RRTO: A High Performance Transparent Offloading System for Model Inference on Robotic IoT

Zekai Sun, Xiuxian Guan Junming Wang, Fangming Liu, Heming Cui

***Abstract*—Fundamental robotic tasks, such as object identification and robot control, increasingly rely on Machine Learning (ML) models deployed on wireless robots. The heavy computation of these tasks, resulting from the large number of parameters and complex operations (e.g., matrix multiplication, convolution, pooling) in ML models, is often offloaded to powerful GPU devices (e.g., cloud data centers) via the Internet of Things for these robots (robotic IoT) to achieve fast and energy-efficient inferences. Existing computation offloading systems can be categorized into two types: transparent and non-transparent methods, depending on whether source code modification is required to enable offloading. While non-transparent offloading systems have yielded success across various ML models, they also pose significant obstacles to the deployment of robots since they require considerable coding effort to modify the source code for each application and are inapplicable to closed-source applications. Meanwhile, existing transparent offloading methods can offload the function call of each operator within the ML models via Remote Procedure Call (RPC) to avoid source code modifications, but such one-by-one handling of RPCs leads to substantial communication costs in robotic IoT.**

We present RRTO, the first high performance transparent offloading system optimized for ML model inference on robotic IoT with a novel record/replay mechanism. Recognizing that the operators invoked by ML models in robotic tasks often follow a fixed order, RRTO automatically records and identifies the order of operators invoked by the ML model and replays the execution of these fixed-order operators during model inference. In this way, RRTO correctly calls all involved operators in the recorded order of each inference of the ML model via one RPC, circumventing the inherent communication costs caused by existing transparent offloading mechanism without the burden of source code modification of the non-transparent offloading methods. Evaluations demonstrate that RRTO improved the performance of our real-world robotic application by 4.9x $\sim$ 48.5x and saved 8% $\sim$ 53% power consumption on our robot compared to other baselines without modifying any source code, achieving similar high performance as state-of-the-art non-transparent offloading methods.

***Index Terms*—Computation Offloading, Model inference, Robotic IoT, Distributed system and network**

I. Introduction

The rapid advancement of machine learning (ML) methods has achieved remarkable success in various fundamental robotic tasks, such as object detection [1]–[3], robot control [4]–[6], and environmental perception [7]–[9]. Deploying these ML methods onto real-world robots typically requires additional hardware support due to the computationally-intensive nature of ML models (e.g., the large number of parameters, complex operations). However, directly integrating computing accelerators (e.g., GPU [10], FPGA [11], SoC [12]) onto real-world robots not only introduces additional economic costs, but also leads to increased energy consumption (e.g., 62% for KAPAO [3] on our robot), shown as "Local" in Fig. 2. Consequently, instead of performing computations locally, these ML applications often offload the computation of ML models to cloud data centers or edge devices equipped with powerful GPUs through the Internet of Things for these robots (robotic IoT).

Existing computation offloading systems can be categorized into two types: transparent and non-transparent methods, depending on whether source code modification is required to enable offloading. **Non-transparent** offloading systems, such as MCOP [13], have demonstrated success across various ML models. MCOP adaptively schedules the computation of ML models to be offloaded to cloud servers based on the model's

Heming Cui is the corresponding author.

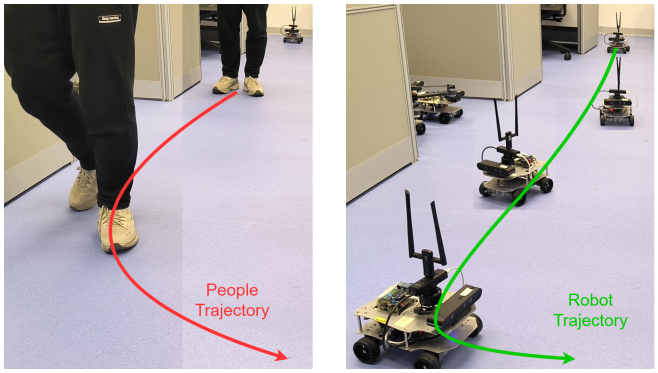
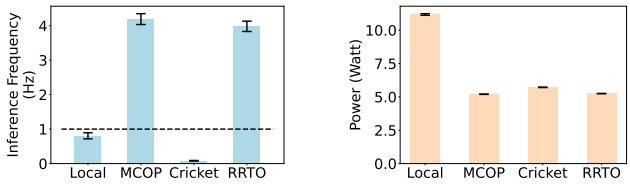


Fig. 1. A real-time people-tracking robotic application on our robot based on a well-known human pose estimation ML model, KAPAO [3].



(a) Average inference frequency with standard deviation.

(b) Average power consumption on our robot with standard deviation.

Fig. 2. Comparison between RRTO and the baselines on the application in Fig. 1. The higher frequency indicates higher people recognition frequency and more real-time people tracking. "Local" refers to inference executed using an integrated computing accelerator on the robot, while "MCOP" [13] is a state-of-the-art (SOTA) non-transparent offloading system, and "Cricket" [14] is a SOTA transparent offloading system.

workload, network bandwidth, and the computing power of cloud servers. This approach successfully improved the performance of our real-world robotic application by 5.2x and saved 54% of power on our robot in our experiments, shown as "MCOP" in Fig. 2. Such non-transparency provides them with a simple and efficient scheduling method but demands significant coding effort to modify the source code for each application and can not be used on closed-source applications.

Transparent offloading methods, such as Cricket [14], provide a more convenient but less efficient approach to offload computation to the cloud servers. ML model inference consists of a series of operators (e.g., addition, convolution). Transparent offloading methods intercept the call of each operator to the corresponding system functions (e.g., *torch.add()*,*torch.convolution()* for PyTorch [15]) and offload all these calls to the cloud servers through Remote Procedure Call (RPC). In this way, they avoid modifications to the source code by intercepting at the system layer but have to offload the RPC calls to the cloud servers one by one and add one Round-Trip Time (RTT) of RPC to the completion time of each operator, bringing extra communication costs.

Moreover, such inherent communication costs caused by the transparent offloading mechanism becomes substantial in robotic IoT networks. ML models commonly have hundreds of operators (e.g., 522 for KAPAO [3]), and each operator usually requires several RPC functions to complete based on different ML frameworks (e.g., cudaGetDevice, cudaLaunchKernel for PyTorch [3]), resulting in hundreds or thousands of RPC calls for a single inference (e.g., 5895 in our experiments). In robotic IoT networks, which typically rely on wireless communication for robots, each RPC RTT usually takes a few milliseconds [16] (average 2.6 milliseconds in our experiments). Consequently, transparent offloading in robotic IoT networks, shown as "Cricket" in Fig. 2, leads to prolonged communication time (98% inference time in our experiments) and may even increase the total energy consumed for each inference, despite reducing power consumption on the robot by offloading.

The key reason for this problem is that existing transparent offloading methods suffer from significant communication costs because they only invoke RPCs when an operator is utilized during the inference process, rather than anticipating their use in advance. This is because these methods are designed for general applications to leverage remote GPUs, rather than specifically designed for ML model inference. Such communication costs are unavoidable in general applications, as they cannot predict future operators from the upper-layer applications and thus cannot call RPCs in advance. Fortunately, in the context of ML model inference of robotic applications, we observe that these ML models are often static, meaning that the operators invoked during inference follow a fixed and predictable order (see Sec. III-A for more details), allowing for a more efficient inference process.

Based on this observation, we present **RRTO**, a **T**ransparent **O**ffloading system for model inference on robotic IoT with a novel **R**ecord/**R**eplay mechanism: record the order of operators invoked by ML model automatically and replay the execution of these fixed-order operators during model inference. In this way, RRTO correctly calls all involved operators in the recorded order of each inference of the ML model via one RPC, without waiting for the operator to be called via RPCs during inference. Based on this mechanism, RRTO dramatically reduces the inherent communication cost caused by traditional transparent offloading mechanism and achieves nearly the same communication cost as non-transparent offloading methods, allowing the outstanding scheduling algorithms in non-transparent offloading methods to be utilized in transparent offloading.

However, it is non-trivial to identify the specific operators invoked during each inference, as from the system layer only the continuous sequence of operator calls is visible to RRTO. RRTO must explicitly identify the operators forming an inference from the the log records of operators sequence without the hints from the upper-layer applications, so that RRTO can correctly handle the offloading of inference via one RPC call.

To address this challenge, RRTO proposes a novel algorithm called *Data Dependency Search* to identify the sequence of operators invoked for each inference. This algorithm first constructs a relationship graph based on the data dependencies between operators (i.e., the previous operator's calculation result serves as the next operator's input). It then takes operators that do not depend on any other operators as starting points and operators that no other operators have any dependencies on as ending points. The algorithm subsequently searches for the longest covering operator sequence between starting and ending operators and verifies whether such an operator sequence can constitute a complete model inference process (i.e., the entire log records of operators can be covered by repeating this sequence). Another observation to this challenge is that the final computation result needs to be sent back from GPU to CPU, and this observation also aids this algorithm in identifying the correct ending operator and dramatically reduces its search space.

