ENABLING HIGH-QUALITY MOBILE IMMERSIVE COMPUTING THROUGH EDGE SUPPORT

BY LUYANG LIU

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Marco Gruteser

and approved by

New Brunswick, New Jersey

October, 2019

ABSTRACT OF THE DISSERTATION

Enabling High-quality Mobile Immersive Computing through Edge Support

by Luyang Liu

Dissertation Director: Marco Gruteser

Emerging Mobile Immersive Computing applications, such as Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), are changing the way human beings interact with the world. Such systems promise to provide unprecedented immersive experiences in the fields of video gaming, education, and healthcare. However, several key processes, such as rendering and object detection, are highly computational intensive, which make them extremely hard to run on mobile devices. Offloading these bottleneck processes to the edge or cloud is also very challenging due to the stringent requirements on high quality and low latency.

In order to achieve high quality and low latency performance of mobile immersive computing applications on mobile thin clients, the system requires to finish the entire offloading pipeline within very short end-to-end latency. Offloading Vision tasks to the edge cloud typically involves several main processes: Sensing, Uplink Transmission, Processing, and Downlink Transmission. These four processes form a round trip from the mobile device to the edge cloud and back to mobile devices. Compared to traditional offloading approaches that execute these processes in a sequential way, our key contribution is to design new video streaming and processing pipelines that can significantly reduce the offloading latency and improve vision quality of VR and AR systems.

High-quality VR systems generate graphics data at a data rate much higher than those supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. The necessary image encoding, makes it challenging to maintain the stringent VR latency requirements. To address this issue, we introduces an end-to-end untethered VR system design and open platform that can meet virtual reality latency and quality requirements at 4K resolution over a wireless link.

Most existing Augmented Reality (AR)/Mixed Reality (MR) systems are able to understand the 3D geometry of the surroundings but lack the ability to detect and classify complex objects in the real world. Such capabilities can be enabled with deep Convolutional Neural Networks (CNN), but it remains difficult to execute large networks on mobile devices. Offloading object detection to the edge or cloud is also very challenging due to the stringent requirements on high detection accuracy and low end-to-end latency. The long latency of existing offloading techniques can significantly reduce the detection accuracy due to changes in the users view. To address the problem, we design a system that enables high accuracy object detection for commodity AR/MR systems running at 60fps.

Furthermore, we build EdgeSharing, an object sharing system leveraging large computational resources at the edge cloud. Beyond the capability of providing object detection service to nearby mobile clients, EdgeSharing holds a real-time 3D feature map of its coverage region on the edge cloud and uses it to provide accurate localization and object sharing service to the client devices passing through this region. By sharing a moving object's location between different camera-equipped devices, it effectively extends the vision of participants beyond their field of view.

We further propose several optimization techniques to increase the localization accuracy, reduce the bandwidth consumption and decrease the offloading latency of the system. The result shows that EdgeSharing is able to achieve high quality localization and object sharing accuracy with a low bandwidth and latency requirement.

Acknowledgements

I would like to express my sincere gratitude towards those who provide valuable support, guidance and help for me. It is with all their love and support that leads me to reach this successful destination.

First, I would like to express my gratefulness to my advisor, Marco Gruteser. Marco taught me key skills needed for an researcher: how to identify and formulate a research problem, tackle a research problem from a unique angle, think critically about a potential solution, develop a big research vision, etc. He always generously shares his experiences and suggestions to guide me through many crossroads during my PhD journey. This work cannot be finished without his insightful comments.

I would also like to thank Prof. Yingying Chen and Prof. Dipankar Raychaudhuri for serving on my dissertation committee. Their insightful suggestions and valuable comments have greatly improve the quality and completeness of my work.

I am also honored to have the opportunity to be mentored by and work with Dr. Yunxin Liu and Dr. Jiansong Zhang from Microsoft Research on the low latency VR project. It is a reaally valuable internship experience that shapes my direction of this dissertation. I really appreciate their time and advice every week to give me insightful comments from different angles regarding my progress. Without their help, my work wont be so complete.

Special thanks to Hongyu Li and Cagdas Karatas, who has worked with me for more than five years and was always willing to give me hand when I needed help. I am very lucky to be surrounded by a brilliant group of peers and friend at

WINLAB Viet Nguyen, Ali Rostami, Shubham Jain, Gorkem Kar, Jing Zhong, Wuyang Zhang, and Xiaoran Fan.

I would like to give my thank to my family and friends for their endless love. My parents are always the one to share happiness and sorrow. They stand with me no matter what happens. Their encouragement let me grow to be strong.

Lastly, thank my girlfriend Hairong for all the love and accompany. I can share all my feelings with you whenever I met difficulties or frustrations. You are always the special one in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

1.1 Background and Motivation

Emerging Mobile Immersive Computing applications, such as Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), are changing the way human beings interact with the world. Such systems promise to provide unprecedented immersive experiences in the fields of video gaming, education, and healthcare. Reports forecast that 99 million Virtual Reality (VR) and Augmented Reality (AR) devices will be shipped in 2021 [1], and that the market will reach 108 billion dollars [2] by then.

To achieve an immersive experience, AR/VR headsets are required to have high computational capability. Several key processes within the entire processing pipeline are highly computational intensive, which make it extremely challenging to run such systems on commercial mobile thin clients. In particular, high-quality AR/VR systems require huge amount of computation resources to render life-like virtual scenes or overlays that are indistinguishable from physical reality, and AR or MR systems further require to have a comprehensive understanding of the surroundings by running high computational vision tasks.

Existing AR/VR systems can be divided into two categories: Light weight and High-quality systems, as shown in Figure 4.1. Light weight AR/VR systems (e.g. Google Daydream and Magic Leap One) run simple AR/VR applications locally on power efficient and light-weight mobile SoCs. These systems provide high mobility to users but the quality is limited by the capability of the headset.

On the other hand, high-quality AR/VR systems (e.g. HTC Vive and HoloLens) leverage high performance graphic processing units to achieve immersive graphic rendering and accurate vision processing. However, such systems typically provide very poor mobility to users. They may either connect to a powerful desktop PC or equip with heavy computation resources on board.

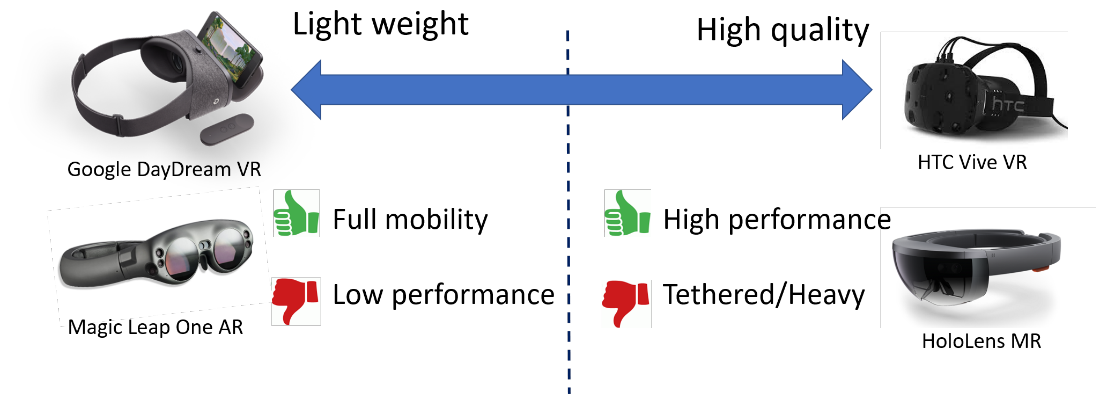


Figure 1.1: Light weight vs High quality AR/VR devices.

High-quality and light weight AR/VR is highly desirable but extremely challenging. While offloading these bottleneck processes to the edge or cloud through wireless network is a feasible solution, it is still very challenging due to the stringent requirements on high quality and low latency for such applications. Offloading vision tasks to the edge cloud requires transmitting huge amount vision data between the server and client through the wireless link. Figure 1.2 illustrates the network bandwidth consumption of VR, AR, and MR. In the case of edge VR (or untethered VR) system, the VR headset sends its orientation data to the edge cloud, and the edge cloud uses this information to render high quality VR frames leveraging its powerful graphic processors. Then, the edge cloud transmits the render frame back to the VR headset to display, which consumes large network resources on the downlink. Compare to VR system, AR system consumes more network resources on the network uplink. AR system is requires to have a more comprehensive understanding our the surroundings. Therefore, it needs to transmit the video frame captured by the camera up to the edge cloud for multiple detection tasks, while the edge cloud only transmits detection results back to the headset. In addition, as AR system's requirement of graphics approaches the complexity of VR, the Mixed Reality system further requires the edge cloud to send high quality visual elements back to the headset as well, which generate large traffic on both uplink and downlink of the network. Such large date volume generates long latency

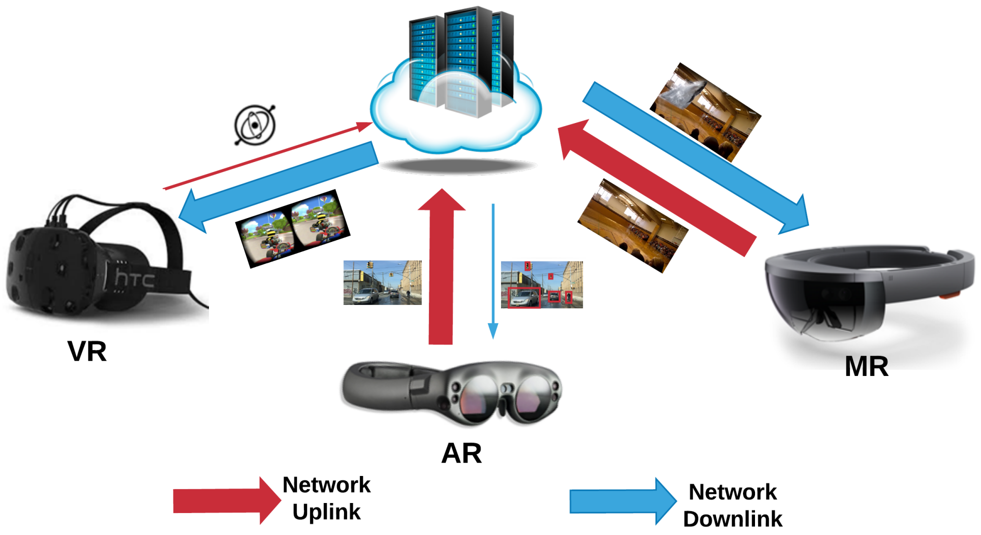


Figure 1.2: Network bandwidth consumption of VR, AR, and MR.

However, it is very challenging to support these immersive computing applications on most today's wireless networks, e.g. WIFI, LTE, and even Millimeter waves. First, Designing a high-quality untethered VR system is extremely challenging due to the stringent requirements on data throughput and end-to-end latency. Assuming we use three bytes to encode the RGB data of each pixel, for HTC Vive and Oculus Rift with a frame rate of 90Hz and a resolution of 2160x1200, the raw data rate is 5.6Gbps, much higher than the data rate (e.g., less than 2Gbps) supported by existing wireless-communication products such as

Wi-Fi and 60GHz wireless communication. For future VR targeting at a resolution of 4K UHD or even 8K UHD, the required data rate would be as high as 17.9Gbps and 71.7Gbps, respectively. To address the challenge of high data throughput, data compression is necessary. However, high-quality VR also requires a very tight total end-to-end (i.e., motion-to-photon) latency of 20-25ms [3] to reduce motion sickness. That is, once the HMD moves, the system must be able to display a new frame generated from the new pose of the HMD within 20-25ms. As compressing and decompressing frames introduce extra latency, it is even more challenging to meet the end-to-end latency requirement.

Second, For AR/MR system, offloading object detection to the edge or cloud is also very challenging due to the stringent requirements on high detection accuracy and low end-to-end latency. High quality AR devices require the system to not only successfully classify the object, but also localize the object with high accuracy. Even detection latencies of less than 100ms can therefore significantly reduce the detection accuracy due to changes in the user's view—the frame locations where the object was originally detected may no longer match the current location of the object. In addition, as mixed reality graphics approach the complexity of VR, one can also expect them to require less than 20ms motion-to-photon latency, which has been found to be necessary to avoid causing user motion sickness in VR applications [3]. Furthermore, compared to traditional AR that only renders simple annotations, mixed reality requires rendering virtual elements in much higher quality, which leaves less latency budget for the object detection task.

In addition, existing vision based object detection methods are only feasible for objects within the device's field of view, which largely limits the user's awareness of the surrounding environment. For example, many accidents occur at intersections involve visual occlusions of cars. Accurately sharing objects between moving clients is extremely challenging due to the high accuracy and low latency requirement for localizing both the client position and positions of its detected objects. Compared to GPS and other inertial sensing methods that are widely recognized to be less accurate in dense city scenarios, visual odometry solutions (e.g. SLAM) are more feasible in such situations where rich visual features exist. These solutions typically determine the position and orientation of a client device by analyzing the associated vision inputs (e.g. camera image, Lidar depth map, etc.) with a map constructed by 3D features. However, these solutions require large amounts of computation and storage resources on the end devices to store the large feature map and run computational intensive tasks on captured frames, which are less common to appear on commercial vehicles or smart devices. In addition, in order to continuously benefit from the evolving localization and detection algorithms, it is more feasible to run those intensive algorithms on the cloud, which can easily expand to large amounts of computation resource. However, offloading vision tasks to the cloud can incur long transmission delays, which make the feedback less useful to the mobile nodes.

To address these challenge, we build our solutions based on the following hypothesis: Innovative streaming and processing algorithms are able to enable high quality and low latency mobile immersive computing applications, such as VR/AR/MR and other vision analytics, on mobile thin clients with the support of commercial edge computation platforms and wireless connections.

1.2 Thesis Contribution

My thesis tackles these challenges and seeks to enable high quality and low latency mobile immersive computing applications on mobile thin clients with the support of commercial edge computation platforms and existing wireless connections. Offloading Vision tasks to the edge cloud typically involves several main

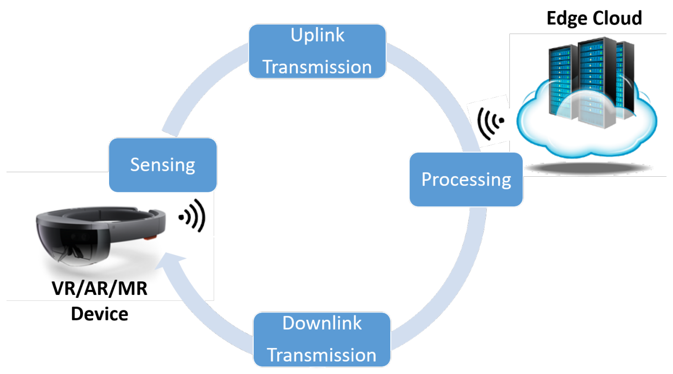


Figure 1.3: Offloading pipeline of VR/AR/MR.

processes: Sensing, Uplink Transmission, Processing, and Downlink Transmission. As shown in Figure 1.3, these four processes form a round trip from the mobile device to the edge cloud and back to mobile devices. Compared to traditional offloading approaches that execute these processes in a sequential way, our key contribution is to design new streaming and processing algorithm that can significantly improve the quality while reduce the offloading latency of VR and AR system.

1.2.1 Contribution Summary

Overall, the key systems and contributions of this thesis are:

- An end-to-end untethered VR system design that can meet virtual reality latency and quality requirements at 4K resolution over a wireless link.
- An AR system that enables high accuracy object detection for commodity AR/MR system running at 60fps.
- An edge assisted object sharing platform that provides device localization and object sharing services to travelers leveraging computational resources at the edge cloud.

1.2.2 Designing a High-quality Untethered VR System with Low Latency Remote Rendering

We first introduces an end-to-end untethered VR system design that can meet virtual reality latency and quality requirements at 4K resolution over a wireless link. This system employs a Parallel Rendering and Streaming mechanism to reduce the add-on streaming latency, by pipelining the rendering, encoding, transmission and decoding procedures. In addition, we introduce a Remote VSync Driven Rendering technique to minimize display latency. We implement an end-to-end remote rendering platform on commodity hardware over a 60Ghz wireless network, and show the system can support current 2160x1200 VR resolution at 90Hz with less than 16ms end-to-end latency, and 4K resolution with 20ms latency, while keeping a visually lossless image quality to the user. Furthermore, our system outperforms previous untethered VR system [4, 3] in all four aspects: end-to-end latency, frame rate, visual quality, and resolutions.

1.2.3 Edge Assisted Real-time Object Detection for Mobile Augmented Reality

We design an AR system that enables high accuracy object detection for commodity AR/MR system running at 60fps. To achieve this, we propose a system that significantly reduces the offloading detection latency and hides the remaining latency with an on-device fast object tracking method. To reduce offloading latency, it employs a *Dynamic RoI Encoding* technique and a *Parallel Streaming and Inference* technique. The *Dynamic RoI Encoding* technique adjusts the encoding quality on each frame to reduce the transmission latency based on the Regions of Interest (RoIs) detected in the last offloaded frame. It provides higher quality encodings in areas where objects are likely to be detected and uses stronger compression in other areas to save bandwidth and thereby reduce latency. The

Parallel Streaming and Inference method pipelines the streaming and inference processes to further reduce the offloading latency. On the AR device, the system decouples the rendering pipeline from the offloading pipeline instead of waiting for the detection result from the edge cloud for every frame. To allow this, it uses a fast and lightweight object tracking method based on the motion vector extracted from the encoded video frames and the cached object detection results from prior frames processed in the edge cloud to adjust the bounding boxes or key points on the current frame in the presence of motion. Taking advantage of the low offloading latency, we find this method can provide accurate object detection results and leave enough time and computation resources for the AR device to render high-quality virtual overlays. Besides, we also introduce an *Adaptive Offloading* technique to reduce the bandwidth and power consumption of our system by deciding whether to offload each frame to the edge cloud to process based on the changes of this frame compare to the previous offloaded frame. The result shows that the system can improve the detection accuracy by 20.2%-34.8% for the object detection and human keypoint detection tasks, and only requires 2.24ms latency for object tracking on the AR device. Thus, the system leaves more time and computational resources to render virtual elements for the next frame and enables higher quality AR/MR experiences.

1.2.4 Designing an Augmented Intersection

We further build EdgeSharing, a first collaborative localization and object sharing system leveraging the resources of an edge cloud platform and the visual inputs from participating mobile clients (e.g., vehicles and pedestrians). Beyond the capability of providing object detection service to nearby mobile clients, EdgeSharing holds a real-time 3D feature map of its coverage region on the edge cloud and uses it to provide accurate localization and object sharing service to the client devices passing through this region. In particular, EdgeSharing holds a

3D feature map of its coverage region constructed from images and depth readings from a dedicated data collection vehicle or crowdsourced from participating clients. This 3D feature map is then used to provide accurate localization services to the client devices passing through this region. Besides, EdgeSharing also leverages the computation power on the edge cloud to detect object locations on the images offloaded by participating clients. These locations are stored in a sharing database and can be shared with other clients in the same region. With EdgeSharing installed on the edge cloud, nearby vehicles are able to learn extra object (e.g., traffic participant) locations from the edge cloud, which are outside the vehicles field of view. This improves their situational awareness and safety. To realize this, we propose several optimization techniques. In particular, we propose a Context-Aware Feature Selection method to filter out potential moving objects in the offloaded images to increase the localization accuracy. We also introduce a Collaborative Local Tracking mechanism to significantly reduce the bandwidth consumption of frame transmission by only offload selected keyframes to the edge cloud, while using a lightweight local tracking method to keep track of the location of the client and its detected objects on the end device. In addition, we design a parallel streaming and processing method to enable parallel video streaming and cloud processing, which largely reduces the end-to-end latency of EdgeSharing. The result shows that the system is able to achieve a mean vehicle localization error of 0.2813-1.2717 meters, an object sharing accuracy of 82.3%-91.44%, and a 54.68% object awareness increment in urban streets and intersections. In addition, the proposed optimization techniques are able to reduce 70.12% of bandwidth consumption and reduce 40.09% of the end-to-end latency.

Chapter 2
Background

This thesis discusses how innovative streaming and processing algorithms are able to enable high quality and low latency mobile immersive computing applications. To better understand the background and challenges, we present background material of (1) Mobile immersive computing, (2) Edge computing, and (3) Offloading pipeline. More detailed related work sections are also provided in the remaining chapters.

2.1 Mobile Immersive Computing

Emerging Mobile Immersive Computing applications, such as Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), promise to provide unprecedented immersive experiences in the fields of video gaming, education, and healthcare. Designing mobile VR/AR/MR system has also attracted strong interest from both industry and academia.

Figure 2.1 shows differences between VR, AR and MR. Virtual Reality completely replaces reality with a virtual world, while seeks to let the user believe the virtual world is a real one. Augmented Reality enhances reality by layering information or virtual aspects over a direct view of actual reality. And Mixed Reality further combines rendered interactive objects with the real environment.

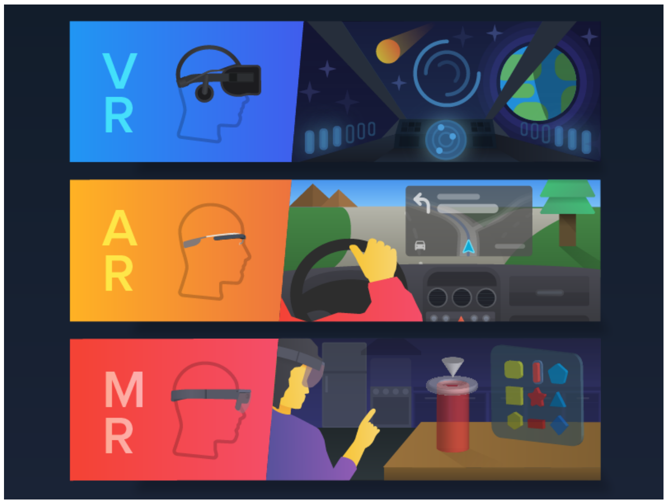


Figure 2.1: Comparisons between VR, AR and MR

2.1.1 Virtual Reality

Existing VR systems can be divided into two categories: High-quality VR and Light-weight VR systems. Due to the requirements of high quality and low latency, most high-quality VR systems, such as HTC Vive [5] and Oculus Rift [6], leverage a powerful desktop PC to render rich graphics contents at high frame rates and visual quality. However, most of these solutions are *tethered*: they must be connected to a PC via a USB cable for sending sensor data from the Head Mounted Display (HMD) to the PC and an HDMI cable for sending graphics contents from the PC back to the HMD. These cables not only limit the user's mobility but also impose hazards such as a user tripping or wrapping the cable around the neck. Standalone, portable VR systems such as Samsung Gear VR [7] and Google Daydream [8] run VR apps and render graphics contents locally on the headset (or a smartphone slide in the headset). Those VR systems allow untethered use but the rendering quality is limited by the capability of the headset or smartphone. There has been extensive demand for such untethered high-quality VR systems.

2.1.2 Augmented / Mixed Reality

Augmented Reality, and in particular Mixed Reality systems enhance the real world by rendering virtual overlays on the user's field of view based on their understanding of the surroundings through the camera. Existing mobile AR solutions such as ARKit and ARCore enable surface detection and object pinning on smartphones, while more sophisticated AR headsets such as Microsoft HoloLens [9] and the announced Magic Leap One [10] are able to understand the 3D geometry of the surroundings and render virtual overlays at 60fps. These AR headsets further promise to support an unprecedented immersive experience called Mix Reality. Compared to tradition AR system, MR requires the system to have a comprehensive understanding of different objects and instances in the real world, as well as more computation resources for rendering high quality elements.

2.2 Edge Computing

While it remains challenge to achieve high quality VR/AR/MR experience on existing mobile devices, edge computing provides a feasible way to overcome the challenge. Compared to the traditional cloud computing that are far away from the mobile devices, edge cloud are computation resources that lies on the edge of the network. For example, a desktop PC can be the edge cloud at your home, and a mini-datacenter connected to the base station can be the edge cloud of the surrounding area. Most edge clouds lie just one hop away from the data producer devices, so that it allow end devices to offload their high computational tasks to it with very low latency. Edge computing has the potential to address the concerns of response time requirement, battery life constraint, bandwidth cost saving, as well as data safety and privacy.

There has been extensive works on enabling computational intensive application with edge cloud offloading. MAUI [11] enables fine-grained energy-aware offload of mobile code to the edge infrastructures. Kahawai [12] improves graphics quality of video games on mobile devices through edge offloading. Real-time video analytics systems [13] can also benefit from the massive computation resources on the edge cloud.

Recently, edge clouds also provide the potential to enable high quality mobile immersive computing applications, such as VR, AR and MR. Several works [3, 4] enables high quality VR system, while another group of work [14, 15, 16] enables AR experience on mobile devices. However, most of previous works only aims to optimize the latency on single process, but fail to consider the systematic way to improve the performance of the entire offloading pipeline. As a result, most of these system cannot achieve the latency and quality requirement for high-quality VR/AR/MR system.

2.3 Offloading Pipeline

Offloading Vision tasks to the edge cloud involves several main processes: Sensing, Uplink Transmission, Processing, and Downlink Transmission. In the case of the Virtual Reality system, the mobile device senses its position and orientation information and transmit it to the edge cloud through the wireless network link. The edge cloud processes the data and renders the VR frames accordingly. Then the frame is transmitted back to the mobile devices and display on the screen. For Augmented and Mixed Reality system, the mobile device senses the surrounding environment based on its camera, and transmits video frames to the edge cloud to process. The edge cloud then applies various vision analysis techniques to understand the frame and transmit the result back to the devices. To reduce the network bandwidth consumption of video frame transmission, the system typically uses video compression techniques to encodes the video frames on the sender side and decodes them on the receiver side. These entire offloading pipeline provides us the opportunities to apply different optimization techniques and reduce the offloading latency.

Chapter 3

Designing a High-quality Untethered VR System with Low Latency Remote Rendering

3.1 Introduction

Virtual reality systems have provided unprecedented immersive experiences in the fields of video gaming, education, and healthcare. Reports forecast that 99 million Virtual Reality (VR) and Augmented Reality (AR) devices will be shipped in 2021 [1], and that the market will reach 108 billion dollars [2] by then. Virtual and augmented reality is also a key application driver for edge-computing and high-bandwidth wireless networking research.

Existing VR systems can be divided into two categories: High-quality VR and standalone VR systems. Due to the requirements of high quality and low latency, most high-quality VR systems, such as HTC Vive [5] and Oculus Rift [6], leverage a powerful desktop PC to render rich graphics contents at high frame rates and visual quality. However, most of these solutions are *tethered*: they must be connected to a PC via a USB cable for sending sensor data from the Head Mounted Display (HMD) to the PC and an HDMI cable for sending graphics contents from the PC back to the HMD. These cables not only limit the user's mobility but also impose hazards such as a user tripping or wrapping the cable around the neck. Standalone, portable VR systems such as Samsung Gear VR [7] and Google Daydream [8] run VR apps and render graphics contents locally on the headset (or a smartphone slide in the headset). Those VR systems allow untethered use but the rendering quality is limited by the capability of the headset or smartphone. There has been extensive demand for such untethered high-quality VR systems.

Untethered high-quality VR is highly desirable but extremely challenging. Ideally, the cables between the VR HMD and PC should be replaced by a wireless link. However, even existing high-quality VR systems operate at 2160x1200 resolution and 90Hz, which generates a data rate much higher than those supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. The necessary image encoding, makes it difficult to maintain the stringent VR motion-to-photon latency requirements, which are necessary to reduce motion sickness.

VR applications have motivated much wireless research to realize robust, high-capacity connectivity, for example in the 60 GHz range. Most existing research has independently focused on optimizing the wireless link [17, 18, 19, 20] or the VR graphics pipeline [4, 3]. To enable high-quality VR on smartphones, Furion [4] separates the rendering pipeline into tasks to render the image foreground and tasks to render the image background, so that the background rendering tasks can be offloaded over commodity Wi-Fi. Other emerging systems such as TP-CAST [21] and DisplayLink [22] replace the HDMI cable with a wireless link to enable untethered VR experience with remote rendering and streaming. However, none of them studied systems issues and optimization opportunities that arise when combining rendering, streaming, and display. Furthermore, it is still challenging to enable untethered VR for future 4K or 8K systems with framerates larger than 90Hz. Additionally, we discover that displaying remote-rendered frames on an HMD may also introduce extra latency due to the VSync driven rendering and display policy.

To overcome these challenges and facilitate such research, we propose an open remote rendering platform that can enable high-quality untethered VR with low latency on general purpose PC hardware. It reduces the streaming latency caused by frame rendering, encoding, transmitting, decoding and display through a Parallel Rendering and Streaming Pipeline (PRS) and Remote VSync Driven Frame Rendering (RVDR). PRS pipelines the rendering, encoding, transmission, and decoding process. It also parallelizes the frame encoding process on GPU hardware encoders. The RVDR technique carefully schedules the start time of sensor acquisition and rendering new frames on the server side so that the result arrives at the display just before the VSync screen update signal, thus reducing latency caused by the display update.

To evaluate the system, we implemented the end-to-end remote rendering platform on commodity hardware over a 60GHz wireless network. The result shows that the system supports existing high-quality VR graphics with a latency of less than 16ms. We further show promise to support future VR systems with 4K resolutions with a latency up to 20ms. To facilitate widespread use of this platform, it is currently designed as a software system using only general-purpose GPU and network hardware. Performance could also be further improved through hardware implementations of key components.

