AccuMO: Accuracy-Centric Multitask Offloading in Edge-Assisted Mobile Augmented Reality

Abstract

Immersive applications such as Augmented Reality (AR) and Mixed Reality (MR) often need to perform multiple latency-critical tasks on every frame captured by the camera, which all require results to be available within the current frame interval. While such tasks are increasingly supported by Deep Neural Networks (DNNs) offloaded to edge servers due to their high accuracy but heavy computation, prior work has largely focused on offloading one task at a time. Compared to offloading a single task, where more frequent offloading directly translates into higher task accuracy, offloading of multiple tasks competes for shared edge server resources, and hence faces the additional challenge of balancing the offloading frequencies of different tasks to maximize the overall accuracy and hence app QoE.

In this paper, we formulate this accuracy-centric multitask offloading problem, and present a framework that dynamically schedules the offloading of multiple DNN tasks from a mobile device to an edge server while optimizing the overall accuracy across tasks. Our design employs two novel ideas: (1) task-specific lightweight models that predict offloading accuracy drop as a function of offloading frequency and frame content, and (2) a general two-level control feedback loop that concurrently balances offloading among tasks and adapts between offloading and using local algorithms for each task. Evaluation results show that our framework improves the overall accuracy significantly in jointly offloading two core tasks in AR —depth estimation and odometry —by on average 7.6%–14.3% over the best baselines under different accuracy weight ratios.

Keywords: Mobile Augmented Reality, Edge Computing, DNN Offloading, Multitask Offloading, Depth Estimation, Odometry.

1 Introduction

Immersive mobile applications, such as augmented reality (AR), often need to perform many challenging tasks to provide an enhanced user experience. For example, Pokémon Go is a representative AR application that allows Pokémon to interact with real scenes on a mobile phone. To achieve this interaction, the mobile device is required to perform several basic computer vision tasks for each frame at a high frame rate of once every 16.7 milliseconds. There are some representative tasks, such as depth estimation, odometry, and object detection. These tasks are critical to providing a realistic, interactive, and immersive user experience because they need to be processed in real time on every frame captured

by the camera to ensure that virtual objects can be accurately integrated into the real world. In order for virtual characters such as Pokémon in Pokémon Go to interact realistically with the real world, it is necessary to accurately track the camera pose and estimate the distance to real-world objects. The computational requirements of these tasks are very high, and the results must be available within the current frame time, otherwise they will not meet the strict frame rate requirements of AR applications (such as 60 FPS), thus affecting the user experience [1].

The paper further discusses the limitations of traditional AR frameworks (such as ARCore, etc.), which have the disadvantages of low resolution and limited accuracy of depth estimation. In contrast, although DNN-based solutions can provide higher accuracy, they require huge computational workloads, and running directly on mobile devices may result in hundreds of milliseconds or even seconds of latency, which is unacceptable for real-time AR applications. To resolve this contradiction, the paper proposes an edge computing-assisted solution, which is to offload the camera frames to the cloud or edge server for DNN inference to reduce the computational burden on mobile devices [1].

However, most existing research focuses on single-task offloading, while multi-task offloading requires a more complex balance between shared edge server resources to maximize overall accuracy and application quality (QoE). This is because multi-task offloading not only reduces the end-to-end offloading latency of each task, but also balances the offloading frequency of different tasks to ensure overall accuracy. The authors propose the AccuMO framework, an accuracy-centric solution to the multi-task offloading problem, which dynamically schedules the offloading of multiple DNN tasks from mobile devices to edge servers while optimizing the overall accuracy across tasks [1].

The design of the AccuMO framework adopts two innovations: first, lightweight accuracy models that support different tasks, which can predict the drop in offloading accuracy under different offloading frequencies and frame contents; second, a two-level control feedback loop that can simultaneously balance offloading between tasks and control the offloading of each task or use local algorithms. This design enables the AccuMO framework to effectively support multiple computationally intensive tasks in AR applications on resource-constrained mobile devices, improve the accuracy of the overall task, and thus enhance the user experience [1].

2 Related works

The related work mentioned in the paper mainly focuses on the problem of computational offloading in augmented reality (AR) applications, especially how to efficiently handle latency-sensitive DNN tasks on mobile devices and edge computing environments. These research works attempt to resolve the contradiction between the limited computing resources of mobile devices and the high computing requirements of AR applications, and improve performance by offloading computationally intensive tasks to edge servers [1].