We implemented RRTO on the Cricket's codebase [14] and also incorporated the outstanding scheduling algorithms of MCOP [13] into RRTO. We evaluated RRTO on a real-world Jetson Xavier NX [17] robot capable of GPU accelerated computation with a robotic application that tracks people in real time [3]. We compared RRTO with local computation, a SOTA non-transparent offloading method (MCOP [13]) and a SOTA transparent offloading method (Cricket [14]) when offloading computation to different GPU devices (namely edge devices with high bandwidth and cloud servers with limited bandwidth). The evaluation shows that:

- RRTO is fast. It reduced inference time by 80% $\sim$ 98% compared to other baselines, similar to the reduction achieved by the SOTA non-transparent offloading method (MCOP).
- RRTO is energy-efficient. It reduced 91% $\sim$ 98% energy consumption per inference compared to other baselines, similar to the performance of MCOP.
- RRTO is robust in various robotic IoT environments. When the robotic IoT environment (the bandwidth to the GPU devices and the computing power of the GPU devices) changed, RRTO's superior performance remained

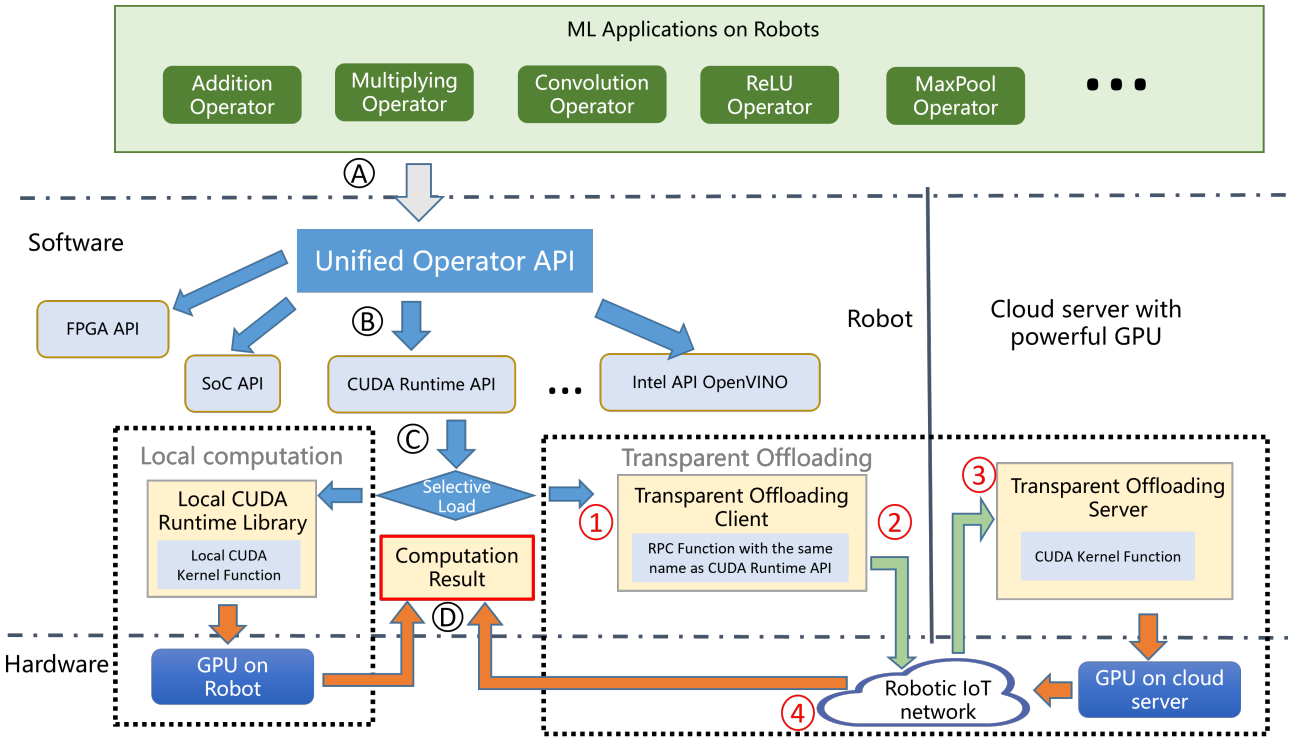


Fig. 3. Workflow of Transparent Offloading System for Model Inference on Robotic IoT.

consistent as MCOP.

Our main contribution is RRTO, the first high-performance transparent offloading system designed for model inference on robotic IoT. RRTO dramatically reduces the inherent communication cost caused by the traditional transparent offloading mechanism via the novel record/replay mechanism, achieving the same high performance as the SOTA non-transparent offloading method without modifying any source code. We envision that RRTO will foster the deployment of diverse robotic tasks on real-world robots in the field by providing fast, energy-efficient, and easy-to-use inference capabilities. RRTO's code is released on https://github.com/rss24p23/RRTO.

In the rest of this paper, we introduce the background of this paper in Sec. II, give an overview of RRTO in Sec. III, present the detailed design of RRTO in Sec. IV, evaluate RRTO in Sec. VI, and finally conclude in Sec. VII.

II. Background

A. Workflow of Transparent Offloading

When a robot performs GPU computations locally, the system call flow of the entire application can be depicted as the left part in Fig. 3:

A The robot application completes the entire computation process for each service by sequentially calling different operators.

B Based on the application's running device (GPU), each operator passes through a unified operator API to find the local CUDA runtime library (NVIDIA GPUs provide high-performance parallel computing capabilities to applications using the CUDA runtime library [18]).

C The local CUDA runtime API is loaded by default.

D The robot's local CUDA library launches the corresponding CUDA kernel functions on the robot's GPU and returns the computation results to the upper-layer application.

Transparent offloading methods [14], [19], [20] usually takes the approach of rewriting dynamic link libraries, defining functions with the same name and using the *LD_PRELOAD* environment variable to prioritize loading the custom dynamic link library. The dynamic linker will then parse the original library function as the custom library function, thereby achieving library function interception. Subsequently, by modifying the management of GPU memory and the launch of CUDA kernel functions, the computation-related data and specific parameters of the corresponding kernel functions are sent to the remote server via RPC, realizing GPU computing transparent offloading. The primary modification occurs in step C. Similar to completing computations locally using the robot's GPU, after steps A and B, each operator's call to the corresponding kernel functions is intercepted by the functions with same name in the dynamic link library and offloaded to the cloud server to execute. The detailed steps (depicted in the right part in Fig. 3) are as follows:

(1) By modifying the dynamic link library, each operator prioritizes calling the RPC functions with the same name as the CUDA runtime API in transparent offloading client, thereby identifying and intercepting all CUDA kernel

function calls.
(2) The transparent offloading client transmits the called CUDA runtime API and required parameters to the cloud server through the robotic IoT network via RPC.
(3) The transparent offloading server launches the corresponding CUDA kernel functions on the cloud GPU and completes the respective computation.
(4) The transparent offloading server sends the computation results back to the client and the transparent offloading system returns the results to the upper-layer application.

B. Non-Transparent Offloading

Non-transparent offloading methods require access to source code to enable the interception of the entire inference process and to optimize it via layer partition [13], [21], [22]. Layer partitioning involves placing parts of models on robots and parts on GPU servers at the granularity of layers to achieve the fastest possible inference time, exploiting the fact that the output data in some intermediate layers of a DNN model is significantly smaller than its raw input data [23]. To employ layer partition, these methods need to obtain the model's structure (e.g., the computation time on each layer's robot and GPU, and the transfer time to pass intermediate results of this layer to the GPU server) and constitute various trade-offs between computation and transmission, considering the model's workload, network bandwidth, and the computing power of the cloud server, as well as application-specific requirements. A layer of the model usually corresponds to several fixed operators (e.g., the convolution layer in [3] will call a convolution operator through the "cudaLaunchKernel" function during inference as described in Sec. VI), and the model's structure can be obtained by recording operators, allowing the scheduling algorithm of layer partition to be adopted on transparent offloading systems by downgrading its granularity from layer to operator.

The non-transparency of these methods brings inconvenience when deploying various applications on robots, which can be solved by transparent offloading. Non-transparent offloading requires modifying the source code, demanding coding effort and engineering experience, while transparent offloading intercepts function calls at the system layer without modifying the source code. The process must be performed for every application and cannot be used on closed-source applications, while transparent offloading works at the system layer and is independent of the upper-layer application.

In summary, non-transparent offloading systems are high-performance but not easy to use, while transparent offloading systems are convenient but have poor performance. RRTO combines the advantages of both approaches, offering a solution that is both high-performance and user-friendly.