The contributions of this work are:

- Quantifying component latency in an end-to-end wireless VR system and identifying the impact of the screen refresh (VSync). §3.2
- Designing a pipeline of Parallel Rendering and Streaming to reduce the streaming latency caused by encoding, transmitting and decoding the frames. §3.4
- Developing a method of Remote VSync Driven Rendering to adjust the timing of sensor data acquisition and rendering of a new frame based on the screen refresh timing (VSync signal) on the client side. §3.5
- Implementing and evaluating an open end-to-end remote rendering and streaming platform based on the system on commodity hardware over a

60GHz wireless network. §3.6

- Showing that the platform can support the current 2160x1200 VR resolution and the 4K resolution at 90Hz within 20ms latency over a stable 60GHz link even without complex modifications of the rendering process. §3.7

3.2 Challenges and Latency Analysis

Designing a high-quality untethered VR system is extremely challenging due to the stringent requirements on data throughput and end-to-end latency. Assuming we use three bytes to encode the RGB data of each pixel, for HTC Vive and Oculus Rift with a frame rate of 90Hz and a resolution of 2160x1200, the raw data rate is 5.6Gbps, much higher than the data rate (e.g., less than 2Gbps) supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. For future VR targeting at a resolution of 4K UHD or even 8K UHD, the required data rate would be as high as 17.9Gbps and 71.7Gbps, respectively.

To address the challenge of high data throughput, data compression is necessary. However, high-quality VR also requires a very tight total end-to-end (i.e., motion-to-photon) latency of 20-25ms [3] to reduce motion sickness. That is, once the HMD moves, the system must be able to display a new frame generated from the new pose of the HMD within 20-25ms. As compressing and decompressing frames introduce extra latency, it is even more challenging to meet the end-to-end latency requirement.

Latency analysis. We use the following equations to model the end-to-end latency of our proposed untethered VR system with remote rendering:

$$T_{e2e} = T_{sense} + T_{render} + T_{stream} + T_{display} \tag{3.1}$$

$$T_{stream} = T_{encode} + T_{trans} + T_{decode} \tag{3.2}$$

$$T_{trans} = \frac{FrameSize}{Throughput} \tag{3.3}$$

T_{e2e} is the total end-to-end latency in generating and displaying a new frame. It consists of four parts: the time for the rendering server to retrieve sensor data from the HMD (T_{sense}); the time for the rendering server to generate a new frame (T_{render}); the time to send the new frame from the rendering server to the HMD (T_{stream}); and the time for the HMD to display the new frame ($T_{display}$).

T_{stream} is the extra latency introduced by cutting the cord of a tethered VR system. It has three parts: the time to compress a frame on the rendering server (T_{encode}); the time to transmit the compressed frame from the rendering server to the HMD over a wireless connection (T_{trans}); and the time to decompress the received frame on the HMD (T_{decode}). T_{trans} is decided by the compressed frame size and the data throughput of the wireless connection.

$T_{display}$ also introduces significant latency. In modern graphics systems, frame displaying is driven by VSync signals that are periodically generated based on the screen refreshing rate. If a frame misses the current VSync signal after it is received and decoded on an HMD, it must wait in the frame buffer for the next VSync signal before it can actually be displayed on the screen (see more details in §3.5). For a frame rate of 90Hz, the average waiting time is 5.5ms. Such an extra latency may significantly impact the performance of a high-quality untethered VR system, and thus must be carefully mitigated as much as possible.

For the total 20-25ms budget of T_{e2e}, T_{sense} is small (less than 400μs in our system with a WiGig network). T_{render} may be 5-11ms depending on the rendering load. With a $T_{display}$ of 5.5ms, we have less than 10ms left for T_{stream}, including encoding, transmitting and decoding a frame, which makes it a very challenging task to meet the latency requirement of high-quality VR.

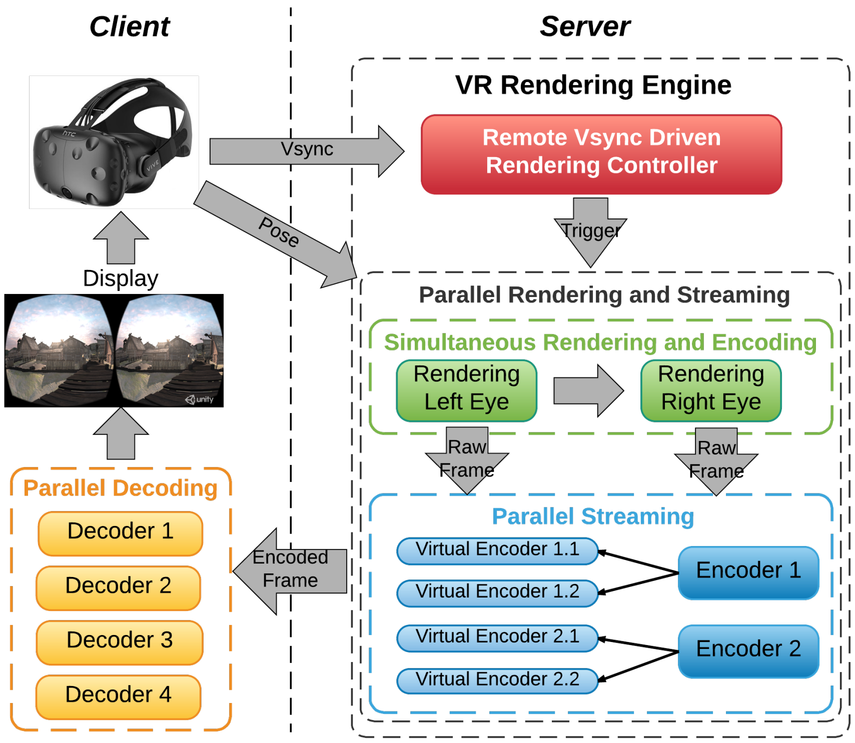


Figure 3.1: System architecture.

In this paper, we focus on minimizing T_{stream} to reduce the end-to-end latency T_{e2e}. This is mainly done through parallelizing frame rendering and frame encoding by leveraging the hardware capability. We cannot change T_{sense} and T_{render}. However, by carefully arranging the timing of rendering new frames with the latest pose of the HMD, we are able to reduce the frame waiting time for VSync signals, and thus reduce $T_{display}$.

Next, we describe our system design and the key techniques on how to address the challenges.

3.3 System Overview

Figure 4.4 shows the system architecture of our proposed untethered VR system with remote rendering. At a high level, it has two parts connected through a wireless link: an HMD as the client and a PC as the rendering server. The HMD client tracks the pose of the player and the timing of its VSync signals and sends the recorded data to the rendering server. If the player uses extra controllers to play the game, the client also sends the controller data to the server. Using the data received from the client, the server renders new frames, compresses and transmits them to the client. Upon receiving a new frame, the client decompresses and displays it on the HMD. The wireless link can be WiFi or 60GHz wireless communication such as WiGig. In our implementation, we use WiGig for its high data throughput and low latency.

To reduce the latency of streaming frames from the rendering server to the HMD client, we develop two key techniques. The first technique is *Parallel Rendering and Streaming* (PRS). PRS takes advantage of the two-eye image rendering nature in VR. Once the left eye image is rendered, PRS immediately sends it to the hardware encoder for compression. At the same time, PRS continues to render the right eye image, enabling simultaneous rendering and encoding. PRS further divides the image of each eye into two slides for parallel four-way encoding to fully utilize the hardware encoding capability. After a frame slide is encoded, it is immediately sent to the wireless link for transmission, without waiting for the whole frame to be compressed. Similarly, once the HMD receives a frame slide, it also immediately starts to decompress it, without waiting for the other frame slides. Consequently, we achieve parallel frame rendering, encoding, transmission, and decoding, and thus significantly reduce the latency.

The second technique is *Remote VSync Driven Rendering* (RVDR). The key idea of RVDR is to reduce display latency by deciding when to acquire head tracking sensor data and render a new frame based on the timing of the VSync signals of the HMD client. To do so, the client keeps tracking the time of its last VSync signal and the display delay of the last frame. Based on these timing information, the rendering server decides whether to start earlier to render the next frame so that the next frame can meet the next VSync signal on the client, or to slightly postpone the sensor acquisition and rendering of the next frame so that it arrives closer to the VSync signal and display latency is reduced. This is effective because the frame can be rendered with the latest possible pose of the HMD and thereby reduce motion-to-photon latency. As a result, we further reduce the display latency and minimize the rate of missing frames to maximize the user experience.

In our design, the HMD client uses dedicated hardware video codecs (e.g., H.264) to decode the frames rendered on the rendering server. We choose this design because hardware video codecs have a small size and consume a low power. Prior studies [23, 4] have shown that it is not practically feasible to decode frames using CPU or GPU on an HMD, due to the high decoding latency and high power consumption. Hardware video codecs are mature and cost-effective techniques, widely used in smartphones, tablets, laptops and many other devices. Thus, integrating hardware video codecs into HMDs is a practical solution to enable high-quality untethered VR systems.

3.4 Parallel Rendering and Streaming Pipeline

In this section, we introduce how we use the *Parallel Rendering and Streaming* (PRS) mechanism to minimize the streaming latency (T_{stream}) on commercial VR platforms. PRS consists of two parts: *simultaneous rendering and encoding*, and *parallel streaming*. Together, they build a parallel frame rendering and streaming pipeline to reduce the streaming latency.

3.4.1 Simultaneous Rendering and Encoding

Rendering a frame with rich graphics contents may take a long time (e.g., longer than 5ms) even on a powerful VR-ready GPU (e.g., Nvidia Titan X). As we cannot simply reduce frame quality to save frame-rendering time, we must find other ways to mitigate the impact of the long frame-rendering time on the end-to-end latency. To this end, we propose the approach of simultaneous rendering and encoding, to allow starting to encode a frame while it is still being rendered.

Simultaneous rendering and encoding is feasible due to two reasons. First, we observe that rendering a VR frame is typically done in three sequential steps: (1) render the left eye image, (2) render the right eye image, and (3) apply lens distortion on the whole frame so that the frame can be correctly displayed on a VR headset. This sequential rendering of the two-eye images provides an opportunity for us to start to encode the left eye image before the right eye image is fully rendered. Second, modern GPUs have dedicated hardware encoders and decoders that are separate from the GPU cores used for rendering VR frames (e.g., CUDA cores in Nvidia GPUs). Therefore, we may leverage the dedicated hardware encoders to compress a frame without impacting the performance of VR rendering.

Specifically, in simultaneous rendering and encoding, we redesign the VR rendering procedure to the following 6 steps: (1) render the left eye image, (2) apply lens distortion on the left eye image, (3) pass the distorted left eye image to the encoding pipeline in a separate thread, and at the same time (4) render the right eye image, (5) apply lens distortion on the right eye image, (6) pass the distorted right eye image to the encoding pipeline in another separate thread. Note that only steps (1), (2), (4), and (5) execute on the main rendering thread, while steps (3) and (6) execute on two separate encoding threads using hardware-based encoders. These encoding operations mainly consume the hardware-based encoder resources with light CPU usage, and thus do not block or slow down the frame-rendering pipeline inside the GPU.

Traditional video streaming approaches usually also use hardware-based encoders to accelerate the video-encoding procedure. However, they keep waiting for a frame being fully rendered before passing the entire frame to a hardware

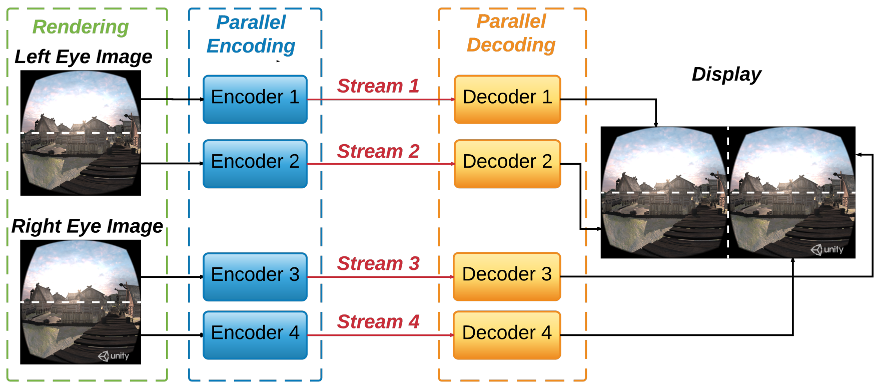


Figure 3.2: Simultaneous rendering and encoding with 4-way parallel streaming.

encoder. Such a design largely increases the end-to-end latency, which is fine to video streaming with a low frame rate (e.g., 30 fps) and without user interactions, but it is not acceptable in high-quality interactive VR systems.

3.4.2 Parallel Streaming

To further reduce the streaming latency, we propose to use a multi-threaded streaming technique to encode the image of each eye in multiple encoding threads. This is because almost all high-performance GPUs support more than one video encoding session [24] and each encoding session generates its own encoding stream independently. Consequently, by dividing an image into multiple slides and encoding each slide using different encoding sessions in parallel, we can reduce the total encoding time. In our system, we cut the image of each eye into two slides and compress each slide in a separate video stream. In total, we have four slides of the two eyes for 4-way parallel streaming. On the client side, multiple decoding sessions are used to decode each image slide in parallel as well.

This *parallel streaming* mechanism can be combined with *simultaneous rendering and encoding*. Figure 3.2 shows the process of simultaneous rendering and encoding together with 4-way parallel streaming. The image of each eye is divided into two slides: the upper one and the bottom one. The total four image slides

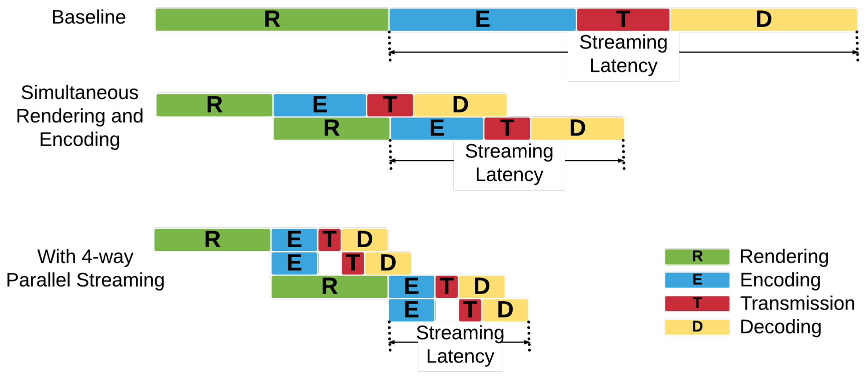
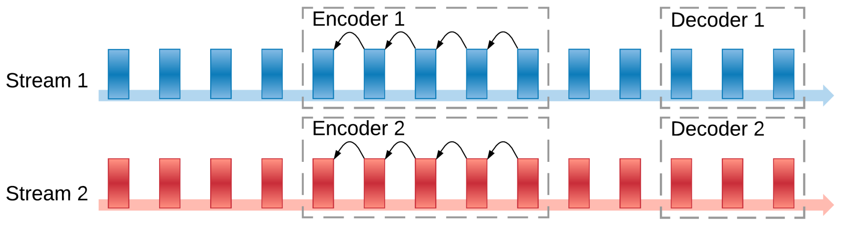


Figure 3.3: Illustration of parallel rendering and streaming (PRS) pipeline.

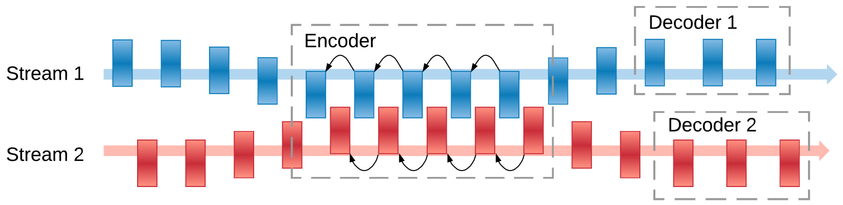
are encoded into four video streams using four encoders. Accordingly, the HMD client uses four decoders to decode the four video streams, composites the four image slides into a full frame, and displays the frame on the HMD.

Figure 3.3 illustrates how the PRS mechanism can reduce the streaming latency through simultaneous rendering and encoding and 4-way parallel streaming, in comparison to a baseline approach. Four main tasks (rendering, encoding, transmission, and decoding) are represented with rectangles in different colors. The length of each rectangle is the rough execution time of the corresponding task. Note that here we analyze only the streaming latency rather the total end-to-end latency. Thus, we do not show the time of fetching the sensor data before rendering a frame and the time that the frame waits in the frame buffer after it is decoded but before it is displayed. In the baseline approach, the four tasks execute sequentially. The streaming latency, as shown in Figure 3.3, is the total execution time of encoding, transmitting and decoding the whole frame.

With *simultaneous rendering and encoding* (middle in Figure 3.3), the encoding of the left eye image starts immediately after the left eye image was rendering and in parallel with the rendering of the right eye image. As a result, this two-way parallel approach reduces the user-perceivable *add-on streaming latency*, i.e., the extra streaming latency after the whole frame is rendered, by 1/2. By combining the *simultaneous rendering and encoding* with 4-way *parallel streaming* together



(a) Encode two streams with two encoding sessions.



(b) Encode two streams with only one encoding session.

Figure 3.4: Encoder multiplexing.

(bottom in Figure 3.3), we use two encoders to compress the image of one eye in parallel. The add-on streaming latency further reduces to around 1/4 of the one in baseline approach.

The parallel streaming technique may be further extended to 8-way or even 16-way for more parallelisms. However, we do not recommend doing so because 1) it requires more simultaneous encoding sessions that may not be possible on many GPUs as we will show later, 2) it reduces the performance of motion estimation in H.264 and thus leads to a lower compression rate, and 3) it makes the implementation more complex.

Encoder Multiplexing

Ideally, the 4-way parallel streaming approach requires four encoding sessions. However, in practice, many GPUs including some popular commercial VR-ready GPUs may not support four simultaneous encoding sessions. For example, the Nvidia's GeForce 9 and 10 series and TITAN X GPUs [25] can support maximum two encoding sessions running simultaneously on a single GPU [1]. Consequently, the 4-way parallel streaming approach cannot directly work on those GPUs.

From Figure 3.3, we observe that even though the 4-way parallel streaming approach uses four encoding streams, there are only up to two streams overlapped at any time. This is because the rendering of the left eye image and the rendering of the right eye image are sequential. As rendering an image usually takes a longer time than compressing the image, encoding the half image of the left eye is expected to be much faster than rendering the image of the right eye. Therefore, only the encoding of the upper and bottom slides of the same eye image overlap. That is, only two encoding sessions are actually needed at the same time.

However, we still cannot directly encode four streams using only two encoding sessions due to the inter-frame compression in video encoding (e.g., H.264 and H.265). Compared to image compression that compresses independent images (e.g., JPEG), video compression encodes a set of images into a video stream containing 3 types of frames: I-frames, P-frames, and B-frames. An I-frame is encoded from a single image and can be decoded independently from other frames. However, a P-frame contains references from the previously encoded frame for a higher compression ratio, and therefore cannot be decoded by itself. A B-frame further uses bi-directional prediction that requires both its prior frame and its latter frame as its references, introducing more inter-frame dependencies in its encoding and decoding [2]. These inter-frame dependencies make it hard to encode two video sources using one single encoder.

To solve the problem, we propose *encoder multiplexing* [3] to temporally allow

[1]This limitation may be just due to cost or marketing strategy considerations, as consumer devices are mostly designed for decoding existing video streams rather than encoding new streams.

[2]As a result, B-frames can only be encoded and decoded when the next frame is available. Waiting for the next frame significantly increases the streaming latency and makes it infeasible to use B-frames for low latency VR systems. We do not use B-frames in our system.

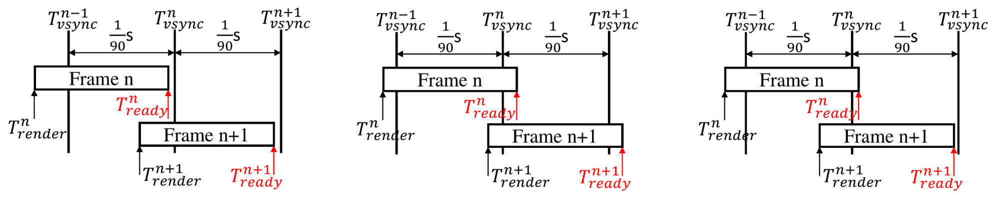
[3]Most devices including smartphones are able to decode four or even more (e.g., eight) video

two video streams to share the same encoding session, as shown in Figure 3.4. Figure 3.4(a) shows the standard usage of a single encoding session for a single video stream. The two video streams are encoded in two different encoding sessions separately. Each encoder compresses its raw input frames rendered from the rendering server into an encoded H.264 stream, and sends the encoded stream through the network link to the client. Each P-frame references to its previous frame in the same stream and thus can be correctly decoded by the corresponding decoder.

As shown in Figure 3.4, with *encoder multiplexing*, we encode the two video streams in the same encoding session. To encode each P-frame correctly, we set the previous frame in the same video stream as the long-term reference frame (LRF) of the P-frame. As a result, each P-frame references to the previous frame in its own video stream rather than the previous frame that is encoded in the encoding session (which is from the other stream). The outputs of the encoder will be divided into two streams that are sent to two different decoders on the client. Even though each decoder only receives half of the encoded outputs of the encoder, it has the reference frame needed to decode received P-frames. What we need to do is only changing the list of decoded picture buffer (DPB) before passing each stream to the decoder, to let it ignore those missed frames (i.e., the frames of the other steam).

Specifically, we use one encoding session to encode the upper half images of two eyes and another encoding session to encode the bottom half images of two eyes, respectively. As a result, we enable four virtual encoders for the 4-way parallel streaming using only two encoding sessions and make our approach work on most GPUs.

streams simultaneously [4]. Thus, we do not need *decoder* multiplexing.



(a) Ideal case (b) Long waiting time (c) Missing a frame

Figure 3.5: Displaying two consecutive frames that are remotely rendered. (a) The ideal case where the frames are displayed immediately after they are ready. (b) The frames failed to meet their VSync signal and must wait for a long time before actually being displayed. (c) The frames become ready in the same VSync interval and thus frame n is dropped.

3.5 Remote VSync Driven Rendering

Modern computer systems use VSync (Vertical Synchronization) signals to synchronize the rate of rendering frames (i.e., frame rate) and the refresh rate of a display. To ensure a smooth user experience (e.g., avoid screen tearing), the *double buffering* technique is usually used with two frame buffers: a front frame buffer containing the frame that is being displayed on the screen, and a back frame buffer containing the frame that is being rendered. Upon receiving a VSync signal, the system swaps the two buffers to display the newly rendered frame and continues to render the next frame in the new back frame buffer. If the system renders a frame very fast, the frame must wait for the next VSync signal in the back frame buffer, before it is sent to the display. If rendering a frame takes too long and misses the next VSync signal, it must wait for the following VSync signal to be displayed.

Problems. The above VSync-driven rendering and displaying mechanism works well on a local system. However, in remote rendering, the frame displaying is driven by VSync signals of the HMD client but the frame rendering is driven by VSync signals of the rendering server. Due to the asynchronized frame rendering and displaying and the extra streaming latency, it may cause problems.

To illustrate the problems, we show the processing procedure of two consecutive frames n and $n+1$ in Figure 3.5. The rendering server starts to render frame n at time T^n_{render}. The frame is then encoded and transmitted to the client. The client decodes the frame and presents it to the frame buffer swap chain at time T^n_{ready}. We define the time interval between T^n_{render} and T^n_{ready}, i.e., $T^n_{ready} - T^n_{render}$, as the *generating time* of frame n. Ideally, the frame is ready just before VSync signal n at time T^n_{vsync}, so that it can be displayed immediately. This ideal case is shown in Figure 3.5(a). However, if frame n missed VSync signal n (i.e., $T^n_{ready} > T^n_{vsync}$), it must wait for VSync signal $n+1$ and thus the end-to-end latency is increased by $T^{n+1}_{vsync} - T^n_{ready}$. Such a long waiting time case is shown in Figure 3.5(b) [4]. Furthermore, if frame n missed the VSync signal n, and at the same time frame $n+1$ becomes ready before time T^{n+1}_{vsync} (i.e., $T^{n+1}_{ready} < T^{n+1}_{vsync}$), frame n will become useless and thus will be dropped. Instead, frame $n+1$ will be displayed upon VSync signal $n+1$. This case is called *frame missing* as shown in Figure 3.5(c) [5]. In this case, the time and resources used to render, encode, transmit and decode frame n are wasted.

To confirm that the two problems are real, we conduct an experiment. In this experiment, we cap the frame rate on the rendering server to 90 Hz. On the client, for each frame n, we record the time interval ΔT^n between the frame-ready time T^n_{ready} and the next VSync signal time after T^n_{ready}. We define such a time interval ΔT^n as the *waiting time* of frame n. Figure 3.6 plots the waiting time of more than 200 consecutive frames. It shows that the frame waiting time keeps drifting from 0 ms to 11.1 ms periodically. This is because that the rendering server is unaware of the VSync signals of the client and thus cannot synchronize its rendering with the frame displaying on the client. This phenomenon not only

[4] This case may also happen in local systems without remote rendering. However, the extra streaming latency in remote rendering makes this case happen more frequently.

[5] This case is caused by remote rendering and will not happen in a local system unless VSync is disabled.

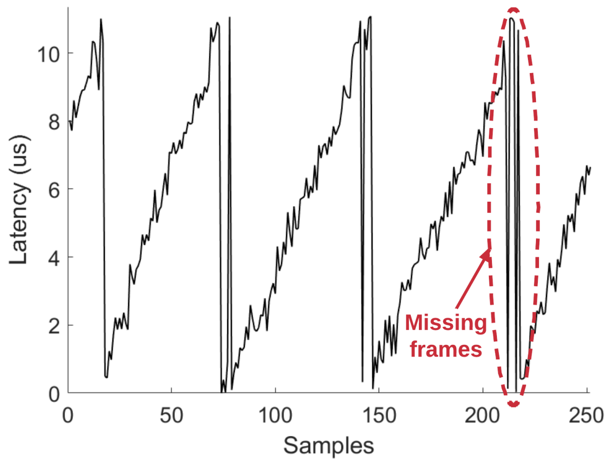


Figure 3.6: Time till next VSync signal.

introduces additional latency as shown in Figure 3.5(b), but also causes frame missing when ΔT^n jumps in consecutive frames, which is shown in the red dotted circle in Figure 3.6. In the red dotted circle, for a frame n with a very large ΔT^n close to 11 ms, the ΔT^{n+1} of the next frame $n+1$ may immediately become very small close to 0 ms. In this case, the two frames n and $n+1$ are ready to display within the same VSync interval and thus frame n will not be displayed, which is the frame-missing case in Figure 3.5(c).

Solution. To solve the problems, we propose to drive the frame rendering of the server using the VSync signals of the client. The key idea is adjusting the timing of rendering the next frame according to the feedback from the client on how the previous frame was displayed, how long its waiting time was, and how fast the HMD moved. Specifically, we use the following equations to decide when to start to render a new frame $n+1$:

$$T_{render}^{n+1} = T_{render}^{n} + \frac{1}{90}s + T_{shift} \tag{3.4}$$

$$T_{shift} = (T_{vsync}^{n} - T_{ready}^{n} - T_{conf} - T_{motion}) * cc \tag{3.5}$$

$$T_{motion} = k * \Delta\theta^n \tag{3.6}$$

In Equation 3.4, besides using a constant time interval of 1/90 seconds to maintain the frame rate of 90 Hz, we further introduce a time shift T_{shift} that dynamically modifies the start time of rendering frame $n+1$ (i.e., T^{n+1}_{render}). T_{shift} is decided by several factors. The first factor is the waiting time of frame n, $T^n_{vsync} - T^n_{ready}$. We intend to postpone T^{n+1}_{render} for time interval $T^n_{vsync} - T^n_{ready}$. Assume frames n and $n+1$ have the same generating time, i.e., $T^{n+1}_{ready} - T^{n+1}_{render} = T^n_{ready} - T^n_{render}$, the time of placing frame $n+1$ to the swap frame buffer chain T^{n+1}_{ready} exactly equals to T^{n+1}_{vsync}. This way, the waiting time of frame $n+1$ is minimized. However, if it takes a slightly longer time period to generate frame $n+1$, frame $n+1$ may miss its VSync signal $n+1$, resulting in a very long waiting time or even missing frame $n+1$. To mitigate this issue, we introduce T_{conf} to shift T^{n+1}_{ready} back a lit bit to tolerate the variance in generating frames.

We take a data-driven approach to decide a proper value for T_{conf}. Initially, we set T_{conf} to zero. We track the frame-generating time of the last 1,000 frames. We calculate the value that covers the variances of the frame-generating time of the last 1,000 frames with a confidence of 99% (a.k.a 99% confidential interval). We set the value of T_{conf} to the half of the 99% confidential interval. Over time, T_{conf} acts as an adaptive safeguard to handle the variance in generating consecutive frames.

Another factor we consider in T_{shift} is how fast the HMD moves. The intuition behind this consideration is as follows: when the HMD moves fast, the view of the VR game may change fast. As a result, the content of the next frame may have significant changes and thus its rendering time might be longer than that of its previous frame. To accommodate the large rendering cost of the next frame, we need to start to render it early to avoid missing VSync. We use T_{motion} for this purpose in Equation 3.5. However, as the frame rate is as high as 90 Hz, the absolute distance that the player may move along a direction in a frame time (i.e., 1/90 seconds) is pretty small and thus has limited impact on the content changes of the next frame in practice. But, the rotation of the HMD may significantly affect the content of the next frame due to the change of the viewing angle. Therefore, we only consider the viewing angle changes in our design. As shown in Equation 3.6, T_{motion} is determined by the change of viewing angle $\Delta\theta^n$ together with a constant scaling parameter k. We empirically set the value of k to 100.

