DISTRIBUTED PLACEMENT AND RESOURCE ORCHESTRATION OF REAL-TIME EDGE COMPUTING APPLICATIONS

By

WUYANG ZHANG

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

Dipankar Raychaudhuri

And approved by

_____

_____

_____

New Brunswick, New Jersey

January 2021

ABSTRACT OF THE DISSERTATION

Distributed Placement and Resource Orchestration of Real-time Edge Computing Applications

by Wuyang Zhang

dissertation Director: Prof. Dipankar Raychaudhuri

The recent emergence of a broad class of deep learning based augmented and virtual reality applications motivates the need for real-time mobile cloud services. These real-time, mobile applications involve intensive computation over large data sets, and are generally required to provide low end-to-end latency for acceptable quality-of-experience at the end-user. Limited battery life, computation and storage capacity constraints inherent to mobile devices mean that application execution must be offloaded to cloud servers, which then return processed results to the mobile devices through the Internet. When cloud servers reside in remote data centers, end-to-end communication may translate into long delays characteristic of multi-hops transmissions over the Internet. Moving cloud computing to the edge of a network has helped to lessen these otherwise unacceptable delays while leveraging the benefits of a high-performance cloud. While this improvement is significant, there are several technical challenges that need to be addressed in order to achieve low end-to-end latency.

In the thesis, we aim to address the following problems. First, how to efficiently distribute the real-time application between the mobile device, edge servers and data center to meet the latency constraints? Second, as the edge cloud architecture is inherently distributed and heterogeneous, how to perform resource allocation and task orchestration in a latency-

constrained design? Finally, existing cloud computing solutions often assume there exists a dedicated and powerful server, to which an entire job can be offloaded. In reality, we may not be able to find such a server, which motivates an investigation of techniques for use of multiple less powerful edge servers to achieve a parallel job offloading.

In the first part of the thesis, we take virtual reality massively multiplayer online games (VR MMOGs) as a driving example and design a hybrid service architecture that achieves a good distribution of workload between the mobile devices, edge clouds, core cloud for low latency and global user scalability, We also propose an efficient service placement algorithm based on a Markov decision process to dynamically place a user's gaming service on edge clouds. This dynamic service placement can help to further reduce the latency under user mobility.

In the second part of this thesis, we present the design and implementation of a *latency-aware* edge computing platform, aiming to minimize the end-to-end latency for edge applications. The proposed platform is built on Apache Storm, an open source distributed computing framework, and consists of multiple edge servers with heterogeneous computation (including both GPUs and CPUs) and networking resources. Central to our platform is an orchestration framework that breaks down an edge application into Storm tasks as defined by a directed acyclic graph (DAG) and then maps these tasks onto heterogeneous edge servers for efficient execution.

In the last part of this thesis, we take a closer look at these computing intensive deep learning-based computer vision jobs. We propose to partition the video frame and offload the partial inference tasks to multiple servers for parallel processing. This work presents the design and implementation of Elf, a framework to accelerate the mobile deep vision applications with any server provisioning through parallel offloading. Elf employs a recurrent region proposal prediction algorithm, a region proposal centric frame partitioning, and a multi-offloading scheme.

ACKNOWLEDGEMENTS

Through the journey at WINLAB over the past six years, I have received tremendous help and guidance which finally lead me to complete this dissertation.

I wish to extend my deepest gratitude to my advisors Prof. Dipankar Raychaudhuri and Prof. Yanyong Zhang for the invaluable guidance and support at every step through the journey. The PhD training is never about doing the research itself. Prof. Ray taught me how to identify a meaningful and practical problem with critical thinking, how to narrow it down to a proper scope from a big picture to move forward, how to present a work and deliver its impacts, how to always stand on the front lines with judgement and courage, and how to collaborate efficiently with a team. His insights and wisdom always drive me across this amazing journey. One of his sayings "Nothing is there naturally and you need to explore and understand how people gradually develop a technique from the ground" motivates me to always think about the backend insights of techniques. Prof. Zhang encouraged me, back to 2015, to join WINLAB as a PhD student, which opened the door to my later PhD journey. I could not thank more for her hands-on training with respect to how to extend an initial research idea to a solid and practical system work, which is the foundation and the core of proceeding this dissertation work. She taught me how to better write each sentence in a research paper and how to better speak every single word in a presentation. Also, her research insights always push me beyond the knowledge boundary, leading me to further explore and extend a research work to be a more rewarding conclusion. I would like to express my admiration to Prof. Roy Yates who always inspired me by the depth of his technical insight, and his passion about exploring and solving problems. It is invaluable to have the opportunity of discussing with and learning from him. I also would like to thank Dr. T. V. Lakshman for his thoughtful and generous feedback towards the dissertation work. My infinite gratitude to all the committee members.

This journey towards Ph.D. would not have been such amazing and smooth without

the support and company from my mentors, friends, and collaborators: Ivan Seskar, Dr. Richard E. Howard, Dr. Yunxin Liu, Prof. Marco Gruteser, Prof. Richard Martin, Prof. Zoran Gajic, Prof. Janne Lindqvist, Prof. Kristin Dana, Prof. Hana Godrich, Prof. Maria Striki, Dr. Charlie Zhang, Dr. Russell Ford, Dr. Yibo Zhu, Dr. Shiqiang Wang, Dr. Kai Su, Dr. Francesco Bronzino, Dr. Feixiong Zhang, Dr. Yi Hu, Dr. Jiachen Chen, Dr. Bin Cheng, Dr. Sugang Li, Dr. Shreyasee Mukherjee, Dr. Zhenhua Jia, Dr. Luyang Liu, Dr. Jing Zhong, Dr. Sumit Maheshwari, Dr. Xiaoran Fan, Dr. Parishad Karimi, Hongyu Li, Dragoslav Stojadinovic, Shalini Choudhury, Prasad Netalkar, Vishakha Ramani, Mohsen Rajabpour.

I would like to acknowledge the staff members at WINLAB for their generous help: Noreen DeCarlo, Elaine Connor, Lisa Musso, Jennifer Shane, Michael Sherman, Jakub Kolodziejski.

Last but not least, I thank my family for the support and continuous encouragement throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Background

The recent emergence of a broad class of deep learning based augmented and virtual reality applications motivates the need for real-time mobile cloud services. These real-time, mobile applications involve intensive computation over large data sets, and are generally required to provide low end-to-end latency for acceptable quality-of-experience at the end-user. Limited battery life, computation and storage capacity constraints inherent to mobile devices mean that application executions must be offloaded to cloud servers, which then return processed results to the mobile devices through the Internet. When cloud servers reside in remote data centers, end-to-end communication may translate into long delays characteristic of multi-hops transmissions over the Internet. For a client instance in New Jersey which connects to Amazon EC2 cloud servers located in West Virginia, Oregon and California, the round-trip latency *alone* is 17, 104 and 112ms, with achievable bandwidths of 50, 18 and 16Mbps, respectively. In order to support these emerging edge applications, edge cloud computing has been proposed as a viable solution [1, 2, 3], which moves the computing towards the network edge to reduce the response latency while also avoiding edge-to-core network bandwidth constraints.

1.2 Challenges of Deploying and Orchestrating Real-time Edge Applications

Despite the earlier and ongoing work on various aspects of edge computing, we intend to consider this problem in a fully latency-constrained design. Such a perspective introduces the following challenges.

First, how to efficiently distribute the function of real-time applications into mobile

devices, edge servers and central clouds in order to fully utilize the computing resources while to provide a global user scalability.

We take virtual reality massively multiplayer online games (VR-MMOGs) as a representative example and it has been widely adopted as a killer application in the era of edge computing. VR-MMOGs can leverage edge cloud computing to meet their QoS requirements [4], but simply moving all the gaming tasks to the edge makes it harder for players to share games across the network since it is difficult to synchronize users' profiles and game worlds among widely distributed edge clouds.

Second, the problem of how to efficiently deploy these new edge applications within an edge cloud has not been systematically studied. Duplicating the successful cloud computing design will not work for the edge applications. This is mainly due to the highly heterogeneous nature of edge clouds. Unlike central clouds, edge clouds are often comprised of heterogeneous computation nodes with widely diverse network bandwidths. For example, the studies in [5, 6], assume the computation nodes and their interconnects are relatively homogeneous in central clouds, while the edge servers considered in [7, 8] exhibit widely varying capabilities. Thus, an important new challenge associated with edge clouds is that of efficiently orchestrating these heterogeneous resources in order to meet application latency constraints.

Finally, existing methods only consider offloading tasks to a single server, assuming that the server has sufficient resources to finish the tasks in time. Nevertheless, in practice, a single edge server is equipped with costly hardware, for example, Intel Xeon Scalable Processors with Intel Deep Learning Boost or NVIDIA EGX A100, which are typically shared by multiple clients (i.e., multi-tenant environment). Moreover, the heterogeneous resource demands of applications running on edge servers and highly dynamic workloads by mobile users lead to resource fragmentation. If the fragmentation cannot be efficiently utilized, it may produce significant resource waste across edge servers. To this point, in order to meet the requirements of real-time applications, especially those deep mobile vision

applications with heterogeneous edge computing resources, *it is advantageous to offload smaller inference tasks in parallel to multiple edge servers*. This mechanism can benefit many real-world deep vision tasks, including multi-people key point detection for AR applications and multi-object mask detection for autonomous driving tasks, where objects can be distributed to different servers for parallel task processing.

1.3 Proposed Solutions

First, we take a closer look at the game flows in VR-MMOGs in this study. We discover that the events initiated by the players can generally be classified into two categories based on the tolerance levels of response latency. The response to the user's local *view change events* (which has effect only on his/her screen, *e.g.*, mouse movements, map scrolls, selection of a game object without changing it) has much more stringent timeliness requirements compared to the response to the *game events* (which involves global game state updates, *e.g.*, updated scores, bleeding on shot targets). In VR-MMOGs, view change events are a lot more frequent compared to non-VR-MMOGs since the orientation of the VR device is changing all the time, and it requires immediate (~20ms) feedback on the screen. In comparison, players can tolerate more than 100ms latency towards game events, and in some games this value can be as large as 1 second [9].

Based on the fundamental differences between view change and game events, we believe that they should be treated differently in order to provide the best VR-MMOG user experience. In this paper, we propose EC+, an architecture for Edge Cloud augmented VR-MMOGs. EC+ exploits edge clouds for view change events rendering to satisfy the ultra low delay requirement. The rendering on edge clouds can also provide higher resolution and refresh rate compared to the rendering on the mobile devices since edge clouds have more computation power and they are close to the players. As for the game events, EC+ still uses a central cloud to manage global game and game logic to provide a wide coverage with minimal overhead in maintaining the consistency of game states. In addition to proposing

the EC+ architecture, we also devise an efficient algorithm that selects an edge cloud for each player to handle player mobility and dynamic edge cloud workload. Modeled upon a Markov decision process (MDP), the proposed algorithm periodically makes edge cloud placement decisions, taking into consideration the overall QoS (the latency and the bandwidth between client and edge, *and* between edge and game server), mutual impact among players (*e.g.*, edge load, game world sharing), and player mobility patterns. To ensure the feasibility, we come up with the approaches that can reduce the algorithm complexity in both storage and execution time. We also design a mechanism to ensure seamless handoff when a gaming service is migrated from one edge cloud to another.