C. Related Work

Model Compression. Quantization and model distillation are the two most commonly used methods of ML model compression on the robots. Quantization [24]–[26] is a technique that reduces the numerical precision of model weights and activations, thereby minimizing the memory footprint and computational requirements of deep learning models. This process typically involves converting high-precision (e.g., 32-bit) floating-point values to lower-precision (e.g., 8-bit) floating-point representations, with minimal loss of model accuracy. Model distillation [27]–[29], on the other hand, is an approach that involves training a smaller, more efficient "student" model to mimic the behavior of a larger, more accurate "teacher" model by minimizing the difference between the student model's output and the teacher model's output. The distilled student model retains much of the teacher model's accuracy while requiring significantly fewer resources. These model compression methods are orthogonal to offloading methods, because they achieve faster inference speed by modifying the model and sacrificing the accuracy of the result, while offloading realizes fast inference without loss of accuracy by scheduling the calculation tasks.

RPC Optimization. RPC [30] is a communication protocol that enables one process to request a service from another process located on a remote computer, typically over a network. To improve RPC performance, several optimization strategies can be employed to achieve more efficient communication between remote processes and an overall enhancement in system performance: Batching [31] (aggregate multiple RPC calls into a single request), Asynchronous RPC [32] (decouple the request and response processing), and Caching [33] (Store the results of previous RPC calls). However, these optimization strategies are not effective in reducing the communication cost during model inference in robotic IoT. During the model inference process, the next operator is typically called after the previous operator has completed its execution, which renders Batching ineffective. While Asynchronous RPC and Caching enable the client to continue executing other tasks (subsequent operators) without waiting for the server's response, they lack the ability to determine when to stop and obtain the correct computation results. Compared with the traditional RPC optimization strategies, RRTO further reduces the communication cost by avoiding most operator's corresponding RPC communication, which will be described with more details in Sec. III and can be considered as a specific co-design for RPC optimization strategy and transparent offloading system for model inference.

Multiple Inference Scheduling. Multiple Inference Scheduling has been a significant research focus, aiming to accelerate multiple DNN inference tasks by optimizing their execution on various devices under different network bandwidths while considering application-specific inference speed requirements and energy consumption demands. Methods such as [34]–[36] support online scheduling of offloading inference tasks based on the current network and resource status of mobile systems, meeting user-defined energy constraints, and optimizing DNN inference workloads in cloud computing using deep reinforcement learning based schedulers for QoS-aware scheduling of heterogeneous servers. Although these methods target overall optimization in multi-task scenarios involving multi-robots, they are orthogonal to the optimization of an individual offloading system for single inference, but instead focus on coordinating

the overall offloading systems on a cluster of robots for multiple inference tasks to optimize the overall inference latency and power consumption. A higher-performance offloading system can provide them with a larger, more flexible scheduling space. Due to the poor performance of traditional transparent offloading systems caused by communication costs, existing multiple inference scheduling methods are mainly based on non-transparent offloading systems. However, if a transparent offloading system, such as RRTO, achieves the same high performance as these non-transparent offloading systems, these multiple inference scheduling methods can also be adopted to provide a high-performance, transparent offloading solution supporting advanced scheduling techniques. And we leave it as RRTO's future work.

III. Overview

A. ML Models with Fixed-Order Operators

In machine learning, many models have a fixed order in activating their layers during the inference process, referred to as static ML models. These include: 1. feed-forward neural networks (e.g., Multi-Layer Perceptrons [37], Convolutional Neural Networks [38]) with a fixed structure where neurons in each layer are activated sequentially given an input; 2. Recurrent Neural Networks (e.g., simple RNNs [39], Elman networks [40]) that have a fixed computation process at each time step, despite their ability to handle variable-length sequences; 3. Autoencoders (e.g., basic autoencoders [41], Variational Autoencoders [42]) with fixed encoder and decoder parts that activate the same components during each inference; 4. Generative Adversarial Networks (e.g., basic GANs [43], DCGANs [44]) with fixed generator and discriminator structures; 5. shallow machine learning models (e.g., linear regression [45], logistic regression [46], Support Vector Machines [47]) that typically have a single fixed layer. These models with fixed structures and no dynamic mechanisms have relatively simple and regular computation processes and are the targeted models of our RRTO and the baselines.

On the other hand, some models may activate different layers during different inferences depending on the input, and are referred to as dynamic ML models. These include 1. models with attention mechanisms (e.g., Transformers [48], BERT [49]) that dynamically compute attention weights to focus on different parts of the input; 2. gated models (e.g., LSTM [50], GRU [51]) that control information flows through gating units, leading to different activated parts based on the input; 3. conditional computation models (e.g., Mixture of Experts [52]) that select different experts to activate based on input conditions; and 4. dynamic network models (e.g., those obtained through Neural Architecture Search [53]) that can dynamically adjust their structure based on the input. The dynamic nature of some ML models allows for better adaptability and expressiveness, but this nature also makes it difficult to profile their running statistics (time consumed, size of input and output, etc.) and thus few optimizing systems are targeted on these models.

Static ML models are widely used in robotic applications, such as Convolutional Neural Networks for computer vision tasks [3], [54], [55], and Multi-Layer Perceptrons, Recurrent Neural Networks, and Support Vector Machines for robot manipulation and automatic navigation [56]–[58]. On the other hand, dynamic ML models often require more computing resources and GPU storage, leading to their deployment mainly in data centers rather than on robots [48], [49], [52].

B. Workflow of RRTO

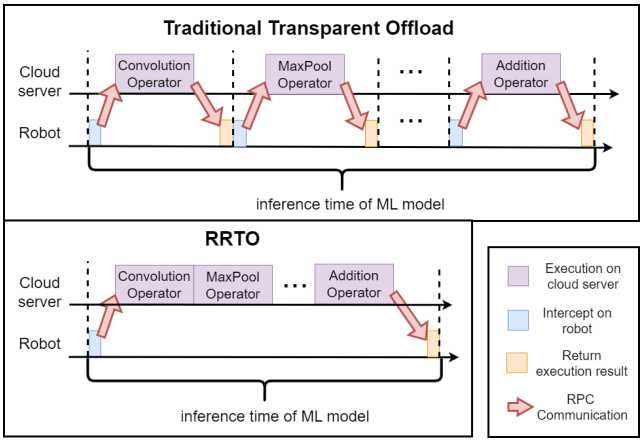


Fig. 4. Workflow of RRTO.

Fig. 4 illustrates the workflow of RRTO and contrasts it with traditional transparent offloading systems during model inference in robotic IoT networks. Traditional transparent offloading systems suffer from frequent RPC communication of operators, resulting in substantial communication cost and diminished system performance, including reduced GPU utilization on cloud servers, extended model inference time, and increased energy consumption per inference. The RTT communication cost for each operator is relatively minimal in data center networks, where devices are connected with high-speed (e.g., 40 Gbps ˜500 Gbps) networking technologies, such as InfiniBand [59] or PCIe [60]. However, in robotic IoT networks, the bandwidth between robots and powerful GPU devices is more limited and ML models commonly have hundreds of operators (e.g., 522 for [3]). For edge devices connected via Wi-Fi 6, the actual bandwidth reaches only 450 Mbps, leading to an average RTT of 2.6 milliseconds for each operator and the communication cost accounts for 98% of the total inference time in our experiments. For cloud servers, the even lower bandwidth imposed by the internet further hinders performance, as evaluated in Sec. VI.

To address the communication cost issue in transparent offloading processes of ML models, RRTO introduces an automatic recording and replay mechanism. Since ML model inference can be regarded as a complex function calculation, ML models typically invoke corresponding operators (e.g., addition, convolution, max-pool) in a fixed order to obtain accurate computation results and repeat these operators for each subsequent inference process. RRTO records the operators called during the first few inferences and replays the execution of this recorded sequence, referred to as the *inference operator sequence*, for subsequent inferences.

By employing this approach, RRTO only requires the first and last operators in the inference operator sequence to be offloaded via RPC, as in traditional transparent offloading systems, to obtain the correct input and output of ML models. For the operators in the middle of the inference operator sequence, RRTO directly calls these operators on the offloading server side without requiring any extra RPC communication from the offloading client side, thus avoiding the inherent communication cost associated with these operators.

Notice that, although there has been substantial work on optimizing RPC communication [31]–[33], RRTO goes a step further by directly eliminating the RPC communication of operators in the middle of the sequence. While existing RPC optimization methods still wait for RPCs from the offloading client to instruct the offloading server on the subsequent functions to be executed, RRTO preemptively calls the corresponding operators' functions on the offloading server side.

IV. Design

A. Architecture of RRTO

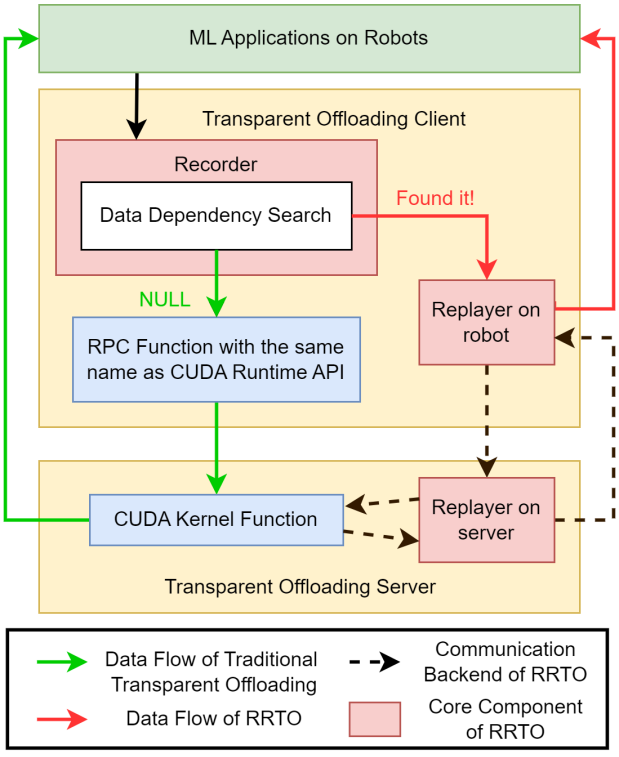


Fig. 5. Architecture of RRTO.