In terms of single-task offloading, researchers have proposed a variety of strategies to optimize the latency and accuracy of task offloading. For example, some works focus on reducing transmission time through frame compression algorithms while maintaining or improving task accuracy; other works explore techniques such as key frame selection and region of interest encoding to reduce the amount of

data that needs to be transmitted. Other studies have proposed partial offloading methods that split the DNN model and distribute computing tasks between mobile devices and servers to reduce the size of input data. These methods have achieved some success in applications with high latency tolerance such as video analysis, but for scenarios such as AR applications that require real-time processing, these methods may not be effective enough [1].

In terms of multi-task offloading, although some studies have begun to explore how to process multiple DNN tasks simultaneously on edge servers, these works generally do not take into account the accuracy trade-offs between different tasks, nor do they provide solutions for dynamically adjusting offloading strategies to adapt to real-time requirements. These studies tend to focus on maximizing throughput or reducing service level objective (SLO) violations without optimizing for current frame accuracy (CFA) in AR applications [1].

In addition, local tracking techniques have also been widely studied as a way to optimize DNN offloading. These techniques adjust the results of the server DNN model by running fast algorithms on mobile devices to generate more accurate current frame task results. These local trackers are usually custom designed for specific tasks and can be completed quickly to meet the needs of real-time applications [1].

3 Method

The paper proposes a framework called AccuMO, which is adept at adapting to content and designed for multitask offloading. This framework is capable of enhancing the precision of multiple tasks simultaneously. It achieves this by intelligently adjusting the frequency at which each task is offloaded to the edge server and by making informed decisions on when to offload tasks versus when to utilize local computation methods [1].

3.1 Key Insight

This paper presents two key insights, pointing out that the relationship between accuracy and offloading frequency may vary with frame content [1].

(1)Offloading frequency-accuracy correlation is framedependent. The key insight highlights that the relationship between task offloading frequency and accuracy varies by frame. Shown as Figure 1, Analyzing the local tracking accuracy drop for two tasks in sample videos reveals that the drop rate changes over time and is influenced by frame content. For instance, frames with more cars show a greater accuracy drop in depth estimation due to the local tracker's inaccuracy with moving objects. The accuracy drops for the two tasks are almost uncorrelated, with the odometer's drop rate being affected by camera angular velocity. The first observation is that local tracking accuracy drop rates depend on content, with different tasks being impacted by different features [1].

(2)Local algorithms may have lower average accuracy than offloading solutions but can still outperform on certain video frames. Shown as Figure 2, Experiments show that while offloading generally outperforms a lightweight model (FastDepth) in terms of depth estimation accuracy, there are instances where the local model performs better on individual frames, especially when content changes

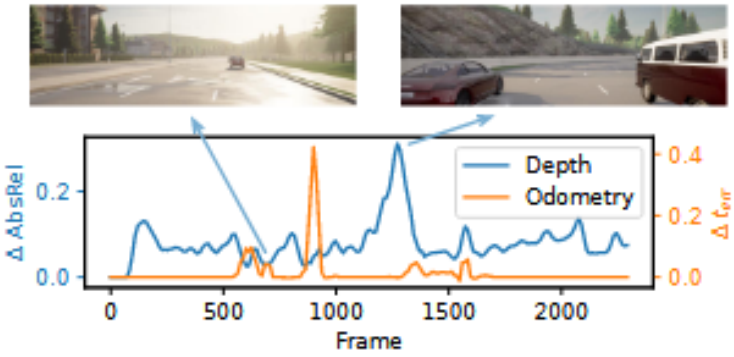

Figure 1. Accuracy drop timeline for depth estimation and odometry on a sample video when increasing the stride from 6 to 12. [1]

rapidly or when server resources are occupied. The key takeaway is that local algorithms, despite being less accurate on average, can sometimes surpass offloading solutions for specific frames, and there's little correlation between the two methods' accuracies due to their differing sensitivities to frame content [1].