To address the second challenge, we set out to build and test such an edge computing orchestration platform. Our design is driven by the requirement of deploying and accelerating this new class of edge applications – e.g., processing large volumes of data such as video data generated by mobile/IoT sensors (including 3D cameras) in real time. We first build an edge cloud testbed that consists of four different CPU settings, four different GPU settings and five different link bandwidth settings. On these nodes, we run Apache Storm [10] as the baseline distributed edge computing framework. Apache Storm provides real-time support, but has an implicit assumption that the underlying computing/networking resources are homogeneous. Also, it does not provide proactive support for GPUs. In this work, we address these shortcomings. Note that our platform design is not specific to Apache Storm. In fact, it can easily interface with other distributed computing frameworks such as Apache Flink.

The design of Hetero-Edge mainly focuses on distributed resource orchestration for edge computing. Specifically, we intend to answer the following important questions. Firstly, if an edge cloud consists of both GPUs and CPUs, when do we serve requests on GPUs and when do we use CPUs? How do we partition our jobs so that we can most efficiently utilize the available resources? Secondly, after partitioning the job to several pipelined and parallel tasks, how can we map them to appropriate computing nodes (including both GPUs and

CPUs) to minimize their overall latency? Thirdly, how can we effectively prevent a parallel task from completing significantly slower than its peers and becoming a straggler [11]? Since resources of edge clouds are highly diverse, the likelihood of having stragglers is much higher than in a homogeneous setting. By carefully studying these questions, we devise the resource orchestration schemes in Hetero-Edge, featuring: (1) matching a task's resource demand with the underlying node's resource availability, (2) matching a task's workload level with the underlying node's resource availability, and (3) suitably splitting work on processors with vastly different processing power (GPUs vs CPU).

To address the aforementioned challenges, we propose and design **ELF**[1], a framework to accelerate high-resolution mobile deep vision offloading in heterogeneous client and edge server environment, by distributing the computation to available edge servers adaptively. ELF adopts three novel techniques to enable both low latency and high quality of service. To eliminate the accuracy degradation caused by the frame partitioning, we first propose a content-aware frame partitioning method. It is promoted by a fast recurrent region proposal prediction algorithm with an attention-based LSTM network that predicts the content distribution of a video frame. Additionally, we design a region proposal indexing algorithm to keep track of the motion across frames and a low resolution compensation solution to handle new objects when first appear. Both work jointly to help understand frame contents more accurately. Finally, ELF adopts lightweight approaches to estimate the resource capacity of each server and dynamically creates frame partitions based on the resource demands to achieve load balance. Overall, ELF is designed as a plug-and-play extension to the existing deep vision networks and requires minimal modifications at the application level.

[1]Elf is a small creature in stories usually described as smart, agile, and has magic power

CHAPTER 2
DISTRIBUTED PLACEMENT OF EDGE COMPUTING APPLICATIONS

2.1 Introduction

The rapid rise of Massively Multiplayer Online Games (MMOGs) calls for gaming platforms that support ultra low latency and intensive 3D world rendering [12]. With emerging Virtual Reality (VR) technologies (*e.g.*, HTC Vive, Oculus, Google Cardboard), VR based MMOGs, *i.e.*, VR-MMOGs, are quickly looming on the surface, demanding even faster gaming interaction and image rendering. In addition, VR-MMOGs place a new set of requirements on the underlying system design due to a union of VR and MMOGs: *1*) the need of simultaneously rendering two images with different perspectives for both eyes, *2*) the need of supporting wider angles of visual field (120 degree compared to 60 degree in normal games), *3*) the need of providing ultra-low latency that prevents people from having motion sickness (<30ms compared to 100–1000ms in normal games), and *4*) the need of rendering with a higher refresh rate (60–120 frames per second (fps), compared to 24–60fps in normal games).

Meanwhile, to enable players with "thin" mobile devices (*e.g.*, smartphones, pads, TVs) to enjoy high-quality gaming, the paradigm of cloud gaming (also known as gaming on-demand [12]) that delivers games from the cloud to players has been proposed and developed. GeForce NOW (for Nvidia Shield clients [13]), PlayStation NOW [14], GamingAnywhere [15], as the industrial pioneers of cloud gaming, are drawing substantial players to move from traditional gaming to this cloud-based paradigm [16]. These cloud gaming services perform game logic computation on cloud servers and stream encoded views, over the Internet, to apps on heterogeneous mobile devices. Cloud gaming allows users to access games anywhere via mobile devices, without periodically upgrading their hardware

to satisfy the ever increasing hardware demands. Also, cloud gaming significantly reduces the energy consumption incurred by the heavyweight rendering tasks on mobile devices. Finally, high resolution frames that are cumbersome, if not impossible, to be generated locally can be streamed from the cloud.

While cloud gaming can provide a multitude of benefits to players, the service providers face a set of serious challenges to ensure the demanded quality of service (QoS). The first and foremost challenge stems from the *latency* between cloud servers and players. Responses that are not fast enough in VR gaming can result in dissatisfying game experiences and may further contribute to player motion sickness. According to [9], ~20ms is an acceptable end-to-end latency for such applications. A latency of 50ms can still support responsive services, but with noticeable lagging. Nevertheless, the average network latency in today's Internet between the Amazon EC2 cloud and mobile devices is more than 80ms [12], which already exceeds the tolerable latency level even without performing any computation.

The second challenge is the high *bandwidth* demand of MMOGs – they generally require a bandwidth of 100Mbps to stream VR games with 1080p resolution at 60 fps [17], while the wireless Internet bandwidth available to a mobile device is 2Mbps [18]. Network jitters cause decreased refreshing rates and increased packet delays, both worsening the user experiences. Moreover, users with mobile devices are more inclined to move around compared to those connected to fixed hosts, and some may even play while sitting in cars or trains. The disconnections caused by changing network access points also lead to deteriorated gaming performance.

Edge cloud computing [19, 18, 20] moves cloud services closer to the users. Naturally, it has the potential to bring down otherwise unacceptable network delays and to provide high downlink bandwidth while taking advantage of high performance computing resources. VR-MMOGs can leverage edge cloud computing to meet their QoS requirements [4], but simply moving all the gaming tasks to the edge makes it harder for players to share games across the network since it is difficult to synchronize users' profiles and game worlds among

widely distributed edge clouds.

To address the challenges, we take a closer look at the game flows in VR-MMOGs in this study. We discover that the events initiated by the players can generally be classified into two categories based on the tolerance levels of response latency. The response to the user's local *view change events* (which has effect only on his/her screen, *e.g.*, mouse movements, map scrolls, selection of a game object without changing it) has much more stringent timeliness requirements compared to the response to the *game events* (which involves global game state updates, *e.g.*, updated scores, bleeding on shot targets). In VR-MMOGs, view change events are a lot more frequent compared to non-VR-MMOGs since the orientation of the VR device is changing all the time, and it requires immediate (~20ms) feedback on the screen. In comparison, players can tolerate more than 100ms latency towards game events, and in some games this value can be as large as 1 second [9].

Based on the fundamental differences between view change and game events, we believe that they should be treated differently in order to provide the best VR-MMOG user experience. In this paper, we propose EC+ an architecture for Edge Cloud augmented VR-MMOGs.

EC+ exploits edge clouds for view change events rendering to satisfy the ultra low delay requirement. The rendering on edge clouds can also provide higher resolution and refresh rate compared to the rendering on the mobile devices since edge clouds have more computation power and they are close to the players. As for the game events, EC+ still uses a central cloud to manage global game and game logic to provide a wide coverage with minimal overhead in maintaining the consistency of game states.

In addition to proposing the EC+ architecture, we also devise an efficient algorithm that selects an edge cloud for each player to handle player mobility and dynamic edge cloud workload. Modeled upon a Markov decision process (MDP), the proposed algorithm periodically makes edge cloud placement decisions, taking into consideration the overall QoS (the latency and the bandwidth between client and edge, *and* between edge and game

server), mutual impact among players (*e.g.*, edge load, game world sharing), and player mobility patterns. To ensure the feasibility, we come up with the approaches that can reduce the algorithm complexity in both storage and execution time. We also design a mechanism to ensure seamless handoff when a gaming service is migrated from one edge cloud to another.

Finally, we summarize our contributions below:

- A study of the new requirements of VR-MMOGs, explaining why client-centric and cloud gaming fall short in fulfilling these requirements (§section 2.3);
- A design of a hybrid architecture that leverages both edge and central clouds to satisfy the latency and throughput requirements of VR-MMOGs (§section 2.4);
- A general edge cloud placement algorithm which intends to maximize the game performance for a large number of players with different bandwidth, latencies, edge loads, game world sharing scenarios (§section 2.5); and
- A comprehensive evaluation using both synthetic and real-world topologies to quantify the benefit of the proposed architecture and algorithm (§section 2.6).

2.2 Related Work

We first present the background of (VR-)MMOGs and then review the existing solutions that could potentially support VR-MMOGS, namely, cloud centric gaming and edge cloud assisted gaming.

2.2.1 Massively Multiplayer Online Gaming Meets Virtual Reality

Virtual reality (VR) has been supported by multiple industrial products like PlayStation.VR [14], HTC Vive [21], Oculus [9], Google Cardboard [22]. VR devices extract players' sensory (*e.g.*, eyes and ears) information and accordingly "hijack" the natural

stimulation with the artificial stimulation from a virtual world generator [23]. VR technology remarkably hands a highly immersive experience with substantial depth perceptions. Playing MMOGs through VR devices is the natural next step [24]. In fact, the VR versions of several popular MMOGs have been developed, *e.g.* World of Warcraft, Minecraft Multiplayer, and Grand Theft Auto V Online [25].

As much as VR-MMOGs generate excitement in the gaming community, it also poses the unprecedented demands and the challenges, especially with respect to providing ultra low latency and a high refresh rate, on the underlying system design.

This paper aims to design an architecture that is carefully tuned to satisfy these demands.

2.2.2 Supporting Gaming through Cloud

Many cloud gaming solutions [26, 27, 13, 14, 15] have been proposed to reduce the computation and/or storage requirements on game terminals. These solutions can be broadly classified into two categories: file-streaming games and video-streaming games. In file-streaming gaming (*i.e.*, progressive downloading), a small portion of the game is initially downloaded to a user device. While this portion runs, the rest of game can be downloaded and installed in parallel [26, 27]. While it is true that file-streaming games can reduce the game boot time and the storage required on game devices, it still requires devices to process game logic and perform 3D rendering. Therefore, it is difficult to support VR-MMOGs on mobile devices like Google cardboard.