Fig. 5 presents the architecture of RRTO. Compared with Fig. 3, RRTO implements its record/replay mechanism based on the core components of existing transparent offloading systems, and retains transparency to upper-layer applications, meaning that RRTO still does not require modifications to the source code to enable offloading. The pseudo codes for the client and the server sides of RRTO are provided in Sec. IV-B.

During the first several inferences, RRTO enters the recording phase, following the same execution pattern as traditional transparent offloading systems by offloading the operator's execution to the cloud server via RPC, as depicted by the green lines in Fig. 5. Upon identifying and intercepting CUDA kernel function calls of operators from upper-layer ML applications, RRTO first records the function called by the operator, including the required parameters and the return value, using its recorder. It then attempts to find the inference operator sequence through a data dependency search, which will be described in more detail in Sec. IV-C.

Once the recorder identifies the inference operator sequence, RRTO transitions to the replaying phase, initiating the replaying of the execution of the inference operator sequence for subsequent inferences using the replayer on both the robot and server, as illustrated by the red lines in Fig. 5. Similar to Caching (described in Sec. II-C) in existing RPC optimization methods, the replayer on the robot returns the execution results of previous RPC calls to the upper-layer applications, allowing the offloading client to continue execution until it is blocked at the ending operator to receive the final computation result from the offloading server. Meanwhile, the offloading server replays the execution of the inference operator sequence sent by the client and returns the final computation result to the offloading client. This approach enables RRTO to achieve a communication cost nearly equivalent to that of non-transparent offloading methods, as it needs to transmit almost the same input and output as these methods, as depicted by the black dotted lines in Fig. 5.

We have adopted existing optimizations of layer partitions from non-transparent offloading systems [13] onto RRTO, as described in Sec. II-B, by downgrading the granularity of its scheduling algorithm from layer to operator. Instead of requiring the model structure like non-transparent offloading systems, RRTO obtains the model structure expressed as operators (the operator sequence for a single inference) via its record/replay mechanism. Similar to layer partition, RRTO's operator partition places some operators to be executed on robots, while others are executed on GPU servers via RRTO's communication backend, as a layer of the model usually corresponds to several fixed operators. This approach allows RRTO to utilize optimizations from non-transparent offloading systems (mainly layer partition) to enhance its performance. Furthermore, RRTO's operator partition provides a fine-grained and more flexible offloading schedule method, which can be utilized to handle fluctuations in the wireless network bandwidth of robotic networks, and we leave it as future work, as described in Sec. VI-D.

RRTO's record/replay mechanism only fails when the operator sequence changes, which occurs in dynamic ML models. Upon prediction failure (the replayer on robot found that the actual operator calls are inconsistent with the sequence of operators found), RRTO stops the record/replay mechanism and restarts the above process until a new operator sequence is found. Fortunately, ML models in robotic applications are often static (see Sec. III-A), making RRTO's record/replay mechanism rarely fail in robotic applications, preventing RRTO from degenerating into traditional transparent offloading systems. Notice that optimizing inference for dynamic ML models with changing operator sequences remains an open issue for offloading systems, and dynamic ML models are

not the targeted models of our RRTO and existing offloading systems. Non-transparent offloading systems cannot actively optimize such models due to their reliance on the same assumption as RRTO of fixed operator order, as they have to obtain the model's structure and running statistics to schedule and optimize (see Sec. II-B), while traditional transparent offloading systems still suffer from high communication costs. Some work [61] try to optimize the inference for dynamic ML models with changing operator sequences by statistically predicting which layers will be activated during the next inference, which is beyond the scope of this paper.

B. Record/Replay Mechanism

Algorithm 1 RRTO_on_Client

KwInputInput KwOutputOutput Cuda kernel function called by the corresponding operator $func$ and the required parameters $args$ The execution result ret inference operator sequence $IOS = \emptyset$
$IOS.empty()$ recorder
$SendRPCtoServer(func, args)$
$IOS = DataDependencySearch(func, args)$
$ret = GetRPCExecutionResult()$
$RecordReturn(ret)$ replayer on robot
$func == IOS.start()["func"]$
$ret = StartRRTO(args, IOS)$
start a new inference
$func == IOS.end()["func"]$
$ret = WaitingForRRTO()$
Waiting for the final computation result
$ret = IOS.find(func)["ret"]$
ret

Here we present how RRTO achieves the record/replay mechanism. The transparent offloading client part is given in Alg. 1 and the server part is given in Alg. 2. Details of data dependency search are mainly described in the next subsection IV-C.

In Alg. 1, on the offloading client side, RRTO takes as input a CUDA kernel function called by the corresponding operator and the required parameters. The algorithm first checks whether the recorder has already identified the inference operator sequence (line 1). If the sequence has not been found, RRTO proceeds with the recorder phase, which includes sending an RPC to the server (line 2), performing a data dependency search (line 3), and obtaining and recording the RPC execution result (lines 4, 5). To enhance system efficiency, RRTO overlaps the data dependency search with the execution of the RPC, allowing the DataDependencySearch algorithm calculation to be completed while the client awaits the RPC execution result. If the inference operator sequence has been identified, RRTO proceeds with the replayer phase on the robot. This phase involves initiating RRTO for a new inference at the first operator (line 9), returning the execution results of previous RPC calls at the intermediate operators within the inference operator sequence (line 15), and waiting for the final computation result at the last operator (line 12).

In Alg. 2, the RRTO offloading server continuously awaits tasks from the client and returns the final execution results. If

Algorithm 2 RRTO_on_Server

KwInputInput KwOutputOutput
client task $task$ inference operator sequence $IOS = \emptyset$, the execution result ret $task == SendRPCtoServer$
$func, args = GetClientInput()$
$ret = CUDARuntimeLibrary(func, args)$
replayer on server
$args, IOS = GetClientInput()$
$Op \in IOS$
$args = RRTOFixArgs(Op["args"], ret, args)$
$ret = CUDARuntimeLibrary(Op["func"], args)$
$SendExecutionResultBack(ret)$

the client is still in the recorder phase, the RRTO offloading server processes RPC requests in the same manner as traditional transparent offloading systems (lines 2, 3). Once the client enters the replayer phase on the robot, the RRTO offloading server correspondingly transitions into the replayer phase on the server, replaying the execution of the inference operator sequence identified by the client (lines 8, 9). During this process, RRTO needs to adjust the parameters required by the corresponding operators, which typically consist of data or addresses of the computation results from the previous operators within the current inference.

C. Algorithm of Data Dependency Search

The performance of RRTO is heavily dependent on its ability to identify the correct inference operator sequence. If the sequence is not found accurately, even with a discrepancy of just one operator more or less, RRTO will not be able to obtain the correct inference result. Identifying the inference operator sequence is challenging, as RRTO must maintain its transparency and cannot receive any hints from upper-layer applications regarding which operators are invoked for each inference. Instead, it can only rely on log records of operators for the first few inferences to identify the starting and ending operators and operators between the two.

Algorithm 3 DataDependencySearch

KwInputInput KwOutputOutput
Cuda kernel function called by the last operator $func$ and the required parameters $args$ inference operator sequence IOS Log records $history = \emptyset$ and relationship graph $map = \emptyset$
$history.add(func)$
$map.update(func, args)$
$StartPoses = map.startposes()$
$EndPoses = map.endposes()$
$Sequence = FindLongestPair(StartPoses, EndPoses)$
$Verify(history, Sequence)$ $Sequence \quad NULL$
cannot find

The pseudo code for data dependency search is provided in Alg. 3. To identifying the inference operator sequence, RRTO first records the data dependencies between operators (i.e., the calculation result of the previous operator serves as input for the next operator) and constructs a relationship graph (line 2). These dependencies can be established by comparing whether

parameters and calculated results between operators are the same (i.e., having the same address). Then, RRTO attempts to find the inference operator sequence based on this relationship graph.

RRTO considers operators that do not depend on any other operators as starting operators (line 3), which are candidates for the first operator in the inference operator sequence. It also considers operators that no other operators depend on as ending operators (line 4), which are candidates for the last operator in the inference operator sequence. RRTO searches for the longest covering operator sequence between starting and ending operators (line 5). Finally, RRTO verifies whether such an operator sequence can constitute a complete model inference process (line 6) by checking if the entire log records of operators can be covered by repeating this sequence.

