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

Author Names Omitted for Anonymous Review. Paper-ID [add your ID here]

***Abstract*—The abstract goes here.**

I. Introduction

The rapid advancement of machine learning (ML) methods has achieved remarkable success in various robotic tasks, such as intelligent decision-making[], environmental perception[], and human-robot interaction[]. Deploying these ML methods onto real-world robots typically necessitates additional hardware support due to the computationally-intensive nature of ML models (e.g., xx% computation of the whole robot task[]). However, directly integrating computing accelerators (e.g., GPU[], NPU[], FPGA[], SoC[]) onto real-world robots not only demands specialized hardware adaptation within the application's source code but also leads to increased energy consumption (e.g., xx% for []). Consequently, these ML methods 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).

Computation offloading methods, such as MCOP[2], 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 workload, network bandwidth, and the computing power of cloud servers. This approach successfully reduces xx%inference time and saves xx%energy consumption in our experiments. However, these methods are typically **untransparent**, meaning that they require modifications to the source code to enable such offloading methods. This requirement 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[1], provide a more convenient, but also worse performance approach to offload computation to the cloud servers. ML model inference consists of a series of operators (e.g., addition, convolution). Transparent offloading intercepts each operator's call to the corresponding system functions (e.g., *torch.add()*,*torch.convolution()*) and offloads all these calls to the cloud servers through Remote Procedure Call (RPC). In this way, transparent offloading avoids modifications to the source code by intercepting at the system layer but has to offload the RPC calls to the cloud servers one by one, bringing worse performance.

However, in robotic IoT networks, outstanding scheduling algorithms for untransparent offloading cannot be applied to transparent offloading methods due to the significant communication cost caused by the transparent offloading mechanism. ML models commonly have hundreds of operators (e.g., 451 for []), resulting in hundreds of RPC calls for a single inference. In robotic IoT networks, which typically rely on wireless communication for robots, each RPC call usually takes 2-5 milliseconds[]. Consequently, transparent offloading in robotic IoT networks leads to significant communication costs (e.g., xx% inference time for xxx) and energy wastage (e.g., xx% for xxx).

The key reason why existing transparent offloading methods suffer from communication cost is that they only call RPC when the operator has been used in the inference process. This is because they are designed for general applications to use remote GPUs, rather than specifically for ML model inference. Such communication costs cannot be avoided in general applications, as they do not know the future operators from the upper-layer applications and cannot call RPCs in advance. Fortunately, in the scenario of ML model inference, we find that the operators invoked by ML models are usually fixed in order and can be predicted in advance for the inference process.

Base 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. It automatically records the order of operators invoked by ML model and replays the execution of these fixed-order operators during model inference. In this way, RRTO call the RPCs of the correspondin operators during inference in advance, without waiting for the operator to be called during inference. Based on this approach, RRTO achieves nearly the same communication cost as untransparent offloading methods, allowing the outstanding scheduling algorithms in untransparent offloading methods to be used in transparent offloading. However, the design of RRTO is confronted with a major challenge: how to find the correct order of operators for inference at the system level without the help of hints from the upper-layer applications?

To address this challenge, RRTO formulates this problem into a sub-problem of finding the longest common substring and discovers the correct order of operators via a novel algorithm: *find_operators_per_inference()*. This algorithm first identifies the longest common operator sequence based on the history of the called operators and then verifies whether this operator sequence can constitute a complete model inference process according to the operators' data dependencies (i.e., the calculation result of the previous operator serves as the input for the next operator). Finally, RRTO replays the execution of the operators sequence found by the *find_operators_per_inference()* algorithm.

We implemented RRTO on the Cricket's codebase[1] and also incorporated the outstanding scheduling algorithms of MCOP[2] into RRTO. We evaluated RRTO on a real-world robot with a robotic application that tracks people in real time[]. We compared RRTO with locally computation, a state-of-the-art (SOTA) untransparent offloading method (MCOP[2]) and a SOTA transparent offloading method (Cricket[1]) when offloading computation to different GPU devices (namely edge devices with high bandwidth and cloud servers with limited bandwidth) Evaluation shows that:

- RRTO is fast. It reduces inference time by xx% compared to other baselines, similar to the xx% reduction achieved by the SOTA untransparent offloading method.
- RRTO is energy-efficient. It saves xx% in energy consumption compared to other baselines, similar to the xx% savings of the SOTA untransparent offloading method.
- 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) changes, RRTO's superior performance remains consistent.