Video-streaming games, on the other hand, place all the processing in a cloud, including user profile management, game update calculation, game frame rendering and encoding, *etc.* The cloud then streams the encoded frames to players over the Internet [13, 14, 15]. This mechanism enables players to enjoy high-quality games even on the devices with limited computing and power resources (*e.g.*, smartphones, pads, TVs) – the game terminals merely need to decode frames just like watching a Youtube video. With the advent of GPU grids [28], game processing has become more efficient than purely using CPU-based clouds.

Many studies have been conducted to further improve the user experience of video-streaming games. In [29] video games are classified into CPU-consuming and memory-consuming types to increase the resource utilization in a cloud. Lee *et al.* [30] use High Efficiency Video Coding (HEVC) to reduce the bandwidth requirement by 59% without compromising video quality. Solutions in [31, 32] reduce the response time by predicting possible game updates and rendering speculative frames ahead of time.

The main disadvantage of cloud gaming, however, involving both file-streaming and video-streaming gaming, is the need to transmit a large amount of data, either a game itself or game frames, through the core network. Due to the massively multiplexing nature of the Internet design, the available bandwidth and latency between a cloud and a player may change dramatically over time [33]. This often leads to jitters, lags, frame drops or low-quality frames (glitches) in the middle of a game, and resulting in a poor gaming experience, especially for video-streaming games [34].

2.2.3 Edge Cloud Computing

Edge cloud (or fog computing [19, 18, 20]) moves computing and storage closer to clients, promising to deliver shorter latency and higher bandwidth. It can benefit applications which require high bandwidth, low latency but without large-scale aggregation, *e.g.*, preprocessing of surveillance camera data [35], image classification [36], smart traffic light control [19], *etc.* Edge cloud computing also has the potential to better serve (VR-)MMOGs, if carefully designed to solve the challenge of large-scale aggregation (*i.e.*, game state synchronization among all players).

Work in [37, 38, 39] proposes peer-to-peer (P2P) MMOGs wherein the delegate of the players (game consoles or edge cloud servers) form a P2P network to synchronize gaming states shared among the players directly. This distributed architecture incurs a large amount of synchronization overhead and may potentially limit the number of concurrent players in a game. To address these challenges, the authors in [40] propose to move the rendering

process from a cloud to idle desktops which are close to clients. Indeed, this technique can reduce the network latency by 20% and reduce the network traffic volume by 90%. Nevertheless, any user events, including those only need local updates, are dispatched to the center cloud altogether. This solution performs well for non-VR games because there aren't many local update events in these games. However, VR-MMOGs have a significantly larger number of such events, which makes this solution ill-suited.

2.2.4 Service Migration Among Clouds

To satisfy specific application requirements, tasks have to be initially placed on assigned machines of a cloud [41, 42, 43]. Later, the tasks may be migrated (reassigned) to under-utilized machines to meet particular optimization targets. Concerning where to migrate, distinct migration strategies have been proposed on the basis of expected optimal targets. Lim *et al.* [44] propose a performance aware migration schema in respond to dynamic server workloads. Ghribi *et al.* [45] investigate an energy efficient scheduling to achieve significant energy savings. With respect to how to migrate, Douglis [46] comes up with a process migration schema that moves a process from a source machine to a destination machine, which encounters the difficulty of separating a process from its operating system. Clark *et al.* [47] design a live virtual machine (VM) migration mechanism that effectively overcomes this barrier. Yet, a core cost of VM migration is a short downtime during which an application is compulsively paused. The downtime changes among different applications, ranging from several milliseconds to several seconds [48]. To reduce the downtime, Jin *et al.* [49] investigate a memory compression approach and Ha *et al.* [50] study a pipelining processing of VM migration .

2.3 A Closer Look at VR-MMOGs

A VR-MMOG is essentially a large-scale event driven system. Even though each VR-MMOG may have unique and complicated game logic, they do have similar game events

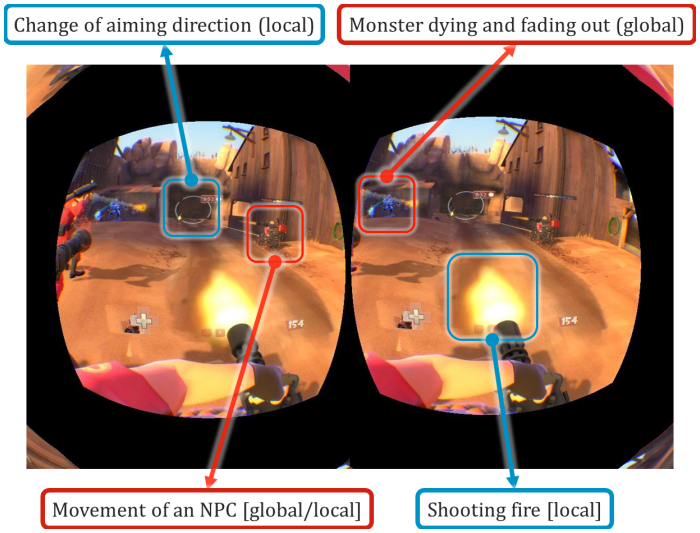


Fig. 2.1 A game view usually contains both view change updates local to a player (*e.g.*, change of look direction, immediate feedback on action like firing) and game world updates synchronized among all players (*e.g.*, monsters' dying, non-player characters' actions)

and share an identical underlying game flow. In this section, we first study the new features and challenges of VR-MMOGs and then present several existing MMOG models to help understand why they fail to satisfy these new requirements.

2.3.1 View Change Events vs Game Events

Any game flow begins with a particular user event. When playing a game, a player can trigger a user event through external devices including mouse, keyboard, VR headset, *etc.* Clicking a mouse at a certain point in a game world, pressing a particular key, or changing the orientation of head (while wearing a VR headset), for example, each entails a user event. However, players have different delay expectations on different kinds of events, and therefore a game architecture should treat these events differently.

We realize that there are two fundamental types of user events, namely, (local) view change events and (global) game events. The first type of user events– view change events – only causes transient changes to a user's perspective, but leaving his/her game world intact. For instance, a user event of clicking a mouse at the location $(153, 85)$ might be interpreted as selecting a troop from a player's army. The chosen soldiers will be highlighted on the

Table 2.1 Comparison between event types in VR-MMOGs

	View change	Game
Tolerable latency (ms)	20	100
Event size (bytes)	180	90
Frequency (events/sec)	95	5

player's screen, but this event is invisible to other players.

The second type of user events– game events – not only causes changes to a player's perspective, but also cause permanent updates to a player's game world, which we refer to as game events. In (VR-)MMOGs, such updates should be synchronized among all the players who can see this game event. For example, the same mouse click at $(153, 85)$ might be interpreted as "player A punches player B" or "player A collects 100 golds from the ground", both causing changes to a game world and deemed as a game event. Since a same user event can be interpreted differently based on a particular game logic and a particular game world, MMOGs usually include a module to distinguish the type of user events before sending them all the way to a game server.

As shown in Fig. Figure 2.1, all user events, no matter which type, will eventually be reflected on a user interface. However, players do have different expectations on the feedback delay. According to [9], players have different tolerance levels for game events, ranging from 100 milliseconds (*e.g.*, first person shooting games) to 1 second (*e.g.*, real-time strategy games). A number of studies have been conducted to reduce the game event response latency by optimizing server scheduling [29], improving rendering algorithm and hardware [28] and optimizing network dissemination [51].

Compared to game events, we find it counterintuitive that players expect much shorter feedback delays for view change events. Immediate local view updates lead to a smooth game control and a seamless user experience. Here, the tolerable latency varies from tens of milliseconds (*e.g.*, orientation changes in 3D games) to a couple of hundred milliseconds (*e.g.*, keystrokes). This latency is much more critical to video-streaming games since a renderer resides much farther away in a cloud rather than in a local GPU.

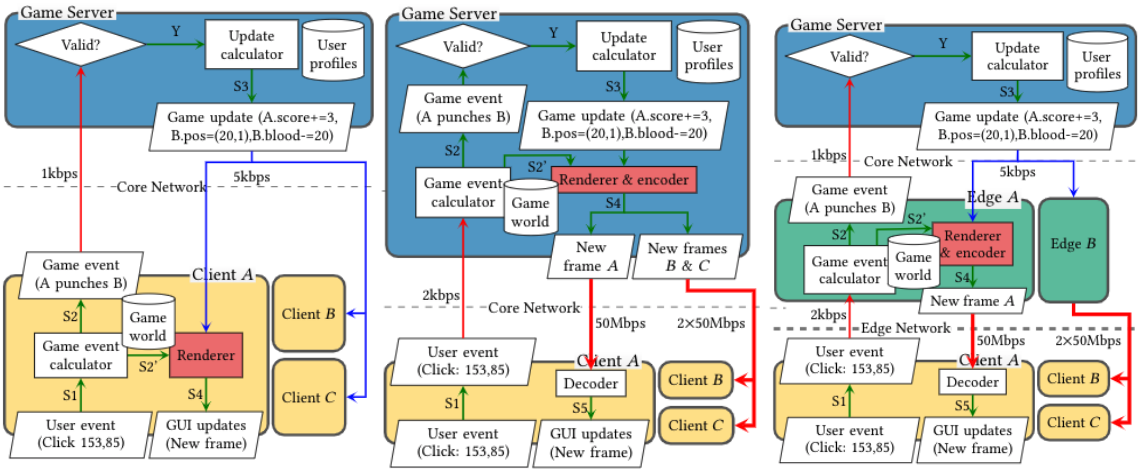


Fig. 2.2 Traditional gaming

Fig. 2.3 Video streaming gaming

Fig. 2.4 Edge cloud augmented gaming

Fig. 2.5 Comparison of Different MMOG Architectures (red line: unicast, blue line: multicast)

The high frequency of such events of VR-MMOGs makes matters even worse. We compare the features between view change events and game events in Table Table 2.1 based on several studies on games [9, 52, 53, 54]. With constant orientation changes in a VR-MMOG (view change events), such changes usually require the feedback delay of less than 20ms if possible [9], to ensure a pleasant user experience. Players would experience dizziness when the latency increases beyond 50ms. This ultra-low latency makes it almost infeasible for a central clouds to support VR video-streaming games as the average latency between Amazon EC2 and clients is already above 80ms. The amount of orientation change events is another challenge as games usually try to get accelerometer and gyroscope readings more than 50 times/second, some high-end devices like HTC VIVE can even reach 100. This frequency is much higher compared to view change events. Game events, estimated by game actions per minute (APM), is usually around 50 and maximized at 300 with proficient players [54].

Therefore, we believe that view change events should be treated as “first-class citizens” and a gaming architecture should be carefully designed to provide a better support for these events in VR-MMOGs.