Finally, we use a scaling parameter cc as a low-pass filter in calculating the value T_{shift}, as shown in Equation 3.4. We empirically set its value to 0.1.

With this *remote VSync driven rendering* approach, we try to ensure that the system can stay in the ideal case shown in Figure 3.5(a). We expect that most frames become ready to display just before the next VSync signal and thus have a very short waiting latency, and that very few frames are dropped. Furthermore, even we may postpone the rendering of a frame, we always use the latest possible pose of the HMD to render the frame. This is achievable because the sampling rate of HMD pose is as high as 1,000 Hz, one order of magnitude higher than the frame rate. As we display the postponed frame with the best possible VSync signal, we minimize the user-perceived latency and thus provide the best user experience. Indeed, we may even be able to do better than the tethered system. As we will show in Section 3.7, if the rendering time of a frame is very short, we may delay its rendering to reduce its waiting time. As we render it with a fresher HMD pose, doing so not only achieves a lower end-to-end latency, but also provide a better user experience, compared to the tethered system.

3.6 Implementation

We implement our system on Windows for its rich supports on VR. We use Qualcomm WiGig solution to achieve 2.1 Gbps wireless transmission throughput.

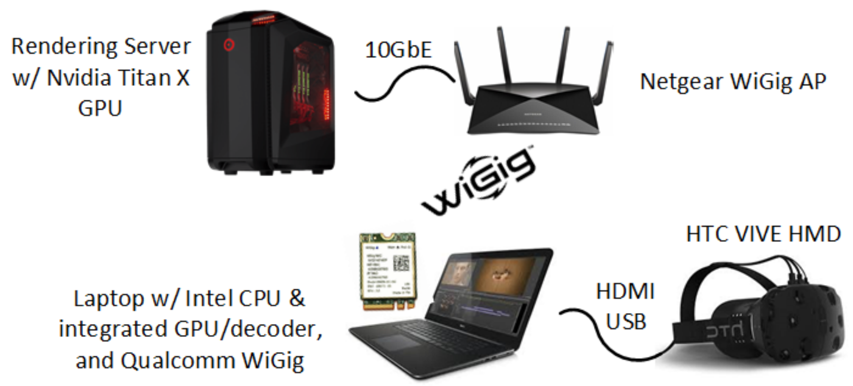

Rendering Server w/ Nvidia Titan X GPU
10GbE
Netgear WiGig AP
WiGig
HTC VIVE HMD
Laptop w/ Intel CPU & integrated GPU/decoder, and Qualcomm WiGig
HDMI
USB

Figure 3.7: Hardware setup.

Our implementation is entirely based on commodity hardware and consists of around 7,000 lines of code.

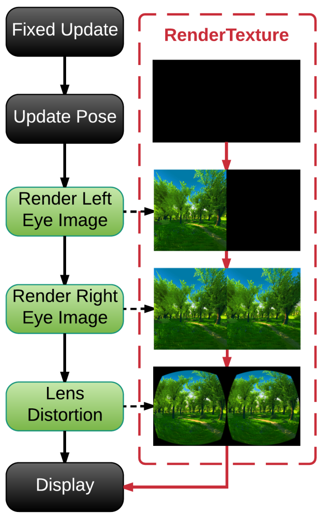
3.6.1 Hardware Setup

As shown in Figure 3.7, the rendering server is an Intel Core i7 based PC with a Nvidia TITAN X GPU. It has a Mellanox 10Gbps network interface card to connect to a Netgear Nighthawk X10 WiGig AP using a 10Gbps Ethernet cable. We use a ThinkPad X1 Yoga laptop to act as the client that connects to the Wigig AP through a Qualcomm QCA6320/QCA6310 WiGig module. The laptop equips an Intel i7-7600U CPU and an HD 620 Integrated GPU with H.264 hardware decoder ASIC included. The laptop connects to an HTC Vive HMD using HDMI and USB.

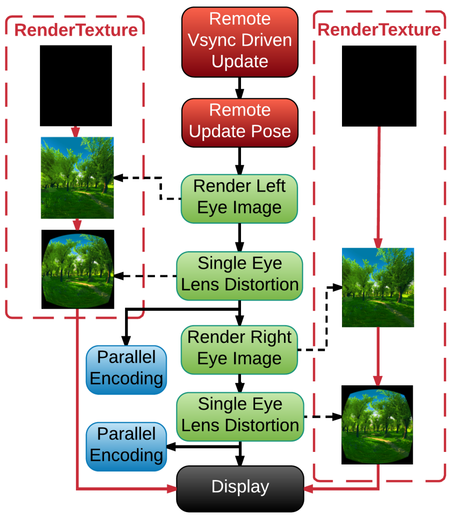
3.6.2 Software Implementation

We implement our proposed techniques based on the OpenVR SDK [26], the Unity game engine [27], and the Google VR SDK for Unity [28].

Remote Rendering VR Camera. The core of the server-side implementation is a VR camera that leverages Unity's rendering solution and Nvidia's Video Codec for low-latency remote rendering. This VR camera is modified based on the



(a) Normal VR Camera.



(b) Our VR Camera.

Figure 3.8: Life cycle comparison between our VR camera and normal VR camera.

solution from the Google VR SDK [28]. Figure 3.8 shows the life cycle comparison between the normal VR camera used in the Google VR and our VR camera. The normal VR camera starts rendering each frame from a fixed update callback function, driven by the periodic VSync signals of the system. Then, the camera updates user's pose and rotates itself towards the correct direction in the 3D scene. After that, the camera renders the left eye image and the right eye image sequentially, then applies lens distortion on the whole frame. On the right side of Figure 3.8(a), we show the changes of the RenderTexture in each step. A RenderTexture is a texture that can be rendered to by D3D or OpenGL. In the normal VR camera case, a screen-size RenderTexture is allocated when the VR camera is created. The camera renders left eye image and right eye image to each side of the render texture sequentially. Then, a dedicated vertex shader is applied to the whole texture to do the VR lens distortion [6].

Our VR camera works as shown in Figure 3.8(b). The fixed update function

[6]For illustration purpose, we set the original RenderTexture in Figure 3.8 as a black frame. In real systems, it should be the last frame when starting rendering the current frame.

is changed to a remote VSync driven update function to optimize the rendering time based on the VSync signals on the HMD client. VSync time can be retrieved from the client side by calling the GetTimeSinceLastVsync() in the OpenVR API [26]. Similarly, the pose update function is changed to the remote pose update to get the pose information from the client. In this function, the server sends a request packet to the client through the wireless network. The client calls the GetDeviceToAbsoluteTrackingPose() function in the OpenVR API to get the current pose of both the HMD and controllers, and send them back to the server. Instead of using a single RenderTexture as the render target, we create two RenderTextures of half screen size for the image of each eye separately. We modify the distortion shader to work for only single eye image, and apply it on the RenderTexture immediately after rendering the image of one eye.

Parallel Encoding. Our parallel encoding module is developed as a Unity native plugin attached to the remote rendering VR camera. This module uses zero-copy between rendering and encoding in the GPU. We register the RenderTexture of each eye using the CUDA function cudaGraphicsD3D11RegisterResource() so that it can be accessed from CUDA. We directly pass the registered memory address to Nvidia's hardware video codec for encoding, and thus do not need any memory copy. As mentioned in Section 3.4, the system passes each RenderTexture to two separate encoders for parallel encoding, and creates total 4 encoding streams. The encoding operation is executed asynchronously without blocking the rendering of frames. To maintain a high image quality with low latency, we use the Two-Pass Rate Control setting with 400KB maximum frame size to encode each frame. With this setting, the encoded frame size is capped to 400KB.

Parallel Decoding. We implement the parallel decoding module on the commercial Intel-based low power video codec, using the Intel Media SDK [29] that can be run on any 3rd generation (or newer) Intel Core processors. In this module, four decoding sessions are created to decode four encoded streams in parallel. A separate display thread is developed to keep querying the decoded frame blocks from the decoder sessions and display them on the corresponding positions on the HMD screen. This module also calls the OpenVR API to retrieve the pose and VSync data for sending to the rendering server.

3.7 Evaluation

We evaluate the performance of our system in terms of end-to-end latency, the trade-off between visual quality and add-on streaming latency, frame missing rate, and resource usage on the client. We demonstrate that our system is able to achieve both low-latency and high-quality requirements of both the tethered HTC Vive VR and future VR at 4K resolution at 90Hz over a stable 60GHz wireless link. The result shows the system can support current 2160x1200 VR resolution with 16ms end-to-end latency and 4K resolution with 20ms latency. Furthermore, we show that our system misses very few frames in different VR scenes and uses only a small portion of CPU and GPU resources on the client.

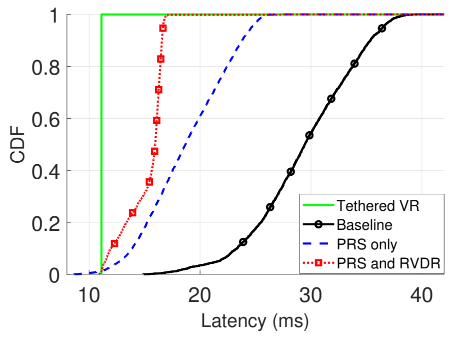
As shown in Table 3.1, our system outperforms previous untethered VR system [4, 3] in all four aspects: end-to-end latency, frame rate, visual quality, and resolutions. However, as we mentioned, we target to optimize the system issues arise when combining the whole rendering and streaming pipeline, thus is complementary to previous work. Performance may further improve when combining different approaches.

Mobile VR	Latency (ms)	Frame Rate (Hz)	Visual Quality (SSIM)	Resolution (pixels)
Flashback [3]	25	60	0.932	1920x1080
Furion [4]	25	60	0.935	2560x1440
Ours	20	90	0.987	3840x2160

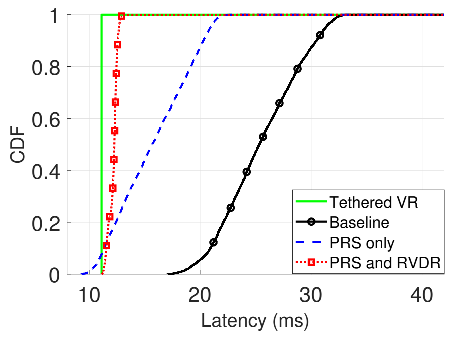
Table 3.1: Comparison between different mobile VR systems.

3.7.1 Experiment Setup

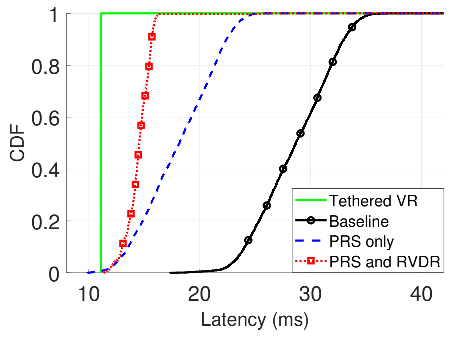
We use the hardware setup described in Section 3.6 to conduct experiments. Since our objective is not to improve 60GHz communication but provide an open platform to do so, we keep the wireless link stationary all the time [7]. We compare our system with the tethered HTC Vive VR system and a baseline untethered solution. As described in Section 3.4, the baseline solution is the typical video streaming approach that executes the rendering, encoding, transmission, and decoding sequentially, without parallel pipelines. We use four different VR scenes in our evaluation: *Viking Village* [30], *Nature* [31], *Corridor* [32], and *Roller Coaster* [33]. These four scenes are carefully selected to cover different kinds of VR applications. Viking Village and Nature are rendering intensive, requiring up to 11ms to render a frame even on our rendering server. Viking Village is a relatively static scene, while Nature has more than hundreds of dynamic objects (leaves, grasses, and shadows) that keep changing all the time. Corridor and Roller Coaster have relatively light rendering loads. Different from the other three scenes, Roller Coaster lets a player sit on a cart of a running coaster. Thus, the player's view keeps changing even if the player doesn't move at all.



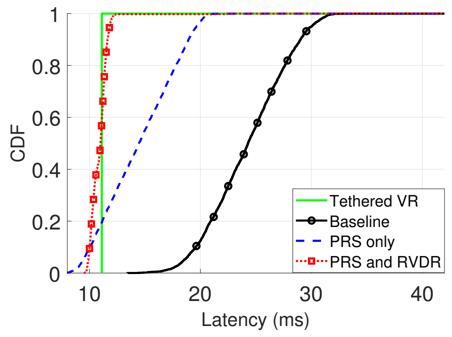
(a) Viking Village



(b) Corridor



(c) Nature



(d) Roller Coaster

Figure 3.9: CDF of end-to-end latency of 4 different approaches in 4 VR scenes.

3.7.2 End-to-end Latency

We first measure the end-to-end latency (T_{e2e}) in four cases: the tethered HTC Vive VR system (Tethered VR), the baseline untethered solution (Baseline), our solution with only the PRS technique (PRS only), and our solution with both PRS and RVDR techniques (PRS and RVDR). For repeatable experiments, we pre-logged a 1-min pose trace for each VR scene and replayed it for the experiments of the same scene in the four cases [8].

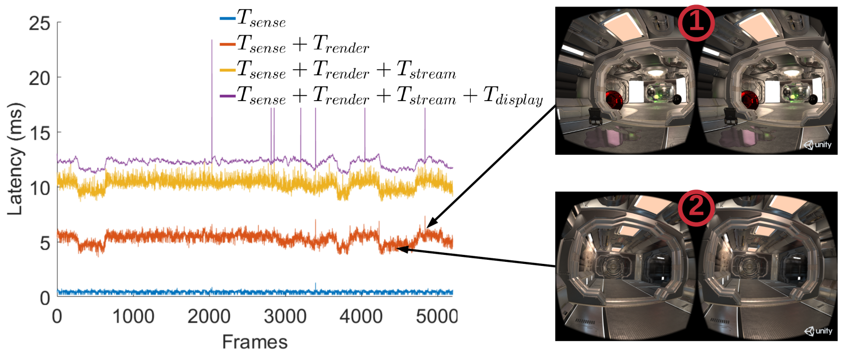
Figure 3.9 shows the CDFs of the measured results. In Tethered VR, T_{e2e} is always 1/90 seconds for the periodic VSync signals at 90Hz. In Baseline, T_{e2e} is very large due to the large extra cost of the sequential frame rendering, encoding,

[7]Mobility is an issue in existing 60GHz wireless products and there is active on-going research on addressing that issue [17, 18, 20].

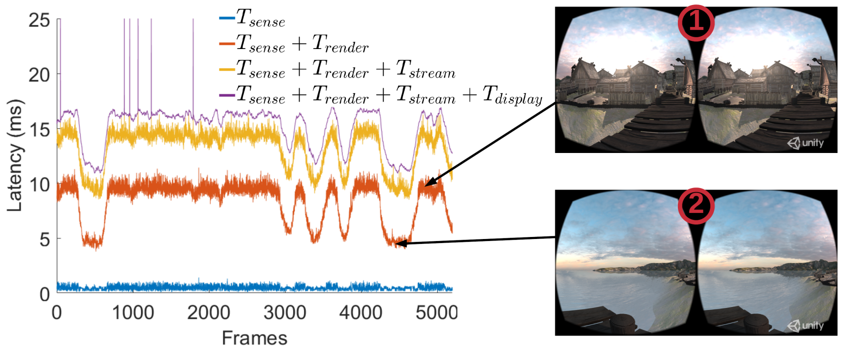
[8]We replay the logged traces on the client upon the requests from the server. Thus, the measured end-to-end latency includes the time cost of sensor data acquisition.

transmission, and decoding. The median value of T_{e2e} is more than 26ms and the maximum value is even larger than 38ms. Due to asynchronized VSync signals on the client and the server, the variance of T_{e2e} is also very large. With the PRS technique, we reduce T_{e2e} for more than 10ms, compared to Baseline. Combing the techniques of both PRS and RVDR, we not only reduce the average T_{e2e} to only 10-14ms depending on which scene is used, but also largely reduce the variance of T_{e2e}. For the scenes with light rendering loads, such as Corridor and Roller Coaster, our solution can achieve a comparable performance to Tethered VR. In Roller Coaster, we may even achieve a lower end-to-end latency (i.e., $< 11ms$) for many frames, because we render them with a fresher pose.

From Figure 3.9, it shows that T_{e2e} varies in different VR scenes. To figure out the reason, we plot the raw latency traces of two VR scenes (Corridor and Viking Village) in Figure 3.10. We breakdown the total end-to-end latency into four parts/tasks: T_{sense}, T_{render}, T_{stream}, and $T_{display}$. Each curve in Figure 3.10(a) and Figure 3.10(b) shows the total latency after each task is finished. For example, the curve at the bottom is T_{sense}, and the curve on the top is the sum of T_{sense}, T_{render}, T_{stream}, and $T_{display}$ (i.e., the total end-to-end latency T_{e2e}). Figure 3.10 shows that the rendering latency T_{render} is a critical part that affects the variance of the total end-to-end latency T_{e2e}. For Corridor, our system achieves a similar T_{e2e} compared to Tethered VR, because the rendering time $T_{rendering}$ is small (around 5ms). However, for rendering-intensive Viking Village, the rendering time $T_{rendering}$ is large (up to more than 10ms), resulting in a large T_{e2e}.



(a) Corridor



(b) Viking Village

Figure 3.10: Raw latency traces of our system in running Corridor and Viking Village.

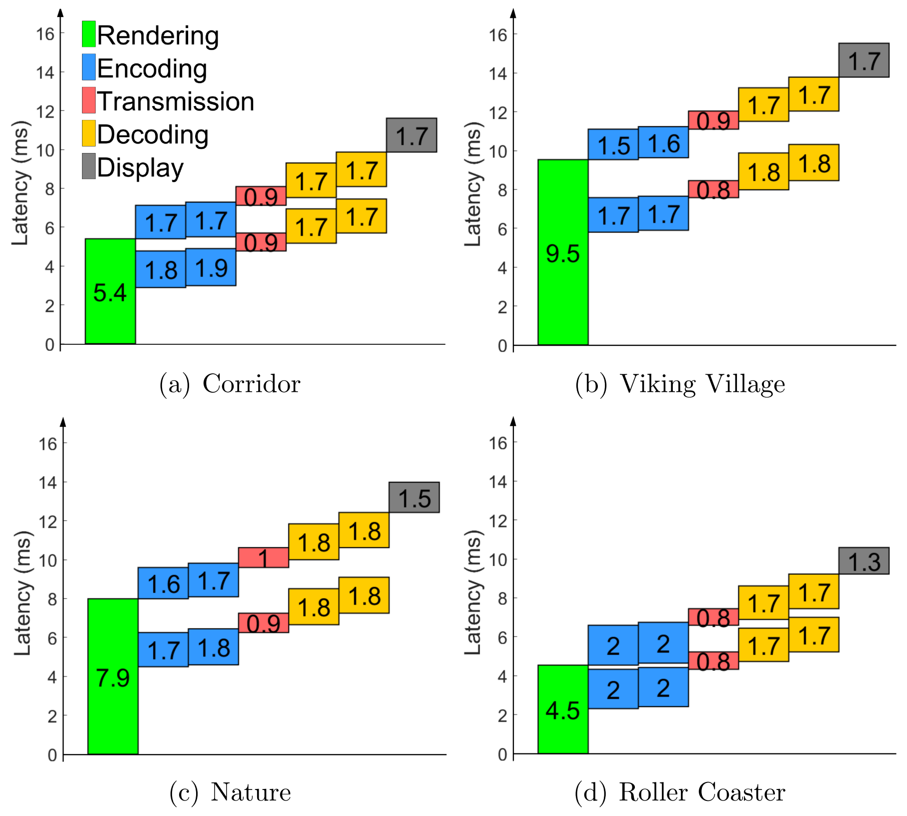
Furthermore, we observed that T_{e2e} has a small variance in Corridor, but a large one in Viking Village, fluctuating from 11ms to 15ms. This difference is also caused by rendering time. For illustration, we show two frames from each game in Figure 3.10. Each frame points to its rendering time on the corresponding rendering latency curve. In Corridor, the rendering time of different frames is relatively constant. In Viking Village, frame #2 (the user looked towards the water) has much less rendering load than frame #1, when the user faced the village. When the user moves, the rendering latency keeps fluctuating, and the total latency changes accordingly. However, as we apply the constant bitrate control setting in encoding each frame, the add-on streaming latency T_{stream} stays at a constant value no matter how the rendering latency changes.

Latency distribution in each step. Figure 3.11 shows the average latency of each step in frame rendering, encoding, transmission, decoding and displaying, and how the steps are overlapped. It shows that our PRS technique is able to build a very effective parallel pipeline to reduce the end-to-end latency. The average display time is only less than 1.7ms, demonstrating the effectiveness of our RVDR technique in reducing the display latency.

Frame rate. With the low add-on streaming latency in our system, we can actually enable a frame rate higher than 90Hz. Indeed, our system is able to achieve 150 frames per second in Corridor, demonstrating the capability of our system in supporting future VR at a higher frame rate, e.g., 120Hz.

3.7.3 Add-on Streaming Latency vs Visual Quality

The visual quality of encoded frames plays a critical role in providing a good user experience. Our system not only achieves low-latency remote rendering, but also keeps a visually lossless (visually identical) experience to users. It is well known that there's a trade-off between the visual quality and the streaming latency. In our system, visual quality is controlled by the bitrate in the Rate Control settings of encoding. To quantify the visual quality of an encoded frame, we use the widely used Structural Similarity (SSIM) to determine how similar the encoded frame is to the original frame. Table 3.2 shows the average SSIM score of the four VR scenes in different bitrate settings (each results in a different encoded frame size). From a previous study [12], a visually lossless encoded visual requires the SSIM score to be larger than 0.98. In Table 3.2, it shows that the encoded frame size of 400KB or larger can achieve an SSIM score of more than 0.98 for all the four VR scenes.



(a) Corridor (b) Viking Village

(c) Nature (d) Roller Coaster

Figure 3.11: Average latency (in ms) of each frame-processing step in the parallel pipeline.

To further build a relationship between add-on streaming latency and visual quality, we calculate the add-on streaming latency in different encoded frame sizes. Figure 3.12 shows how T_{stream} changes with the image SSIM score in Corridor, with the comparison of the Baseline approach and our approach. We also draw the visually lossless quality cutting curve as a red dash line. It shows the system requires an encoded frame size of more than 200KB to achieve visually lossless quality. The add-on streaming latency requirement to achieve this quality in our system is only around 4ms which is much smaller than the one of the Baseline approach.

Frame Size	Nature	Corridor	Viking	Roller
800KB	0.9914	0.9936	0.9927	0.9965
400KB	0.9831	0.9887	0.9865	0.9913
200KB	0.9707	0.9780	0.9763	0.9838
100KB	0.9560	0.9609	0.9643	0.9718
60KB	0.9411	0.9439	0.9478	0.9605

Table 3.2: Encoded frame size and visual quality measurement (SSIM score).

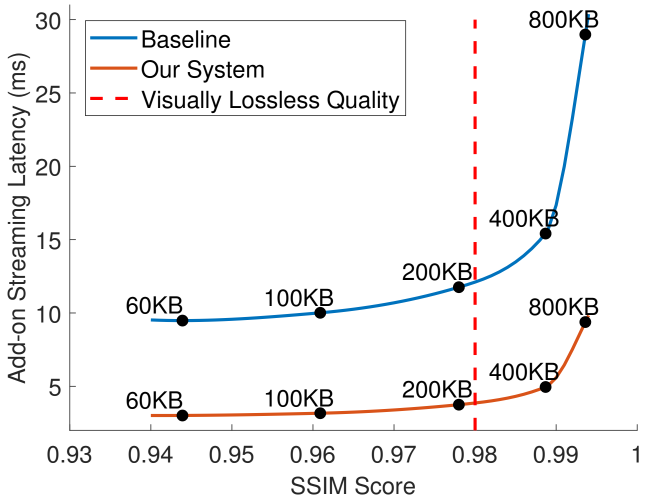


Figure 3.12: Add-on streaming latency vs visual quality for Corridor.

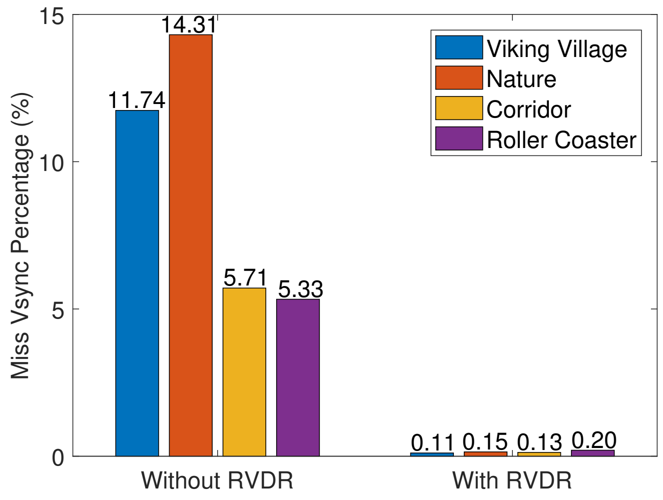


Figure 3.13: Frame missing rate with and without RVDR in the 4 VR apps.

3.7.4 Frame Missing Rate

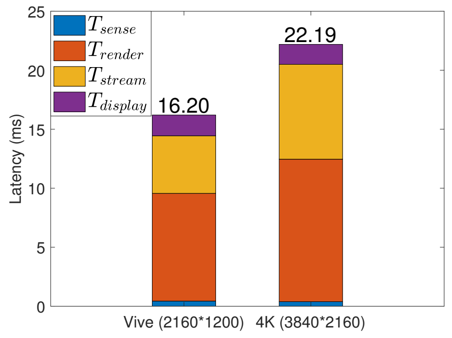
In Figure 3.10, each pin-shape peak on the top purple curve represents a missing frame. To quantify the effectiveness of the RVDR technique in reducing frame missing, we calculate the average frame-missing rate in the four VR apps with and without RVDR enabled. Figure 3.13 shows the results. We can see that RVDR is able to significantly reduce the frame-missing rate. Without RVDR, the average frame-missing rate is from 5.3% to 14.3%. With RVDR, it is reduced to only 0.1% - 0.2%.

3.7.5 Resource Usage on Client

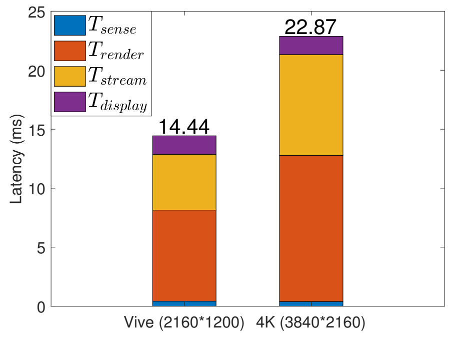
We also measure the resource usage on the client. Without our system, Windows 10 uses 3% to 5% CPU. With our system playing a VR game, the CPU usage is around 36% mainly for handling the network packets. Decoding frames use only 29% of the video-decoding capability of the GPU (not the whole GPU capability such as rendering 3D scenes). In building a future untethered HMD using our techniques, such resource usage may be further significantly reduced by a more integrated hardware design with embedded software. We further discuss the power consumption of the system in Section 3.9.

3.7.6 4K Resolution Support

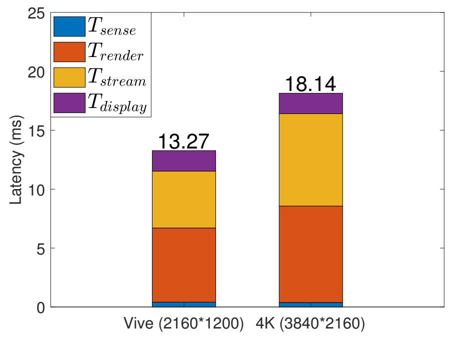
To expand our system towards future VR systems, we further measure whether our system can deliver 4K resolution (3840x2160) VR frames with less than 20ms end-to-end latency. To do it, we conduct experiments on the 4 VR scenes with Vive (2160x1200) and 4K resolutions. We force the rendering engine to render 4K frames with the same graphics quality settings as the Vive case. To achieve visually lossless encoding of 4K, we enlarge the maximum encoded frame size to 500KB. With the 4K resolution, both T_{render} and T_{stream} are larger than those of Vive resolution in all four VR scenes. The average T_{stream} to stream a 4K resolution frame is 8.3ms. The average T_{sense} and $T_{display}$ are 0.4ms and 1.7ms, respectively, not affected by the resolution change. For Corridor and Roller Coaster, our system can still keep the T_{e2e} within 20ms, but the T_{e2e} of Viking Village and Nature exceed 20ms. Figure 3.14(a) and 3.14(b) show the latency breakdown of the 2 scenes. The rendering latency T_{render} is the bottleneck in 4K. This is because the GPU we used is not 4K VR ready, and thus both scenes require more than 11ms to render a 4K frame, which cannot meet the frame rate of 90Hz. While this issue may be solved by using a 4K VR ready GPU, we want to study whether our system has the capability to deliver 4K frames with 90Hz if such a GPU is available. Therefore, we reduce the rendering quality of Viking Village and Nature to squeeze the rendering latency to less than 11ms. As shown in Figure 3.14(c) and 3.14(d), by doing so, our system is able to meet the requirement of 20ms end-to-end latency. With average $T_{sense} = 0.4ms$, $T_{display} = 1.7ms$, and $T_{stream} = 8.3ms$, our system is able to support 4K resolution if the rendering time is within 9.6ms.