Our main contribution is RRTO, the first high-performance transparent offloading system designed for model inference on robotic IoT. RRTO dramatically decrease the communication cost caused by the traditional transparent offloading mechanism via the novel record/replay mechanism, achieving the same high performance as the SOTA untransparent 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/xxxx/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. Transparent offload on Robots

When the robot performs GPU calculation locally, the application specifies the calculation to be performed on the GPU by defining the CUDA kernel function. The specific call process is shown in the figure of 1 :

A The robot application completes the whole calculation process of applying each service by calling different operators in turn.
B Each operator goes through the Unified Operator API to find the local CUDA runtime library based on the data type of the application running device.
C The local CUDA runtime API is loaded by default.
D The robot local CUDA library starts the corresponding CUDA core function on the robot GPU and returns the calculation result to the upper-layer application.

NVIDIA GPU computing power unloading tool This project rewrites a new dynamic link library, in which the function of the same name is defined, and the LD_PRELOAD environment variable is used to make the custom dynamic link library load preferentially, and the dynamic linker will parse the original library function into the library function written by itself, so as to achieve the intercept of the library function. After that, through the management of GPU memory and the modification of CUDA kernel function startup, the call information such as the computation-related data and the specific parameters of the related kernel function is sent to the remote server, so as to realize the transparent unloading of GPU computing power. Step C in the above process is mainly modified. As with the local calculation using the robot GPU, after going through steps A and B, that is, after finding the corresponding CUDA runtime API for each operator in the robot application, the NVIDIA GPU power offloading tool is loaded preferentially. This NVIDIA GPU Power offloading tool performs the following steps:

(1) By modifying the dynamic connection library, each operator through the CUDA runtime API preferentially invokes the computing power with the same name as the CUDA runtime API to unload the client, thus identifying and intercepting all CUDA core function calls.
(2) The computing power unloading client transfers the CUDA runtime API and required parameters to the GPU server over the network.
(3) The server starts the corresponding CUDA core function on the cloud GPU and completes the corresponding self-calculation.
(4) The power offload server passes the calculation result back to the power offload client and returns the calculation result to the upper layer application.

B. Model Compression

Quantification and model distillation

C. RPC optimization

III. Overview

The core problem facing this NVIDIA GPU power offloading system is how to solve the high overhead caused by wireless transmission in transparent offloading in real robot environment. As the figure below shows, in a real robot environment, a single application needs to call the CUDA kernel thousands to tens of thousands of times per minute. For example, one gesture recognition with KAPAO requires about 20 CUDA kernel calls, while smooth gesture recognition calculations require gesture recognition at a frequency of no less than 10hz, which means that KAPAO needs to call the CUDA kernel at least 12,000 times per minute. If you use the simple transparent offloading method, that is, one RPC transfer per CUDA kernel call. In the case of network fluctuations, according to the WiFi6 standard, a round trip delay (RTT) takes 2ms to calculate, and only the wireless network transmission process (that is, the RTT process) consumes 24s in 12,000 RPC transfers. Therefore, the use of simple transparent uninstall methods can introduce serious network transmission overhead, which in turn affects the uninstall computing experience.