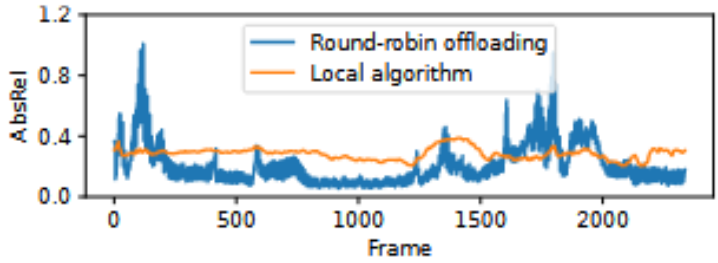

Figure 2. Depth estimation accuracy timeline of round-robin offloading vs. the local algorithm FastDepth on a sample video. [1]

3.2 Architecture Overview

As shown in the Figure3, this is the architecture of AccuMO. AccuMO adopts a modular design to support different numbers of tasks. Solid arrows represent data flow, while dashed arrows represent control flow. The paper designs two loops, a high-level loop and a low-level loop, to control two types of tasks respectively: The low-level control loop targets each task and adjusts between local tracking and using a local algorithm for each frame; The global control loop runs MPC to dynamically balance task offloading [1].

3.3 Low-level control loop

For each task of the AR application, such as depth estimation, odometer, each task has a task component. Each task adopts the mode of task offloading + local tracking. Using the second point of the paper Key insight, each task component may also contain a local algorithm. In order to dynamically switch between DNN models and local algorithms, an accuracy estimation model is needed to estimate

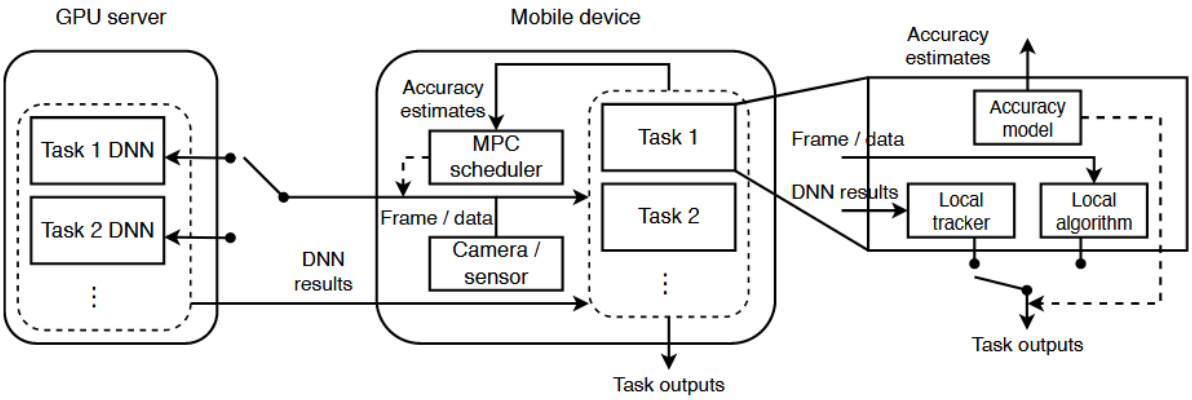

Figure 3. AccuMO architecture. Solid arrows represent data flows, while dashed arrows represent control flows. Inputs to accuracy models are task-specific and not shown. [1]

their accuracy based on the frame content. This model is used to estimate whether the server's DNN model is more accurate or the local algorithm is more accurate. The local tracker adjusts the stale results of the DNN model sent back by the server for the current frame. They need to run quickly to execute in each frame. The local algorithm generates task results directly from the camera or sensor data without relying on the server DNN results. It needs to execute quickly for each frame. The local algorithm can be a streamlined DNN model designed for mobile devices, but it can also be a traditional algorithm without any learning components [1].

Local tracker. Since the calculation and network transmission of the server DNN model takes a certain amount of time, there is a certain lag between the results returned by the server DNN model and the current frame. The role of the local tracker is to adjust the results returned from the server DNN model according to the content of the current frame, so as to achieve more accurate task result output. The local tracker runs on a mobile device and can quickly generate task results for each frame of data, but it may introduce a certain loss of accuracy [1].

Local Algorithm. The local algorithm generates task results directly from the camera or sensor data without relying on the server DNN results. It needs to execute quickly for each frame. The local algorithm can be a streamlined DNN model designed for mobile devices, but it can also be a traditional algorithm without any learning components [1].