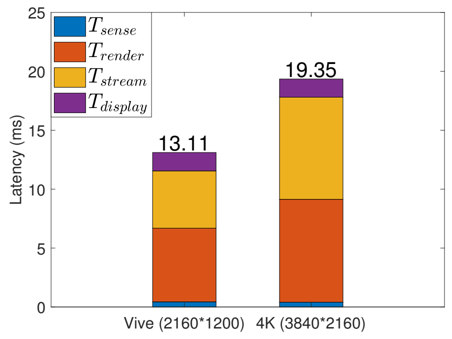
(a) Viking Village (High Quality)



(b) Nature (High Quality)



(c) Viking Village (Low Quality)



(d) Nature (Low Quality)

Figure 3.14: Latency breakdown in 4K resolution.

3.8 Related Work

Cutting the Cord. Cutting the cord of high-quality VR systems has attracted strong interest from both industry and academia. TPCAST [21] and Display Link XR [22] provide a wireless adapter for HTC Vive. They both take a direct "cable-replacement" solution that compresses the display data, transmits over wireless and decompresses on HMD. To our knowledge, they compress whole frames after the frames are fully rendered, and thus it is hard to explore the opportunities to pipeline rendering/streaming and fine-tune rendering/VSync timing. Moreover, their solutions are implemented in ASICs instead of commodity devices. Therefore, they are hardly used by the research community to explore advanced system optimization. Zhong *et al.* [23] explored how to cut the cord using commodity devices and measured the performance of different compression solutions on CPUs and GPUs. Their measurement results provide valuable guidance and are motivating to us.

High-Quality VR on Mobile. Mobile devices are another popular VR platform. Google Daydream [8], Google Cardboard [34] and Samsung Gear VR [7] are examples of this type. As we have discussed, it is very difficult to achieve high-quality VR on mobile due to its limited computing power. Some research works have tried to attack this problem. For example, Flashback [3] performs expensive rendering in advance and caches rendered frames in mobile devices. Doing so provides high-quality VR experience for scenarios that can be pre-computed. Furion [4] enables more scenarios by offloading costly background rendering to a server and only performs lightweight foreground rendering on a mobile device. Such collaborative rendering reduces overall rendering time, which is complementary to our design and can be incorporated into our system to further reduce the latency. Similarly, mobile offloading techniques, e.g., [35, 36, 37, 12, 38, 39] also could be borrowed.

Wireless Performance. Performance of wireless link is critical to wireless VR experience. There are a lot of ongoing research on this issue, especially on mobility and blockage handling for 60GHz/mmWave. For example, MoVR [17] specially designs a mmWave communication system for wireless VR. Agile-Link [18] provides fast beam tracking mechanism. MUST [20] redirects to Wi-Fi immediately upon blockage. Pia [19] switches to different access point proactively. These studies are complementary to our system.

Video Streaming. Our work is also related to video streaming techniques. Nvidia's video codec [40] and Intel's Quick Sync [41] provide fully hardware-accelerated video decoding/encoding capability. Most of these techniques enable *slice-mode* video streaming, which cuts the whole frame into pieces and streams separately. We borrow this idea and combine it into the rendering pipeline to enable parallel rendering and streaming using multiple hardware encoders. 360-degree video streaming [42, 43, 44, 45, 46, 47] pre-caches the panoramic view and allows users to freely control their viewing direction during video playback. Multi-view video coding [48, 49, 50, 51] enables efficient images encoding from multi-viewpoint using both the temporal and spatial reference frames. Video streaming is not extremely latency sensitive as interactive VR unless the streaming is real-time (broadcasting). We may use some techniques in video streaming particularly 360-degree streaming to our system.

3.9 Discussion

In this section, we discuss the following three issues: (1) advantages and the generality of our system, (2) power consumption, and (3) link outage limitations that were out of the scope of this project.

Generality. Our system is a software solution that can be extended to different hardware and operating systems. The server-side rendering module is developed using Unity, which is known to be compatible with D3D and OpenGL. The encoding module on the server side and the decoding module on the client side can be implemented using various hardware codec APIs, such as Nvidia Video Codec [40], Intel Quick Sync [41], Android MediaCodec [52], etc. Compared to previous approaches that still require additional rendering operations on the client side [4, 12, 3, 35], our solution only requires a hardware video codec on the client side, allowing it to work with very thin clients. Our system optimizations of the entire VR offloading pipeline can also be combined with previous techniques (e.g. pre-rendering background scenes [4], robust 60 GHz network [17]) to further improve the performance of the VR offloading task.

Power Consumption. To truly allow cutting the cord of an existing high-quality VR system, it is also important to consider the power consumption on

System component	VR headset	H.264 decoder	WiGig
Power (W)	5.9	4.8	2.3

Table 3.3: Power consumption of three main components in the system.

the client side. Since we use the hardware codec from a laptop in our prototype, it is hard to decouple the codec power consumption from the overall laptop consumption. We therefore estimate power consumption based on reported power consumption data from the three main components (VR headset, H.264 hardware decoder, and WiGig wireless adapter) in our implementation by referring to previous measurement results [53, 54, 55]. The power consumption of an HTC VIVE when running normally is 5.9W [53], and the 802.11ad's power consumption is around 2.3W [54]. We also estimate the power draw of the H.264 decoder in our 4-way parallel decoding scenario based on a prior measurement result of an H.264 video decoder [55]. As shown in Table 3.3, the total power consumption estimate for these key components is 12W, which shows that such a system could be powered from a smartphone-sized Lithium-ion battery for about 3 hours. Note that this power consumption is estimated in a conservative way. The real consumption may be much lower with customized hardware design.

Limitations. This project has not addressed the link outage problem of 60GHz networks, which will likely require orthogonal solutions. To effectively evaluate the performance gain of our solution without the measurements being affected by random movement and link outages, we kept the 60GHz antennas stationary in our experiment to maintain a stable wireless link between the server and the client. It is well known that mobility is still an issue in existing 60GHz wireless products and there is active on-going research to address this [17, 18, 20]. We expect that this project provides a platform that enables such research with realistic end-to-end applications and that ultimately this research will lead to solutions that can be combined with the techniques presented here. We further note that our system can also operate over a Wi-Fi network, which is less susceptible to obstructions, albeit with sacrificing image resolution.

3.10 Conclusion

In this paper, we design an untethered VR system that is able to achieve both low-latency and high-quality requirements over a wireless link. The system employs a Parallel Rendering and Streaming mechanism to reduce the add-on streaming latency, by pipelining the rendering, encoding, transmission and decoding procedures. We also identify the impact of VSync signals on display latency, and introduce a Remote VSync Driven Rendering technique to minimize the display latency. Furthermore, we implement an end-to-end remote rendering platform on commodity hardware over a 60GHz wireless network. The result shows that the system can support 4K resolution frames within an end-to-end latency of 20ms. We plan to release the system as an open platform to facilitate VR research, such as advanced rendering technologies and fast beam alignment algorithms for 60Ghz wireless communication.

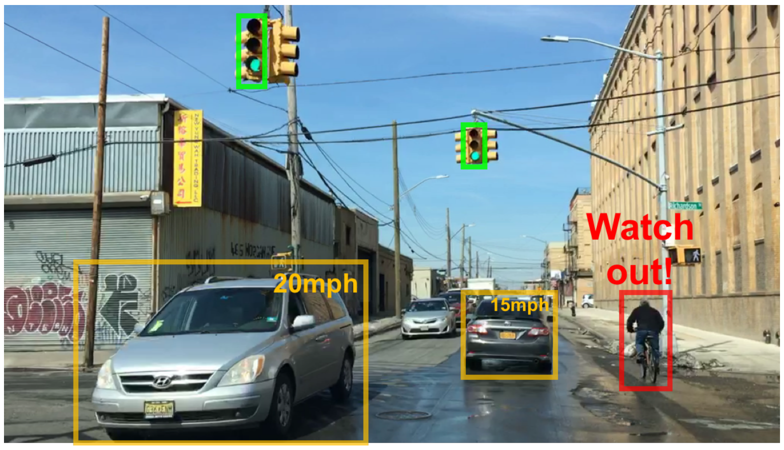
Chapter 4

Edge Assisted Real-time Object Detection for Mobile Augmented Reality

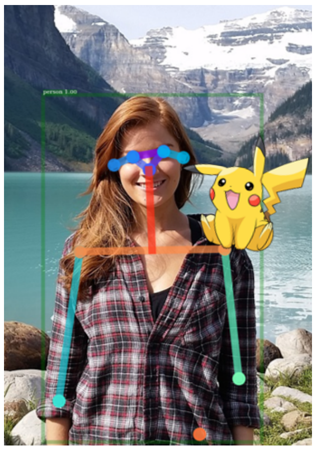
4.1 Introduction

Augmented Reality, and in particular Mixed Reality systems, promise to provide unprecedented immersive experiences in the fields of entertainment, education, and healthcare. These systems enhance the real world by rendering virtual overlays on the user's field of view based on their understanding of the surroundings through the camera. Existing mobile AR solutions such as ARKit and ARCore enable surface detection and object pinning on smartphones, while more sophisticated AR headsets such as Microsoft HoloLens [9] and the announced Magic Leap One [10] are able to understand the 3D geometry of the surroundings and render virtual overlays at 60fps. These AR headsets further promise to support an unprecedented immersive experience called Mix Reality. Compared to tradition AR system, MR requires the system to have a comprehensive understanding of different objects and instances in the real world, as well as more computation resources for rendering high quality elements. Reports forecast that 99 million AR/VR devices will be shipped in 2021 [1], and that the market will reach 108 billion dollars [2] by then.

Most existing AR systems can detect surfaces but lack the ability to detect and classify complex objects in the real world, which is essential for many new AR and MR applications. As illustrated in Figure 4.1, detecting surrounding vehicles or pedestrians can help warn a driver when facing potentially dangerous



(a) Dangerous traffic warning.



(b) Pickachu sits on her shoulder.

Figure 4.1: New AR application supported by object detection algorithms.

situations. Detecting human body key points and facial landmarks allow render virtual overlays on the human body, such as a virtual mask on the face or a Pikachu sitting on the shoulder. Such capabilities could be enabled with CNN, who have shown superior performance in the object detection task [56], but it remains difficult to execute large networks on mobile devices, for example for heat dissipation and power reasons.

Offloading object detection to the edge or cloud is also very challenging due to the stringent requirements on high detection accuracy and low end-to-end latency. High quality AR devices require the system to not only successfully classify the object, but also localize the object with high accuracy. Even detection latencies of less than 100ms can therefore significantly reduce the detection accuracy due to changes in the user's view—the frame locations where the object was originally detected may no longer match the current location of the object. In addition, as mixed reality graphics approach the complexity of VR, one can also expect them to require less than 20ms motion-to-photon latency, which has been found to be necessary to avoid causing user motion sickness in VR applications [3]. Furthermore, compared to traditional AR that only renders simple annotations, mixed reality requires rendering virtual elements in much higher quality, which leaves less latency budget for the object detection task.

Most existing research has focused on enabling high frame rate object detection on mobile devices but does not consider these end-to-end latency requirements for high quality AR and mixed reality systems. Glimpse [15] achieves 30fps object detection on a smartphone by offload trigger frames to the cloud server, and tracks the detected bounding boxes on remaining frames locally on the mobile devices. DeepDecision [16] designs a framework to decide whether to offload the object detection task to the edge cloud or do local inference based on current network conditions. However, they all require more than 400ms offloading latency and also require large amounts of local computation, which leaves little resources to render high-quality virtual overlays. No prior work, can achieve high detection accuracy in moving scenarios or finish the entire detection and rendering pipeline under 20ms.

To achieve this, we propose a system that significantly reduces the offloading detection latency and hides the remaining latency with an on-device fast object tracking method. To reduce offloading latency, it employs a *Dynamic RoI Encoding* technique and a *Parallel Streaming and Inference* technique. The *Dynamic RoI Encoding* technique adjusts the encoding quality on each frame to reduce the transmission latency based on the Regions of Interest (RoIs) detected in the last offloaded frame. It provides higher quality encodings in areas where objects are likely to be detected and uses stronger compression in other areas to save bandwidth and thereby reduce latency. The *Parallel Streaming and Inference* method pipelines the streaming and inference processes to further reduce the offloading latency. On the AR device, the system decouples the rendering pipeline from the offloading pipeline instead of waiting for the detection result from the edge cloud for every frame. To allow this, it uses a fast and lightweight object tracking method based on the motion vector extracted from the encoded video frames and the cached object detection results from prior frames processed in the edge cloud to adjust the bounding boxes or key points on the current frame in the presence of motion. Taking advantage of the low offloading latency, we find this method can provide accurate object detection results and leave enough time and computation resources for the AR device to render high-quality virtual overlays. Besides, we also introduce an *Adaptive Offloading* technique to reduce the bandwidth and power consumption of our system by deciding whether to offload each frame to the edge cloud to process based on the changes of this frame compare to the previous offloaded frame.

Our system is able to achieve high accuracy object detection for existing AR/MR system running at 60fps for both the object detection and human keypoint detection task. We implement the end-to-end AR platform on commodity devices to evaluate our system. The results show that the system increases the detection accuracy by 20.2%-34.8%, and reduce the false detection rate by 27.0%-38.2% for the object detection and human keypoint detection tasks. And the system requires only 2.24ms latency and less than 15% resources on the AR device, which leaves the remaining time between frames to render high quality virtual elements for high quality AR/MR experience.

The contributions of this work can be summarized as follows:

- Quantifying accuracy and latency requirements in an end-to-end AR system with the object detection task offloaded.
- Proposing a framework with individual rendering and offloading pipelines.
- Designing a *Dynamic RoI Encoding* technique to reduce the transmission latency and bandwidth consumption in the offloading pipeline.
- Developing a *Parallel Streaming and Inference* method to reduce the offloading latency.

- Building a *Motion Vector Based Object Tracking* technique to achieve fast and lightweight object tracking on the AR devices.
- Implementing and evaluating an end-to-end system based on commodity hardware.
- Showing that the proposed system can achieve 60fps AR experience with accurate object detection.

4.2 Challenge Analysis

Offering sophisticated object detection in mobile augmented reality devices is challenging because the task is too computationally intensive to be executed on-device and too bandwidth intensive to be easily offloaded to the edge or cloud. Most lightweight object detection models require more than 500ms processing time on current high-end smartphones with GPU support. Even on a powerful mobile GPU SoCs (such as the Nvidia Tegra TX2 which is reportedly used on the Magic Leap One), object detection on an HD frame still takes more than 50ms. This is too long to process every frame on a 60Hz system and likely to lead to energy consumption and heat dissipation issues on the mobile device.

Latency Analysis. When offloading the detection tasks to more powerful edge or cloud platforms the image encoding and transfer add significant latency. Longer latency not only reduces the detection accuracy but also degrades the AR experience. To better understand these challenges, we model the end-to-end latency of a baseline AR solution with offloading as follows:

$$\begin{aligned} t_{e2e} &= t_{offload} + t_{render} \\ t_{offload} &= t_{stream} + t_{infer} + t_{trans\_back} \\ t_{stream} &= t_{encode} + t_{trans} + t_{decode} \end{aligned} \tag{4.1}$$

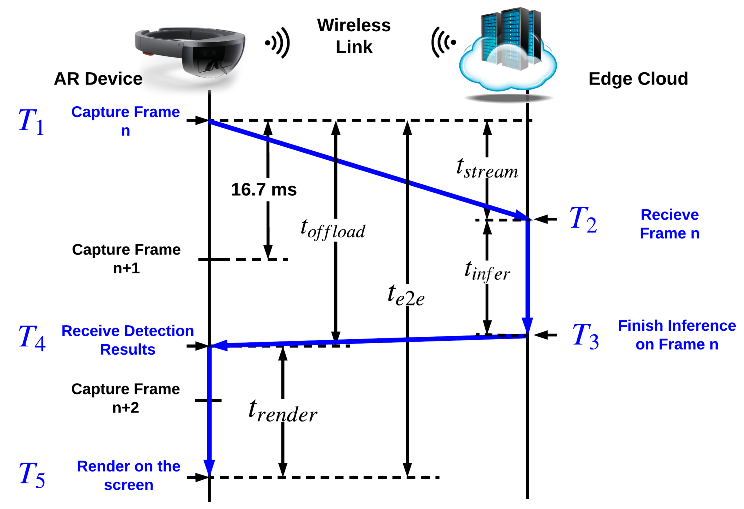
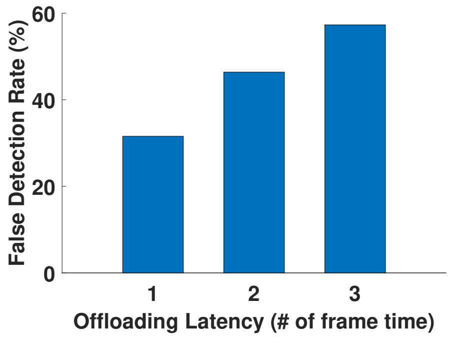


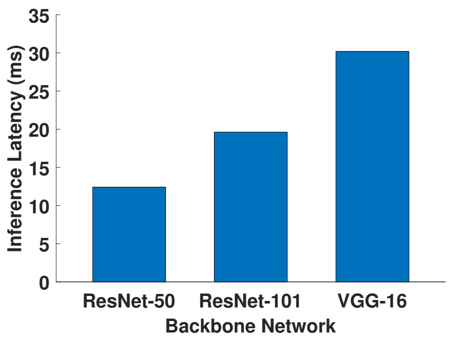
Figure 4.2: Latency Analysis.

As shown in Figure 4.2, the AR device (i.e. smartphone or AR headset) is assumed to be connected to an edge cloud through a wireless connection (i.e. WiFi or LTE). The blue arrow illustrates the critical path for a single frame. Let t_{e2e} be the end-to-end latency, which includes the offloading latency $t_{offload}$ and the rendering latency t_{render}. $t_{offload}$ is determined by three main components: (1) the time to stream a frame captured by the camera from the AR device to the edge cloud $t_{stream} = T_1 - T_1$, (2) the time to execute the object detection inference on the frame at the edge cloud $t_{infer} = T_3 - T_2$, and (3) the time to transmit the detection results back to the AR device $t_{trans\_back} = T_4 - T_3$. To reduce the bandwidth consumption and streaming latency t_{stream}, the raw frames are compressed to H.264 video streams on the device and decoded in the edge cloud [57]. Therefore, t_{stream} itself consists of encoding latency (t_{encode}), transmission latency (t_{trans}) and decoding latency (t_{decode}).

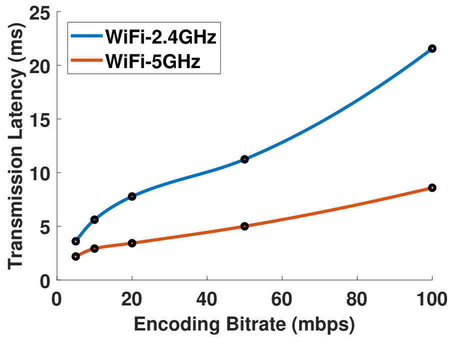
We conduct an experiment to measure the latency and its impact on detection accuracy in the entire pipeline, and find that it is extremely challenging for existing AR system to achieve high object detection accuracy in 60fps display systems. In the experiment, we connect a Nvidia Jetson TX2 to an edge cloud



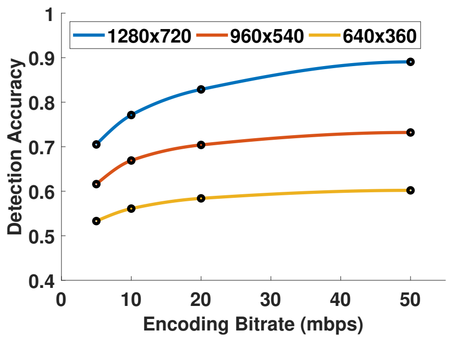
(a) False detection rate with respect to different offloading latency.



(b) Inference latency of Faster R-CNN for different backbone CNN networks.



(c) Transmission latency of different encoding bit rate in two different WiFi networks.



(d) Encoding Bitrate vs Inference Accuracy for two different resolutions.

Figure 4.3: Latency and accuracy analysis.

through two different WiFi protocols (WiFi-5GHz, WiFi-2.4GHz) and stream encoded frames of a video [58] at 1280x720 resolution from the Jetson to the edge cloud for inference. The edge cloud is a powerful PC equipped with a Nvidia Titan Xp GPU.

Detection Accuracy Metrics. To evaluate the detection accuracy in terms of both object classification and localization, we calculate the IoU of each detected bounding box and its ground truth as the accuracy of this detection. We also define the percentage of detected bounding boxes with less than 0.75 IoU [59] (the strict detection metric used in the object detection task) as false detection rate. Similarly, we use the Object Keypoint Similarity (OKS) [60] metric to measure the accuracy of each group of keypoints in the human keypoint detection task.

We find that low latency object detection is highly beneficial for achieving a high detection accuracy. Figure 4.3(a) shows the impact of $t_{offload}$ on the false detection rate. We can find that even a latency of a frame time (16.7ms) will increase the false detection rate from 0% to 31.56%. This is because during the time that the detection result is sent back to the AR device, the user's view may have changed due to user motion or scene motion.

However, it is very challenging to achieve very low latency object detection with commodity infrastructures. We first measure the latency spend on inference (t_{infer}), and show the result in Figure 4.3(b). To push the limit of t_{infer} on the edge cloud, we use TensorRT [61] to optimize three pretrained Faster R-CNN models[1] using INT8 precision. These three models use three different backbone CNN networks (ResNet-50, ResNet-101, and VGG-16) for feature extraction. As shown in Figure 4.3(b), we can observe that all three models require more than 10ms for object detection.

Figure 4.3(c) shows the additional latency imposed by transmitting a single HD frame with different encoding bitrate from the AR device to the edge cloud (t_{trans}) through two different WiFi connections (WiFi-2.4GHz and WiFi-5GHz). Here, bitrate is a codec parameter that determines the quality of video encoding. Encoding with small bitrate will result in a lossy frame after decoded. We can observe that the average t_{trans} requires to transmit an encoded frame with 50mbps bitrate is 5.0ms on 5GHz WiFi and 11.2ms on 2.4GHz WiFi. Inference plus transmission latency therefore already exceeds the display time for one frame. One may think that decreasing resolution or encoding bitrate may reduce the transmission latency, however, this also reduces the detection accuracy of an object detection model.

To validate this issue, we show the detection accuracy of the ResNet-50 based

[1]We choose Faster R-CNN because it is much more accurate than other alternatives, such as SSD and R-FCN.

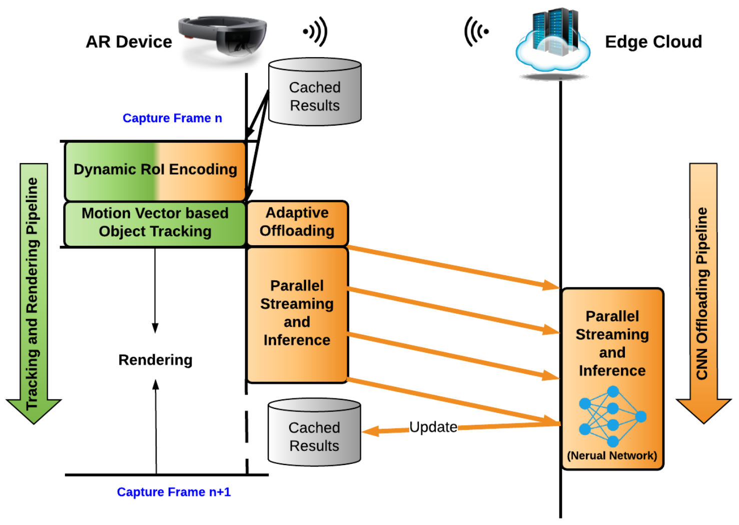


Figure 4.4: System Architecture.

Faster R-CNN model under different encoding bitrate and resolution in Figure 4.3(d). In this case, we use the detection result on raw video frames (without video compression) as the ground truth to calculate the IoU. The result shows that it requires at least 50Mbps encoding bitrate to achieve a high detection accuracy (i.e. 90). We also compare the detection result on two lower resolution frames (960x540 and 640x320), and show that lower resolution has much worse detection accuracy than the original 1280x720 frame. Lowering resolution therefore also does not improve detection accuracy. Note that this accuracy drop can be stacked together with the drop caused by the long offloading latency to get a much lower detection accuracy.

Based on the above analysis, we find that it is extremely challenging for existing AR system to achieve high object detection accuracy in 60fps display systems. This can lead to poor alignment of complex rendered objects with physical objects or persons in the scene.

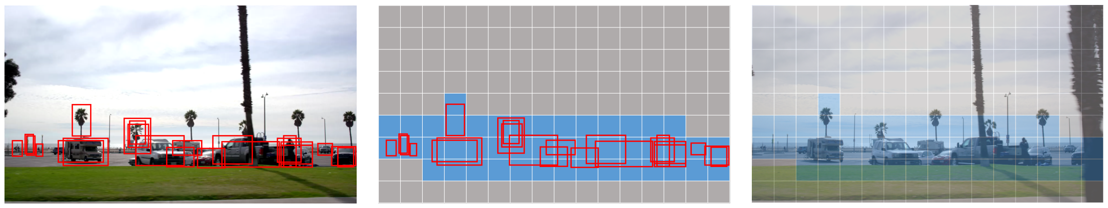

(a) Detect RoIs on the last offloaded frame. (b) Mark macroblocks that overlap with RoIs. (c) Change encoding quality on the current frame.

Figure 4.5: Three main procedures of RoI encoding.

4.3 System Architecture

To overcome these limitations, we propose a system that is able to achieve high accuracy object detection with little overhead on the rendering pipeline of mobile augmented reality platforms, by reducing the detection latency with low latency offloading techniques and hiding the remaining latency with an on-device fast object tracking method. Figure 4.4 shows the architecture of our proposed system. At a high level, the system has two parts connected through a wireless link: a local tracking and rendering system on a mobile device (a smartphone or an AR headset) and a pipelined objected detection system on the edge cloud. To hide the latency caused by offloading the object detection task, our system decouples the rendering process and the CNN offloading process into two separate pipelines. The local rendering pipeline starts to track the scene and render virtual overlays while waiting for object detection results, and then incorporates the detection results into the tracking for the next frame when they arrive.

As shown in Figure 4.4, both pipelines start with a *Dynamic RoI Encoding* technique that not only compresses raw frames for the CNN offloading pipeline (yellow arrow), but also provides its meta data for the on-device tracking module in the tracking and rendering pipeline (green arrow). *Dynamic RoI Encoding* is an efficient video encoding mechanism that is able to largely reduce the bandwidth consumption and thereby reduce the transmission latency to the edge cloud, while maintaining detection accuracy. The key idea of *Dynamic RoI Encoding (DRE)* is to decrease the encoding quality of uninteresting areas in a frame and to maintain high quality for candidate areas that may contain objects of interest based on earlier object detection results. Due to the spatiotemporal correlation over subsequent video frames, the system uses the intermediate inference output of the last offloaded frame as candidate areas. These candidate areas are where it maintains high encoding quality and are also referred to as regions of interest (RoIs).

In the CNN offloading pipeline as illustrated by the yellow blocks and arrow, we propose an *Adaptive Offloading* and a *Parallel Streaming and Inference (PSI)* technique to further reduce the latency and bandwidth consumption of the offloading task.

Adaptive Offloading is able to reduce the bandwidth and power consumption of our system by deciding whether to offload each frame to the edge cloud based on whether there are significant changes compared to the previous offloaded frame. For efficiency, this technique reuses the macroblock type (inter-predicted blocks or intra-predicted blocks) embedded in the encoded video frame from the *Dynamic RoI Encoding* to identify significant changes that warrant offloading for object detection.

Once the frame is marked for offloading, the *Parallel Streaming and Inference (PSI)* method parallelizes the transmission, decoding and inference tasks to further reduce the offloading latency. It splits a frame into slices and starts the convolutional neural network object detection task as soon as a slice is received, rather than waiting for the entire frame. This means that reception, decoding, and object detection can proceed in parallel. To solve the dependency issues across slices during object detection, we introduce a *Dependency Aware Inference* mechanism that determines the region on each feature map that has enough input features to calculate after each slice is received, and only calculate features lie in this region. The detection results are sent back to the AR device and cached for future use.

In the tracking and rendering pipeline (green blocks and arrow in Figure 4.4), instead of waiting for the next detection result, we use a fast and light-weight *Motion Vector based Object Tracking (MvOT)* technique to adjust the prior cached detection results with viewer or scene motion. Compared to traditional object tracking approaches that match image feature points (i.e. SIFT and Optical Flow) on two frames, the proposed technique again reuses motion vectors embedded in the encoded video frames, which allows object tracking without any extra processing overhead. Given the aforementioned optimizations to reduce offloading latency, tracking is needed only for shorter time frames and a lightweight method can provide sufficiently accurate results. Using such a lightweight method leaves enough time and computational resources for rendering on the device, in particular to render high-quality virtual overlays within the 16.7ms (for 60Hz screen refresh rate) latency requirement.