During our implementation, we discovered that the final computation result needs to be sent back from GPU to CPU, and the ending operator can be identified by comparing the address of the final computation result. This specific observation aids RRTO in identifying the correct ending operator and dramatically reduces its search space. However, this does not entirely eliminate the need for establishing the relationship graph, because when the model has multiple outputs, an inference will involve multiple data copies from GPU to CPU and the Data Dependency Search algorithm is still needed to find the correct ending operator.

V. Implementation

We implemented RRTO within Cricket's codebase [14], a transparent offloading system that provides a virtualization layer for CUDA applications, enabling remote execution without the need for recompiling applications. RRTO employs the same Remote Procedure Call (RPC) for communication operations as Cricket: Libtirpc [30], a transport-independent RPC library for Linux. We integrate RRTO's recorder and replayer into the corresponding RPC functions in Cricket and adapts MCOP's scheduling approach [13] by refining the scheduling granularity from MCOP's layers of ML models to the more fine-grained operators.

VI. Evaluation

Testbed. The evaluation was conducted on a customized four-wheeled robot (Fig. 6), equipped with a Jetson Xavier NX [17] 8G onboard computer serving as the ROS master. The system runs Ubuntu 20.04 and utilizes a SanDisk 256G memory card, with ROS1 Noetic installed for application development and a dual-band USB network card (MediaTek MT76x2U) for wireless connectivity. The Jetson Xavier NX interfaces with a Leishen N10P LiDAR, ORBBEC Astra depth camera, and an STM32F407VET6 controller via USB serial ports. Both the LiDAR and the depth camera facilitate environmental perception, enabling autonomous navigation, obstacle avoidance, and SLAM mapping. The onboard computer processes environmental information in ROS1 Noetic, performing path planning, navigation, and obstacle avoidance before transmitting velocity and control data to corresponding

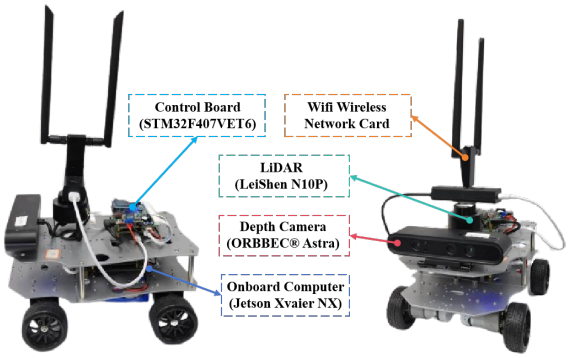


Fig. 6. The detailed composition of the robot platform.

	inference	communication	standby
Power (W)	13.35	4.25	4.04

TABLE I
Power (Watt) of our robot in different states.

ROS topics. The controller then subscribes to these topics and executes robot control tasks.

We documented the overall on-board energy consumption (excluding motor energy consumption for robot movement) of the robot in various states, as presented in Table I. These states include: inference, which refers to model inference with the full utilization of GPU and encompasses the energy consumption of both the CPU and GPU; communication, which involves communication with the server and includes the energy consumption of the wireless network card; and standby, during which the robot has no tasks to execute.

We evaluated two prevalent offloading scenarios for ML applications on robots, referred to as edge and cloud scenarios. In the edge scenario, computation is offloaded to an edge device, which is a PC equipped with 8xIntel(R) Core(TM) i7-7700K CPU @ 4.20GHz and NVIDIA GeForce GTX 1080 Ti 11GB, connected to our robot via Wi-Fi 6 with an average bandwidth of 450 Mbps over 160MHz channel at 5GHz frequency in our experiments. In the cloud scenario, computation is offloaded to a cloud server, which is a GPU server equipped with 48xIntel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz and 4xNVIDIA GeForce RTX 2080 Ti 11GB, connected to our robot via the Internet with an average bandwidth of 160 Mbps in our experiments.

Real-World Robotic Application. We evaluated a real-time people-tracking robotic application on our robot as depicted in Fig. 7. To achieve seamless tracking of individuals, a minimum detection frequency of 1 Hz for the target person is required, illustrated by the black dotted line in Fig. 2a and Fig. 9a. The detailed workflow is described as follows: The ORBBEC Astra depth camera on our robot generates both RGB images and corresponding depth images. First, we obtain a person's key points in the RGB image using a well-known human pose estimation model based on Convolutional Neural Networks, KAPAO [3]. Then, by utilizing the depth values corresponding to these key points in the depth image, the points are mapped to a three-dimensional map constructed by the robot's LiDAR. A Kalman filter [62] is applied to filter

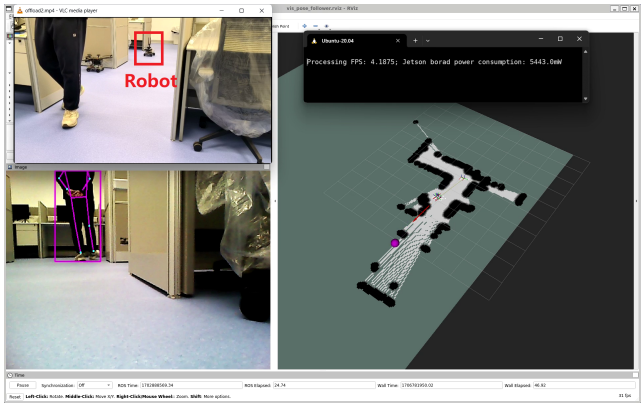
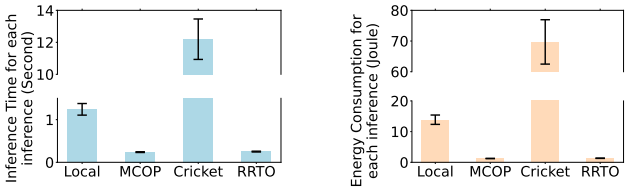


Fig. 7. A screenshot of our real-world experiment. The upper right corner displays real-time FPS and on-board energy consumption, the lower right corner shows the map created by the robot using its LiDAR, the lower left corner features the real-time view from the robot's camera, and the upper left corner provides a third-angle observation of the entire experimental process.



(a) Average Inference Time with standard deviation.

(b) Average Energy Consumption with standard deviation.

Fig. 8. Comparison between RRTO and the baselines per inference in the edge scenario.

out noise and obtain a more accurate position of the person. Finally, the STM32F407VET6 controller directs the robot to the target position, enabling real-time tracking of the person. KAPAO continuously performs inference to achieve the fastest possible inference speed using both baselines and RRTO.

Baselines. We compared RRTO with local computation, MCOP [13] (a SOTA non-transparent offloading system) and Cricket [14] (a SOTA transparent offloading system) when offloading computation to different GPU devices (an edge device with high bandwidth and a cloud server with limited bandwidth).

The evaluation questions are as follows:

- RQ1: How does RRTO benefit real-world robotic applications compared to baseline systems in terms of inference time and energy consumption?
- RQ2: How does RRTO's record/replay mechanism work?
- RQ3: How sensitive is RRTO to various robotic IoT environments (the bandwidth to the GPU devices and the computing power of the GPU devices)?
- RQ4: What are the limitations and potentials of RRTO?

A. End-to-End Performance

Our evaluation results for the edge scenario, as presented in Fig. 2, demonstrate that RRTO achieved performance comparable to MCOP for our robotic application. We further compared RRTO with baseline methods in terms of inference time and energy consumption per inference, as illustrated in Fig. 8.

In terms of inference time, RRTO achieved 80% reduction compared to local computation and 98% reduction compared to Cricket, as shown in Fig. 8a, leading to a higher frequency in Fig. 2a. Despite the additional data transfer required, the powerful GPU's shorter computation time allowed RRTO to perform inferences faster than local computation. The significant communication cost incurred by Cricket's transparent offloading mechanism dramatically slowed down its inference time, which will be discussed in more details in Sec. VI-B. RRTO successfully reduced this extremely heavy communication cost using its record/replay mechanism, achieving a similar inference time to MCOP with nearly the same communication cost.

In terms of energy consumption, RRTO saved 90% energy compared to local computation and 98% energy compared to Cricket to finish the inference of one frame, as shown in Fig. 8b. Although RRTO only saved 53% power consumption compared to local computation and 8% power compared to Cricket in Fig. 2b, the shorter inference time in Fig. 8a allowed RRTO to consume significantly less energy per inference.

Notice that, the average power consumption values in Fig. 2b are not equal to those in Table I. Our application cannot fully utilize the GPU of the Jetson Xavier NX, resulting in the average power of local computation being lower than during the inference stage. Additional CPU computing tasks, such as robot control, cause the average power of offloading methods to be higher than during the communication and standby stages. Furthermore, the extremely frequent RPC function calls of Cricket generated 8% more power consumption on the CPU, and the significant communication cost caused Cricket to spend excessive time in the communication stage, wasting 98% of energy for each inference.