Accuracy Model. The accuracy model is mainly to help make effective offloading decisions between local computing and edge computing. Because offloading to edge servers will cause network latency, and local processing may lead to poor accuracy, a model is needed to accurately estimate the accuracy drop in both offloading and local processing. When tasks are offloaded to edge servers for processing, the returned results need to be updated by the local tracker to adjust for the lagging results of the server DNN. In this process, the local tracker introduces accuracy degradation as it tries to adjust with old server inference results in a changing scenario. For the accuracy of offloaded tasks, the focus is on how the local tracker adjusts to the delayed results returned by the server and the resulting accuracy drop; while for the accuracy of local algorithms, the accuracy drop is estimated by comparing with the results of the server DNN [1].

3.4 Global control loop

For each frame, a series of computationally intensive subtasks need to be processed for that frame. However, the sequence of these subtasks requires a certain scheduling to maximize the overall accuracy [1].

Model Predictive Control (MPC). A control theory optimization algorithm, and the global control loop runs MPC to dynamically balance task offloading. Leveraging the first point of the paper's Key insight, the paper adopts a global scheduler that controls when to offload these tasks belonging to a frame based on the accuracy estimate of each task.

The reason for using MPC instead of machine learning algorithms to design a global scheduler is that machine learning algorithms require hard-coded number of tasks, task design, and optimization goals, and then must be trained offline, algorithms based on control theory are more flexible in dynamically adapting to different tasks and application requirements, such as how to merge task accuracy [1].

4 Implementation details

4.1 Comparing with the released source codes

Compared with the source code, the MPC scheduler is now mainly optimized. Source code In the schedule method, the logic of path cost calculation and task status simulation are mixed together, which makes the code lengthy and difficult to optimize. Each time the path is traversed, the Task[] tc object is recreated, increasing memory overhead. There is no clear pruning logic, and the calculation will continue even if the current path is no longer the optimal path.

The optimized code adds early pruning logic: if the cost of the current path exceeds the known minimum cost, the calculation is stopped immediately. In the calculatePathCost method, taskCopy is used to copy the task status to avoid frequent object creation. The path cost calculation logic is extracted to the calculatePathCost method to make the main logic more concise.

Original source code

```
public class MpcScheduler extends Scheduler {
    private final int lookahead = 30;
    private final double discount = 1;
    private final List<List<Integer>> trails;
    public MpcScheduler() {
        trails = generateTrails(Collections.emptyList());
    }
    private List<List<Integer>> generateTrails(List<Integer> prefix) {
        List<List<Integer>> ret = new ArrayList<>();
        for (int i = 0; i < tasks.length; i++) {
            List<Integer> t = new ArrayList<>(prefix);
            t.add(i);
            t.addAll(Collections.nCopies(tasks[i].latency - 1, -1));
            if (t.size() < lookahead) {
                ret.addAll(generateTrails(t));
            } else {
```

```
                ret.add(t.subList(0, lookahead));
            }
        }
        return ret;
    }

    public int schedule(int frame, double... accDropRates) {
        int minTask = -1;
        double minDrop = Double.MAX_VALUE;
        for (List<Integer> tr : trails) {
            Task[] tc = copy();
            double drop = 0;
            double d = 1;
            for (int i = 0; i < tr.size(); i++) {
                int f = frame + i;
                for (int j = 0; j < tc.length; j++) {
                    if (tc[j].pendingOffload >= 0 && f - tc[j].pendingOffload >= tc[j].latency
                        tc[j].finishOffload();
                    }
                    if (f - tc[j].latestOffload > tc[j].strideLimit) {
                        drop = Double.MAX_VALUE;
                        break;
                    }
                    if (tr.get(i) == j) {
                        tc[j].offload(f);
                    }
                    int s = Integer.MAX_VALUE;
                    if (tc[j].latestOffload >= 0) {
                        s = f - tc[j].latestOffload;
                    }
                    drop += d * Math.min(accDropRates[j] * s, tc[j].accDropLimit);
                }
                if (drop == Double.MAX_VALUE) {
                    break;
                }
                d *= discount;
            }
            if (drop < minDrop) {
                minDrop = drop;
                minTask = tr.get(0);
            }
        }
        if (minTask == -1) {
            if (tasks[0].latestOffload < tasks[1].latestOffload)
                minTask = 0;
            else
                minTask = 1;
        }
```

```
        return minTask;
    }
}
```