4.4 Dynamic RoI Encoding

Dynamic RoI Encoding reduces the transmission latency of the offloading pipeline while maintaining a high object detection accuracy. Transmitting the frames with high visual quality from the mobile to the edge/cloud leads to a high bandwidth consumption and thereby transmission latency. *Dynamic RoI Encoding* selectively applies higher degrees of compression to parts of the frame that are less likely to contain objects of interest and maintains high quality in regions with candidate objects. This largely reduces the size of encoded frames with only a small tradeoff in object detection accuracy. The key lies in identifying the regions with potential objects of interest, which we will refer to as regions of interest. The design exploits candidate regions that have been generated internally by the convolutional neural network on prior frames.

4.4.1 Preliminaries

RoI Encoding. While the building block of RoI encoding has been used in other applications, current methods to select regions of interest are not suitable for this augmented reality object detection task. RoI encoding is already supported by most video encoding platform, which allows the user to adjust the encoding quality (i.e. Quantization Parameter - QP) for each macroblock in a frame. It has been largely adopted in surveillance camera video streaming and 360-degree video streaming, where the RoIs are pre-defined or much easier to predict based on user's field of view. For example, the RoI can be derived as the area that a user chooses to look at. This region would then receive near-lossless compression to maintain quality while lossier compression is used for the background or non-RoI area. Augmented reality includes use cases that should draw users attention to other areas of the view and therefore regions of interest cannot just be based on the current objects a user focuses on.

Object Detection CNNs. Due to impressive performance gains of state-of-the-art object detection is largely based on CNN. While several networks exist (e.g., Faster-RCNN, Mask-RCNN), they share a similar architecture, which firstly utilizes a CNN network to extract the features of the input image, then internally propose candidate regions (also called regions of interest) and their corresponding possibilities through a region proposal network, and finally perform and refine the object classification. The CNN network is also called backbone network and there are multiple options for its implementation, including VGG, ResNet, and Inception. The region proposal network usually generates hundreds of regions of interest which are potential objects locations in the frame.

Note that the term **RoIs** is used both in object detection and video compression. For the object detection task, RoIs are usually the output proposals of the region proposal network. While in the field of video compression, RoIs are the areas inside video frames that may contain more visual information and will be encoded with fewer losses. This presents an opportunity to exploit this similarity and tie these concepts together.

4.4.2 Design

In order to reduce the bandwidth consumption and data transmission delay, we design a dynamic RoI encoding mechanism that links internal RoI generated in the object detection CNNs to the image encoder. Specifically, it uses the CNN candidate RoIs generated on the last processed frame for determining encoding quality on the next camera frame. It accommodates a degree of motion by slightly enlarging each region of interest by one macroblock but largely benefits from the similarity between two frames captured a short moment apart in time. While one may expect that even greater bandwidth savings are possible by choosing RoIs only in areas where object were detected on the previous frame, this approach frequently misses new objects that appear in the scene because the image areas containing these objects end up too heavily compressed. Changes in such a heavily compressed area, however, are often still identified as part of the much larger set of candidate RoIs of the CNN, the outputs of the region proposal network. We therefore use the RoIs from the region proposal network, filtered with a low minimum prediction confidence threshold (i.e., 0.02). A sample output of our RoI detection method is shown in Figure 4.5(a).

In order to use these selected RoIs to adjust the encoding quality on the current frame, we calculate a QP map that defines the encoding quality (QP) for each macroblock on the frame. The QP map indicates for each macroblock whether it overlaps with any RoI. In the example in Figure 4.5(b), all overlapping macroblocks are marked in blue and non-overlapping ones in grey. Since object detection is offloaded to the edge, cloud the object detection pipeline sends this QP map back to the AR device, which uses it for the next captured frame.

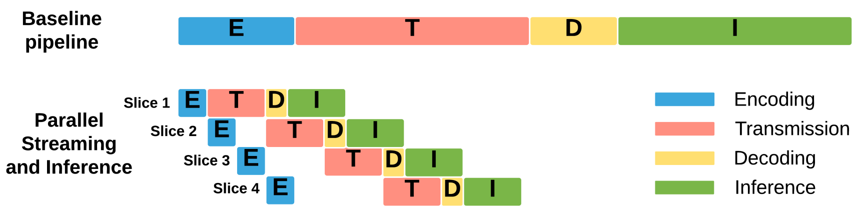


Figure 4.6: Parallel Streaming and Inference.

As shown in Figure 4.5(c), the encoder applies lossy compression on those non-overlapping (grey) regions, while maintaining high visual quality on overlapping (blue) regions.[2] Specifically, our implementation reduces the QP value by 5 for lossy encoding.

4.5 Parallel Streaming and Inference

We offload the heavy deep neural network computation to the edge cloud. This requires transmitting the camera frames from the mobile side to the edge cloud. Conventional architectures, however, can only start the object detection process when the entire frame is received, as the deep neural networks are designed with neighborhood dependency. This will add to the latency, since both the streaming and the inference process take considerable time and run sequentially, as discussed in section 4.2. To mitigate this long latency, we propose a *Parallel Streaming and Inference* technique which enables inferences on slices of a frame, so that the streaming and inference can be effectively pipelined and execute in parallel. This technique can significantly reduce the latency since streaming and inference consume different resources that do not affect each other: transmission consumes bandwidth on the wireless link, decoding uses edge cloud hardware decoders, and the neural network inference mainly consumes GPUs or FPGAs resources on the

[2]Note that Figure 4.5(b) and 4.5(c) uses a grid of 16x9 macroblocks for illustration purposes. In the H.264 standard, a macroblock is usually 16x16 pixels, so a 1280x720 resolution frame has 80x45 macroblocks.

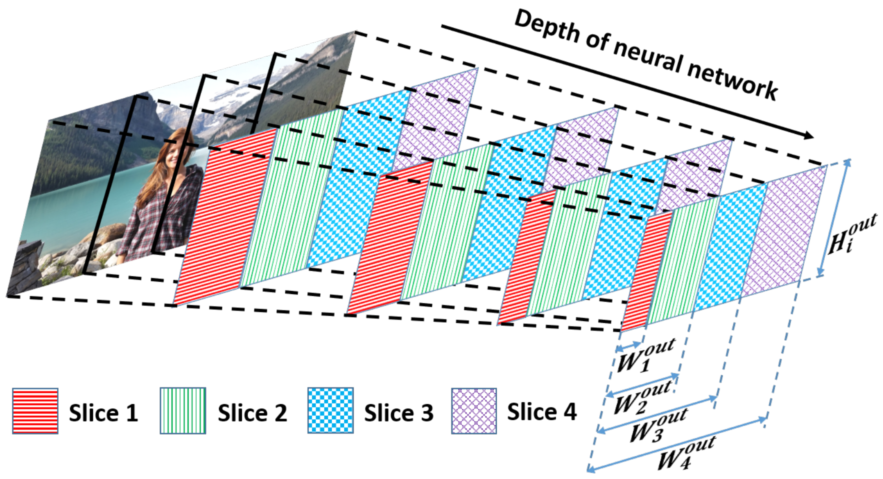


Figure 4.7: Dependency Aware Inference.

edge cloud.

The challenge for deep neural networks to execute on a slice of frame is the dependency among inputs, which is caused by the neuron operations that take neighborhood values as input. To address this problem, we propose **Dependency Aware Inference** to automatically analyze the dependencies of each layer, and only infer on the regions which have enough neighbor values. Figure 4.6 shows how the Parallel Streaming and Inference method reduces the offloading latency. Compared with encoding and inference on the entire frame, we encode the whole image into multiple slices, each slice will be sent to the edge cloud immediately after it is encoded. The edge cloud will start to infer once it receives and decodes the first slice of the frame.

4.5.1 Dependency Aware Inference

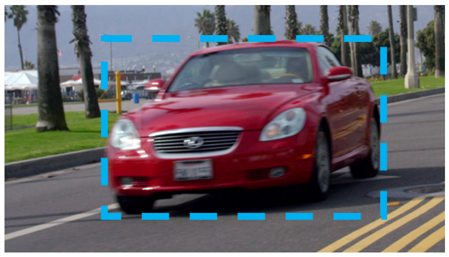
Due to the computational dependency among neighbor values of the input frame, simply running inference and then merging based on slices of a frame will cause significant wrong feature values near boundaries. To solve this problem, we design a *Dependency Aware Inference* technique which only calculates the regions of feature points in each layer with enough input feature points available. Dependencies are caused by the convolutional layers (as well as pooling layers sometimes), where the feature computation around the boundary of each frame slice requires also adjacent slices. This effect propagates for the standard convolutional layers and pooling layers structure. We experimentally find that the boundary feature computation of the last convolutional layer on VGG-16, Resnet-50, and Resnet-101, requires 96, 120, 240 pixels respectively. One naive solution for parallelizing inference is to recompute such regions when the next slice arrives at the edge cloud. However, this requires significant extra computations for every convolutional layer, which inflates the inference latency.

To solve this dependency issue, we calculate the size of the *valid region* for the output feature map of each layer, and only infer based on valid regions. Valid regions are defined as the areas of each convolutional feature map that have enough input features available and their sizes can be determined in equation 4.2.

$$\begin{aligned} H_i^{out} &= (H_i^{in} - 1)/S + 1 \\ W_i^{out} &= \begin{cases} \frac{W_i^{in} - (F-1)/2 - 1}{S} + 1, & i = 1, 2, ..., n-1 \\ \frac{W_i^{in} - 1}{S} + 1, & i = n \end{cases} \end{aligned} \tag{4.2}$$

H_i^{out} and W_i^{out} are the height and width of *valid region* of the output feature map of a convolutional layer after slice i arrives at the edge cloud (i is the number of slice, n is the number of slices we divided.). Similarly, H_i^{in} and W_i^{in} are the *valid region* on the input feature map of this convolutional layer. We also define the spatial extent and stride of this conv layer as F and S correspondingly[3]. Note that we empirically set n to 4 in our system to archive a balance between transmission and inference.

[3]Note that we assume the number of zero padding of a conv layer is equal to $(F-1)/2$ in most cases.


(a) Cached detection result of the last offloaded frame.

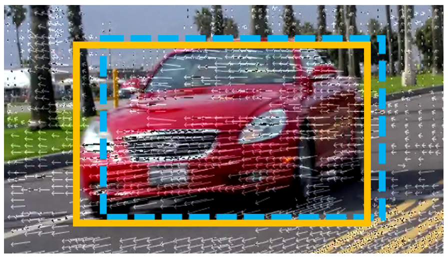

(b) Shift the bounding box based on the motion vectors.

Figure 4.8: Main procedures of RoI encoding.

Figure 4.7 illustrates the concept of the *Dependency Aware Inference* technique. Since our system cuts the whole frame into 4 slices with 1/4 of the original width, H_i^{out} of one conv layer is constant and only affected by H_i^{in} and S as shown in the first equation, while W_i^{out} keeps increasing as more slices arrive at the edge cloud. For example, in the case of a standard 3x3 convolutional layer with stride 1, we will not calculate the very right column of features for slice 1,2 and 3, due to those features requiring inputs from the next slice of the frame. As shown in Figure 4.7, our system only calculates the red regions in each conv layer after slice 1 arrives at the edge cloud. As more slices arrive, the *valid region* keeps increasing on each feature map, and the system continuously calculates those new features included in the *valid region*. We can observe that the number of features that can be calculated for slice 1 keeps decreasing as the network goes deeper. Slice 2 and 3 are able to compute more features than slice 1, and all the remaining features will be calculated after slice 4 arrives. Note that we also defined similar logic to process pooling layers, which will not calculate the rightmost column in the output feature map for slice 1,2 and 3 if the input feature map is an odd number.

4.6 Motion Vectors Based Object Tracking

In this section, we introduce Motion Vector Based Object Tracking that is able to estimate the object detection result of the current frame using the motion vector extracted from the encoded video frames and the cached object detection result from the last offloaded frame.

Motion vectors are broadly used by modern video encoding approaches (e.g. H.264 and H.265) to indicate the offset of pixels among frames to achieve a higher compression rate. Commodity mobile devices are usually equipped with specific hardware to accelerate video encoding and compute the motion vectors. Figure 4.8 shows the key steps of the Motion Vector based Fast Object Tracking technique. For each new frame captured by the camera, the system passes the frame to the *Dynamic RoI Encoding* session. The encoder uses the frame corresponding to the last cached detection result (Figure 4.8(a)) as its reference frame for inter-frame compression. After that, the system extracts all motion vectors from the encoded frame, as illustrated in Figure 4.8(b). To track the object in the current frame, we get the bounding box of this object in the last offloaded frame, calculate the mean of all motion vectors that reside in the bounding box, and use it to shift the old position (in blue) to the current position (in yellow). Similarly, we also apply this technique to the human keypoint detection task, in which we calculate the mean motion vector in the closest 9x9 macroblock region of each keypoint, and use it to shift each keypoint.

In our experiment, we find that the accuracy of the motion vector decreases as the time interval between the current frame and reference frame increases. However, due to the low offloading latency achieved by the proposed latency optimization techniques, we found that this method can provide accurate object detection results with very short latency. The system we implemented on Nvidia Jetson TX2 requires only 2.24ms for this motion tracking process, which leaves enough time and computation resources for the AR device to render high-quality virtual overlays within the 16.7ms latency requirement.

Note that this technique cannot hide the latency to first detection of an object. Since this is already well under the response time that human observers notice, this technique focuses on accurate tracking so that virtual objects can follow the motion of physical ones.

4.7 Adaptive Offloading

To effectively schedule the offloading pipeline, we propose an *Adaptive Offloading* mechanism to determine which encoded frame should be offloaded to the edge cloud. The *Adaptive Offloading* mechanism is designed based on two principles: (1) a frame will only be eligible to be offloaded if the previous offloaded frame has been completely received by the edge cloud, (2) a frame will be considered for offloading if it differs significantly from the last offloaded frame. The first principle eliminates frames queuing up to avoid network congestion, while the second principle ensures that only necessary views with enough changes will be offloaded to minimize communication and computing costs. Therefore, if a frame satisfies both principles, it will be offloaded to the edge cloud.

The first principle requires the system to be aware of the transmission latency of previous offloaded frames. The edge cloud therefore signals the AR device once it receives the last slice of the offloaded frame. Based on this time difference between the reception time and the transmission time, the AR calculates the transmission latency and uses it to decide whether to offload the next encoded frame.

To fulfill the second principle, it is necessary to estimate the differences between two frames. We evaluate such differences from two perspectives with either of them satisfying the second principle: (1) whether large motions (including both user's motion and objects' motion) occur among the frames, (2) whether there are considerable amounts of changed pixels appearing in the frame. The motion of a frame is quantified by the sum of all the motion vectors, and the number of new pixels is estimated by the number of intra-predicted macroblocks within an encoded frame. Between the two types of macroblocks (inter-predicted block and intra-predicted block) within an encoded frame, we experimentally find that intra-predicted macroblocks usually refer to newly appeared regions, since these macroblocks fail to find reference pixel blocks in the reference frame during encoding.

4.8 Implementation

Our implementation is entirely based on commodity hardware and consists of around 4000 lines of code.

4.8.1 Hardware Setup

In the hardware setup, we use a mobile development board Nvidia Jetson TX2 as the AR device, which contains the same mobile SoC (Tegra TX2) as the Magic Leap One AR glass. The Jetson board is connected to a TP-Link AC1900 router through a WiFi connection. We emulate an edge cloud with a PC equipped with an Intel i7-6850K CPU and a Nvidia Titan XP GPU, which connects to router through a 1Gbps Ethernet cable. Both the AR device and the desktop PC run an Ubuntu 16.04 OS.

4.8.2 Software Implementation

We implement our proposed techniques based on Nvidia JetPack[62], Nvidia Multimedia API [63], Nvidia TensorRT [61], and the Nvidia Video Codec SDK [40].

Client Side. We implement the client side functions on the Nvidia Jetson TX2 with its JetPack SDK. The implementation follows the design flow in Figure 4.4. We first create a camera capture session running at 60fps using the JetPack Camera API, and register a video encoder as its frame consumer using the Multimedia API. To realize the RoI encoding module, we use the setROIParams()

function to set the RoIs and their QP delta value for encoding the next frame, based on the RoIs generated on the edge cloud. We also enable the external RPS control mode to set the reference frame of each frame to the source frame of the current cached detection results, so that the extracted Motion Vectors can be used to shift the cached detection results. To implement the Parallel Streaming and Inference module, we enable the slice mode for the video encoder and use the setSliceLength() function with a proper length to let the encoder split a frame into four slices. After frame slices are encoded, the system extracts motion vectors and macroblock types from each slice using the getMetadata() function. This information is used as the input for Adaptive Offloading and MvOT in two different threads (Rendering thread and offloading thread). In the offloading thread, if the Adaptive Offloading module decides to offload this frame, its four slices will be sent out to the server through the wireless link one by one. In the rendering thread, the Motion Vector based Object Tracking module uses the extracted motion vectors and cached detection results to achieve fast object tracking. The system then renders virtual overlays based on the coordinates of the detection result.

Server Side. The server side implementation contains two main modules: Parallel Decoding and Parallel Inference, which are designed to run in two different threads to avoid blocking each other. In the Parallel Decoding thread, the system keeps waiting for the encoded frame slices from the AR device. Once a slice is received, it immediately passes it to the video decoder for decoding in asynchronous mode, which won't block the system to continue receiving other slices. We use Nvidia Video Codec SDK to take advantage of the hardware accelerated video decoder in the Nvidia Titan Xp GPU. After each slice is decoded, the system passes it to the parallel inference thread in a callback function attached to the decoder. The Parallel Inference module is implemented using the Nvidia TensorRT, which is a high-performance deep learning inference optimizer designed for Nvidia GPUs. To push the limit of inference latency on the server side PC, we use the INT8 calibration tool [64] in TensorRT to optimize the object detection model, and achieves 3-4 times latency improvement on the same setup. To achieve the proposed *Dependency Aware Inference* method, we add a PluginLayer before each convolutional layer and pooling layer to adjust their input and output regions based on Equation 4.2. After the inference process of a whole frame, the edge cloud sends the detection results as well as the QP map back to the AR device for future processing.

4.9 Evaluation

In this section, we evaluate the performance of the system in terms of detection accuracy, detection latency, end-to-end tracking and rendering latency, offloading latency, bandwidth consumption, and resource consumption. The results demonstrate that our system is able to achieve both the high accuracy and the low latency requirement for AR headsets and hand-held AR system running at 60fps. The result shows that the system increases the detection accuracy by 20.2%-34.8%, and reduce the false detection rate by 27.0%-38.2% for the object detection and human keypoint detection tasks, respectively. To achieve this high accuracy, the system reduces the offloading latency by 32.4%-50.1% and requires only an average of 2.24ms to run the MvOT method on the AR device, which leaves the remaining time between frames to render high quality virtual elements.

4.9.1 Experiment Setup

We use the setup and implementation described in Section 4.8 to conduct experiments. Two different detection tasks are designed to evaluate the performance of our system: an object detection task and a keypoint detection task. Both of them follow the flow in Figure 4.4. In the first task, the edge cloud runs a Faster

R-CNN object detection model with ResNet-50 to generate bounding boxes of objects for each offloaded frame. In the second task, the edge cloud runs a Keypoint Detection Mask R-CNN model with ResNet-50 to detect the human body keypoints. Based on the detection result, the AR device renders a complex 3D cube on the user's left hand, as shown in Figure 4.10. Both detection tasks run local object tracking and rending at 60fps on the AR device. Compared to the first task, the second task incurs higher rendering loads on the AR device. For repeatable experiments, we extract raw YUV frames at 1280x720 resolution from ten videos[4] in the Xiph video dataset [65] as the camera input for evaluation. In total, 5651 frames have been processed in our evaluation. In addition, we use two different WiFi connections (2.4GHz and 5GHz) as the wireless link between the AR device and the edge cloud. The bandwidths measured with iperf3 are 82.8Mbps and 276Mbps correspondingly.

4.9.2 Object Detection Accuracy

Our system is able to achieve high detection accuracy and low false detection rate under various network conditions. We first measure the object detection accuracy in four approaches: the baseline solution (Baseline), our solution with only the two latency optimization techniques (DRE + PSI), our solution with only the client side motion vector based object tracking method (Baseline + MvOT), and our overall system with all three techniques (DRE + PSI + MvOT). The baseline approach follows the standard pipeline we introduced in Section 2. We evaluate the detection accuracy of our system with two key metrics: mean detection accuracy and false detection rate. Specifically, we feed extracted frames of each video to the client side video encoder at 60fps to emulate a camera but

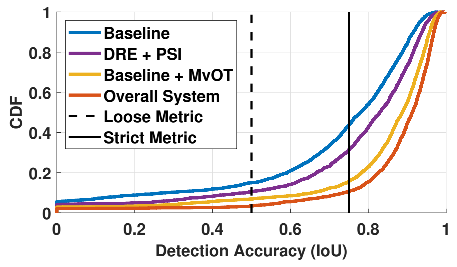
[4]DrivingPOV, RollerCoaster, BarScene, FoodMarket, and SquareAndTimelapse for object detection task. Crosswalk, BoxingPractice, Narrator, FoodMarket, as well as SquareAndTimelapse for the human keypoint detection task.

Detection Model	Approaches	WiFi 2.4GHz	WiFi 5GHz
Faster R-CNN Object Detection	Baseline	0.700	0.758
	DRE + PSI	0.758	0.837
	MvOT only	0.825	0.864
	Overall System	0.864	0.911
Mask R-CNN Keypoint Detection	Baseline	0.6247	0.6964
	DRE + PSI	0.7232	0.7761
	MvOT only	0.7667	0.8146
	Overall System	0.8418	0.8677

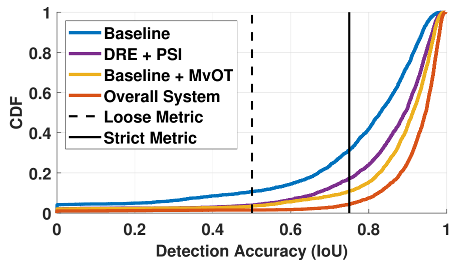
Table 4.1: Mean Detection Accuracy (IoU/OKS) of two models with two WiFi connections.

allow experiments with repeatable motion in the video frames. To calculate the detection accuracy for each frame, we calculate the mean Intersection over Union (IoU) or Object Keypoint Similarity (OKS) between the detection result from the MvOT and the ground truth detection result of each frame (without frame compression and emulating no latency). Recall that IoU is 0 when the detected object labels do not match (e.g., vehicle vs pedestrian) and otherwise represent the degree of position similarity within the frame. More precisely, it is the intersection area over the union area of the detection bounding box and ground truth bounding box. Similar to IoU, OKS also varies from 0 to 1, describing the normalized Euclidean distances between detected positions of keypoints and groundtruth labels. In the experiment, we connect the server and the client devices through two WiFi connections: WiFi-2.4GHz and WiFi-5GHz.

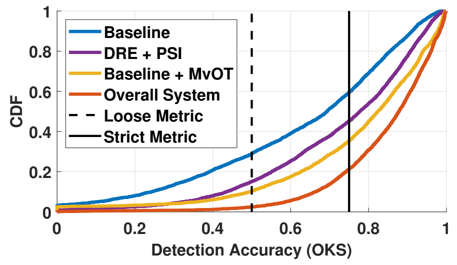
Table 4.1 shows the mean detection accuracy of two models with two different WiFi connections. In the object detection case, we can observe that our system achieves a 23.4% improvement for the WiFi-2.4GHz connection and a 20.2% improvement for the WiFi-5GHz connection. In the human keypoint detection case, our system achieves a 34.8% improvement for WiFi-2.4GHz and a 24.6% improvement for WiFi-5GHz. The results also show that the three main techniques (DRE, PSI, and MvOT) are able to effectively increase the detection accuracy of



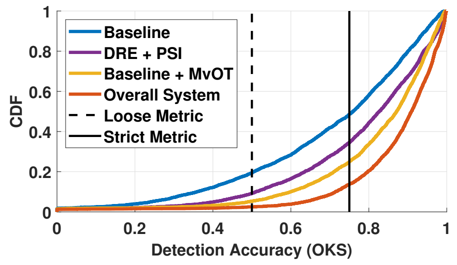
(a) Object Detection - WiFi 2.4GHz



(b) Object Detection - WiFi 5GHz



(c) Keypoint Detection - WiFi 2.4GHz



(d) Keypoint Detection - WiFi 5GHz

Figure 4.9: CDF of detection accuracy (IoU/OKS) for object detection and keypoint detection task.

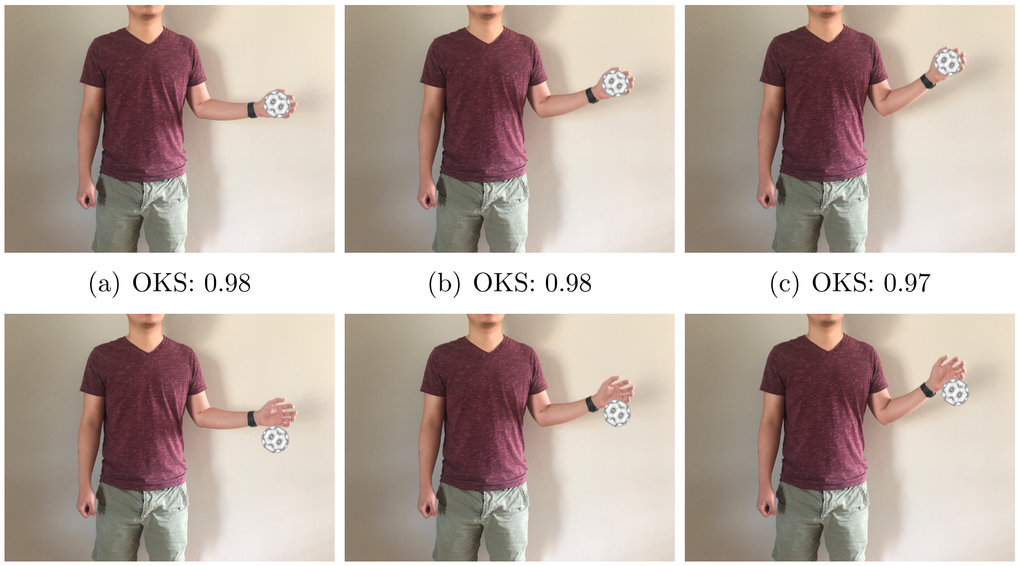
the system. By comparing the DRE + PSI approach with the Baseline approach, we find that the low latency offloading solution helps to achieve high detection accuracy. By comparing the Baseline + MvOT with the Baseline approach, we also see that our fast object tracking technique increases accuracy. The gains of these two approaches accumulate in the overall system accuracy.

In addition, we show the CDF of the measured detection accuracy results in Figure 4.9. To determine acceptable detection accuracy, we adopt two widely used thresholds in the computer vision community: 0.5 as a loose accuracy threshold and 0.75 as the strict accuracy threshold [66]. A detected bounding box or a set of keypoints with a detection accuracy less than the accuracy metric is then considered a false detection. Due to the high quality requirement of AR/MR system, we mainly discuss the false detection rate in terms of the strict accuracy metric, but we also mark the loose metric in each figure with the black dashed line.

Figure 4.9(a) and Figure 4.9(b) show the CDF of IoU for the object detection task. Result shows that our system only has 10.68% false detection rate using WiFi-2.4GHz and 4.34% using WiFi-5GHz, which reduce the false detection rate of the baseline approach by 33.1% and 27.0% correspondingly. Figure 4.9(c) and Figure 4.9(d) show the CDF of OKS for the human keypoint detection task. Compared to object detection task that only tracks the position of each object bounding box, this task requires to track 17 human keypoints of each human using embedded motion vector, which is much more challenging. However, our system can still reduce the false detection rate by 38.2% with WiFi-2.4GHz and 34.9% with WiFi-5GHz.

To understand how the detection accuracy affects the AR experience, we show several frames with their detection accuracy (OKS) from a sample AR the human keypoint detection task in Figure 4.10. In this sequence, the person is moving the left hand while the system seeks to render virtual object in the palm of the hand. The three frames in the first row are the rendering results based on our system, while the bottom three frames are based on the baseline approach. We can observe that the rendered cube is well-positioned in our system but trailing behind the palm due to delayed detection results in the baseline approach.

Impact of background network traffic. Results further show that our system is less affected by the background network load, and accuracy degrades more gracefully even in congested networks. Figure 4.11 shows our measurement results of the false detection rate in WiFi networks with different background traffic loads. In the experiment, we gradually increase the background traffic in the network, and record the corresponding false detection rate with both WiFi-5GHz and WiFi-2.4Hz connections. When raising the traffic load from 0% to 90%, the false detection rate for baseline increases by 49.84% and 35.60% in WiFi-2.4GHz and WiFi-5GHz, respectively. For our system, the increase is only 21.97% and 15.58%, which shows the higher tolerance of our system too network congestion.



(a) OKS: 0.98 (b) OKS: 0.98 (c) OKS: 0.97

(d) OKS: 0.83 (e) OKS: 0.76 (f) OKS: 0.73

Figure 4.10: (a)-(c) Rendering results based on our system. (d)-(f) Rendering results based on the baseline approach.

4.9.3 Object Tracking Latency

Our system only requires 2.24ms to adjust the positions of previously detected objects in a new frame, which leave enough time and computation resources for the AR device to render high-quality virtual overlays with the time between two frames. Figure 4.12 compares our MvOT method with two standard optical flow based object tracking approaches—the Lucas Kanade and Horn Schunck methods. Both methods have been optimized to take advantage of the on-board GPU of Nvidia Jetson TX2. We can observe that our 2.24ms MvOT method is significantly faster than traditional optical flow approaches and requires 75% less GPU resources compared to the Lucas Kanade based optical flow method. While their tracking may be more accurate, the delay would mean missing the frame display time, which leads to lower accuracy because objects can have moved even further in the next frame.