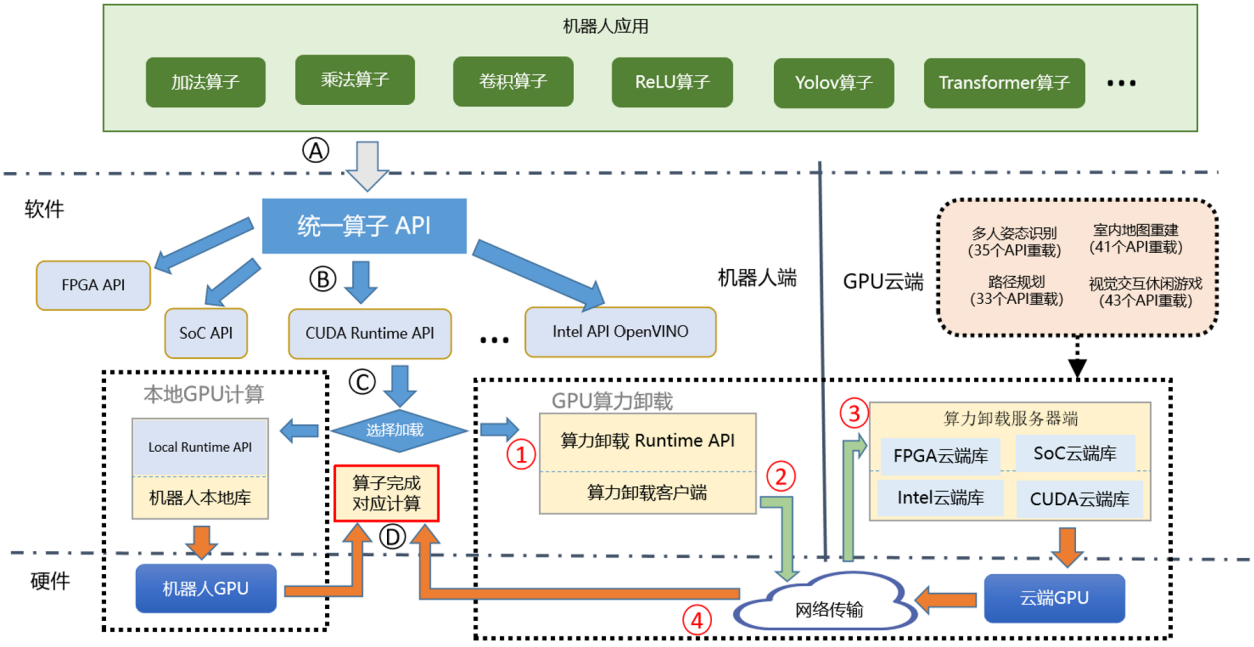

Fig. 1. Transparent Offloading System for Model Inference on Robotic IoT

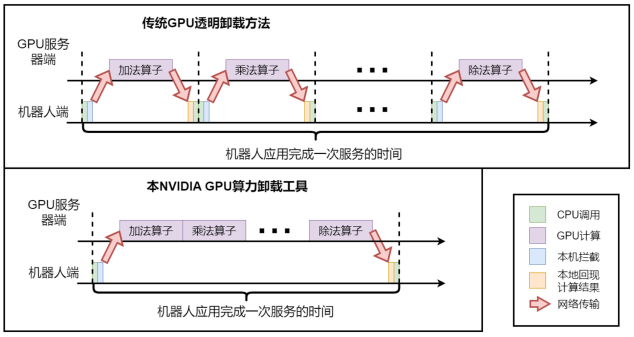

Fig. 2. Record/Replay Mechanism

For example, for vacuum cleaners, high latency can lead to sluggish path planning, making the robot slower to complete cleaning tasks or even run into obstacles.

Therefore, this NVIDIA GPU power offloading system proposes an automatic recording and replay mechanism to reduce the network transmission overhead during the transparent offloading process of GPU power. For the same household robot application, the order of CUDA cores invoked to process different data inputs is usually fixed. For example, multi-person pose recognition requires different images to be input into the same neural network model for recognition. The automatic recording replay mechanism automatically records the CUDA kernel call sequence of the model used by the home robot application and automatically repeats the use process of the model on the GPU service area. This mechanism does not need to wait for the robot to repeat the specific CUDA kernel call during RPC transmission, which greatly reduces the number of repeated RPC transfers and thus reduces the network transmission overhead.

IV. Design

This demo file is intended to serve as a "starter file" for the Robotics: Science and Systems conference papers produced under LaTeX using IEEEtran.cls version 1.7a and later.

V. Implementation

This demo file is intended to serve as a "starter file" for the Robotics: Science and Systems conference papers produced under LaTeX using IEEEtran.cls version 1.7a and later.

VI. Evaluation

This demo file is intended to serve as a "starter file" for the Robotics: Science and Systems conference papers produced under LaTeX using IEEEtran.cls version 1.7a and later.

VII. Conclusion

The conclusion goes here.

Acknowledgments

References

[1] Niklas Eiling, Jonas Baude, Stefan Lankes, and Antonello Monti. Cricket: A virtualization layer for distributed execution of cuda applications with checkpoint/restart support. *Concurrency and Computation: Practice and Experience*, 34(14), 2022. doi: 10.1002/cpe.6474.

[2] Huaming Wu, William J. Knottenbelt, and Katinka Wolter. An efficient application partitioning algorithm in mobile environments. *IEEE Transactions on Parallel and Distributed Systems*, 30(7):1464–1480, 2019. doi: 10.1109/TPDS.2019.2891695.