Modified code

```
public class MpcScheduler extends Scheduler {
    private final int lookahead = 30;
    private final double discount = 1;
    private final List<List<Integer>> trails;
    public MpcScheduler() {
        this.trails = generateTrails(Collections.emptyList());
    }

    private List<List<Integer>> generateTrails(List<Integer> prefix) {
        List<List<Integer>> ret = new ArrayList<>();
        for (int i = 0; i < tasks.length; i++) {
            List<Integer> t = new ArrayList<>(prefix);
            t.add(i);
            t.addAll(Collections.nCopies(tasks[i].latency - 1, -1));
            if (t.size() < lookahead) {
                ret.addAll(generateTrails(t));
            } else {
                ret.add(t.subList(0, lookahead));
            }
        }
        return ret;
    }

    public int schedule(int frame, double... accDropRates) {
        int minTask = -1;
        double minDrop = Double.MAX_VALUE;
        for (List<Integer> trail : trails) {
            double drop = calculatePathCost(frame, minDrop, trail, accDropRates);
            if (drop < minDrop) {
                minDrop = drop;
                minTask = trail.get(0);
            }
        }
        if (minTask == -1) {
            minTask = (tasks[0].latestOffload < tasks[1].latestOffload) ? 0 : 1;
        }
        return minTask;
    }

    private double calculatePathCost(int frame, double minDrop, List<Integer> trail, double[]
        Task[] taskCopy = copy();
        double drop = 0;
        double discountFactor = 1;
        for (int i = 0; i < trail.size(); i++) {
```

```
            int currentFrame = frame + i;
            for (int j = 0; j < taskCopy.length; j++) {
                if (taskCopy[j].pendingOffload >= 0 && currentFrame - taskCopy[j].pendingOfflo
                    taskCopy[j].finishOffload();
                }
                if (currentFrame - taskCopy[j].latestOffload > taskCopy[j].strideLimit) {
                    return Double.MAX_VALUE;
                }
                if (trail.get(i) == j) {
                    taskCopy[j].offload(currentFrame);
                }
                int stride = (taskCopy[j].latestOffload >= 0) ? currentFrame - taskCopy[j].lat
                drop += discountFactor * Math.min(accDropRates[j] * stride, taskCopy[j].accDro
            }
            discountFactor *= discount;
            if (drop > minDrop) {
                break;
            }
        }
        return drop;
    }
}
```

4.2 Experimental environment setup

Server setup: The system is based on the conda environment and the following components need to be installed:

Python 3.9,
TensorFlow-GPU 2.7.0,
PyTorch 1.11.0 with CUDA support,
TorchVision (compatible version with PyTorch),
CUDA Toolkit,
CUDA Toolkit Development,
Scikit-Image, OpenCV,
AV (for video processing),
Pandas,
TQDM,
Matplotlib

Laptop setup: Android debug bridge, evo, scikit-image, pandas, numpy, Pillow

SmartPhone setup: Android 12

4.3 Module introduction

Depth estimation:

Local tracker: Use warping algorithm, implemented in C++ and running on the phone's CPU.

Local algorithm: Use FastDepth, a depth estimation neural network designed for embedded devices, running on the phone's CPU, relying on the ncnn inference framework.

Optical flow estimation: Use FlowNet2S, a lightweight neural network model, to estimate optical flow in the accuracy model. FlowNet2S runs on the phone's GPU and is based on TensorFlow Lite. It used in the accuracy model of the depth estimation task to help evaluate the error introduced by warping.

AdaBins: A depth estimation DNN model based on PyTorch, runs on the server.

Odometry:

Visual odometry task: Implemented using a Kalman filter, using the insfilterErrorState object in the MATLAB Sensor Fusion and Tracking Toolbox, and converted to C code through MATLAB Coder to run on mobile devices.

Visual odometry model: Use DAVO, a visual odometry DNN model based on TensorFlow, runs on the server.

MPC Scheduler:

Control loop and MPC scheduler: Implemented in Java, responsible for managing the scheduling of task offloading.

4.4 Interface design

Accumo's server side is running, as shown in the following two figures 4 5.