B. Micro-Event Analysis

To obtain a deeper understanding of the performance improvement achieved by RRTO, we performed an analysis of the RPC function calls generated by Cricket during different stages of KAPAO inference to present the features of the traditional transparent offloading mechanism. The detailed breakdown of these calls is presented in Table II.

By comparing the different stages of function calls in Table II, we can see that KAPAO needs to go through an initialization stage of inference different from the inference loops. This is because the working process of KAPAO [3] follows the default detection model in Yolo v5 [63]: the inference pipeline is first initialized by generating a mesh grid of a certain size that fits the input image size, which serves as the storage of intermediates; then in the following loop iterations through the inference pipeline the mesh grid is reused and the operator call sequence is fixed. RRTO recorded all involved operators during the first few inferences, not just the initial process, and ignored the different operator

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 II
COMPOSITION OF RPC FUNCTION CALLS DURING DIFFERENT STAGES OF KAPAO INFERENCE.

sequences from the initializing inference when the correct operator sequence was found.

For the following loop inference in Table II, we observed that a significant portion, specifically 90.62%, of the RPC function calls were attributed to cudaGetDevice and cudaGetLastError. These calls were generated by PyTorch [15] due to our application's reliance on this framework and served the purpose of determining the data's location, which are essential for executing computations across multiple GPUs and parallel tasks. Even when restricting PyTorch to utilize only a single GPU sequentially and employing Caching (described in Sec. II-C) to avoid invoking these RPC functions, transparent offloading systems can only achieve inference times similar to local computation according to our experiments. However, they still cannot match the inference times achieved by MCOP or RRTO. This is evident from the fact that cudaLaunchKernel still accounts for 8.85% of the total RPC function calls, as it informs the server about subsequent computing tasks, such as additional convolution or maxpool operations. While existing RPC optimization methods rely on waiting for RPCs of cudaLaunchKernel from the client to instruct the server to fullfill the subsequent computing tasks, RRTO records these cudaLaunchKernel function calls and directly executes the subsequent computing tasks on the server without the need for 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, accounting for 0.19% of the total RPC calls, are primarily used for data transmission between the CPU and GPU, which are mainly employed for the input and output of the ML model and cannot be replayed by RRTO.

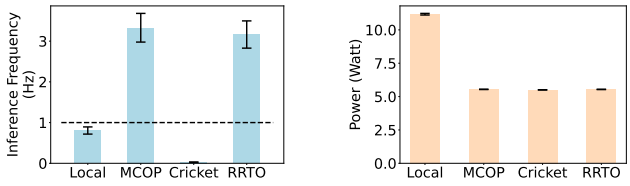
	MCOP	Cricket	RRTO
RPCs for each inference	N\A	5895	11
Average GPU utilization on the edge device	29.0%	1.1%	27.5%

TABLE III
COMPARISON BETWEEN RRTO AND THE BASELINES ABOUT NUMBERS OF RPC CALLS AND AVERAGE GPU UTILIZATION ON THE EDGE DEVICE

.

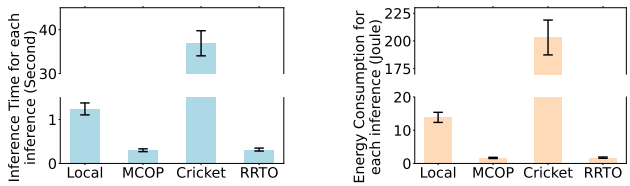
To provide an additional perspective on the performance gain achieved by RRTO, we compared it with the baselines about numbers of RPC calls and the resulting average GPU utilization on the edge device during the execution of the robotic application (measured using pynvml [64], as shown in Table III. MCOP synchronized input and output data of the ML model between CPU and GPU via cudaMemcpyHtoD and cudaMemcpyDtoH at the application layer by modifying the source code instead of using RPC. Cricket incurred a higher communication cost, leading to lower GPU utilization on the edge device. Although RRTO still needed to handle cudaMemcpyDtoH and cudaMemcpyHtoD as Cricket does, resulting in 11 RPCs per inference, the benefits of RRTO in terms of performance improvement are evident.

C. Sensitivity Studies



(a) Average Inference Frequency with standard deviation.

(b) Average Power with standard deviation.



(c) Average Inference Time with standard deviation.

(d) Average Energy consumption with standard deviation.

Fig. 9. Comparison between RRTO and the baselines in the cloud scenario.

We conducted an empirical evaluation of RRTO's performance across various robotic IoT environments, the cloud scenario as illustrated in Fig. 9. While cloud servers are equipped more powerful GPUs, their limited bandwidth causes offloading methods to incur greater communication time, resulting in slower inference times. In the cloud scenario, Cricket experiences additional performance degradation due to the significant proportion of the communication cost caused

by its transparent offloading mechanism. However, with a transmission cost similar to that of MCOP, RRTO still manages to achieve nearly same high performance as MCOP in the cloud scenario.

D. Lessons Learned

Changing Network Bandwidth. During the implementation and evaluation of RRTO, we discovered that the performance of computation offloading heavily rely on the network bandwidth of robotic IoT networks. Unlike GPU clusters equipped with fast interconnects such as InfiniBand [59], robotic IoT often lack fast and stable network connections for computation offloading. Many factors may cause network fluctuations, including wireless signal interference for Wi-Fi (due to obstacles [65], robot movement [66], channel preemption by other wireless devices [67], etc.) and network congestion for Internet connections (caused by multiple users accessing simultaneously). The fine-grained offloading schedules for operators in RRTO, as described in Sec. IV-A, have the potential to handle such random and violent fluctuations in network bandwidth, and we leave it as future work.

Fixed Calculation Logic.
Since RRTO leverages the fact that operators involved in the inference of a DNN model are often invoked in a fixed order, its record/replay mechanism enables support for other computation tasks [68], [69], not just for static ML models only if they have fixed calculation logic. However, RRTO cannot support tasks with unfixed calculation logic, such as dynamic ML models with changing operator sequence, due to the inability to replay varying operator sequences. Nonetheless, tasks with complex logic and branching are better suited for CPU execution rather than GPU [70], and dynamic ML models are mainly deployed in data centers rather than on robots (see Sec. III-A).

Future Work.
We would like to apply and evaluate RRTO on a broader variety of real-world applications on various robots (e.g., unmanned aerial vehicles, legged robots) in the future. Also, RRTO, being the first transparent offloading system to achieve the same high performance as non-transparent offloading systems, opens up new possibilities for further improvements. It is of interest to explore how to implement existing approaches based on non-transparent offloading systems on the basis of RRTO, such as building an edge computing power network that offloads computation to other idle robots or edge devices [71] through fine-grained operator partition scheduling to ensure full utilization of GPU resources and deploying multiple inference scheduling approaches for simultaneous inference tasks [34]–[36]. These optimizations could potentially enhance RRTO's performance and capabilities in various real-world scenarios.

VII. Conclusion

In this paper, we introduce RRTO, a high-performance transparent offloading system designed for model inference on robotic IoT. RRTO tackles the inherent communication cost associated with traditional transparent offloading mechanisms by introducing a novel record/replay mechanism, achieving the same high performance as the SOTA non-transparent offloading method without modifying any source code. We envision that RRTO will significantly streamline the deployment of a diverse array of ML applications on mobile robots in real-world settings. By offering fast and energy-efficient inference capabilities, RRTO enables these robots to execute complex tasks with high efficiency and effectiveness.

Appendix

The revised parts of our manuscript are highlighted in blue. We have comprehensively addressed the reviewers' valuable comments by enhancing the overall writing quality, including but not limited to rectifying typographical errors, updating figures in Sec. VI with standard deviations, and expanding the theoretical background in SecII and Sec. III. Additionally, we have provided point-by-point responses to each reviewer's comments, with corresponding hyperlinks directing to the revised sections for easy reference.

Q1. (Chair qHQW) The presented application scenario is quite simplistic and the theoretical background is quite poor.

A1: Sec. II now includes more comprehensive background information on non-transparent offloading methods and related scenarios, as suggested by the reviewers. In Sec. III-A, we provide detailed observations about RRTO and the case study, supporting the fact that ML models in robotic applications often perform operations in a fixed order. We have also updated our evaluation based on the reviewers' comments, ensuring it fully represents the scenarios described in the case study. As for the other scenarios mentioned by the reviewers, we will address them individually in the following questions.

Q2. (Reviewer sZZS) This paper lacks scientific findings and groundings to support that ML models typically perform operations in a fixed order.

A2: Sec. III-A now presents detailed observations about RRTO and the case study, supporting the notion that ML models in robotic applications often perform operations in a fixed order.

Q3. (Reviewer sZZS) How the existing method and RRTO can perform with ML models with dynamic operation sequences or fixed order operation with a different initial sequence?

A3: For ML models with dynamic operation sequences, we addressed this case in the last paragraph of Sec. IV-A. Regarding ML models with fixed order operations but a different initial sequence, RRTO recorded all involved operators during the first few inferences, not just the initial process, and ignored the different operator sequences from the initializing inference once the correct operator sequence was found (see Sec. IV-A). Since KAPAO [3] also undergoes a different initialization inference, we supplemented our analysis with a more complete examination of the system calls at different stages of its inferences in Sec. VI-B, including its different initial sequence. This additional analysis proves RRTO's ability to support ML models with fixed order operations but a different initial sequence and helps readers better understand the entire workflow of RRTO.