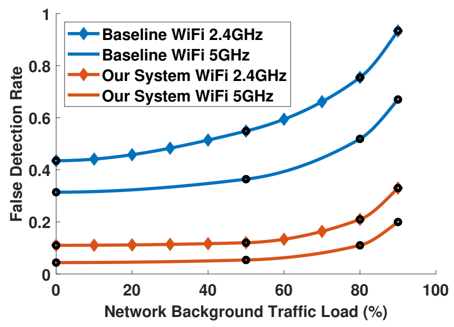


Figure 4.11: False detection rate vs Network background traffic load.

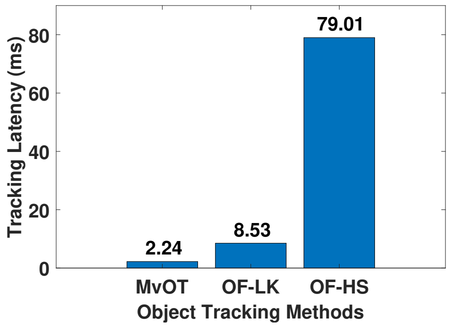


Figure 4.12: Latency of MvOT compare with optical flow tracking methods.

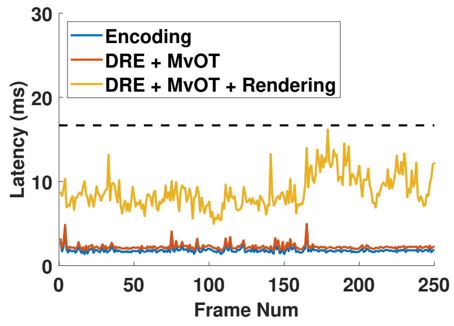


Figure 4.13: Raw latency traces of our system running a keypoint detection task.

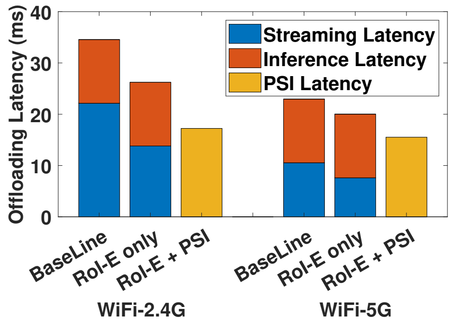


Figure 4.14: Offloading latency of three approaches using WiFi.

4.9.4 End-to-end Tracking and Rendering Latency

Our system is able to achieve an end-to-end latency within the 16.7ms inter-frame time at 60fps to maintain a smooth AR experience. To validate this, we run the keypoint detection task with 3D cube rendering on the *BoxingPractice* video and plot the raw latency traces in Figure 4.13. The black dashed line in the figure is the 16.7ms deadline for 60fps AR devices, and the yellow curve is the end-to-end latency of this application. Due to our low latency object detection method (Encoding + MvOT) requires an average latency of only 2.24ms, we leave more than 14ms for the AR device to render high quality elements on the screen. We can find that our system is able to finish the detection and rendering tasks within 16.7ms for all 250 test frames.

4.9.5 Offloading Latency

Our RoI Encoding and Parallel Streaming and Inference techniques can effectively reduce the offloading latency. Figure 4.14 shows the offloading latency of three methods (Baseline, DRE, and DRE + PSI) with two different WiFi connections. We divide the offloading latency into streaming latency and inference latency for the first two methods, and use a PSI latency for the third method, because the streaming and inference processes run in parallel. The streaming latency contains time spending on encoding, transmission, and decoding tasks. The mean encoding latency to encode an HD frame on Jetson TX2 is 1.6ms and the mean decoding latency on our edge cloud server is less than 1ms.

In the baseline approach, the mean offloading latency is 34.56ms for WiFi-2.4G and 22.96ms for WiFi-5G. With the RDE technique, our system is able to reduce the streaming latency by 8.33ms and 2.94ms, respectively. Combine the techniques of both RDE and PSI, the system further reduces the offloading latency to 17.23ms and 15.52ms. We find that our latency optimization techniques are especially effective to reduce the offloading latency on lower bandwidth connections, such as on the 2.4GHz WiFi network.

4.9.6 Bandwidth Consumption

Our system is able to reduce the bandwidth consumption of the offloading task through the *Dynamic RoI Encoding (DRE)* and *Adaptive Offloading* techniques. We conduct an experiment to measure the bandwidth consumption of three different offloading approaches (Baseline, DRE only, and DRE plus Adaptive Offloading) in the object detection task. In all three approaches, we use seven different QPs (5, 10, 15, 20, 25, 30, and 35) to control the base quality to encode each frame. The approaches with the RoI Encoding technique will adjust the encoding quality based on the detected RoIs, and the adaptive offloading approach further makes the decision whether to offload each frame to the edge cloud. We record the mean detection accuracy and the bandwidth consumption of these approaches for each QP.

Figure 4.15 shows how the mean detection accuracy changes with the bandwidth consumption for the object detection task, with the comparison of these three approaches. For the same bandwidth consumption, our RoI Encoding plus Adaptive Offloading approach can achieve the highest mean detection accuracy. Similarly, we can observe that this approach also requires the least bandwidth consumption given a mean detection accuracy, e.g. to achieve the mean detection accuracy of 0.9, our system reduces 62.9% bandwidth consumption compared to the baseline approach.

4.9.7 Resource Consumption

Our solution consumes very few computation resources on the AR devices. To calculate the resource consumption of our system, we run an object detection task without any local rendering tasks on the DrivingPOV video repeatedly for 20 minutes and use the *tegrastats* tool from JetPack to measure the resource CPU and GPU usage. Figure 4.16 shows the raw resource usage traces for 20 minutes. Results show that our system requires only 15% of the CPU resource and 13% of the GPU resource, which leaves all the remaining resources to rendering rich graphic overlays for AR/MR system.

4.10 Related Works

Mobile AR. Designing mobile Augmented Reality system has attracted strong interest from both industry and academia. ARCore [67] and ARKit [68] are two mobile Augmented Reality platforms, while HoloLens [9] and Magic Leap One [10] further promise to achieve an experience called Mixed Reality. However, none of

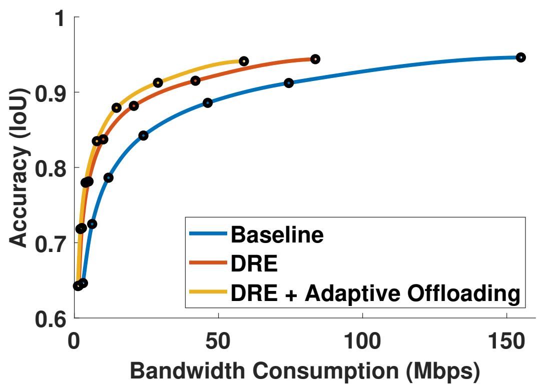


Figure 4.15: Bandwidth consumption of three approaches.

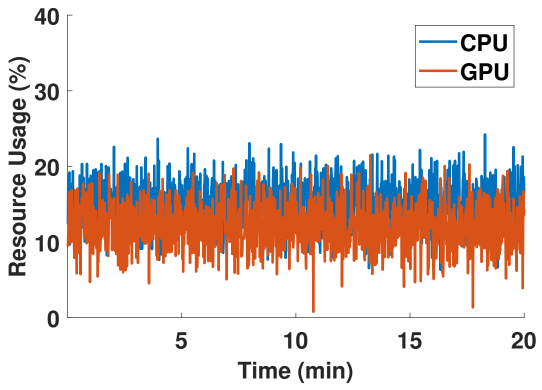


Figure 4.16: CPU/GPU Resource consumption of our system.

these platforms support object detection due to the high computation demands. To enable such experience, Vuforia [69] provides an object detection plugin on the mobile devices based on the traditional feature extraction approach. Overlay [70] uses sensor data from the mobile device to reduce the number of candidate objects. VisualPrint [71] aims to reduce the bandwidth consumption of image offloading by only transmit the extracted feature points to the cloud. However, none of them is able to run in real-time (e.g. 30fps or 60fps). Glimpse [15] is a continuous object recognition system on the mobile device running at 30fps. This system only offloads trigger frames to the cloud, and uses an optical flow based object tracking method to update the object bounding boxes on the remaining frames. However, Glimpse requires a more than 600ms end-to-end latency, as well as significant computational resources on the smartphone for object tracking, which is not feasible for high quality AR/MR system. Other mobile AR works [72, 73, 74] also provide useful insights for us.

Deep Learning. In recent year, Convolutional Neural Network (CNN) has been proven to achieve better performance than traditional hand-crafted feature approaches on various detection tasks. Huang et al. [75] compare the speed and accuracy trade-offs for modern CNN object detection models, including Faster

R-CNN [76], R-FCN [77] and SSD [78]. Thanks to the idea of multitask learning, current CNN can further reuse the deep features inside the object bounding box for more fine-grained detection, such as instance segmentation [79], human key points detection [79], facial landmark detection [80], etc. There have been extensive works on how to efficiently run these CNN models on mobile devices [81, 82, 83, 84, 85, 86, 87, 88]. However, none of them can satisfy the latency requirement for high quality AR/MR system.

Mobile Vision Offloading. Offloading computation-intensive tasks to cloud or edge cloud infrastructures is a feasible way to enable continuous vision analytics. Chen et al. [89] evaluate the performance of seven edge computing applications in terms of latency. DeepDecision [16] designs a framework to decide whether to offload the object detection task to the edge cloud or do local inference based on the network conditions. Lavea [90] offloads computation between clients and nearby edge nodes to provide low-latency video analytics. VideoStorm [91] and Chameleon [92] achieve higher accuracy video analytics with the same amount of computational resources on the cloud by adapting the video configurations. Most of these works focus on a single aspect in the whole vision offloading pipeline, while we focus more on improving the performance of the whole process.

Adaptive Video Streaming. Adaptive video streaming techniques have been largely exploited to achieve better QoE. Several 360-degree video streaming works [93, 94, 42, 95] also adopt the idea of RoI encoding to reduce the latency and bandwidth consumption of the streaming process. Other video adaptation techniques [96, 97, 98, 99] are also complementary to our work.

4.11 Conclusion

In this paper, we design a system that enables high accuracy object detection for AR/MR systems running at 60fps. To achieve this, we propose several low latency offloading techniques that significantly reduce the offloading latency and bandwidth consumption. On the AR device, the system decouples the rendering pipeline from the offloading pipeline, and uses a fast object tracking method to maintain detection accuracy. We prototype an end-to-end system on commodity hardware, and the results show that the system increases the detection accuracy by 20.2%-34.8%, and reduce the false detection rate by 27.0%-38.2% for two object detection tasks. The system requires very few resources for object tracking on the AR device, which leaves the remaining time between frames for rendering to support high quality AR/MR experiences.

Chapter 5

EdgeSharing: Edge Assisted Real-time Localization and Object Sharing in Urban Streets

5.1 Introduction

Urban streets and intersections are notorious traffic trouble spots. According to the U.S. Department of Transportation, 51 percent of all injury crashes and 28 percent of all fatal crashes in the United States occur at intersections. Many accidents, which occur at intersections involve visual occlusions of cars or vulnerable road users. This is a challenge that instrumenting individual vehicles with sensors, as in current automated driving and advanced driver assistance system, cannot fully solve since they can suffer from similar occlusions.

Ubiquitous sensors and computational resources on the road raise the possibility of smart intersections that address such occlusions through object sharing. Object sharing systems promise to extend traffic participants awareness beyond their field of views by sharing the moving object's detected from different camera perspectives and positions. For example, a leading vehicle can share its detected front vehicle or pedestrian locations to a following vehicle whose field of view has been blocked by the leading vehicle. Meanwhile, the following vehicle can also share the traffic participants detected in the blind spot of the leading vehicle to extend its awareness. Furthermore, a deployed camera at an intersection can localize each object in the intersection and provides this information to all clients in the nearby region.

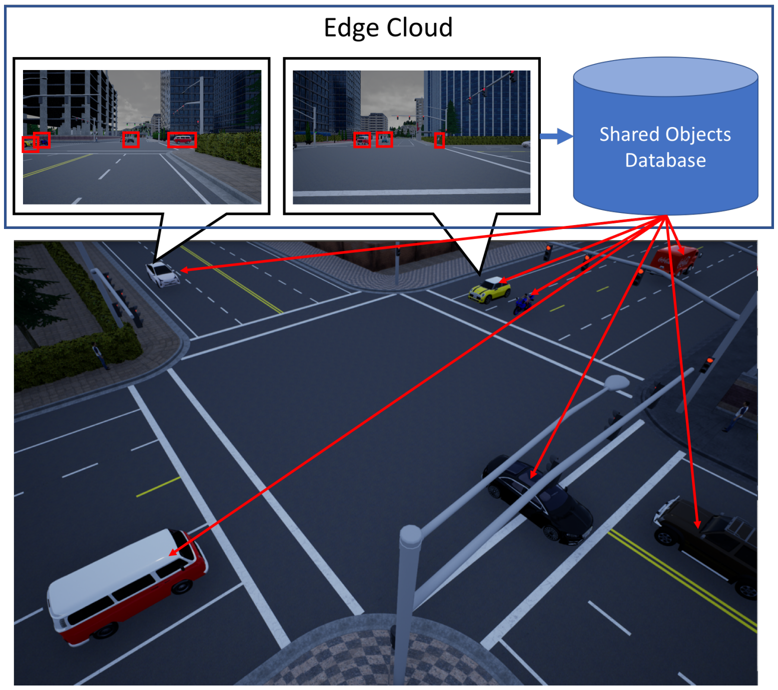


Figure 5.1: Object Sharing in Urban Street

Accurately sharing objects between moving clients is extremely challenging due to the high accuracy and low latency requirement for localizing both the client position and positions of its detected objects. Compared to GPS and other inertial sensing methods that are widely recognized to be less accurate in dense city scenarios, visual odometry solutions (e.g. SLAM) are more feasible in such situations where rich visual features exist. These solutions typically determine the position and orientation of a client device by analyzing the associated vision inputs (e.g. camera image, Lidar depth map, etc.) with a map constructed by 3D features. However, these solutions require large amounts of computation and storage resources on the end devices to store the large feature map and run computational intensive tasks on captured frames, which are less common to appear on commercial vehicles or smart devices. In addition, in order to continuously benefit from the evolving localization and detection algorithms, it is more feasible to run those intensive algorithms on the cloud, which can easily expand to large amounts of computation resource. However, offloading vision tasks to the cloud can incur long transmission delays, which make the feedback less useful to the mobile nodes.

Today's visual odometry method on mobile devices either run in small spaces or offload image features up to the cloud for assistance. Most of existing AR/VR platforms, including Apple ARKit [68], Google ARCore [67], Microsoft Hololens [9], and Oculus Go [100], are able to achieve real-time SLAM in small spaces with mostly stationary scenes. Mapping in large and open spaces are even more challenging due to the large variations in the environment. Existing outdoor mapping technologies adopted by self-driving cars requires the vehicle to equipped with high computational resources (e.g. GPU or FPGA) and also store a huge precomputed 3D feature maps onboard to achieve accurate visual based localization. Other applications such as Google Map's AR navigation app [101] offloads features from the captured image to the cloud and match them with the feature map stored in the cloud server, which effectively offloads the computation task to the cloud. For object sharing, AVR [74] is a pioneer work that allows a vehicle to share its point cloud to another vehicle through V2V communication. However, it requires each vehicle to have large computation resources and comprehensive 3D feature map on board, which is really hard to achieve with different manufacturers. Besides, The direct sharing method between mobile nodes is hard to apply any sophisticated system or upgrade with improved algorithms that exceed the computational capabilities of legacy devices. While cloud offload is a feasible way to overcome these limitations of the device's resources, it requires a long transmission latency to upload the visual information to the remote cloud. It is therefore highly desirable to offload localization and object sharing platform on in a way that reduces latency incurred by frame transmission so that it can support driving and vehicle control applications.

In this paper, we introduce EdgeSharing, a first collaborative localization and object sharing system leveraging the resources of an edge cloud platform and the visual inputs from participating mobile clients (e.g., vehicles and pedestrians). In EdgeSharing, the edge cloud holds a 3D feature map of its coverage region constructed from images and depth readings from a dedicated data collection vehicle or crowdsourced from participating clients. This 3D feature map is then used to provide accurate localization services to the client devices passing through this region. Besides, EdgeSharing also leverages the computation power on the edge cloud to detect object locations on the images offloaded by participating clients. These locations are stored in a sharing database and can be shared with other clients in the same region. With EdgeSharing installed on the edge cloud, nearby vehicles are able to learn extra object (e.g., traffic participant) locations from the edge cloud, which are outside the vehicles field of view. This improves their situational awareness and safety. To realize this, we propose several optimization techniques. In particular, we propose a Context-Aware Feature Selection method to filter out potential moving objects in the offloaded images to increase the localization accuracy. We also introduce a Collaborative Local Tracking mechanism to significantly reduce the bandwidth consumption of frame transmission by only offload selected keyframes to the edge cloud, while using a lightweight local tracking method to keep track of the location of the client and its detected objects on the end device. In addition, we design a parallel streaming and processing method to enable parallel video streaming and cloud processing, which largely reduces the end-to-end latency of EdgeSharing.

Compared to direct sharing between vehicles, an edge cloud-based object sharing system has allowed for easier software and hardware upgrading and maintenance, allowing the system to run more sophisticated algorithms at key intersections, even if legacy vehicles do not have the computational resources to run them fully in-vehicle. In addition, the large storage on the edge cloud also allows the system to store longer-lived object locations even when traffic is sparse.

EdgeSharing is able to achieve high-quality real-time localization and accurate object sharing with only small amounts of cloud computation and bandwidth resources. Our evaluation result demonstrates that the system is able to achieve a mean vehicle localization error of 0.2813-1.2717 meters, an object sharing accuracy of 82.3%-91.44%, and a 54.68% object awareness increment in urban streets and intersections. In addition, the proposed optimization techniques are able to reduce 70.12% of bandwidth consumption and reduce 40.09% of the end-to-end latency.

The contributions of this work can be summarized as follows:

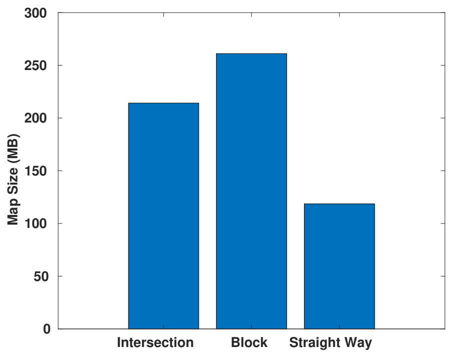
- Designing the first real-time collaborative localization and object sharing system EdgeSharing, leveraging the support of the edge cloud.
- Proposing a practical solution to localize objects detected by dynamic moving devices.
- Improving the localization accuracy with context-aware feature selection mechanism.
- Reducing offloading bandwidth with a collaborated local tracking method.
- Decreasing the end-to-end latency using a parallel streaming and processing pipeline.
- Implementing and evaluating the EdgeSharing system based on commodity hardware and showing that the proposed system can achieve high-quality real-time localization and accurate object sharing with only small amounts of cloud computation and bandwidth resources.

5.2 Motivation and Challenge

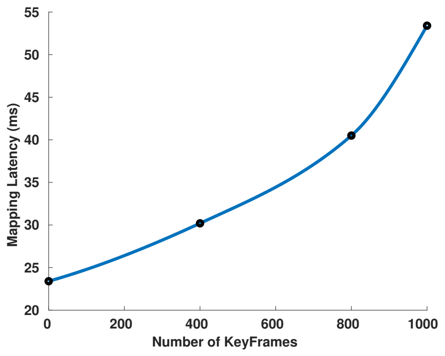
Many traffic accidents at intersections involve occlusions of cars or other vulnerable road users. This situation becomes even more severe in urban intersections where large amounts of vehicles and pedestrians heavily interact together. A left turning vehicle may obscure oncoming traffic. A large truck may block the view of following vehicles, therefore prevent them from making the right decisions. As ubiquitous sensors and computation resources appear on the road, this creates an opportunity for object detection and sharing between different mobile nodes, which promises to address such occlusion by extending the vision beyond their field of view.

In order to achieve a high-quality object sharing experience, the system requires to accurately localize both the positions of mobile nodes (e.g. vehicles and pedestrians) as well as their detected objects. While GPS and inertial sensor readings can achieve decent localization accuracy in an open area, it is widely recognized to have poor accuracy in urban canyon scenarios, due to multi-path and building obstructions. Therefore, visual odometry solutions (e.g. visual mapping and SLAM) offer higher accuracy in such situations, particularly in such urban areas where rich visual features exist. In addition, CNN based object detection is also required to localize the object position in the observer's perspective.

However, running the whole system on mobile nodes is very challenging because the task is memory and computationally intensive. First, the system requires the mobile nodes to have large storage resources to store the large 3D feature map for visual mapping. To demonstrate this, we conduct this analytic based on a popular open-sourced SLAM algorithm ORB-SLAM, which generates its feature maps using scene features (map points) among a subset of selected frames (keyframes) captured by a dedicated map generation vehicle. As the spatial area of the whole map increases, the number of keyframes increases and the



(a) Storage requirement for three scenarios.



(b) Mapping latency increases with the number of keyframes in the map.

Figure 5.2: Challenges of running object sharing system on mobile nodes.

storage consumption of the feature map increases consequently. Figure5.2(a) provides an illustration of the map storage requirement in three different scenarios: an urban intersection, a block, and a 100 meter straightway. We find that the map of only one intersection requires more than 200MB to store. Therefore, it is challenging to store many such feature map on mobile nodes.

Besides, it is challenging to run sophisticated mapping and object detection algorithm on mobile nodes due to their limited computation resources. While many existing works have already shown that running object detection on mobile devices results in very low frame rate [102], we also find that the mapping latency of ORB-SLAM increases with the number of keyframes in the map, as shown in Figure 5.10(c). A feature map with around 800 keyframes (similar to the number of keyframes in an intersection) requires around 40ms to track a frame on a normal CPU (i.e. Intel I7 5500U), which makes it extremely hard to run at 30fps on current mobile nodes.

In addition, Current SLAM algorithms (e.g. ORB-SLAM) have very poor performance running in the urban street due to the dense traffic. It is well known that SLAM algorithms are expected to fail when many moving objects occur in the scene. Based on experiments, we also observed that the accuracy of both tracking and mapping decrease as the number of moving vehicles increases in the scene. In order to provide a robust and accurate localization method, the object sharing system needs a more sophisticated algorithm that might require more computation resource.

While vehicles may be equipped with significantly higher computational resources, long replacement cycles for vehicles will make it challenging for older vehicles to be continuously upgraded with state-of-the-art algorithms that will likely require additional resources.

It is therefore highly desirable to have a localization and object sharing platform on the edge cloud that can be more easily upgraded and can overcome resource limitation on commercial mobile nodes. Compared to traditional cloud platforms that are typically far from the end devices, edge clouds are located at the network edge, thus require less latency for offloading tasks of mobile nodes. Previous works have already demonstrated the benefit of the edge cloud of vision based offloading tasks, such as VR, AR, and video analytics. In this paper, we ask whether it is also possible to use achieve cloud-based localization and object sharing system that aims to achieve high-quality object sharing with latencies low enough to be able to support vehicle control applications.

Note that this is not meant to replace local processing in vehicles but can augment in-vehicle processing at key intersections, particularly for legacy vehicles that may lack the resources for state-of-the-art algorithms.

5.3 EdgeSharing Design

In this paper, we propose EdgeSharing, a system that aims to provide high-quality real-time object sharing services to travelers in dense city traffic scenarios leveraging large computational resources at the edge cloud. First, this platform allows mobile clients, such as vehicles and pedestrians to offload computations of the visual localization task onto the edge cloud while maintaining low latency

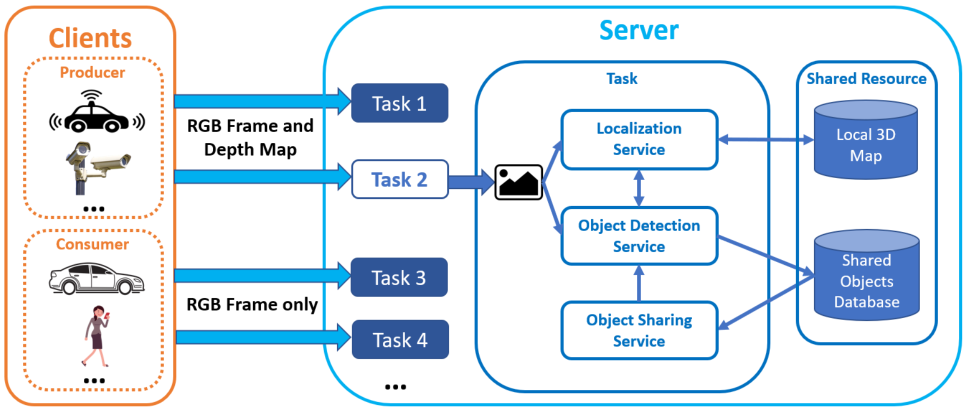


Figure 5.3: System Design

and high accuracy. Second, it collects sensor data from these mobile clients and other infrastructure sensors near the edge that offer different perspectives onto the intersection and merges this. Merging accuracy benefits from sharing rich data streams so that perspectives can be accurately aligned based on a large set of feature points. Merging information from different perspectives allows vehicles to see into blind spots and other areas of the intersection that are occluded by objects. More complete information about traffic participant positions and trajectories will allow improving efficiency and safety in advanced driver assistance as well as automated driving systems. The whole system consists of two parties: clients and an edge server, as illustrated in Figure 5.3.

Clients. Clients can be any mobile nodes with visual sensors (e.g. autonomous or human-driven cars, pedestrian's smartphones or smart glasses, and street cameras for commercial or security purposes) that have connectivity to the edge cloud server. In EdgeSharing, these mobile clients can be split into two categories: producer and consumer, with some devices acting as both. Producers are mobile clients with both RGB camera and depth sensors that can localize the position of captured objects in the 3D environment, while consumers are mobile clients that keep receiving the information of surrounding objects that it might interact within the local region. Participation as a consumer is possible without depth-sensing. Typical producer clients include advanced self-driving cars, street cameras or flying drones with RGB cameras and depth sensing capability. These devices are able to continuously transmit captured RGB frames and depth readings that contain information of its current perspective to the edge server. While consumers are mobile clients that keep receiving the information of surrounding objects that it might interact within the local region. Differently, consumer clients can be any existing commercial mobile devices such as smartphones, smartglasses, and commercial vehicles. These clients only need to offload the RGB frame captured by its equipped camera to localize itself, and then benefit from the objects shared by producers. Note that producers such as self-driving cars can also act as consumers in the meantime to receive object locations shared by other producers.

Edge Server. An edge server is a regional edge cloud or data center that gathers sensor readings of all participating mobile clients, detects objects from different perspectives, and provides such information back to mobile clients for safety purposes. As illustrated in Figure 5.3, the edge server consists of three key services: Device Localization, Object Detection, and Object Sharing. In order to benefit from the objects detected by different mobile clients, EdgeSharing requires estimating the accurate 3D pose information (location and orientation) of each participating client in the world coordinate system. While GPS and inertial sensor data provide low-cost location and orientation data, it is widely recognized to be noisy and inaccurate in urban canyons. The Device Localization procedure of EdgeSharing estimates the 3D transformation matrix with visual odometry techniques leveraging the offloaded frames from the client device. EdgeSharing redesigns a popular ORB-SLAM algorithm to work in this edge offloading scenario – the edge server collects the offloaded frames from mobile clients to generate the 3D map and uses it to help localize these devices. In the Object Detection Service, the system first leverages a state-of-the-art object detection algorithm to detect locations of a new object in the image frame. EdgeSharing uses the depth information provided by the producer client to estimate the 3D localization of each such object in its camera coordinate system, and then projects the position to the world coordinate system with the perspective projection from the device localization service. All detected object locations are stored in an object database on the edge server. The last component is Object Sharing Service which shares the stored object locations in the database with nearby clients. EdgeSharing projects the position of each object in the database back to the client's coordinate system and returns this information back to the client. We further detail this system in the following subsections.

5.3.1 Device Localization Service

To enable accurate object sharing in urban streets with dense traffic, EdgeSharing requires estimating the position and orientation of each participating client in the coverage region. This allows the system to calculate the relative transformation between two different objects and share the information with clients.

EdgeSharing leverages a popular SLAM framework (i.e. ORB-SLAM) as its device localization solution. The original purpose of ORB-SLAM is to leverage ORB features extracted from continuous captured frames for real-time construction or updating of a 3D feature map of an unknown environment, while simultaneously keeping track of an agent's location within it. In EdgeSharing, we use this ORB-SLAM solution to generate a high-quality 3D feature map of the coverage region of an edge cloud server (e.g. an intersection) and use this map to localize the participating mobile clients in this local region.