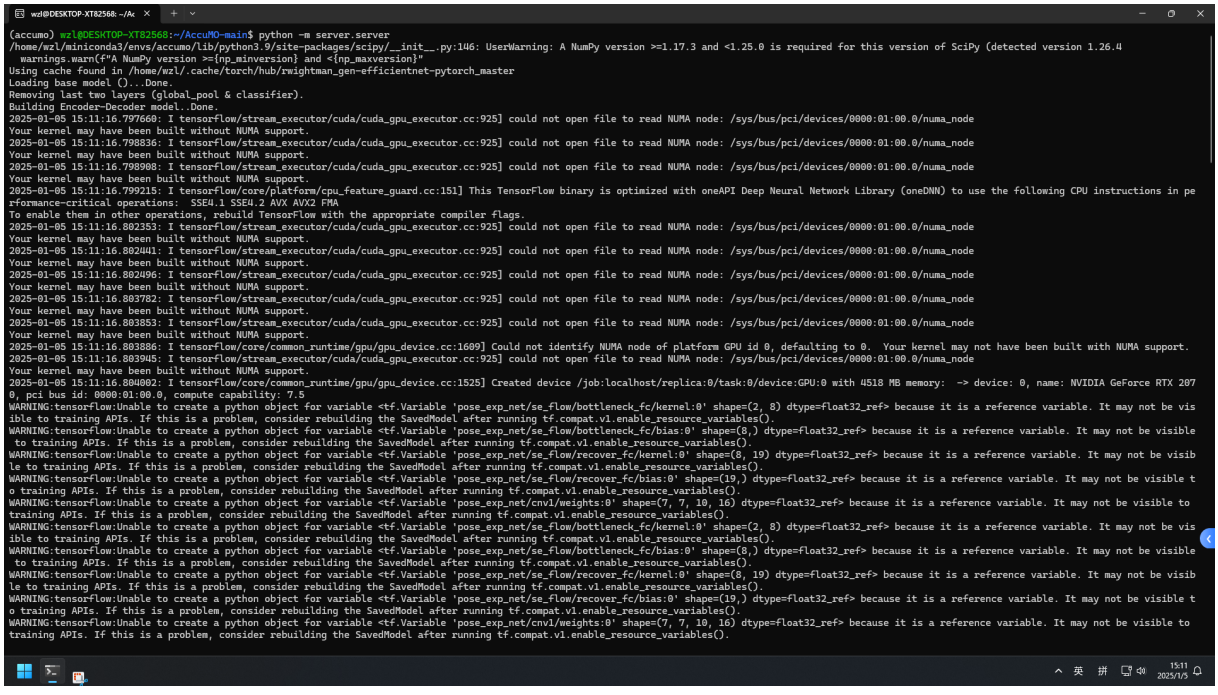

Figure 4. Server interface

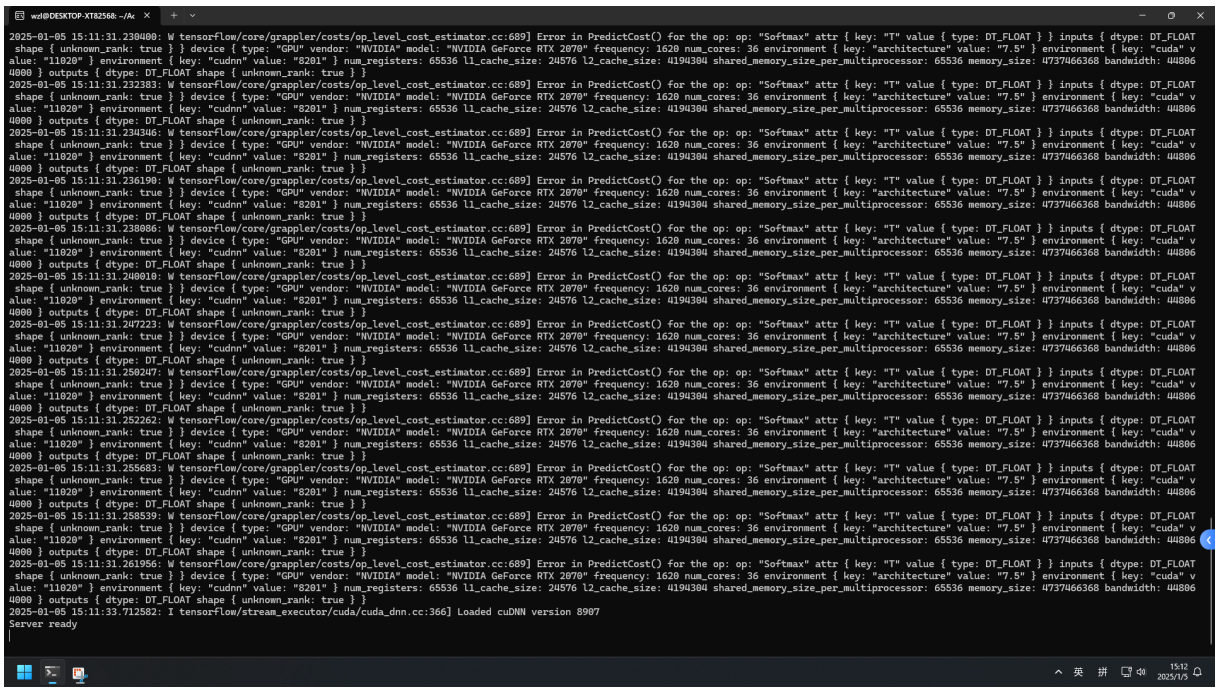

Figure 5. Server interface

4.5 Main contributions

The core contribution is the introduction of the advance pruning method, which significantly optimizes the performance of the scheduling algorithm. By checking in real time whether the cost of the current path exceeds the known minimum cost in the ‘calculatePathCost‘ method, once it is found that the current path cannot become the optimal solution, subsequent calculations will be terminated immediately to avoid unnecessary calculation overhead. This pruning strategy is particularly effective when there are a large number of paths, which can significantly reduce the amount of calculation and improve the execution efficiency of the code. At the same time, the introduction of the pruning method does not affect the functional integrity of the algorithm, and it can still accurately find the optimal task scheduling path.

5 Results and analysis

The dataset given in the paper is a series of pictures generated by an autonomous driving simulator, including the camera’s picture information and the IMU data at that time. The method for comparing the experimental results is to compare the result data of the original code with the result data of the modified code to determine the efficiency improvement brought about by the code modification. The comparison metrics are the same as those used in the original paper. For depth estimation, we use absolute relative error (AbsRel), and for visual odometry, we use the KITTI odometry metric (terr).

As can be seen from the table 1, for the task errors of depth estimation and visual odometer, the results before and after the modification are not much different, and the differences can be attributed to the error range. However, the overall running time has been slightly reduced. Compared with the original code, the overall running time has been reduced by 1.21% when processing the same data set.

This is because pruning checks in real time whether the cost of the current path exceeds the known minimum value during the calculation of the task sequence. If it determines that the path cannot produce the optimal solution, it will stop further calculations to avoid unnecessary computing costs. However, due to the small number of tasks in each frame of the task sequence, the effect of pruning is limited. If more visual tasks are performed in each frame, the effect of the optimized pruning algorithm may be better.