Q4. (Reviewer 6MCn) What would happen if the inference model triggers a new sequence of inference operations that start and end with similar GPU calls following the result of a previous inference?

A4: RRTO can distinguish operators belonging to different inference tasks to provide each task with high-performance inference via its record/replay mechanism. Firstly, RRTO's code base, Cricket [14], can already distinguish RPC functions from different processes, as it is designed for multiple general applications to share remote GPUs. By distinguishing RPC functions from different processes, the calculation results in the GPU servers can be returned to the correct process. Secondly, for inference tasks within the same process, RRTO can distinguish operators belonging to different tasks via the special design within its record/replay mechanism (see Sec. IV-A). During the recording phase, RRTO's data dependency search is a robust tool that efficiently finds the correct sequence. This is because inference tasks based on the same model have the same operator sequence (see Sec. IV-C). During the replaying phase, the replayer on the robot, which checks whether the prediction of operator sequence fails (see Sec. IV-A), can also check for new inference tasks and start a new inference via RRTO's record/replay mechanism. However, performance downgrade caused by computation resource or network constraints due to multiple inference tasks should be handled by multiple inference scheduling methods first (see Sec. II-C), such as batching several inference tasks together, rather than directly letting RRTO handle multiple inference tasks as above. We leave such deployment onto RRTO as future work.

Q5. (Reviewer 6MCn) Could the optimizations performed in frameworks like MCOP be adapted to RRTO, as this would significantly improve the performance?

A5: Yes, the optimization performed in MCOP, which is layer partition (placing parts of models on robots and parts on GPU servers at the granularity of layers), has been adapted to RRTO by downgrading the granularity of its scheduling algorithm from layer to operator, as described in the fourth paragraph of Sec. II-B and Sec. V. We leave it as future work to implement more existing approaches based on non-transparent offloading systems on the basis of RRTO, such as multiple inference scheduling (see Sec. II-C and Sec. VI-D).

Q6. (Reviewer 6MCn) Would RRTO adapt to false predictions on the list of GPU operations to be performed?

A6: When the prediction fails, RRTO's record/replay mechanism cannot find the correct operator sequence for the current inference process, preventing it from correctly replaying the inference process. In such cases, RRTO degenerates into the traditional non-transparent method, executing each operator one by one via RPC to complete the current inference task. However, RRTO's predictions only fail when the operator sequence changes, which occurs in dynamic ML models. We addressed this case in the last paragraph of Sec. IV-A.

Q7. (Reviewer 6MCn) How about the scenario where robots using edge computing utilize multiple inference models simultaneously

A7: As described in Sec. II-C, multiple inference scheduling methods (e.g., [34]–[36]) are orthogonal to optimizing an individual offloading system for single inference. Instead, they focus on coordinating the overall offloading systems on a cluster of robots to optimize the overall inference latency and power consumption for multiple inference tasks. Multiple inference scheduling only puts forward requirements for the performance of offloading systems, where a higher-performance offloading system can provide them with a larger, more flexible scheduling space, and they are typically deployed on high performance offloading systems, which are only non-transparent offloading systems currently. If a transparent offloading system, such as RRTO, achieves the same high performance as these non-transparent offloading systems, these multiple inference scheduling methods can also be adopted to provide a high-performance, transparent offloading solution supporting advanced scheduling techniques. We leave the deployment of multiple inference scheduling onto RRTO as future work.

Q8. (Reviewer 6MCn) What is the benefit of such an offloading when the images from the robot can be transmitted at a similar or higher rate to perform inference on an edge device?

A8: Notice that transferring raw images to the GPU server and deploying entire ML models onto it are special cases for baselines and RRTO, where entire layers (or operators) are placed onto the GPU servers. Under varying network bandwidth constraints, different layer (or operator) partition scheduling strategies are implemented to achieve the fastest possible inference time (see Sec. II-B). In our experiments, the application KAPAO continuously performs inferences to maximize speed, aiming for the lowest possible end-to-end latency for real-time tracking, using both baselines and RRTO, achieving the highest inference frequency under real-world network conditions. If the achieved inference frequency is still insufficient for the application's needs, multiple inference scheduling strategies could be a solution, such as batching several inference tasks together (see Sec. II-C), which is beyond the scope of this paper, as we focus primarily on the performance of single inferences in this paper.

Q9. (Reviewer De3N) The case study of RRTO.

A9: we provide the case study of RRTO in Sec. III-A

Q10. (Reviewer De3N) Stressing the proposed framework to more complex robotic systems (like UAVs or legged robots) requiring more computationally demanding machine learning performances.

A10: RRTO provides a convenient and high-performance offloading deployment for computation-intensive models that need to be offloaded to GPU servers. However, setting up RRTO on complex robotic systems (like UAVs or legged robots) requires additional hardware support and extra coding efforts. Due to the limited time for rebuttal, we leave the evaluation of RRTO on various robotic systems as future work. Nevertheless, we believe RRTO's high performance will remain consistent across different robotic systems.

Q11. (Reviewer De3N) Proposed writing comments to help readers deeply understand your work

A11: We have revised the corresponding sections of the manuscript based on the reviewer's valuable suggestions to improve the clarity and quality of the writing.

REFERENCES

[1] K. J. Joseph, S. Khan, F. S. Khan, and V. N. Balasubramanian, "Towards open world object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 5830–5840.

[2] S. Liu, X. Li, H. Lu, and Y. He, "Multi-object tracking meets moving uav," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 8876–8885.

[3] W. McNally, K. Vats, A. Wong, and J. McPhee, "Rethinking keypoint representations: Modeling keypoints and poses as objects for multi-person human pose estimation," *arXiv preprint arXiv:2111.08557*, 2021.

[4] Q. Li, F. Gama, A. Ribeiro, and A. Prorok, "Graph neural networks for decentralized multi-robot path planning," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2020, pp. 11 785–11 792.

[5] B. Wang, Z. Liu, Q. Li, and A. Prorok, "Mobile robot path planning in dynamic environments through globally guided reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 5, no. 4, pp. 6932–6939, 2020.

[6] Y. Yang, L. Juntao, and P. Lingling, "Multi-robot path planning based on a deep reinforcement learning dqn algorithm," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 3, pp. 177–183, 2020.

[7] A.-Q. Cao and R. de Charette, "Monoscene: Monocular 3d semantic scene completion," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022, pp. 3991–4001.

[8] Y. Li, Z. Yu, C. Choy, C. Xiao, J. M. Alvarez, S. Fidler, C. Feng, and A. Anandkumar, "Voxformer: Sparse voxel transformer for camera-based 3d semantic scene completion," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 9087–9098.

[9] Z. Xia, Y. Liu, X. Li, X. Zhu, Y. Ma, Y. Li, Y. Hou, and Y. Qiao, "Scpnet: Semantic scene completion on point cloud," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 17 642–17 651.

[10] 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 Robotics and Automation Letters*, vol. 6, no. 2, pp. 2335–2342, 2021.

[11] T. Ohkawa, K. Yamashina, H. Kimura, K. Ootsu, and T. Yokota, "Fpga components for integrating fpgas into robot systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 101, no. 2, pp. 363–375, 2018.

[12] V. Honkote, D. Kurian, S. Muthukumar, D. Ghosh, S. Yada, K. Jain, B. Jackson, I. Klotchkov, M. R. Nimmagadda, S. Dattawadkar *et al.*, "2.4 a distributed autonomous and collaborative multi-robot system featuring a low-power robot soc in 22nm cmos for integrated battery-powered minibots," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 48–50.

[13] H. Wu, W. J. Knottenbelt, and K. Wolter, "An efficient application partitioning algorithm in mobile environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 7, pp. 1464–1480, 2019.

[14] N. Eiling, J. Baude, S. Lankes, and A. Monti, "Cricket: A virtualization layer for distributed execution of cuda applications with checkpoint/restart support," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 14, 2022.

[15] pytorch, "pytorch," https://www.pytorch.org, 2024.

[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, leading manufacturing systems transformation – Proceedings of the 55th CIRP Conference on Manufacturing Systems 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827122003146

[17] NVIDIA, "The world's smallest ai supercomputer," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[18] ——, "cuda runtime apis," https://docs.nvidia.com/cuda/cuda-runtime-api/index.html, 2024.

[19] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A gpgpu transparent virtualization component for high performance computing clouds," in *Euro-Par 2010-Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31-September 3, 2010, Proceedings, Part I 16*. Springer, 2010, pp. 379–391.

[20] Y.-S. Lin, C.-Y. Lin, C.-R. Lee, and Y.-C. Chung, "qcuda: Gpgpu virtualization for high bandwidth efficiency," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019, pp. 95–102.

[21] X. Chen, J. Zhang, B. Lin, Z. Chen, K. Wolter, and G. Min, "Energy-efficient offloading for dnn-based smart iot systems in cloud-edge environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 3, pp. 683–697, 2021.