2.3.2 Overview of Existing MMOG Architectures

We study the underlying communication and computation models of the existing MMOG architectures to understand why these solutions fall short in supporting VR-MMOGs. We classify them into two categories: traditional client-centric gaming and cloud-centric gaming, based on where rendering happens. While file-streaming gaming also gets support from a cloud, its game model is almost equal to client-centric games as it requires a local powerful game consoles for rendering. The flowcharts of the two game models are shown in Fig. Figure 2.2 and Figure 2.3.

Traditional Client-Centric Gaming

A traditional gaming architecture performs most of the tasks on the client side except the synchronization of a shared game world. As shown in Fig. Figure 2.2, a game loop starts with a capture of a user event (*e.g.*, mouse click at $(153, 85)$). This event is firstly sent to a local game event calculator (step $S1$) that detects whether it is a game event. In most games, a game event also triggers a view change event (*e.g.*, shooting fire in Fig. Figure 2.1), and therefore, a view change rendering request is sent to a render (step $S2'$). If the event is a game event (*e.g.*, player A punches B), this game event will be forwarded to a game server (step $S2$).

Once receiving the game event at the server, a validation module checks the validity of the event according to a game logic (*e.g.*, if A is allowed to punch B, if A is close enough to reach B, etc). It will discard suspiciously cheating events (possibly generated by a game bot) and stale events (caused by network delays). A update calculator at the server then computes the "consequences" (game updates, *e.g.*, A gets 3 points, B retreats by 1 step and looses 20 blood) of each valid game event and accordingly updates users' profiles. A user profile usually consists of individual state information of a game character (*e.g.*, the current location, the experience point and the skill level). To avoid transmitting too many small game updates,

Table 2.2 Refresh rate (fps) comparison between mobile devices and desktop machines in different games

Game	Resolution	Refresh rate (fps)	
		Desktop	Mobile
StarCraft II	1024×768	380	55
	1280×1024	136	13
	1920×1080	119	5
GTA V	1024×768	168	10.8
	1280×1024	161	<5
	1920×1080	136	<5

the server usually accumulates game updates for a small period of time and sends all the updates in the period as a batch (step *S*3). The length of the period is usually determined by the smallest frame interval among all players (*e.g.*, 33ms for 30fps clients). When all players share a same synchronized game world, the same updates should reach all of them, and therefore they can be sent via multicast or broadcast to improve the efficiency in dissemination.

A renderer on the client side usually renders a game world periodically (30-60 frames per second) to reflect outputs of view change events (from *S*2) and game updates (from *S*3). In most games, a renderer may generate frames even without any updates to reflect, for example, variations of luminous intensity, flowing of water and/or moving of non-player characters (NPC). Rendering projects geometry, viewpoint, texture, lighting, and shading upon 3D skeletal objects in a game world, and finally outputs a game frame in a game interface like screens or VR devices (step *S*4).

This architecture usually requires a game client to have abundant CPU, GPU and RAM resources since rendering a frame involves a large number of matrix multiplication and floating point operations. Thus, it is not friendly for players who prefer to enjoy games with their mobile devices. For example, an average desktop GPU like AMD Radeon HD 7970M, can reach 380, 136 and 119 fps for StarCraft II at resolutions 1024x768, 1280x1024 and 1920x1080 respectively. However, the refresh rate on Intel HD Graphics Cherry Trail (a GPU adopted on Microsoft Surface 3 tablets) can only deliver 55, 13 and 5 fps with the same resolution (see Table Table 2.2). It means that, if a player tries to play this game on a

Microsoft Surface 3 tablet, he/she can only choose the resolution of 1024x768 or below. A same restriction can be found in most popular MMOGs like GTA V, Minecraft, *etc*. Yet, to enjoy an immersive VR gaming experience, players should not be constrained by desktop machines and cables.

Cloud-Centric Video Streaming Gaming

Cloud-centric video streaming gaming [14, 13, 15] significantly reduces the resource requirement on user devices. All rendering tasks will be executed in a central cloud as shown in Figure Figure 2.3. Specifically, a client device merely sends user events directly to a cloud, and receives subsequent updated frames. This video streaming gaming architecture promises to enable players to enjoy MMOGs on mobile devices, while several important challenges must be addressed to support VR-MMOGs by this architecture.

Firstly, it is demanding for this architecture to satisfy the ultra low latency requirement of VR-MMOGs. In particular, a user may desire a view change rendering to be completed within 20ms [9]. With latency of 50ms, VR games can still respond to a user's input, but with noticeable lagging, which may lead to an undesirable user experience. Unfortunately, the average network delay between Amazon EC2 and a mobile device is around 80ms [12], which is much longer than the preferred latency of view change events.

Secondly, it is also challenging for this architecture to support a high refreshing rate. VR-MMOGs generally require bandwidth of 50Mbps to stream a video with a 1080p resolution at 60 fps, while the available bandwidth in the wireless Internet to a mobile device is merely about 2Mbps [18]. In the traditional gaming architecture, a game server only needs to send game updates to clients, which can be multicasted to minimize the network traffic. In this architecture, however, a game server needs to send distinct rendered frames to each individual client. As a result, unicast is required in this scenario. Everything considered, network bandwidth needed in this architecture increases significantly compared

to the traditional gaming architecture.

2.4 EC+: A VR-MMOG Architecture Augmented by Edge Clouds

In the previous section, we discuss that traditional client-centric gaming places heavy-weight rendering tasks on the game client which essentially prevents users from playing VR-MMOGs on mobile devices. Video-streaming gaming intends to facilitate mobile devices but eventually fails to do so due to slow responses and poor video quality. It is thus desirable to leverage a third computing platform that has sufficient resources while incurring short network latency from and to the clients. We believe edge cloud computing is a good candidate for the following reasons: *1*) it has enough computation power as the servers in edge clouds usually have GPUs that are desktop-level or better, and *2*) it is located in the access network that is close to the users.

Based on this understanding, we propose a new architecture, EC+, that cleverly distributes the work among the central cloud and edge clouds. It has the following salient features: *1*) engaging edge clouds in managing view change updating and rendering to achieve low latency and high refreshing rate, *2*) engaging the central cloud in managing game state updating to support a large number of players and minimize the overhead required to maintain consistent game states, and *3*) handling user mobility and edge-cloud workload imbalance by performing dynamic gaming service migration to provide continued performance.

2.4.1 Flow of Gaming in EC+

Below, we discuss the game flow in the EC+ architecture step by step, which is also shown in Figure Figure 2.4:

- *User event forwarding on the client* ($S1$): VR game devices capture all user inputs and send them to a designated edge cloud. We will discuss how to choose and dynamically change the associated edge cloud for a user in § section 2.5.

- *Local view updating and rendering on the edge cloud* ($S2$, $S2'$): When the edge cloud receives a user event, it passes the event to a "game event calculator", which performs two tasks in parallel: (1) calculating local view updates, and (2) determining whether a global game event is represented. After task (1) is completed, the local view update request is passed to a "renderer and encoder" which will then perform image rendering and encoding. After task (2) is completed and a global game event is needed, then the game event is further passed on to a central cloud.

- *Global game updating on the central cloud* ($S3$): The game server behaves similarly to that in traditional gaming. When the game server on the central cloud receives the game event, it calculates the updates that are caused by this event, updates user profiles accordingly, and then generates one or more update requests to all the players who are involved in this game event. It then sends these requests to the edge clouds that these players are currently connected to. Similar to traditional games, the server here can also take advantage of multicast in game update dissemination.

- *Game world change rendering on the edge cloud* ($S4$): The edge cloud performs all the rendering, including local view change rendering, game world change rendering, and background view refreshing rendering. The rendering performance is critical to the overall performance of EC+. We can leverage techniques such as the one proposed in [55] that involves a scalable parallel rendering framework to simultaneously render for multiple players who share a same game world, which can greatly reduce the overall rendering latency.

In summary, the proposed game flow has the following advantages. Firstly, bypassing the center cloud when dealing with view change events (in step $S2'$) can greatly shorten their response latencies, making it possible to have immediate local view updates. Secondly, by rendering frames on edge clouds (in step $S4$), we can harness their low latency and high bandwidth. Thirdly, the core network traffic can be largely reduced due to the adoption of

edge clouds and the possibility of multicasting game updates to users.

2.4.2 Edge Cloud Migration

When we try to place a player's gaming service (including all the components resided in an edge cloud) onto edge clouds, selecting a suitable edge cloud becomes an important issue. After an initial edge cloud selection, we also need to consider the need of dynamically migrating the services to other edge clouds as the workloads and user locations change. Specifically, we note that service migration becomes necessary when the player moves around while playing games and/or the workload at each edge cloud changes over time.

In our framework, we consider the migration problem by partitioning continuous time into discrete time slots with equal length (say, 2 minutes). With the time partition, we can simultaneously make the optimal migration decisions, upon offline snapshots of the network/server states, for overall clients in the network. In respond to dynamic network/server states, any online solution, nevertheless, introduces the significant computation overhead in highly frequent decision making procedures. We have developed an efficient algorithm based on Markov Decision Process (MDP) to select and migrate a player's edge service, which we will discuss in detail in § section 2.5. However, unlike many of the service migration solutions which assumes an ignorable service transition time, we acknowledge that it is impossible to migrate an edge service from one edge to another instantly given the size of a VR game world. Therefore, we propose a mechanism to ensure a new edge cloud is activated when a player connects to the new one.

To ensure a smooth transition between two edge clouds, the key is the ability to render frames correctly for a player connected to the destination edge cloud. A frame rendering process consists of a series of matrices operations on a game world matrix, a view/perspective matrix as well as a projection matrix, where the game world matrix represents a collection of 3D game models with the particular spatial relations, the view/perspective matrix transfers the relative positions of 3D game models to fit a particular view perspective, and

the projection matrix converts 3D positions of game models into the homogeneous screen space. Service migration in EC+ mainly involves migrating a player's game world as the other matrices are ignorable in size and reproduced easily.

Here, we discuss the migration events of interest within a time slot starting from τ. We assume that a mobile user gets the service from an edge cloud e at τ.

- EC+ starts to make the migration decisions for all clients in the network at the time τ.
- At the time $\tau + \Delta_1$, EC+ finishes the computations of the decision making and determines to migrate the service of this mobile user to the edge cloud e'. e' will get a notification so that it subscribes to the multicast group of this game and start receiving all the game updates. The edge cloud e starts to send a snapshot of the game world at the time $\tau + \Delta_1$.
- At the time $\tau + \Delta_2$, e' successfully receives the game world snapshot (taken at $\tau + \Delta_1$) and start to merge the game updates received since $\tau + \Delta_1$.
- At the time $\tau + \Delta_3$, e' finishes the merging, and it now has the latest game world that is exactly same with the one kept in e. Meanwhile, the e' continues to receive the game updates and keeps the game world up-to-date.
- At the end of the time slot τ, this mobile user connects to e' and successfully gets the gaming service from this new edge cloud. The previous edge cloud e will release all the gaming resources if there is no other client connects to it.