Table 1. The error of depth estimation and visual odometry between the original code and the modified code (lower is better)

Code	Depth(AbsRel)	Odom(KITTI)	Running time(S)
Original code	0.1745	0.0645	658
Modified code	0.1745	0.0646	650

6 Conclusion and future work

6.1 Conclusion

The key improvement is the implementation of an early pruning technique in the scheduling algorithm. This method checks in real-time within the 'calculatePathCost' process if the current path's cost exceeds the known minimum. If it determines that the path cannot yield an optimal solution, it halts further calculations to avoid unnecessary computational expense. This approach significantly reduces computation when handling numerous paths, enhancing efficiency without compromising the algorithm's ability to find the optimal scheduling path. To a certain extent, it speeds up the execution of the overall task.

6.2 Future work

The current scheduler mainly focuses on the frequency of task offloading, but does not consider the dynamic allocation of edge server resources in depth. In the future, more fine-grained resource management mechanisms can be introduced, such as dynamically adjusting the allocation ratio of GPU computing resources according to the real-time needs of tasks.

The current framework is mainly aimed at single-user AR applications. In the future, it can be expanded to multi-user scenarios to study how to optimize multi-task scheduling for multiple users while sharing edge resources.

The introduction of online learning mechanism enables the precision model to be dynamically adjusted according to real-time feedback to adapt to different scenarios and task requirements.

The current scheduler makes decisions based on model predictive control (MPC). In the future, more flexible adaptive algorithms can be introduced to dynamically adjust scheduling strategies based on network conditions, device status, and task requirements.

The current framework mainly focuses on two core tasks: depth estimation and visual odometry, and can be expanded to more AR tasks in the future.

References

[1] Z. Jonny Kong, Qiang Xu, Jiayi Meng, and Y. Charlie Hu. AccuMO: Accuracy-Centric Multitask Offloading in Edge-Assisted Mobile Augmented Reality. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–16. ACM, October 2023.