[22] H. Liang, Q. Sang, C. Hu, D. Cheng, X. Zhou, D. Wang, W. Bao, and Y. Wang, "Dnn surgery: Accelerating dnn inference on the edge through layer partitioning," *IEEE transactions on Cloud Computing*, 2023.

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

[24] A. Défossez, Y. Adi, and G. Synnaeve, "Differentiable model compression via pseudo quantization noise," *arXiv preprint arXiv:2104.09987*, 2021.

[25] Gheorghe and M. Ivanovici, "Model-based weight quantization for convolutional neural network compression," in *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. IEEE, 2021, pp. 1–4.

[26] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "Vecq: Minimal loss dnn model compression with vectorized weight quantization," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 696–710, 2020.

[27] J. Gou, B. Yu, S. J. Maybank, and D. Tao, "Knowledge distillation: A survey," *International Journal of Computer Vision*, vol. 129, pp. 1789–1819, 2021.

[28] T. Lin, L. Kong, S. U. Stich, and M. Jaggi, "Ensemble distillation for robust model fusion in federated learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 2351–2363, 2020.

[29] L. Wang and K.-J. Yoon, "Knowledge distillation and student-teacher learning for visual intelligence: A review and new outlooks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 44, no. 6, pp. 3048–3068, 2021.

[30] S. Dickson, "libtirpc: Transport independent rpc library," https://git.linux-nfs.org/?p=steved/libtirpc.git, 2024, git repository.

[31] 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 *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 36–51.

[32] S. Eyerman and I. Hur, "Efficient asynchronous rpc calls for microservices: Deathstarbench study," *arXiv preprint arXiv:2209.13265*, 2022.

[33] 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 *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 93–105.

[34] M. Altamimi, A. Abdrabou, K. Naik, and A. Nayak, "Energy cost models of smartphones for task offloading to the cloud," *IEEE Transactions on Emerging Topics in Computing*, vol. 3, no. 3, pp. 384–398, 2015.

[35] K. Elgazzar, P. Martin, and H. S. Hassanein, "Cloud-assisted computation offloading to support mobile services," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 279–292, 2014.

[36] Z. Fang, T. Yu, O. J. Mengshoel, and R. K. Gupta, "Qos-aware scheduling of heterogeneous servers for inference in deep neural networks," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, 2017, pp. 2067–2070.

[37] R. Kruse, S. Mostaghim, C. Borgelt, C. Braune, and M. Steinbrecher, "Multi-layer perceptrons," in *Computational intelligence: a methodological introduction*. Springer, 2022, pp. 53–124.

[38] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: analysis, applications, and prospects," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 12, pp. 6999–7019, 2021.

[39] J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber, "A clockwork rnn," in *International conference on machine learning*. PMLR, 2014, pp. 1863–1871.

[40] Y. Wang, L. Wang, F. Yang, W. Di, and Q. Chang, "Advantages of direct input-to-output connections in neural networks: The elman network for stock index forecasting," *Information Sciences*, vol. 547, pp. 1066–1079, 2021.

[41] D. Meyer, "Introduction to autoencoders," 2015.

[42] D. P. Kingma, M. Welling *et al.*, "An introduction to variational autoencoders," *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019.

[43] Z. Pan, W. Yu, X. Yi, A. Khan, F. Yuan, and Y. Zheng, "Recent progress on generative adversarial networks (gans): A survey," *IEEE access*, vol. 7, pp. 36 322–36 333, 2019.

[44] F. Gao, Y. Yang, J. Wang, J. Sun, E. Yang, and H. Zhou, "A deep convolutional generative adversarial networks (dcgans)-based semi-supervised method for object recognition in synthetic aperture radar (sar) images," *Remote Sensing*, vol. 10, no. 6, p. 846, 2018.

[45] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[46] M. P. LaValley, "Logistic regression," *Circulation*, vol. 117, no. 18, pp. 2395–2399, 2008.

[47] I. Steinwart and A. Christmann, *Support vector machines*. Springer Science & Business Media, 2008.

[48] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[49] 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*, 2018.

[50] Y. Yu, X. Si, C. Hu, and J. Zhang, "A review of recurrent neural networks: Lstm cells and network architectures," *Neural computation*, vol. 31, no. 7, pp. 1235–1270, 2019.

[51] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE, 2017, pp. 1597–1600.

[52] S. Masoudnia and R. Ebrahimpour, "Mixture of experts: a literature survey," *Artificial Intelligence Review*, vol. 42, pp. 275–293, 2014.

[53] P. Ren, Y. Xiao, X. Chang, P.-Y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–34, 2021.

[54] L.-E. Montoya-Cavero, R. D. de León Torres, A. Gómez-Espinosa, and J. A. E. Cabello, "Vision systems for harvesting robots: Produce detection and localization," *Computers and electronics in agriculture*, vol. 192, p. 106562, 2022.

[55] S. Wan and S. Goudos, "Faster r-cnn for multi-class fruit detection using a robotic vision system," *Computer Networks*, vol. 168, p. 107036, 2020.

[56] S. M. J. Jalali, S. Ahmadian, A. Khosravi, S. Mirjalili, M. R. Mahmoudi, and S. Nahavandi, "Neuroevolution-based autonomous robot navigation: A comparative study," *Cognitive Systems Research*, vol. 62, pp. 35–43, 2020.

[57] Z. Xie, L. Jin, X. Luo, Z. Sun, and M. Liu, "Rnn for repetitive motion generation of redundant robot manipulators: An orthogonal projection-based scheme," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 615–628, 2020.

[58] F. Piltan, A. E. Prosvirin, M. Sohaib, B. Saldivar, and J.-M. Kim, "An svm-based neural adaptive variable structure observer for fault diagnosis and fault-tolerant control of a robot manipulator," *Applied Sciences*, vol. 10, no. 4, p. 1344, 2020.

[59] NVIDIA, "Infiniband networking solutions," https://www.nvidia.com/en-us/networking/products/infiniband/, 2024.

[60] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 94–110, 2019.

[61] J. M. Tarnawski, A. Phanishayee, N. Devanur, D. Mahajan, and F. Nina Paravecino, "Efficient algorithms for device placement of dnn graph operators," *Advances in Neural Information Processing Systems*, vol. 33, pp. 15 451–15 463, 2020.

[62] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[63] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, K. Michael, TaoXie, J. Fang, imyhxy, Lorna, 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. [Online]. Available: https://doi.org/10.5281/zenodo.7347926

[64] N. Corporation, "pynvml: Python Bindings for the NVIDIA Management Library." [Online]. Available: http://www.nvidia.com/

[65] B. Alam, G. Calò, G. Bellanca, J. Nanni, A. E. Kaplan, M. Barbiroli, F. Fuschini, P. Bassi, J. S. Dehkordi, V. Tralli *et al.*, "Numerical and experimental analysis of on-chip optical wireless links in presence of obstacles," *IEEE Photonics Journal*, vol. 13, no. 1, pp. 1–11, 2020.

[66] Y. Feng, E. Benassi, L. Zhang, X. Li, D. Wang, F. Zhou, and W. Liu, "Concealed wireless warning sensor based on triboelectrification and human-plant interactive induction," *Research*, 2021.

[67] K. Eshteiwi, G. Kaddoum, B. Selim, and F. Gagnon, "Impact of co-channel interference and vehicles as obstacles on full-duplex v2v cooperative wireless network," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7503–7517, 2020.

[68] N. Cautaerts, P. Crout, H. W. Ånes, E. Prestat, J. Jeong, G. Dehm, and C. H. Liebscher, "Free, flexible and fast: Orientation mapping using the multi-core and gpu-accelerated template matching capabilities in the python-based open source 4d-stem analysis toolbox pyxem," *Ultramicroscopy*, vol. 237, p. 113517, 2022.

[69] R. Chowdhury and D. Subramani, "Optimal path planning of autonomous marine vehicles in stochastic dynamic ocean flows using a gpu-accelerated algorithm," *IEEE Journal of Oceanic Engineering*, vol. 47, no. 4, pp. 864–879, 2022.

[70] V. Rosenfeld, S. Breß, and V. Markl, "Query processing on heterogeneous cpu/gpu systems," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–38, 2022.

[71] Y. Bao, Y. Peng, C. Wu, and Z. Li, "Online job scheduling in distributed machine learning clusters," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 495–503.

If you have an EPS/PDF photo (graphicx package needed), extra braces are needed around the contents of the optional argument to biography to prevent the LaTeX parser from getting confused when it sees the complicated `\includegraphics` command within an optional argument. (You can create your own custom macro containing the `\includegraphics` command to make things simpler here.)

If you include a photo:

Michael Shell Use `\begin{IEEEbiography}` and then for the 1st argument use `\includegraphics` to declare and link the author photo. Use the author name as the 3rd argument followed by the biography text.

If you will not include a photo:

John Doe Use `\begin{IEEEbiographynophoto}` and the author name as the argument followed by the biography text.