With this mechanism, we can seamlessly complete service migration in EC+ without any service down-times as long as the time slot is larger than Δ_3 for all the migrations.

2.5 Edge Cloud Selection on User Mobility

We devise an algorithm to efficiently determine where to place and migrate an edge cloud service in the presence of dynamic network states and server workload states, and user mo-

bility (initial edge cloud selection can also be generalized as a migration operation). In EC+, we model our placement/migration algorithm as a Markov decision process (MDP) [56] since a placement/migration decision is only affected by a current state and user mobility. We realize that several MDP-based selection approaches have been studied in the literature [57, 58, 59, 60, 4], but we notice that VR-MMOGs impose new challenges, namely, the changing network status over time, the mutual impact among players, and the existence of an extra entity (central server) in the communication. In this section, we present our modified edge selection algorithm.

2.5.1 Problem formulation with Markov Decision Process

In our selection algorithm, we consider a total of M edge clouds, and N access points through which mobile users connect to the Internet. As we discussed in §section 2.4, we partition continuous time into discrete time slots with equal length. At a time slot τ, a mobile user connects to an access point $n_\tau \in [1, N]$ and receives a gaming service from an edge cloud $m_\tau \in [1, M]$. We define this as a state S_τ=$m_\tau n_\tau$. A player may move and connect to a new access point at the end of a time slot. Due to the user mobility and changing workloads on the edges, we may need to migrate the gaming service to a proper edge to satisfy the user's QoS requirement. To achieve this, an action a_τ upon the state migrates the service from the edge cloud m_τ to $m_{\tau+1}$. The action a_τ is represented by the location of a possible edge cloud $m_{\tau+1}$, thereby $a_\tau \in [1, M]$. The new edge cloud at $m_{\tau+1}$ is anticipated to have the minimal network cost by considering the player's any possible locations ($n_{\tau+1}$) in the next time slot. Note that while we are calculating MDP at the time τ, we assume that the migration happens at $\tau + 1$, as we described in the previous section. As a result, at the time slot $\tau + 1$, the system may enter a transit state: $S_{\tau+1}$=$m_{\tau+1}n_{\tau+1}$, with the transition probability $p(S_\tau, a_\tau, S_{\tau+1})$. We assume that the transition probability is given as the known parameter of our algorithm as there are many studies on mobility prediction including [61, 62, 63] and our earlier work [64] that calculates the probabilities of user movements based

upon the aggregated network-level statistics.

To determine the destination edge cloud $m_{\tau+1}$, a cost function $C(S_\tau, a_\tau, S_{\tau+1})$ is defined to measure the overall network transmission cost as well as the migration cost from a state S_τ to a state $S_{\tau+1}$, when we take an action a_τ^π. We detail this cost function in §subsection 2.5.2. Our objective is to find an optimal action (a_τ^π) for each user in each time slot that minimizes long-term cost. The long-term cost function is given by

$$V(S_0) = \sum_{\tau=0}^{\infty} \gamma^\tau \cdot \sum_{S_{\tau+1}=1}^{M\times N} p(S_\tau, a_\tau^\pi, S_{\tau+1}) \cdot C(S_\tau, a_\tau^\pi, S_{\tau+1}), \tag{2.1}$$

where $\gamma \in [0, 1)$ is a discount factor that controls the impact of future states on the long-term cost counted from the current state. We convert the cumulative sum of the long-term cost given by Equation Equation 2.1 into a recursive definition:

$$V^*(S_\tau) = \min_{a_\tau} \left\{ p(S_\tau, a_\tau, S_{\tau+1}) \cdot \left[C(S_\tau, a_\tau, S_{\tau+1}) + \gamma \cdot V^*(S_{\tau+1}) \right] \right\}, \tag{2.2}$$

It is well known that the optimal action $a_\tau^\pi = m_{\tau+1} \in [1, M]$ for each state S_τ can be obtained by Bellman's value iteration [56] which iteratively update the equation Equation 2.2 until the value of $V^*(S_\tau)$ is converged.

2.5.2 Game-specific Cost Function

While modeling the placement algorithm, we try to minimize the "cost" of actions to provide the best game experience. We believe that the cost function should take different features into consideration, including latency, bandwidth, *etc.*. Here, in order to propose a general framework that can satisfy all kinds of VR-MMOGs, we do not mandate the application requirements on different features. Instead, we assume the game provider can get their cost function based on the studies in [65, 66, 67, 68] and their policies.

In this section, we list a set of features that we have in mind. They come in two categories: transition cost and transmission cost. The transition cost is the cost incurred

when we migrate an edge service. As we already have a mechanism to avoid application down-times, this cost is curtailed to the bandwidth cost (*i.e.*, the size of the game world). The transmission cost is the cost to the communication between an edge cloud and a player. This transmission cost can be further categorized into two sub-types: cost without mutual impact and cost with mutual impact. The cost without mutual impact is merely measured by network latency, bandwidth and server load, while the cost with mutual impact is additionally measured by a count of game world sharing (since a migration decision for one player can meanwhile affect the decision of other players who are sharing one game world). Importantly, we highlight the cost with mutual impact which intends to co-place multiple users in one edge cloud to facilitate game world sharing and to reduce the overall migration overhead.

2.5.3 Optimal Joint Migration Decisions

Many earlier MDP based migration approaches calculate an individual migration decision for each user, assuming a user's migration decision have little impact on others. However, when to consider the co-placement, the assumption fails to be hold. To this point, we have to consider all possible combinations of migration decisions at each step and find the optimal joint migration solution.

Assume the total number of the migration decisions we need to jointly consider at each step is K, which is also the number of users in the system, and denote each decision as $d_k, k \in [0, K)$. We then redefine a state as $S_{global}(t) = \{S_{d_1}(t), S_{d_2}(t), ..., S_{d_{K-1}}(t)\}$, and a joint action as $a_{global}(t) = \{a_{d_1}(t), a_{d_2}(t), ..., a_{d_{K-1}}(t)\}$. The new reward function is the sum of the reward function of each individual decision. Finally, we solve Eqn. Equation 2.1 to compute the optimal joint migration action $a^*_{global}(t)$.

Though this approach provides a globally optimal migration decision, the time complexity to search for the optimal joint solution is much higher than that of treating each migration decision independently. Specifically, the time complexity of the latter is $O(M^3N^2)$, while

the time complexity of the global solution is $O((M^3N^2)^K)$. This cost is prohibitively high, preventing us from finding the optimal joint solution in real-time.

2.5.4 Heuristic Joint Migration Decisions

When to calculate the optimal migration decision for each player, we hold an assumption that all other players remain connecting to their current edge clouds and therefore, the whole edge cloud serving conditions does not change. Only under this assumption, the optimal migration decision can keep being optimal. Yet, we fail to hold this assumption if we consider a collection of migration decisions for multiple players. To be close to the assumption, we can order the migration probability of all players and preferentially calculate the optimal migration decisions for the players with higher migration likelihoods. By doing so, after making a migration decision, we argue that the latter migration decisions are more likely to have players connected to the current edge cloud. To estimate the migration likelihoods, we use the overall cost function value subtracting the migration cost. We argue that this heuristic approach with the time complexity of $O(kM^3N^2)$ can minimize the global migration cost.

2.5.5 Runtime Optimization to Reduce Decision Time

We discover a few characteristics of the MDP calculation in this edge placement problem, which can be explored to optimize the runtime. The first characteristic we find is

$$\forall a_\tau, m_{\tau+1} : p(m_\tau n_\tau, a_\tau, m_{\tau+1} n_{\tau+1}) = 0, where\ a_\tau \neq m_{\tau+1}.$$

It indicates that a migration action is deterministic towards next state. Thus, we can simplify the state transition probability from $p(m_\tau n_\tau, a_\tau, m_{\tau+1} n_{\tau+1})$ to $p(m_\tau n_\tau, m_{\tau+1} n_\tau)$, also simplify the cost function from $C(m_\tau n_\tau, a_\tau, m_{\tau+1} n_{\tau+1})$ to $C(m_\tau n_\tau, m_{\tau+1} n_{\tau+1})$, Accordingly, we can reduce the space complexity of p and C from $O(M^3N^2)$ to $O(M^2N^2)$.

The second characteristic we discover is

$$\forall m_\tau, m'_\tau, m_{\tau+1}, m'_{\tau+1} : p(m_\tau n_\tau, m_{\tau+1} n_{\tau+1}) = p(m'_\tau n_\tau, m'_{\tau+1} n_{\tau+1}).$$

It demonstrates that the state transition probability merely relates to linked access points, but not to server placements . Thus, we can simplify the transition probability from $p(m_\tau n_\tau, m_{\tau+1} n_\tau)$ to $p(n_\tau, n_{\tau+1})$ and accordingly reduce the space complexity of p from $O(M^2N^2)$ to $O(N^2)$.

The third characteristic we discover is

$$\forall n_\tau, n'_\tau : C(m_\tau n_\tau, m_{\tau+1} n_{\tau+1}) = C(m_\tau n'_\tau, m_{\tau+1} n_{\tau+1}).$$

It implies that the cost function merely associates with the connected access point $n_{\tau+1}$ at the time slot $\tau + 1$. Thus, we can simplify the cost function from $C(m_\tau n_\tau, m_{\tau+1} n_{\tau+1})$ to $C(m_\tau, m_{\tau+1} n_{\tau+1})$ and accordingly reduce the space complexity of C from $O(M^2N^2)$ to $O(M^2N)$.

By jointly considering the above 3 propositions, we can simplify Equation 1 to

$$V(m_0 n_0) = \sum_{\tau=0}^{\infty} \gamma^\tau \cdot \sum_{n_{\tau+1}=1}^{N} p(n_\tau, n_{\tau+1}) \cdot C(m_\tau, m^\pi_{\tau+1} n_{\tau+1}), \tag{2.3}$$

where $m^\pi_{\tau+1}$ is our decision at the time slot τ which takes effect at the time slot $\tau + 1$. Therefore, we can reduce the total space complexity from $O(M^3N^2)$ to $O(M^2N + N^2)$, and reduce the time complexity of each MDP iteration from $O(M^3N^2)$ to $O(M^2N^2)$. Since the computation of Equation Equation 2.3 can be converted into the vector multiplication of $p(n_\tau, *) \cdot [C(m_\tau, a_\tau *) + \gamma V(a_\tau, *)]$, we can further reduce the execution time using parallel computing (multi-threading, and GPU). We evaluate the performance improvement of our proposed optimizations in §section 2.6.

