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

[84] 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.

[85] 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.

[86] 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.

[87] 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.

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

[89] 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.

[90] 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.

[91] 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.

[92] 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.