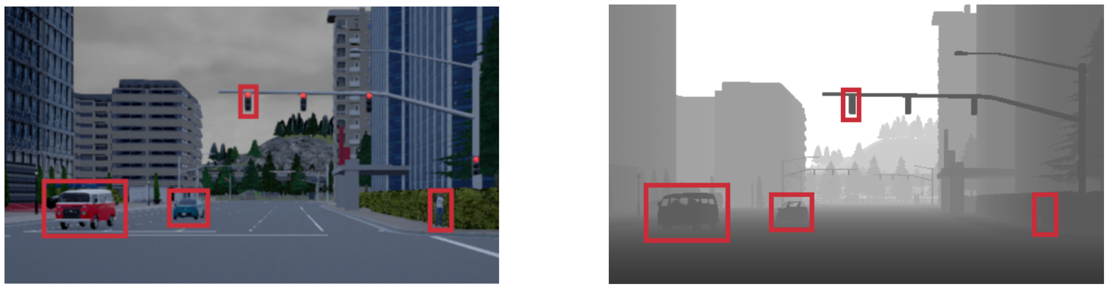
To build the feature map for device localization, we can simply use a dedicated data collection vehicles to drive through the region from different directions for several times and provide the captured frames to the ORB-SLAM algorithm to process. There has been other research that constructs feature map using crowd-sourced frames from travelers in the street [103]. Once the map is constructed, the edge server is able to provide device localization service to the mobile devices by matching the ORB feature points from their offloaded image to features in the 3D feature map. The system further optimizes the pose of the device with RANSAC and motion model constraints. The final output of the device localization service is a world to camera transformation matrix (T_{cw}) that is able to transform a point's 3D location between the world coordinate system and the camera coordinate system. This matrix is also refer to the combination of a rotation matrix (R_{cw}) and a translation matrix (t_{cw}) between the two coordinate system as shown in Equation 5.3

5.3.2 Object Detection and Sharing Service

After localizing the 3D position and orientation of each client, EdgeSharing adopts an Object Detection and Sharing Service to identify the objects in the client's field of view, while also share those NLOS objects that have been detected by other clients on the road. This service has three key components: Object Bounding Box Detection, Object 3D Localization, and Object Sharing.

Object Bounding Box Detection. EdgeSharing leverages state-of-the-art CNN based Object Detection models to identify object locations on each frame's pixel coordinate system. Specifically, we use a ResNet-50 based Faster RCNN Object Detection model to detect object locations of each video frame. This model takes the raw RGB frame as the input and outputs the 2D bounding boxes of objects appears in the frame. The model is trained with Microsoft Coco dataset, which contains 91 different object types, including person, car, motorcycle, bicycle, bus, truck, traffic light, and different street signs that all appear frequently in urban streets.

Object 3D Localization. In order to share the detected object to other clients, EdgeSharing requires to estimate the 3D localization of each object in the world coordinate frame. The Object 3D Localization process works in the following procedures: (1). The system estimates the depth of each object with respect to the camera's coordinate system. This step can be achieved using depth sensors from the vehicle (e.g. Lidar, Radar, RGBD camera or stereo camera), or based on other monocular based depth estimation techniques. (2). The system then uses perspective projection to transform the location of a vehicle from the 2D pixel coordinate frame to the 3D world coordinate frame. Equation 5.1 shows the projection equation of projecting a point from the pixel coordinate system (u, v) to the camera's coordinate system (x_c, y_c, z_c), and then to the world coordinate system (x_w, y_w, z_w). EdgeSharing uses the center point of each bounding boxes as the location of this object (u, v) on the frame's pixel coordinate system, and uses the median depth value inside the bounding box as its depth towards the camera's coordinate system (z_c). With the position information of the object and the intrinsic matrix (K) of the camera, the system is able to transform each object to the camera's coordinate system. After that, EdgeSharing further derives the position of each object in the world coordinate system leveraging the client's Extrinsic Matrix (T_{cw}) calculated by the device localization service. (3). All detected locations of objects are immediately stored in a Shared Object Database, an object collector on the edge cloud that stores the real-time 3D object locations in its coverage area. The Shared Object Database gathers the location of each object from images offloaded by nearby clients, while providing this information for object sharing. To maintain a timely object sharing, each collected object will be expired after 33 ms. Note that this process is only executed for frames offloaded by producer clients, which include both the color and depth information of frames.



(a) Object detection on the RGB frame. (b) Estimate depth of each detected object using the depth map of the frame.

Figure 5.4: Object 3D Localization.

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = KT_{cw} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{5.1}$$

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.2}$$

$$T_{cw} = \begin{bmatrix} R_{cw} & t_{cw} \end{bmatrix} \tag{5.3}$$

Object Sharing. Finally, EdgeSharing shares objects inside the Shared Object Database to the client to provide them more information beyond their line-of-sight. In particular, the system transforms all shared objects within the responsible region from the world coordinate system to the client camera's coordinate system based on the transformation matrix (T_{cw}) calculated from the device localization service. The server combines the shared object locations with the detected object locations, and send them back to the client.

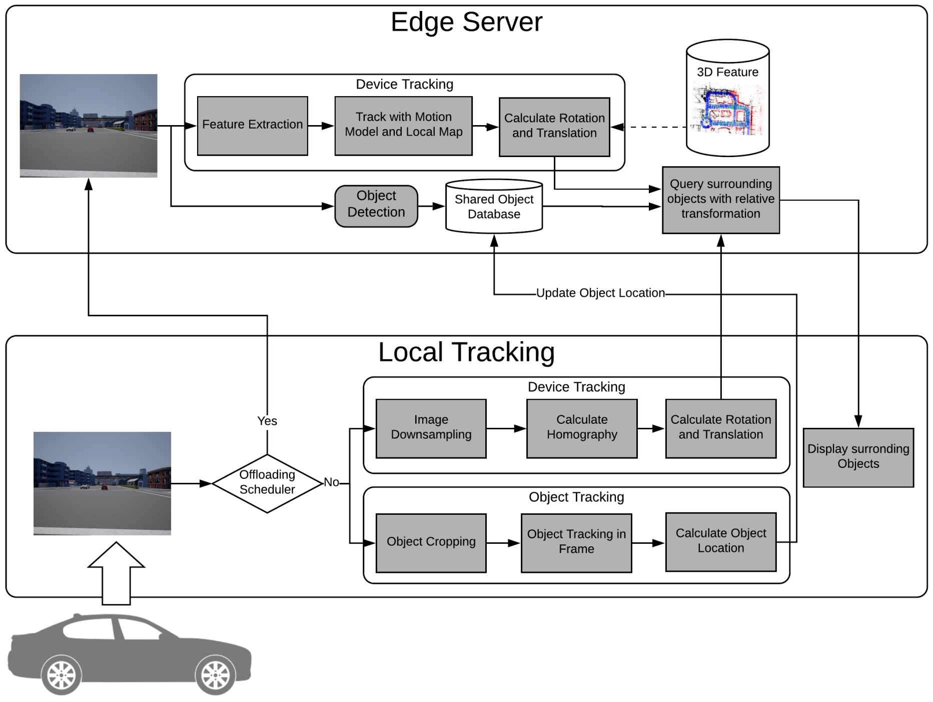


Figure 5.5: Collaborated Local Tracking

5.3.3 System Optimization

In addition, we propose several optimization techniques on our system to improve the performance of EdgeSharing. First, we design a *Context Aware Feature Selection* method to select only stable features on each frame to estimate the pose of the device (Section 5.4). This method takes advantage of the powerful computation resource on the edge server and uses the detected object bounding boxes to do the feature selection. Second, EdgeSharing adopts a *Collaborated Local Tracking* mechanism to reduce offloading bandwidth while maintaining a high tracking accuracy (Section 5.5). Last, we implement a *Parallel Streaming and Processing* pipeline to decrease the end-to-end latency of the whole offloading tasks by efficiently pipelining streaming and image processing processes in parallel (Section 5.6).

5.4 Context Aware Feature Selection

EdgeSharing applies a *Context Aware Feature Selection* mechanism to increase the accuracy of our device localization service in dense traffic scenarios. While SLAM based visual odometry method has been adopted as the key inside-out tracking method for mobile devices, it is still widely recognized to be vulnerable to moving objects in the scene [104]. This method relies on matching feature points between continuous captured frames and uses them with motion model constraints to derive instant changes over time. The key principle underlies this mechanism is that most of the matched feature points are not moving between frames in the world coordinate system. However, based on our observation, the principle is very hard to guarantee in dense traffic scenarios, where large amounts of moving objects (e.g. vehicles, pedestrians, etc.) exist. To overcome this challenge, we propose a *Context Aware Feature Selection* method to filter out potential moving features in the scene and only use stable feature matches to estimate the location changes. The intuition of this approach is that we believe feature points located inside the detected object bounding boxes of movable objects are more likely to move than other background regions. By taking advantage of the object detection service, EdgeSharing directly filtered out the feature points lie in the bounding boxes of potential moving objects. In particular, we consider the following objects as potential moving objects: car, bus, truck, motorcycle, bicycle, and person.

Figure 5.6 shows an illustration of our feature selection method when processing one of the offloaded frames. The green markers are the feature matches between the frame and feature points in the 3D map. The blue markers represent the matches with previous frames and red markers are the discarded matches due to they lie in the bounding boxes of detected potential moving objects. In the device localization service, the system only matches green and blue feature points with the 3D feature points and use them to derive the location of the device. We

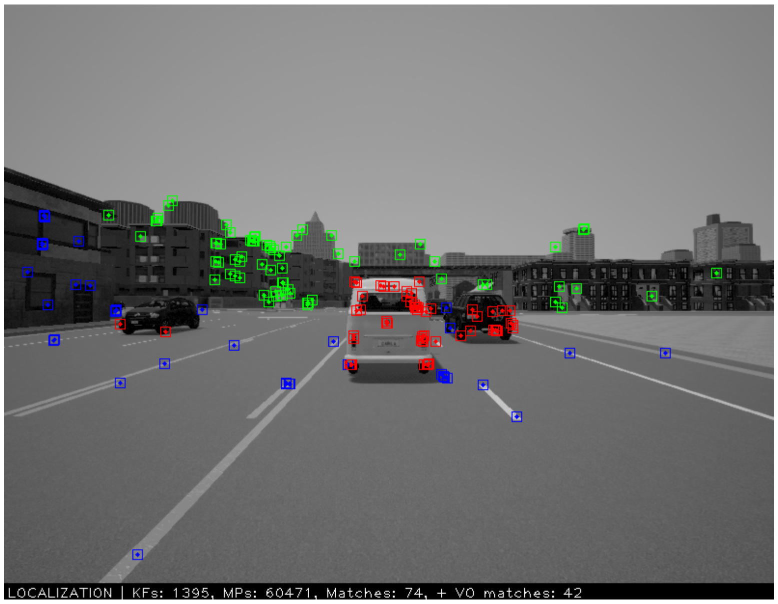

LOCALIZATION | KFs: 1395, MPs: 60471, Matches: 74, + VO matches: 42

Figure 5.6: Context Aware Feature Selection

further show how this mechanism can increase the localization accuracy in dense city traffic scenarios in Section 5.10.

5.5 Collaborated Local Tracking

EdgeSharing requires participants to keep offloading video frames to the server through the wireless link, which require a huge amount of bandwidth support. To eliminate this large overhead, a potential solution is to reduce the offloading frequency. However, this will result in a low device localization accuracy due to the low update rate. To overcome this trade-off between the localization accuracy and bandwidth consumption, we propose a *Collaborated Local Tracking* to reduce the offloading frequency while keeping a high localization accuracy. Instead of continuously offload every frame to the edge server, EdgeSharing leaves some frames on the device for local tracking and updating. The offloaded frames still follow the regular procedures described in Section 5.3, while the local frames go through a fast local tracking process to estimate the location changes of the client as well as the position of its detected objects.

5.5.1 Local Device Tracking

As shown in Figure 5.5, the local device tracking module has three components: (1) Image downsampling, (2) Homography calculation, and (3) Transformation calculation. In the first step, the system downsamples the raw frame to a smaller resolution (from 800x600 to 400*300) to reduce the time complexity of following feature extraction and matching. Then, the system extracts ORB feature points from the downsampled frame and find feature matches with the last offloaded frame. After that, the system runs a RANSAC with projective motion model to get a group of correct matches between the current frame and the last offloaded frame using the matched ORB feature pairs between them. Finally, an SVD method can be used to estimate the homography matrix between two images. As shown in Equation5.4, Homography matrix is a $3x3$ matrix that is an image to image mapping matrix that is able to transform each pixel's homogeneous coordinate on the last offloaded image ($[u, v, 1]^T$) to its corresponding coordinate on the current image's pixel homogeneous coordinate system ($[u', v', 1]^T$).

$$\lambda \begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = H \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{5.4}$$

With the homography matrix between two images, the system then extracts the rotation and translation matrix between two images, and uses them to build the transformation matrix $T_{c'c}$ that can transform a 3D point in the camera's homogeneous coordinate system of the last offloaded frame ($[x, y, z, 1]$) to the corresponding points in the current camera's homogeneous coordinate system ($[x', y', z', 1]$). After that, the system can calculate the transformation matrix from the world coordinate system ($T_{c'w}$) using Equation 5.5.

$$T_{c'w} = T_{c'c}T_{cw} \tag{5.5}$$

With this local tracking method, EdgeSharing is able to localize the position of the client device in real-time without the necessity of offloading every frame to the edge to process. For the local processed frame, the mobile client only needs to send the updated location of the device to the edge server, and the server can use this information to share objects to the client.

5.5.2 Local Object Tracking

In addition to local device tracking, EdgeSharing also enables object tracking when the frame is kept on the device for local processing. The local object tracking has three main procedures. First, the system uses a motion vector based object tracking method to shift the bounding box of each object from the last offloaded to the current frame, based on the match feature points in those regions. Then the system uses the same method as introduced in section 5.3 to find the position of the object in the camera's coordinate frame using the new bounding box location and depth readings of that region. After that, the location of the object can be projected to the world coordinate system using the transformation matrix $T_{c'w}$ calculated in the local tracking module. Finally, the client device transmits the device transformation matrix and all object locations to the cloud to update the share object database and retrieves the shared objects based on its position. This local object tracking technique can help the producer client update its detected object position without continuously uploading frame to the cloud.

5.5.3 Adaptive Offloading

In addition to the local tracking mechanism, we also design an adaptive offloading mechanism to determine whether the client device should offload the current frame to the edge server or keep it for local tracking and only update the location and orientation changes to the server. The key intuition behinds this method

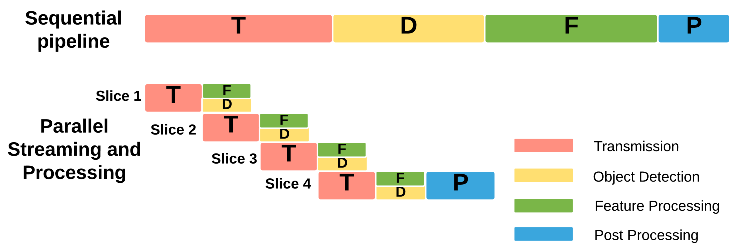


Figure 5.7: Parallel Streaming and Processing

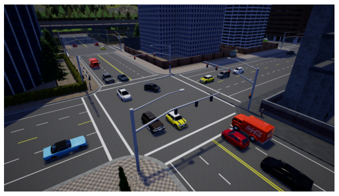
is that we want to offload the frames that have large changes compare to the previous offloaded frame. Therefore, Adaptive Offloading uses the number of feature matches between the last offloaded frame and the last frame to determine how similar is the current frame compared with the last offloaded frame. If the feature matches are too low, the system will be hard to maintain a high accuracy tracking, therefore, should offload the frame to the edge server to process. If there are still a lot of matches with the last offloaded frame, the system can continue to process the frame locally. We empirically pick 40 pairs as the threshold and report the result in section 5.10.

5.6 Parallel Streaming and Processing

EdgeSharing also strives to reduce the end-to-end latency of the whole transmission and processing pipeline. The whole processing pipeline consists of frame streaming and several cloud processing procedures that usually has a The end-to-end latency of more than 50ms. Such long latency may significantly reduce the accuracy of localization and object sharing, since the location of objects may already changes after the system get the detection result. To overcome these challenge we propose a Parallel Streaming and Processing pipeline to parallel the processing pipeline of different tasks on different resources of the edge cloud platform, and therefore, largely reduce the overall latency spend on the task.

As illustrated in Figure 5.7, the baseline of the offloading pipeline is consists

(a) Dataset 1: 4-way Stop Sign.


(b) Dataset 2: City Intersection.

(c) Dataset 3: LA Intersection.

of four key procedures: (1) Frame transmission from the client device to the edge server, (2) Object detection on the offloaded frame, (3) Feature processing on the offloaded frame (i.e. feature extraction and matching), and (4) post processing, including camera pose estimation, object location estimation and object sharing. Since both the transmission and image processing on the server take considerable time and run sequentially, it ends up with a long end-to-end latency for the entire system. A previous work [105] has already shown how to enable parallel streaming and inference for object detection tasks. In EdgeSharing, we further extend this idea to enable parallel streaming and feature processing. In particular, the client device splits each offloaded frame into four slides and transmit them one by one. Once each slide of the image arrives at the edge server, the system can immediately start feature extraction or object detection tasks on the single slide instead of waiting for the whole frame to arrive. In addition, we also parallel the procedures of object detection and feature processing by offloading the object detection to the onboard GPU. After all object detection and feature processing task finished, the system starts to the post processing stage and finish the rest of the task.

Parallel Streaming and Process is able to achieve almost half latency reduction compared to the baseline approach. More results regarding this technique is described in the section 5.10.

5.7 Implementation

Our implementation is entirely based on commodity hardware and consists of more than 5000 lines of code.

5.8 Hardware Setup

We emulate the whole system using a group of commodity hardware. On the client side, We use a mobile development board Nvidia Jetson TX2, which is connected to a TP-Link AC1900 router through a WiFi connection. We emulate an edge cloud with a PC equipped with an Intel i7-6850K CPU and an Nvidia Titan XP GPU, which connects to a router through a 1Gbps Ethernet cable. Both the client device and the edge server run an Ubuntu 16.04 OS.

5.9 Software Implementation

The whole system is built upon the successes of previous open source projects or libraries, including ORB-SLAM2 [104], OpenCV [106], Nvidia JetPack [62], Nvidia Multimedia API [63], Nvidia TensorRT [61], and the Nvidia Video Codec SDK [40]. We further explain the implementation of the edge server and the client side in detail.

Edge Server. The edge server-side implementation contains three key modules: *Frame Decoding*, *Object Detection* and *Frame Tracking*, which are designed to run in three different threads to avoid blocking each other. In *Frame Decoding* thread, the edge server keeps decoding frame slides transmitted from the client device and feeds them to the *Object Detection* thread and the *Frame Tracking* thread immediately after completion. The *Object Detection* thread is implemented using the Nvidia TensorRT, which is a high-performance deep learning inference optimizer designed for Nvidia GPUs. To push the limit of inference latency on the server side PC, we use the INT8 calibration tool [64] in TensorRT to optimize the object detection model, and achieves 3-4 times latency improvement on the same setup. More design detail can be found in a previous work [105]. The *Frame Tracking* thread is developed based on the famous ORB-SLAM2 algorithm [104]. We modify the original project to fit our requirement of parallel processing and feature selection. In particular, we create a Frame object for each frame slide and use it to extract key points and ORB descriptors on each slide. Then the feature matching is performed by matching each feature point on a slide with features on the entire last frame. To implement the *Context-aware Feature Selection* method, we perform a filter on the match feature pairs in the post processing stage, leveraging the object bounding boxes detected in the *Object Detection* thread.

Mobile Client. We implement the client side functions on the Nvidia Jetson TX2 with its JetPack SDK and OpenCV. The implementation follows the design flow in Figure 5.3 and Figure 5.5. We use OpenCV to achieve the image downsampling, cropping, and various transformation matrix calculation. To implement the Parallel Streaming and Inference module, we enable the slice mode for the video encoder and use the setSliceLength() function with a proper length to let the encoder split a frame into four slices. Each slide is immediately transmitted out to the edge server to process.

5.10 Evaluation

In this section, we evaluate the performance of the EdgeSharing in terms of device localization accuracy, object sharing latency, bandwidth consumption, end-to-end latency. The results demonstrate that our system is able to achieve both the high accuracy and the low latency requirement for device localization and object sharing in urban streets. The result shows that the system is able to achieve a mean vehicle localization error of 0.2813-1.2717 meters, an object sharing accuracy of

82.3%-91.44%, and a 54.68% object awareness increment in urban streets and intersections. In addition, the proposed optimization techniques are able to reduce 70.12% of bandwidth consumption and reduce 40.09% of the end-to-end latency.

5.10.1 Experiment Setup

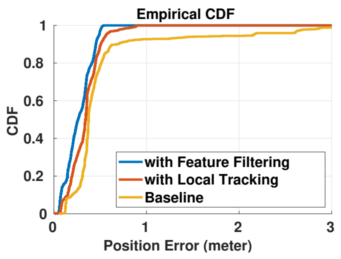
We use the setup and implementation described in Section 5.7 to conduct experiments. Two different object sharing tasks are designed to evaluate the performance of our system: a vehicle to vehicle object sharing and an infrastructure to vehicle object sharing. In vehicle to vehicle object sharing scenario, the EdgeSharing server collects object position from producer vehicles with both cameras and depth sensors and providing object sharing service to all consumer vehicles with only RGB cameras. In the infrastructure to vehicle sharing scenario, the EdgeSharing gets object position from a street camera and depth sensor, and shares detected objects to all vehicles with RGB cameras.

For repeatable experiments, we use three collected datasets as described in the section 5.10.2 to conduct our experiment. During the experiment, we simulate the whole offloading and sharing process follows the workflow described in Section 5.3.

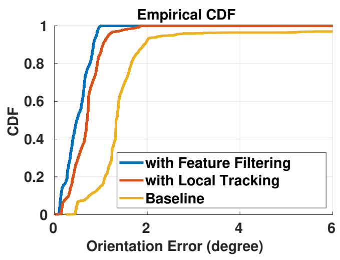
5.10.2 Dataset Description

We collected three different datasets to evaluate the performance of our system. To first evaluate the system under perfect data inputs, we collect two datasets using an open-source simulator for autonomous driving called CARLA [107]. CARLA provides open digital assets (urban layouts, buildings, vehicles) that were created for this purpose and can be used freely. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors and maps generation. We carefully choose one 4-way stop sign intersection and another city intersection from CARLA's map 3 and map 5 to collect dataset for our experiment. Figure 5.8(a) and Figure 5.8(b) show the top view of two intersections. During the data collection, we spawn 400 vehicles equipped with front view cameras in random locations on the map. In the 4-way stop sign intersection, we aim to evaluate vehicle to vehicle object sharing, therefore, each vehicle is equipped an additional depth camera to record the corresponding depth for each pixel on the RGB image. In map 5, we put a street camera and another depth camera at the position shown in Figure 5.8(b) to evaluate infrastructure to vehicle sharing scenario. The street camera keeps capturing 120-degree field of view and 1280x720 resolution frames at 30 fps, while vehicles are capturing frames 90 fov 800x600 resolution frames at 30 fps. All images are recorded with synchronized time-stamps for our evaluation. We also log the position and orientation data of each participant as groundtruth for the following evaluation.

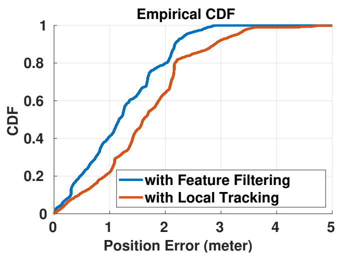
To further evaluate the performance of EdgeSharing in real traffic scenarios, we collect another 120-hour dataset in an intersection in Los Angeles, CA. In this dataset, we use GoPros mounted at the bottom center under the windshield to record the full drivers front view videos with 1280720 resolution at 30 fps. The 30Hz GPS readings are also recorded using the embedded GPS sensor in GoPros. During the data collection, ten different drivers were driving together as a fleet in an urban street back and forth for 30 times. Figure 5.8(c) show an example of collected video data, where three vehicles are driving together in a row. We can observe that the second vehicle can see the first vehicle in its field of view. Similarly, the third vehicle can also see the second vehicle in its front view. We also use this data to evaluate how well EdgeSharing can provide localization and object sharing services to them.



(d) CDF of position error in dataset 2.



(e) CDF of orientation error in dataset 2.



(f) CDF of position error in dataset 3.

Figure 5.8: CDF of device localization of EdgeSharing.

Dataset	Approaches	Position (meter)	Orientation (degree)
4-way Stop Sign	Baseline	0.6481	2.0129
	+ Feature Selection	0.3173	0.5742
	+ Local Tracking	0.3856	0.8215
City Intersection	Baseline	0.5801	1.8812
	+ Feature Selection	0.2813	0.5216
	+ Local Tracking	0.3650	0.7518
LA Intersection	Baseline	Fail	N/A
	+ Feature Selection	1.2717	N/A
	+ Local Tracking	1.6841	N/A

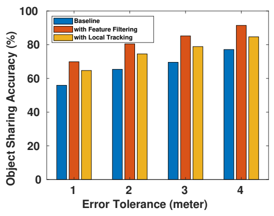
Table 5.1: Mean position and orientation error of three different approaches on three datasets.

5.10.3 Accuracy of Device Localization

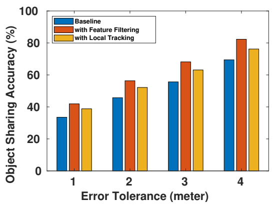
EdgeSharing is able to achieve high accuracy of device localization under many different urban traffic scenarios. We first measure the device localization accuracy of EdgeSharing in three approaches: the baseline solution (Baseline), our solution with the *Context-aware Feature Selection* method (+ Feature Selection), and our solution with both the *Context-aware Feature Selection* and the *Collaborated Local Tracking* method (+ Local Tracking). The baseline approach follows the standard pipeline we introduced in Section 2. We evaluate the detection accuracy of our system with two key metrics: the position error and the orientation error of the device, as shown in Table 5.1. For the 4-way stop sign intersection and the city intersection dataset, we create a dedicated map generation vehicle in the simulator and let it drive through the intersection from different directions and lines for several times. The captured 30fps RGB frames and depth maps are used to generate the map. For the LA intersection dataset, we use two videos from the leading vehicle to construct the feature map. The map is then used to help the other vehicles localize themselves in the same intersection. In the baseline approach, both the map generation and device localization processes use all ORB feature points on the frame, while only selected features are used for those tasks in the last two approaches. We use some reference points to transform the map's coordinate system to the world coordinate system of the groundtruth location reading, and use it unify the position and orientation output of the device Localization service with our groundtruth. As shown in Table 5.1, the proposed *Context-aware Feature Selection* is able to increase the localization accuracy for the first two datasets. In the 4-way stop sign dataset, EdgeSharing reduces the mean position error from 0.6481 meters to 0.3173 meters, and reduces the mean orientation error from 2.0129 degrees to 0.5742 degrees. Similarly, EdgeSharing also reduces the mean position error and mean orientation error to 0.2813 meters and 0.5216 degrees correspondingly. For the LA dataset, we only evaluate the performance of position error since we did not collect orientation groundtruth. For the baseline approach, we find that EdgeSharing fails to build a meaningful feature map due to the interference of a large amount of moving objects and the vehicle's own dash. Therefore, the localization with the map generated by the baseline solution does not make any sense. With the *Context-aware Feature Selection* method, EdgeSharing is able to achieve a pretty good performance of around 1.2717 meters. Note that this error may also be caused by the inaccurate GPS readings collected by the GoPro.

In addition, we show that the proposed *Collaborated Local Tracking* method does not significantly reduce the localization accuracy. As listed in Table 5.1, the *Collaborated Local Tracking* method increases the mean position error from 0,3173 meter to 0.3856 meter for the 4-way stop sign dataset, increases from 0.2813 meter to 0.3650 meter for the city intersection dataset, and increases from 1.2717 meter to 1.6841 meter for the LA intersection dataset. The Orientation error also performs similar behavior.

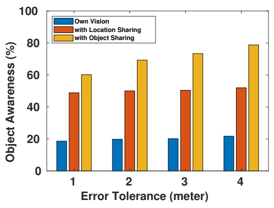
To further understand the distribution of the position and orientation error, we show the cumulative distribution function (CDF) of localization error for the city intersection dataset in Figure 5.8(d) and Figure 5.9(d) respectively. The 95 percentile of the position error distribution is 0.489 meter for the approach with *Context-aware Feature Selection* and 2.156 meters for the baseline approach. Similarly, The 95 percentile of the orientation error distribution is 0.909 degree for the approach with *Context-aware Feature Selection* and 2.579 degrees for the baseline approach. The result demonstrates that our *Context-aware Feature Selection* technique is able to reduce the localization error and keep most of the tracking error in a very low value. Figure 5.8(f) shows the CDF of position error in the LA dataset. We can observe that EdgeSharing is able to achieve a position error of fewer than 3 meters for the majority cases in both the feature selection and local tracking approaches.



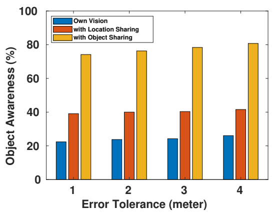
(a) Object Sharing Accuracy of the City Intersection Dataset.



(b) Object Sharing Accuracy of the LA Intersection Dataset.



(c) Object Awareness of the City Intersection Dataset.



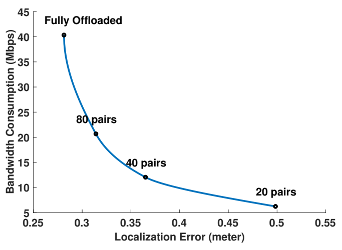
(d) Object Awareness of the 4-way Stop Sign Intersection Dataset.

Figure 5.9: Accuracy of Object Sharing.