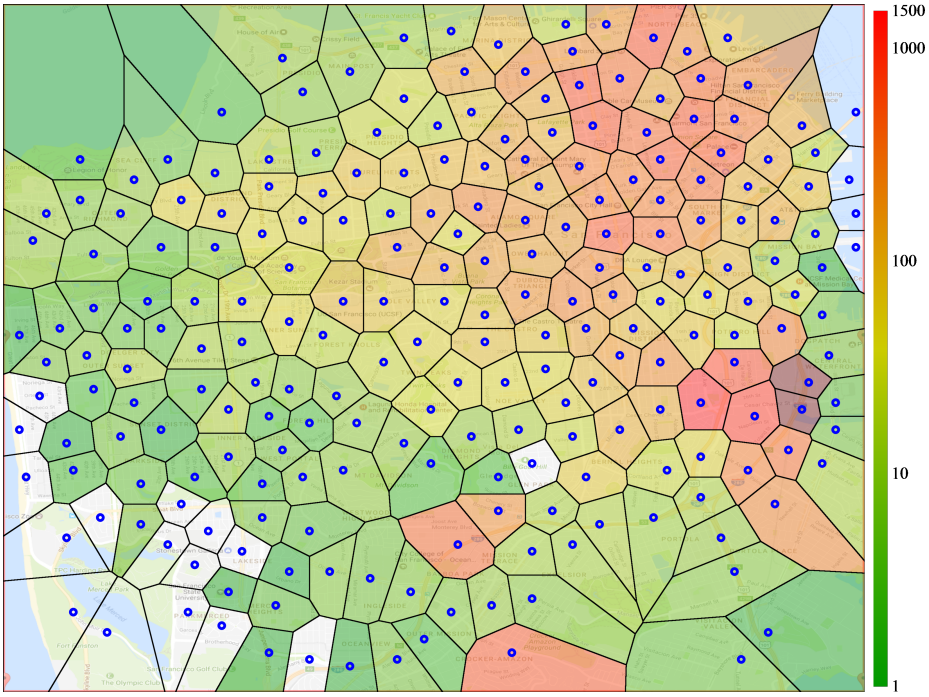
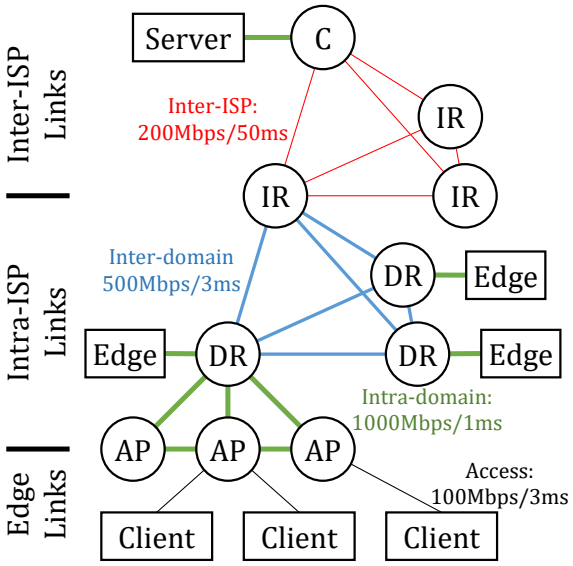


Fig. 2.6 Access points with corresponding effective ranges and heat ($\int Conn(t)dt$)

2.5.6 Further optimizations

Besides the aforementioned optimizations of MDP applied in VR-MMOG migration, we also consider the following optimizations: *1*) Every player moves in a regular activity range and may never, if not impossible, link to a portion of remote access points. Accordingly, the probability table is sparse. We can therefore compress this table as well as the cost table to further reduce the space and time complexity. *2*) In many cases, players can only link to several nearby edge clouds due to the stringent latency and bandwidth requirements. We can identify and exclude remote edge clouds that fail to satisfy the requirements from the possible migration destinations. We can then remove the states associated with the migration destinations and eliminate the calculation of the cost and utility table with respect to the states. Since proposed edge placement algorithm is a framework that is generally applicable to all (VR-)MMOGs, we leave the application-specific optimizations as our future work.



Link	B/w (Mbps)	Latency (ms)
C↔IR	200	50
IR↔IR	200	50
IR↔DR	500	3
DR↔DR	500	3
DR↔AP	1000	1
AP↔AP	1000	3
C↔Server	1000	2
DR↔Edge	1000	1
AP↔Client	100	3

Fig. 2.7 3-layered network topology and corresponding bandwidth and latency

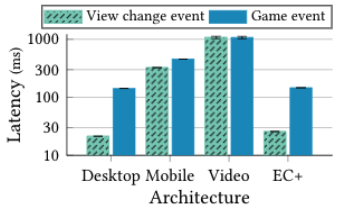


Fig. 2.8 Event latency (95% CI)

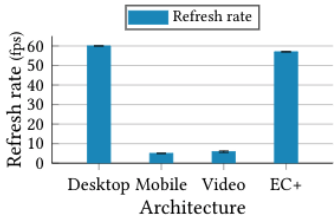


Fig. 2.9 Refresh rate (95% CI)

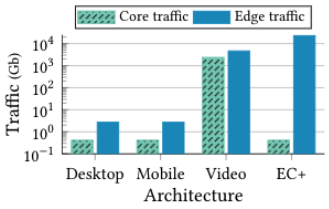


Fig. 2.10 Aggregate network traffic

Fig. 2.11 Result of game simulation without mobility in different architectures: traditional gaming with powerful GPU (Desktop), traditional gaming with mobile devices (Mobile), videostream gaming (Video), and EC+

2.6 Evaluation

We conduct a detailed simulation-based evaluation of the proposed EC+ architecture as well as the MDP based service placement/migration algorithm. We summarize the simulation results in this section.

2.6.1 Comparison of EC+ with Other Gaming Architectures

We use detailed simulation studies to compare our EC+ architecture with the traditional client-centric gaming architecture and the cloud-centric video streaming architecture.

Simulation Setup

We first present our simulation set up for the comparison study.

Network Topology: We use the San Francisco AP map developed in our earlier work [64] as the network topology. We estimate each AP's coverage using Voronoi cells (see Fig. Figure 2.6). We further assign the APs to different domains to represent more realistic network topologies. Specifically, we build a 3-level hierarchical topology as shown in Fig. Figure 2.7.

In simulations, games players connect to the Internet through APs, and edge clouds are assumed co-located with the domain routers. The central cloud is placed at C (see Fig. Figure 2.7). We carefully choose bandwidth and latency parameters for different links in the network. For example, we assume in the intra-domain network, nodes are connected through gigabit switches with millisecond level latencies. The actual capacity for inter-ISP connections is usually much higher, but since the core network is multiplexed with other traffic, and the ISPs might not be directly linked to each other, we choose a bandwidth of 200Mbps (shared bandwidth) and a latency of 50ms (due to the number of hops between two ISPs). As such, we summarize the chosen bandwidth and latency values for different types of links in Fig. Figure 2.7.

Player Location Trace: To model mobile game players, we use the San Francisco cab trace that we adopted in our earlier work [69]. The trace contains the locations of more than 500 cabs between 2008-05-17 and 2008-06-10. We observe noticeable daily mobility patterns in the trace. In our study, we pick the data sets on May 31, 2008, Saturday, as our trace. Since the simulated APs are mainly located in the central San Francisco area (see map in Fig. Figure 2.6), we focus on the 66 cabs that traveled in that area. We assume these 66 players are playing a same MMOG game. Fig. Figure 2.6 shows the "heat" of each AP. The hotness of AP a is calculated as $H(a) = \sum_{u \in U} t_u(a)$, where $t_u(a)$ is the total time a user u is associated with a.

Game trace: We consider a 1-hour synthetic game trace, taking the parameters from the study in [52]. Each player's user events arrive with Poisson distribution (λ between 9.5

and 15), a portion of which are randomly selected as game events. Each player's action per minute is between 30 and 300. In total, we have 18,686,459 user events (average: 78.642 UEs per second per user), and 287,567 (1.53%) game events (average: 72.618 actions per minute per user). Size of user events: Poisson distribution (λ=40); Size of updates (in traditional and edge): Poisson distribution (λ=130) [52]; Size of frames (in video stream and edge): (60Mbps / 60fps) = ~1Mb/f. The games are refreshing at 60fps.

Metrics: We use metrics including event latency (time between when the event happens and the clients see the related update) for both user and game events, core and edge network traffic, and frame rate.

Comparison Results with Stationary Players

To study the fundamental difference among the architectures, we compare them assuming the players are stationary. Specifically, we take the locations of each user at 0:30, 1:30, . . . , and 23:30 on 5/31 and create 24 different traces. We evaluate each architecture using these traces and report the results in Fig. Figure 2.11.

Traditional client-centric gaming fares well when users are equipped with desktops with powerful GPUs (with an average rendering latency pf 10ms). In this case, the view change response latency is rather low, ~20ms. The aggregate network traffic is also low (~400Mb) since only small game update packets are exchanged between the server and the client, with multicast support. Finally, it can obtain a refreshing rate of 60fps. When users with mobile devices (GPUs can render 5 frames per second) try to adopt traditional gaming, the rendering performance degrades significantly. The average view change rendering latency is 300ms, and the average refreshing rate is just around 5 fps.

Client-centric video stream gaming faces performance bottleneck in the core network, since it has to unicast frames to each client (which consumes more than 1Tb core network traffic). The frame drop rate is quite high, and therefore the actual frame rate at the client

side is only around 7 fps. To alleviate the frequent frame drops, we adopt forward error correction (FEC) so that each frame contains all the events that arrive before the frame is rendered. However, even with this technique, the rendering latency is still high (>1s for both view changes, and game events) since the events can only be delivered by the next frame that is successfully delivered. We note that the performance can be even worse in the real world due to the multiplexing on the back haul links.

EC+ can provide event update latency (< 30ms for view change events and around 100ms for game events) and refresh rate (of 56fps) similar to traditional gaming (desktop), since the renderers in edge clouds are powerful and are close enough to the clients. With such a low update latency on the non-game events, our architecture can have good support on VR applications where users need immediate feedback for the non-game events (*e.g.*, look into another direction). While our solution does consume more traffic (>10Tb) in the edge network compared to the traditional gaming, we argue that it is feasible and stable since the ISPs usually have full control of the edge network to ensure QoS.

2.6.2 Validation of Edge Placement

To validate our edge selection solution, we conduct a set of small-scale simulations with synthetic topology and a small number of users.

Simulation Setup

We consider a 7×7 grid (shown in Fig. Figure 2.12), where each of the 24 outermost grid cells represents an AP and the users that are connected to this AP. We consider all the edge clouds are resided in the gray grid cells next to the outermost circle, and the central cloud is resided in the central grid. We assume a link's latency is proportional to the distance between the two endpoints, while its bandwidth is reversely proportional to the distance.

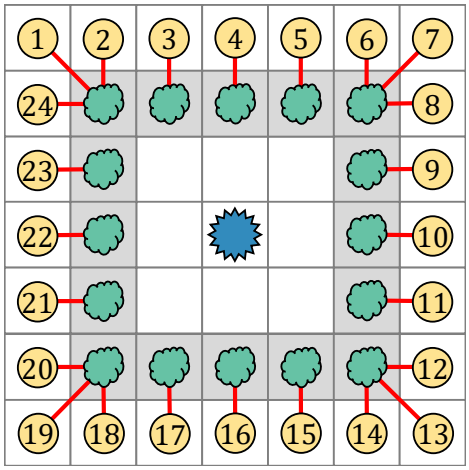


Fig. 2.12 Low migration cost

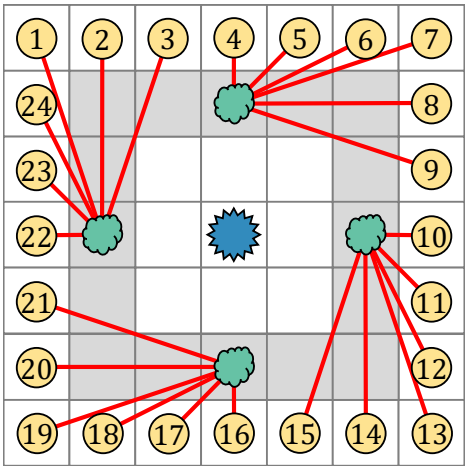


Fig. 2.13 Medium migration cost

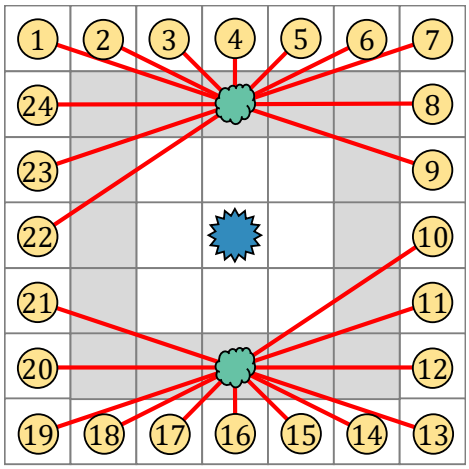


Fig. 2.14 Higher migration cost

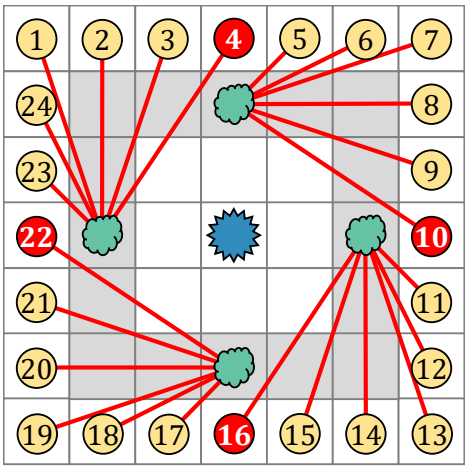


Fig. 2.15 With back probability

Fig. 2.16 Edge placement decision in single user case

Validation for single player scenarios

We first validate our algorithm with a single player and varying the migration cost. We note that the migration cost varies from game to game, dependent on the game world size.

We first consider a simple mobility pattern where the player moves around clock-wise from one grid cell to the next (along with the outermost circle). Fig. Figure 2.12 shows that when the migration cost is small, the placement algorithm always tries to place the service on the nearest edge cloud whenever the play moves. In this example, the player changes the location 24 times, and the corresponding service changes the location 16 times. When the migration cost increases, the algorithm decides to migrate less frequently (see Fig. Figure 2.13). For example, when the migration cost is around twice the current transmission cost – in this case the transmission latency is doubling the frame size in a time slot – the service location changes four times when the player changes location 24 times.

When we further increase the migration cost, there are only two service locations for a total of 24 location changes for the player (see Fig. Figure 2.14). Finally, when the migration cost becomes too large (greater than four times of the current transmission cost), the algorithm does not migrate at all (not shown in the figure).

We next consider a different player mobility pattern: at each time slot, the player moves to the next grid cell in the clockwise direction with a 60% probability, moves to the next grid cell in the counter clockwise direction with a 10% probability, and stay in his/her current cell with 30% probability. This new mobility pattern leads to different migration decisions (shown in Fig. Figure 2.15). Here, we use the same migration cost as in Fig. Figure 2.13. The results show that with the probability of the player moving backward, the placement algorithm becomes more conservative. It still has 4 different service locations, but the location change occurs one-time slot later than that in Fig. Figure 2.13.

In the considered single player scenarios, the algorithm outcomes match our expectation. We thus validate its correctness in the single player case.

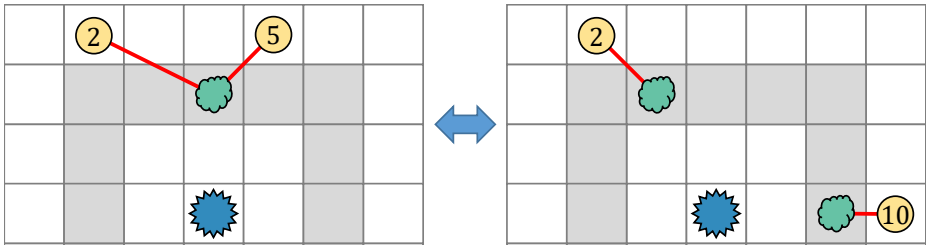


Fig. 2.17 Server sharing when players are close to each other

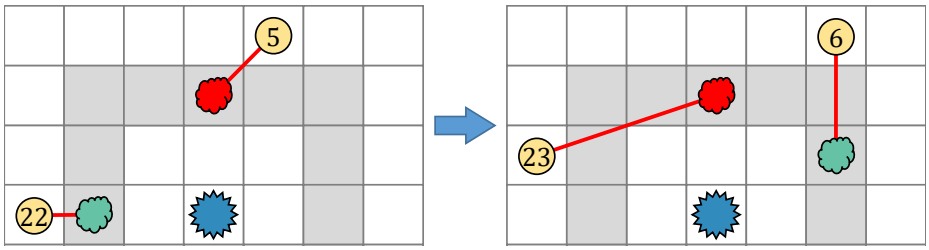


Fig. 2.18 Reuse of existing edge (red) to reduce migration cost even when the original client (at 6) is going to move away

Fig. 2.19 Mutual impact in multiplayer scenario

Validation for Multi-Player Scenarios

Next, we validate our algorithm when we consider two players in the same 7×7 grid topology.

Here we vary the distance between the two players. Fig. Figure 2.17 shows that, when the two players are close to each other (when they are in cells 2 and 5 respectively), our algorithm decides to place them on the same edge cloud to take advantage of shared game world. In Fig. Figure 2.18, when considering where to migrate to, our algorithm tries to migrate a player to an edge cloud that is hosting other players to ruse game worlds and reduce the migration cost. In this example, even though the original player (in cell 6) is about to leave the red edge cloud, our algorithm still migrates the player in cell 23 to the red edge cloud.

Optimal vs Heuristic Placement with Multiple Players

Next, we compare the optimal placement solution vs the heuristic placement solution when we have two players in the 7×7 grid topology. In each time slot, the player moves to the

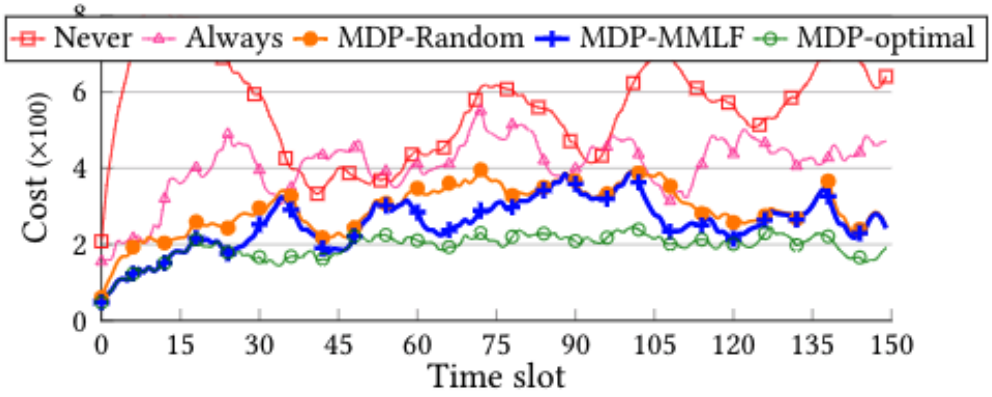


Fig. 2.20 Result of service migration with different strategies

next grid cell clockwise, stay in the same grid cell, and moves to the next grid cell counter clockwise, each with 1/3 probability respectively.

Fig. Figure 2.20 reports the resulting migration cost of the following schemes: (1) MDP-optimal, the global optimal placement, (2) MDP-MMLF, the heuristic MDP placement scheme in which we place the user with the maximum migration likelihood first, (3) MDP-random, the heuristic MDP placement scheme in which we randomly sort the players, (4) Always, an always-migrate scheme, and (5) Never, a never-migrate scheme.

Among these five schemes, MDP-optimal gives the best performance. It also consumes the most memory and CPU resources. Given the 24×24 client locations and 16×16 edge cloud locations, it takes more than 1 minute to compute the migration decisions for two players. When we have a large number of players, the computation cost will be prohibitively costly. We also observe that all three MDP based solutions give much better performance compared to naive always-migrate or never-migrate schemes. Finally, we find that our proposed MDP-MMLF performs closer to the optimal solution than MDP-random.

Evaluating the Runtime Overhead

Finally, we measure the run-time overhead of the three versions of MDP-MMLF implementation: (1) original, (2) optimized. Our hardware platform consists of an Intel Core i7-4790 CPU with the clock rate of 3.60GHz [70], running Ubuntu 14.04.

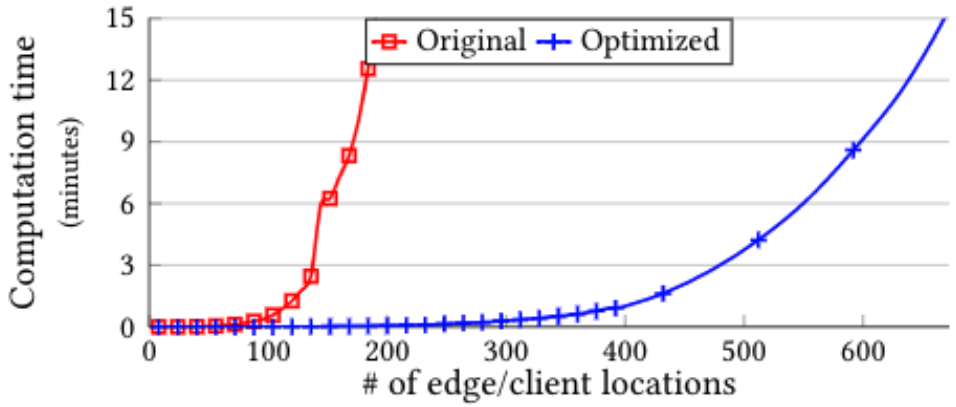


Figure 9: Result computation time.