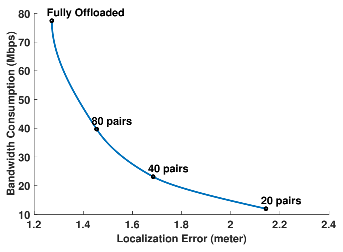
5.10.4 Accuracy of Object Sharing

Our system is able to achieve high localization accuracy for object sharing in urban streets, and provides higher object awareness to individual vehicles to enlarge their vision. We first use object localization accuracy to estimate the performance of EdgeSharing on different datasets. In terms of object localization accuracy, we want to understand what percentage of detected objects is correct among all detected objects. We first show the object sharing accuracy of the first and third dataset with three different approaches in Figure 5.9(a) and Figure 5.9(b) respectively. To calculate the accuracy, we use the object 3D localization method introduced in section 5.3.2 to calculate the 3D location of the object in the world coordinate system, and check whether in an error tolerance range can we find a vehicle from the collected groundtruth data. As shown in Figure 5.9(a), we can observe the approach with *Context Aware Feature Selection* is able to achieve the highest sharing accuracy compare to other methods for the same error tolerance. And for the same approach, the accuracy increases as the error tolerance increases. The sharing accuracy of the *Context Aware Feature Selection* approach with an error tolerance of 4 meters is able to achieve 91.44%. For the LA dataset, we only calculate the sharing accuracy of our data collection vehicles in the scene. The result of *Context Aware Feature Selection* approach and *Collaborated Local Tracking* approach are depicted in Figure 5.9(b), which shows our system is able to achieve 82.3% sharing accuracy with 4-meter error tolerance in the dataset collected in real scenarios with imperfect sensors. Note that the wrong detection can be considered as the combination of device localization error, depth sensor error, object detection algorithm error, and the error caused by vehicle shape. We experimentally find that the error of the CNN based object detection algorithm has the most significant contribution to the entire object sharing error.

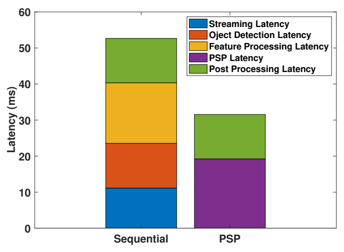
Second, we want to know how many more objects can one vehicle be aware of with the support of EdgeSharing. To achieve this, we define object awareness as the percentage of objects that one vehicle is aware of in the region of the intersection. We compute the object awareness of each vehicle in the intersection at each timestamp with three approaches: own vision, location sharing only, and location and object sharing in three different error tolerance. In the own vision approach, each vehicle only knows the objects in its field of view. With location sharing, the producers in the street are able to store their position in the shared object database for sharing, based on the transformation matrix calculated from the localization service. With object sharing, these producers further store the locations of the detected object to the database. In the experiment of the 4-way stop sign dataset, we randomly assign 25 vehicles as producer and calculate the object awareness for the rest of vehicles in the region of the intersection. For the city intersection dataset, we only use the street camera as the producer and calculate the rate for all vehicles in the region of the intersection. Figure 5.9(c) and Figure 5.9(d) show the object awareness of two dataset correspondingly. We can find that object awareness is significantly increasing with the support of location sharing and object sharing from the producer clients. We do not demonstrate the object awareness since we don't have the groundtruth location data for all vehicles on the street.



(a) Localization error vs Bandwidth consumption for dataset 2.



(b) Localization error vs Bandwidth consumption for dataset 3.



(c) Latency comparison between the sequential approach with the Parallel Streaming and Processing approach.

Figure 5.10: Bandwidth consumption and end-to-end latency of EdgeSharing.

5.10.5 Bandwidth Consumption

EdgeSharing uses the *Collaborate Local Tracking* method to significantly reduce the bandwidth consumption while maintaining a low localization error. To demonstrate this, we tune the offloading threshold (feature matching pair number) used in *Collaborate Local Tracking* to illustrate the relationship between localization error and the bandwidth consumption of our system. Figure 5.10(a) and Figure 5.10(b) show this relationship of the city intersection and LA intersection datasets. Compared to the fully offloaded scenarios, EdgeSharing is able to largely reduce the bandwidth consumption, while only has a slight increment on the localization error. For example, if we choose to use 40 matching pairs as the offloading threshold, we can reduce the bandwidth consumption of the urban intersection dataset from 40.34 Mbps to only 12.05 Mbps, while only increasing the localization error from 0.2813 to 0.365 meters. Similarly, we also observe the same pattern for the LA dataset we collected in real cases. Note that the LA dataset requires higher bandwidth consumption, because it is transmitting a large frame than the simulation dataset we collected from CARLA.

5.10.6 Latency

EdgeSharing includes a *Parallel Streaming and Processing* (PSP) pipeline to reduce the end-to-end latency of the system. We contact a real-time experiment on the city intersection dataset using the setup introduced in section 5.7, and compare the sequential approach and the PSP approach in Figure 5.10(c). As shown in the Figure, we divide the offloading latency into streaming latency, object detection latency, feature processing latency, and post processing latency for the sequential approach, and use a PSP latency for the third method, because the streaming and inference processes run in parallel. The streaming latency contains time spending on encoding, transmission, and decoding tasks. The mean encoding latency to encode an HD frame on Jetson TX2 is 1.6ms and the mean decoding latency on our edge cloud server is less than 1ms.

In the sequential approach, the mean end-to-end latency to finish the entire pipeline of one frame is 52.66ms, while our solution with *Parallel Streaming and Processing* technique requires only 31.55ms, which makes it possible for the system to deliver 30fps object sharing experience to the participated clients.

5.11 Related Works

Collaborated Sensing in Connected Vehicles. Connected vehicles allows cars to share information and sensor reading to other devices both inside as well as outside the vehicle. Traditional vehicular communication systems (e.g. DSRC) has been used to solve plenty of traffic issues by sharing safety messages to other nodes on the road [108, 109, 110, 111]. Many recent works further explore to use millimeter wave to provide much higher bandwidth for the vehicular communication [112, 113, 114]. With such great resources, there have been more works that focus on sharing the camera perceptions through V2V networks not limited for autonomous vehicles but also for other Advanced Driving Assistance Systems

(ADAS). The see-through system [115] uses monocular cameras on the car to share its front view to its following vehicles. The system uses DSRC to transmit the front cars view to its following cars and overlays the occluded region on the augmented view of following cars. Inspired by this vision sharing idea, AVR [74] broadens the vehicles visual horizon by sharing 3D visual information extracted from stereo cameras with other nearby vehicles. This system proposes a novel usage of the SLAM algorithm in vehicle position tracking, and methods to isolate and track dynamic objects to save bandwidth consumption. However, this work requires each vehicle to have large computation resources and comprehensive 3D feature map on-board, which is really hard to achieve with different manufacturers. In contrary, EdgeSharing is a complete edge cloud-based solution for object sharing between vehicles and other mobile nodes. Compared to direct sharing methods, edge cloud-based object sharing system has the benefits of easier system upgrading and maintenance, as well as the better computation resources for more sophisticated algorithms.

Vision Task Offloading. Offloading computation-intensive tasks to cloud or edge cloud infrastructures is a feasible way to enable continuous vision analytics on power and computation constraint devices. Chen et al. [89] evaluate the performance of seven edge computing applications in terms of latency. DeepDecision [16] designs a framework to decide whether to offload the object detection task to the edge cloud or do local inference based on the network conditions. Lavea [90] offloads computation between clients and nearby edge nodes to provide low-latency video analytics. VideoStorm [91] and Chameleon [92] achieve higher accuracy video analytics with the same amount of computational resources on the cloud by adapting the video configurations. Liu et al. [105] propose an edge based AR system that is able to achieve 60fps object detection on commercial mobile devices. These works have demonstrated the benefit of using edge offloading in many different applications, which also supports the idea of EdgeSharing.

Device Localization. Accurate device localization is a key component of EdgeSharing system. While GPS provides low-cost position tracking on smart devices, it is widely recognized to be inaccurate in city traffic where large amounts of interference exist. Some other techniques such as inertial sensor [116], wireless signal [117] and visual sensors [104] on the device to improve the localization accuracy. Among these techniques, visual odometry techniques have shown its great performance in tracking devices in the environment where rich visual features exist. Several popular visual SLAM algorithms, including ORB-SLAM2 [104] and LSD-SLAM [118], have shown their superior performance in mapping and localization in small space such as an indoor environment with very few moving objects. These techniques have also been adopted in commercial AR or VR platforms, such as ARKit [68], ARCore [67], Hololens [9] and Oculus Go [100]. In this work, we use the existing ORB-SLAM2 as our localization solution. In order to make it work in outdoor scenarios with dense traffic, we propose the new feature selection method leveraging the resource of the edge cloud.

Adaptive Video Streaming. Adaptive video streaming techniques have been largely exploited to achieve better QoE. Several 360-degree video streaming works [93, 94, 42] also adopt the idea of RoI encoding to reduce the latency and bandwidth consumption of the streaming process. Adaptive video streaming techniques have also been adopted by mobile gaming [35, 12] and virtual reality system [119] to achieve high-quality experience on mobile thin clients. Other video adaptation techniques [96, 97, 98, 99] are also complementary to our work.

5.12 Conclusion

In this paper, we introduce EdgeSharing, a first collaborative localization and object sharing system leveraging the resources of edge cloud platform and the visual inputs from participating mobile clients (e.g., vehicles and pedestrians). In EdgeSharing, the edge cloud uses a 3D feature map of its coverage region to provide accurate localization services to the client devices passing through this region. Besides, EdgeSharing also leverages the computation power on the edge cloud to detect object locations on the images offloaded by participating clients, localizes them in 3D space, and shares them with other clients in the same region. With EdgeSharing installed on the edge cloud, nearby vehicles are able to learn extra object (e.g., traffic participant) locations from the edge cloud, which are outside the vehicles field of view, which improves their situational awareness and safety. To realize this, we propose several optimization techniques. In particular, we propose a Context-Aware Feature Selection method to filter out potential moving objects in the offloaded images to increase the localization accuracy. We also introduce a Collaborative Local Tracking mechanism to significantly reduce the bandwidth consumption of frame transmission by only offload selected keyframes to the edge cloud, while using a lightweight local tracking method to keep track of the location of the client and its detected objects on the end device. In addition, we design a parallel streaming and processing method to enable parallel video streaming and cloud processing, which largely reduces the end-to-end latency of EdgeSharing.

Chapter 6
Conclusion

In this dissertation, we presented novel techniques and system archtechtures to enable high-quality mobile immersive computing with the support of edge cloud. As emerging mobile immersive computing applications, such as Virtual Reality (VR), Augmented Reality (AR), and Mixed Reality (MR), are changing the way human beings interact with the world. Such systems promise to provide unprecedented immersive experiences in the fields of video gaming, education, and healthcare. However, several key processes, such as rendering and object detection, are highly computational intensive, which make them extremely hard to run on mobile devices. Offloading these bottleneck processes to the edge or cloud is also very challenging due to the stringent requirements on high quality and low latency.

In order to achieve high quality and low latency performance of mobile immersive computing applications on mobile thin clients, the system requires to finish the entire offloading pipeline within very short end-to-end latency. Offloading Vision tasks to the edge cloud typically involves several main processes: Sensing, Uplink Transmission, Processing, and Downlink Transmission. These four processes form a round trip from the mobile device to the edge cloud and back to mobile devices. Compared to traditional offloading approaches that execute these processes in a sequential way, we proposed novel video streaming and processing pipelines that can significantly reduce the offloading latency and improve vision quality of VR and AR system. We further introduced advanced vision based algorithms that can largely improve the quality of these applications leveraging the resources on the edge cloud.

To demonstrate the advantage of offloading vision tasks to the edge cloud, this dissertation makes the following fundamental contributions:

- An open remote rendering platform that can enable high-quality untethered VR with low latency on general purpose PC hardware. High-quality VR systems generate graphics data at a data rate much higher than those supported by existing wireless-communication products such as Wi-Fi and 60GHz wireless communication. The necessary image encoding, makes it challenging to maintain the stringent VR latency requirements. To address this issue, we introduces an end-to-end untethered VR system design and open platform that can meet virtual reality latency and quality requirements at 4K resolution over a wireless link. We found that innovative streaming and processing pipelines are able to enable high-quality and low latency VR experience on mobile thin client, with the support of the edge cloud.

- A system that enables high accuracy object detection for commodity AR/MR system running at 60fps. Most existing Augmented Reality (AR)/Mixed Reality (MR) systems are able to understand the 3D geometry of the surroundings but lack the ability to detect and classify complex objects in the real world. Such capabilities can be enabled with deep Convolutional Neural Networks (CNN), but it remains difficult to execute large networks on mobile devices. Offloading object detection to the edge or cloud is also very challenging due to the stringent requirements on high detection accuracy and low end-to-end latency. To address the problem, we introduce a system that employs low latency offloading techniques, decouples the rendering pipeline from the offloading pipeline, and uses a fast object tracking method to maintain detection accuracy. We found that advanced processing pipelines can enable high-quality object detection for mobile augmented reality system, with the support of the edge cloud.

- We build EdgeSharing, an object sharing system leveraging large computational resources at the edge cloud. Beyond the capability of providing object detection service to nearby mobile clients, EdgeSharing holds a real-time 3D feature map of its coverage region on the edge cloud and uses it to provide accurate localization and object sharing service to the client devices passing through this region. By sharing a moving object's location between different camera-equipped devices, it effectively extends the vision of participants beyond their field of view. We further propose several optimization techniques to increase the localization accuracy, reduce the bandwidth consumption and decrease the offloading latency of the system. This work shows that edge cloud is a perfect location to sharing information between mobile clients to improve the quality of users.

In this dissertation, we explore the usage of the edge cloud offloading on various mobile immersive tasks and show the advantage of the edge cloud on high computational tasks with several real-world applications. We hope this dissertation can serve as the guide of designs and implementations for future edge cloud based mobile immersive applications.

References

[1] "Virtual Reality and Augmented Reality Device Sales to Hit 99 Million Devices in 2021," http://www.capacitymedia.com/Article/3755961/VR-and-AR-device-shipments-to-hit-99m-by-2021.

[2] "The reality of VR/AR growth," https://techcrunch.com/2017/01/11/the-reality-of-vrar-growth/.

[3] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2016, pp. 291–304.

[4] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai, "Furion: Engineering high-quality immersive virtual reality on todays mobile devices," in *Proceedings of the 23rd International Conference on Mobile Computing and Networking (MobiCom17). ACM, Snowbird, Utah, USA*, 2017.

[5] "HTC Vive VR," https://www.vive.com/ .

[6] "Oculus Rift VR," https://www.oculus.com/ .

[7] "Samsung Gear VR," http://www.samsung.com/global/galaxy/gear-vr .

[8] "Google Daydream," https://vr.google.com/.

[9] "Microsoft HoloLens," https://www.microsoft.com/en-us/hololens/.

[10] "Magic Leap One," https://www.magicleap.com/.

[11] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM, 2010, pp. 49–62.

[12] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi, "Kahawai: High-quality mobile gaming using gpu offload," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2015, pp. 121–135.

[13] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-time video analytics: The killer app for edge computing," *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[14] W. Zhang, B. Han, and P. Hui, "Low latency mobile augmented reality with flexible tracking," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking.* ACM, 2018, pp. 829–831.

[15] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems.* ACM, 2015, pp. 155–168.

[16] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *INFOCOM. IEEE*, 2018.

[17] O. Abari, D. Bharadia, A. Duffield, and D. Katabi, "Enabling high-quality untethered virtual reality." in *NSDI*, 2017, pp. 531–544.

[18] O. Abari, H. Hassanieh, M. Rodriguez, and D. Katabi, "Millimeter wave communications: From point-to-point links to agile network connections." in *HotNets*, 2016, pp. 169–175.

[19] T. Wei and X. Zhang, "Pose information assisted 60 ghz networks: Towards seamless coverage and mobility support," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '17. New York, NY, USA: ACM, 2017, pp. 42–55. [Online]. Available: http://doi.acm.org/10.1145/3117811.3117832

[20] S. Sur, I. Pefkianakis, X. Zhang, and K.-H. Kim, "Wifi-assisted 60 ghz wireless networks," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '17. New York, NY, USA: ACM, 2017, pp. 28–41. [Online]. Available: http://doi.acm.org/10.1145/3117811.3117817

[21] "TPCAST Wireless Adapter for VIVE," https://www.tpcastvr.com/product.

[22] "DisplayLink XR: Wireless VR Connection," http://www.displaylink.com/vr.

[23] R. Zhong, M. Wang, Z. Chen, L. Liu, Y. Liu, J. Zhang, L. Zhang, and T. Moscibroda, "On building a programmable wireless high-quality virtual reality system using commodity hardware," in *Proceedings of the 8th Asia-Pacific Workshop on Systems.* ACM, 2017, p. 7.

[24] "Nvidia Video Encode and Decode GPU Support Matrix," https://developer.nvidia.com/video-encode-decode-gpu-support-matrix.

[25] "Nvidia's VR ready GPUs," https://www.geforce.com/hardware/technology/vr/supported-gpus.

[26] “OpenVR SDK,” https://github.com/ValveSoftware/openvr.

[27] “Unity game engine,” https://unity3d.com/.

[28] “Google VR SDK for Unity,” https://github.com/googlevr/gvr-unity-sdk.

[29] “Intel Media SDK,” https://software.intel.com/en-us/media-sdk.

[30] “Viking Village,” https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140.

[31] “Nature,” https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977.

[32] “Corridor,” https://assetstore.unity.com/packages/essentials/tutorial-projects/corridor-lighting-example-33630.

[33] “Roller Coaster,” https://assetstore.unity.com/packages/3d/props/exterior/animated-steel-coaster-plus-90147.

[34] “Google Cardboard,” https://vr.google.com/cardboard/ .

[35] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, “Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming,” in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 151–165.

[36] J. R. Lange, P. A. Dinda, and S. Rossoff, “Experiences with client-based speculative remote display.” in *USENIX Annual Technical Conference*, 2008, pp. 419–432.

[37] B. Wester, P. M. Chen, and J. Flinn, “Operating system support for application-specific speculation,” in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 229–242.

[38] B. Reinert, J. Kopf, T. Ritschel, E. Cuervo, D. Chu, and H.-P. Seidel, “Proxy-guided image-based rendering for mobile devices,” in *Computer Graphics Forum*, vol. 35, no. 7. Wiley Online Library, 2016, pp. 353–362.

[39] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, “Towards efficient edge cloud augmentation for virtual reality mmogs,” in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ser. SEC ’17. New York, NY, USA: ACM, 2017, pp. 8:1–8:14. [Online]. Available: http://doi.acm.org/10.1145/3132211.3134463

[40] “Nvidia Video Codec,” https://developer.nvidia.com/nvidia-video-codec-sdk.

[41] "Intel Quick Sync," https://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html.

[42] F. Qian, L. Ji, B. Han, and V. Gopalakrishnan, "Optimizing 360 video delivery over cellular networks," in *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges.* ACM, 2016, pp. 1–6.

[43] E. Kuzyakov and D. Pio, "Next-generation video encoding techniques for 360 video and vr.(2016)," *Online: https://code. facebook. com/posts/1126354007399553/nextgeneration-video-encoding-techniques-for-360-video-and-vr*, 2016.

[44] M. Berning, T. Yonezawa, T. Riedel, J. Nakazawa, M. Beigl, and H. Tokuda, "parnorama: 360 degree interactive video for augmented reality prototyping," in *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication.* ACM, 2013, pp. 1471–1474.

[45] M. Kressin, "Method and apparatus for video conferencing with audio redirection within a 360 degree view," Aug. 16 2002, uS Patent App. 10/223,021.

[46] A. Pavel, B. Hartmann, and M. Agrawala, "Shot orientation controls for interactive cinematography with 360 video," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology.* ACM, 2017, pp. 289–297.

[47] C.-L. Fan, J. Lee, W.-C. Lo, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu, "Fixation prediction for 360 video streaming to head-mounted displays," 2017.

[48] K. Müller, H. Schwarz, D. Marpe, C. Bartnik, S. Bosse, H. Brust, T. Hinz, H. Lakshman, P. Merkle, F. H. Rhee *et al.*, "3d high-efficiency video coding for multi-view video and depth data," *IEEE Transactions on Image Processing*, vol. 22, no. 9, pp. 3366–3378, 2013.

[49] Y. Chen, Y.-K. Wang, K. Ugur, M. M. Hannuksela, J. Lainema, and M. Gabbouj, "The emerging mvc standard for 3d video services," *EURASIP Journal on Applied Signal Processing*, vol. 2009, p. 8, 2009.

[50] S. Shimizu, M. Kitahara, H. Kimata, K. Kamikura, and Y. Yashima, "View scalable multiview video coding using 3-d warping with depth map," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 11, pp. 1485–1495, 2007.

[51] A. Vetro, T. Wiegand, and G. J. Sullivan, "Overview of the stereo and multiview video coding extensions of the h. 264/mpeg-4 avc standard," *Proceedings of the IEEE*, vol. 99, no. 4, pp. 626–642, 2011.

[52] "Android MediaCodec," https://developer.android.com/reference/android/media/MediaCode

[53] "Vive Power Draw Test Results," https://www.reddit.com/r/Vive/comments/51ir1h/vive$_{pow}$

[54] S. K. Saha, T. Siddiqui, D. Koutsonikolas, A. Loch, J. Widmer, and R. Sridhar, "A detailed look into power consumption of commodity 60 ghz devices," in *A World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2017 IEEE 18th International Symposium on.* IEEE, 2017, pp. 1–10.

[55] "H.264 HD Video Decoder Power Consumption," https://www.soctechnologies.com/ip-cores/ip-core-h264-decoder/.

[56] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He, "Detectron," https://github.com/facebookresearch/detectron, 2018.

[57] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the h. 264/avc video coding standard," *IEEE Transactions on circuits and systems for video technology*, vol. 13, no. 7, pp. 560–576, 2003.

[58] "Netflix DrivingPOV Video," https://media.xiph.org/video/derf/Chimera/Netflix$_{DrivingPOV_{C}opy}$

[59] "Intersection over Union (IoU)," http://cocodataset.org/detection-eval/.

[60] "Object Keypoint Similarity (OKS)," http://cocodataset.org/keypoints-eval/.

[61] "Nvidia TensorRT," https://developer.nvidia.com/tensorrt/.

[62] "Nvidia JetPack," https://developer.nvidia.com/embedded/jetpack/.

[63] "Nvidia Multimedia API," https://developer.nvidia.com/embedded/downloads/.

[64] "Fast INT8 Inference with TensorRT 3," https://devblogs.nvidia.com/int8-inference-autonomous-vehicles-tensorrt/.

[65] "Xiph Video Dataset," https://media.xiph.org/video/derf/.

[66] "Detection Evaluation for Microsoft COCO," http://cocodataset.org/detection-eval/.

[67] "Google ARCore," https://developers.google.com/ar/.

[68] "Apple ARKit," https://developer.apple.com/arkit/.

[69] "Vuforia Object Recognition," https://library.vuforia.com/articles/Training/Object-Recognition/.

[70] P. Jain, J. Manweiler, and R. Roy Choudhury, "Overlay: Practical mobile augmented reality," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2015, pp. 331–344.

[71] ——, "Low bandwidth offload for mobile ar," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies.* ACM, 2016, pp. 237–251.

[72] W. Zhang, B. Han, and P. Hui, "On the networking challenges of mobile augmented reality," in *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network.* ACM, 2017, pp. 24–29.

[73] W. Zhang, B. Han, P. Hui, V. Gopalakrishnan, E. Zavesky, and F. Qian, "Cars: Collaborative augmented reality for socialization," 2018.

[74] H. Qiu, F. Ahmad, F. Bai, M. Gruteser, and R. Govindan, "Avr: Augmented vehicular reality," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2018, pp. 81–95.

[75] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," in *IEEE CVPR*, vol. 4, 2017.

[76] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[77] J. Dai, Y. Li, K. He, and J. Sun, "R-fcn: Object detection via region-based fully convolutional networks," in *Advances in neural information processing systems*, 2016, pp. 379–387.

[78] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision.* Springer, 2016, pp. 21–37.

[79] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Computer Vision (ICCV), 2017 IEEE International Conference on.* IEEE, 2017, pp. 2980–2988.

[80] R. Ranjan, V. M. Patel, and R. Chellappa, "Hyperface: A deep multi-task learning framework for face detection, landmark localization, pose estimation, and gender recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017.

[81] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[82] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Proceedings of the 15th International Conference on Information Processing in Sensor Networks.* IEEE Press, 2016, p. 23.

[83] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2016, pp. 123–136.

[84] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[85] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2017, pp. 82–95.

[86] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2017, pp. 68–81.

[87] R. LiKamWa, Y. Hou, J. Gao, M. Polansky, and L. Zhong, "Redeye: analog convnet image sensor architecture for continuous mobile vision," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 255–266.

[88] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, "On-demand deep model compression for mobile devices: A usage-driven model selection framework," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2018, pp. 389–400.

[89] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky *et al.*, "An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing.* ACM, 2017, p. 14.

[90] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing.* ACM, 2017, p. 15.

[91] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman, "Live video analytics at scale with approximation and delay-tolerance," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation.* USENIX Association, 2017, pp. 377–392.

[92] J. Jiang, G. Ananthanarayanan, P. Bodk, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *Proceedings of the 2018 ACM SIGCOMM Conference.* ACM, 2018.

[93] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han, "Rubiks: Practical 360-degree streaming for smartphones," pp. 482–494, 2018.

[94] X. Xie and X. Zhang, "Poi360: Panoramic mobile video telephony over lte cellular networks," in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies.* ACM, 2017, pp. 336–349.

[95] X. Liu, Q. Xiao, V. Gopalakrishnan, B. Han, F. Qian, and M. Varvello, "360 innovations for panoramic video streaming," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks.* ACM, 2017, pp. 50–56.

[96] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM, 2017, pp. 197–210.

[97] H. Yeo, S. Do, and D. Han, "How will deep learning change internet video delivery?" in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks.* ACM, 2017, pp. 57–64.

[98] B. Han, F. Qian, L. Ji, and V. Gopalakrishnan, "Mp-dash: Adaptive video streaming over preference-aware multipath," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies.* ACM, 2016, pp. 129–143.

[99] Y. Sun, X. Yin, J. Jiang, V. Sekar, F. Lin, N. Wang, T. Liu, and B. Sinopoli, "Cs2p: Improving video bitrate selection and adaptation with data-driven throughput prediction," in *Proceedings of the 2016 ACM SIGCOMM Conference.* ACM, 2016, pp. 272–285.

[100] "Oculus Go," https://www.oculus.com/go/.

[101] "Google Map's AR App," https://www.theverge.com/2019/2/10/18219325/google-maps-augmented-reality-ar-feature-app-prototype-test.

[102] "TensorFlow Lite Performance," https://www.tensorflow.org/lite/models.

[103] F. Ahmad, H. Qiu, X. Liu, F. Bai, and R. Govindan, "Quicksketch: Building 3d representations in unknown environments using crowdsourcing," in *2018 21st International Conference on Information Fusion (Fusion).* IEEE, 2018, pp. 2314–2321.

[104] R. Mur-Artal and J. D. Tardós, "Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.

[105] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *MobiCom.* ACM, 2019.

[106] "OpenCV," https://opencv.org/.

[107] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

[108] Q. Xu, T. Mak, J. Ko, and R. Sengupta, "Vehicle-to-vehicle safety messaging in dsrc," in *Proceedings of the 1st ACM international workshop on Vehicular ad hoc networks*. ACM, 2004, pp. 19–28.

[109] F. Bai and H. Krishnan, "Reliability analysis of dsrc wireless communication for vehicle safety applications," in *2006 IEEE intelligent transportation systems conference*. IEEE, 2006, pp. 355–362.

[110] N. Lu, N. Cheng, N. Zhang, X. Shen, and J. W. Mark, "Connected vehicles: Solutions and challenges," *IEEE internet of things journal*, vol. 1, no. 4, pp. 289–299, 2014.

[111] K. Abboud, H. A. Omar, and W. Zhuang, "Interworking of dsrc and cellular network technologies for v2x communications: A survey," *IEEE transactions on vehicular technology*, vol. 65, no. 12, pp. 9457–9470, 2016.

[112] J. Choi, V. Va, N. Gonzalez-Prelcic, R. Daniels, C. R. Bhat, and R. W. Heath, "Millimeter-wave vehicular communication to support massive automotive sensing," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 160–167, 2016.

[113] V. Va, T. Shimizu, G. Bansal, R. W. Heath Jr *et al.*, "Millimeter wave vehicular communications: A survey," *Foundations and Trends® in Networking*, vol. 10, no. 1, pp. 1–113, 2016.

[114] M. Giordani, A. Zanella, and M. Zorzi, "Millimeter wave communication in vehicular networks: Challenges and opportunities," in *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*. IEEE, 2017, pp. 1–6.

[115] P. Gomes, F. Vieira, and M. Ferreira, "The see-through system: From implementation to test-drive," in *2012 IEEE vehicular networking conference (VNC)*. IEEE, 2012, pp. 40–47.

[116] C. Bo, X.-Y. Li, T. Jung, X. Mao, Y. Tao, and L. Yao, "Smartloc: Push the limit of the inertial sensor based metropolitan localization using smartphone," in *Proceedings of the 19th annual international conference on Mobile computing & networking*. ACM, 2013, pp. 195–198.

[117] J. Wang, N. Tan, J. Luo, and S. J. Pan, "Woloc: Wifi-only outdoor localization using crowdsensed hotspot labels," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[118] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *European conference on computer vision*. Springer, 2014, pp. 834–849.

[119] E. Cuervo, K. Chintalapudi, and M. Kotaru, “Creating the perfect illusion: What will it take to create life-like virtual reality headsets?” 2018.