Fig. 2.21 Result computation time

In our experiments, we consider 1 user and vary the number of edge clouds and client locations. Figure Figure 2.21 shows the computation times for all three implementations.

The results show that the execution time of the original implementation increases fast with the number of edge/client locations (so does its memory consumption). The optimized implementation has a much lower execution time with the same edge/client numbers.

We believe this version can be used for scenarios with many more users since in the real network, there are usually ~10 possible edge clouds that a user can use at any time.

2.7 Conclusion

In this paper, we highlight the main challenges of VR-MMOGs. In response to these challenges, we propose the EC+ architecture that seamlessly distributes the required processing across user devices, edge clouds, and the center cloud to achieve ultra low-latency responses, frequent refreshing, and a large number of concurrent players. To complement our architecture, we also investigate a game service placement algorithm which intends to maximize the gaming performance for all the players by dynamically placing their services on those edge clouds that can lead to the best performance. Finally, we have conducted detailed simulation studies to evaluate our edge-cloud assisted gaming architecture and dy-

namic service placement algorithm. Our results indicate that the proposed approach serves as a viable solution for supporting VR-MMOGs.

CHAPTER 3
EFFICIENT ORCHESTRATION OF EDGE RESOURCES AND JOBS

3.1 Introduction

In the last decade, hosting major computing jobs on central clouds has proven effective since central clouds generally have abundant computing/storage resources [71]. Recently, as mobile devices and Internet of Things (IoT) sensors keep increasing, an unprecedented amount of data have been generated, and a new class of applications is quickly looming on the surface. These applications involve performing intensive computations on sensor data (typically image/video) in real time, aiming to realize much faster interactions with the surrounding physical world and thus providing truly immersive user experiences.

With this trend, central clouds may no longer be the appropriate platform for supporting these applications, in view of performance limitations caused by network bandwidth and latency constraints. For example, a mobile AI assistant application needs responses within tens of milliseconds. An autonomous driving system, as another example, may generate gigabytes of data every second by its stereo camera or LIDAR, and needs responses within a few milliseconds. Yet, for a client instance in New Jersey which connects to Amazon EC2 cloud servers located in West Virginia, Oregon and California, the round-trip latency *alone* is 17, 104 and 112ms, with achievable bandwidths of 50, 18 and 16Mbps, respectively. In order to support these emerging edge applications, edge cloud computing has been proposed as a viable solution [1, 2, 3], which moves the computing towards the network edge to reduce the response latency while also avoiding edge-to-core network bandwidth constraints.

Several aspects of edge computing have been studied in recent years. For example, the study in [7] proposes systems that enable rapid virtual machine (VM) handoff or live migration across edge clouds. Edge computing also raises many interests from the analytic perspective as it introduces a new communication and computing paradigm [72].

In order to minimize service delay in edge computing, works in [73, 74, 75, 76] introduced optimization frameworks to minimize transmission delay and/or processing delay for mobile users. Applications proposed in [3, 77] offloaded intensive computing tasks to edge clouds to achieve low latency image processing.

Despite the earlier and ongoing work on various aspects of edge computing, the problem of how to efficiently deploy these new edge applications within an edge cloud has not been systematically studied. Simply duplicating the successful cloud computing design will not work for the edge applications. This is mainly due to the highly heterogeneous nature of edge clouds. Unlike central clouds, edge clouds are often comprised of heterogeneous computation nodes with widely diverse network bandwidths. For example, the studies in [5, 6], assume the computation nodes and their interconnects are relatively homogeneous in central clouds, while the edge servers considered in [7, 8] exhibit widely varying capabilities. Thus, an important new challenge associated with edge clouds is that of efficiently orchestrating these heterogeneous resources in order to meet application latency constraints.

To address this problem, we set out to build and test such an edge computing orchestration platform. Our design is driven by the requirement of deploying and accelerating this new class of edge applications – e.g., processing large volumes of data such as video data generated by mobile/IoT sensors (including 3D cameras) in real time. We first build an edge cloud testbed that consists of four different CPU settings, four different GPU settings and five different link bandwidth settings. On these nodes, we run Apache Storm [10] as the baseline distributed edge computing framework. Apache Storm provides real-time support, but has an implicit assumption that the underlying computing/networking resources are homogeneous. Also, it does not provide proactive support for GPUs. In this work, we address these shortcomings. Note that our platform design is not specific to Apache Storm. In fact, it can easily interface with other distributed computing frameworks such as Apache Flink.

The design of Hetero-Edge mainly focuses on distributed resource orchestration for edge

computing. Specifically, we intend to answer the following important questions. Firstly, if an edge cloud consists of both GPUs and CPUs, when do we serve requests on GPUs and when do we use CPUs? How do we partition our jobs so that we can most efficiently utilize the available resources? Secondly, after partitioning the job to several pipelined and parallel tasks, how can we map them to appropriate computing nodes (including both GPUs and CPUs) to minimize their overall latency? Thirdly, how can we effectively prevent a parallel task from completing significantly slower than its peers and becoming a straggler [11]? Since resources of edge clouds are highly diverse, the likelihood of having stragglers is much higher than in a homogeneous setting. By carefully studying these questions, we devise the resource orchestration schemes in Hetero-Edge, featuring: (1) matching a task's resource demand with the underlying node's resource availability, (2) matching a task's workload level with the underlying node's resource availability, and (3) suitably splitting work on processors with vastly different processing power (GPUs vs CPU).

We have implemented an example edge application on our Hetero-Edge testbed, i.e., real-time 3D scene reconstruction from two stereo video streams [78]. We use this example application to drive our evaluation effort. We emulate a realistic setting where user streams dynamically join and leave the system and track the detailed system performance for two hours. We show that with seven edge servers, we are able to support all the streams that arrive within the two hours with an average per-frame latency of 32 milliseconds. We also show that our schemes can effectively prevent straggler tasks and can shorten a frame's latency by 40% compared to the state-of-the-art Storm schedulers when we have heterogeneous resources. We summarize our contributions as follows:

- We have designed and implemented a distributed edge computing platform Hetero-Edge that extends the capabilities of a stream processing framework, Apache Storm, for use in heterogeneous distributed edge environments with a focus on latency reduction. We make our code open source and share it through GitHub at .

- We have devised a dynamic task topology generation scheme, a *latency-aware* task

scheduler and a proportional workload partitioning scheme, which, when combined, can proactively minimize the overall latency in heterogeneous distributed edge environments.

- We have implemented 3D scene reconstruction as a driving application example and have shown how to optimize this category of applications on our edge platform to achieve low latency. Note that we only use this application as an example to drive the discussion and evaluation. Other real-time edge vision applications will be readily supported in the same way without changing our system in any way.
- We have learned valuable lessons in deploying real-time edge applications on heterogeneous edge servers. Such lessons will help us realize the wide adoption of edge computing.

3.2 System Model

In this section, we discuss the emerging real-time edge vision applications, summarize the system assumptions for edge clouds, and present the architecture of Apache Storm.

3.2.1 Characteristics of Real-Time Edge Vision Applications

In this study, we focus on supporting a new class of applications, which we refer to as real-time edge vision applications. These applications usually take image/video data that are captured by mobile or IoT devices as input, perform complex processing on each frame and have stringent latency requirements. For examples, consider real-time 3D scene reconstruction [78], virtual reality [75], augmented reality [79], vision-based autonomous driving [80], etc. Though diverse, these applications share quite a few common characteristics, such as low latency requirement and high computation/networking demand. Importantly, they are usually parallel and pipelined by nature.

In the rest of this chapter, we will use 3D scene reconstruction [78] as the example use case to drive the discussion and evaluation. We believe that the ability to perform low latency

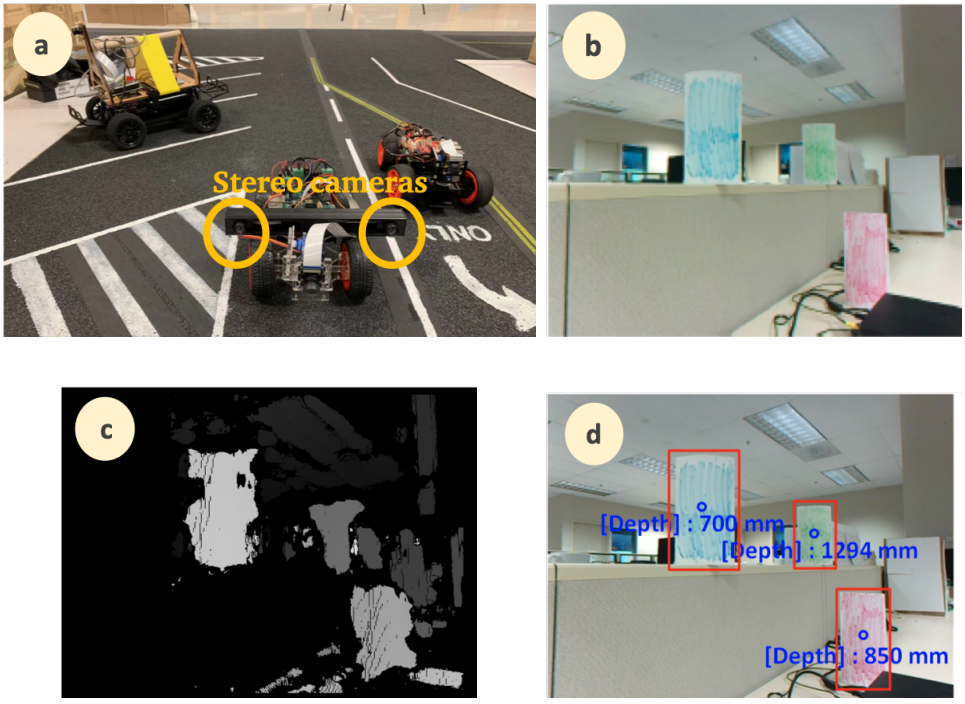


Fig. 3.1 An example 3D reconstruction application. (a) is the mini self-driving cars with the stereo cameras, (b) is the raw input from the left stereo camera, (c) is the resultant disparity map by which we can infer the depth of each pixel, and (d) is the reprojection result which shows the depth of objects in the real world

3D scene reconstruction not only can help build mobile augmented reality or virtual reality to enhance immersive user experience, but also can enable an array of applications with tight feedback loops. For instance, 3D scene reconstruction for autonomous vehicles is used to detect the relative positions of the obstacles and trigger collision-avoidance reactions [81]. It typically consists of the following steps: (1) offline camera calibration, (2) stereo image rectification, (3) disparity calculation, and (4) 3D re-projection. The essence is to infer the disparity of each pixel from multiple 2D images and then use this extra dimension data to reconstruct the object jointly. Disparity measures the difference in retinal position between two points that correspond to the same point on the real object. By definition, a more remote point tends to have a smaller disparity value than a nearer one. This step mainly decides the quality of the reconstruction effect and usually involves heavy computation overhead.

Figure Figure 3.1 illustrates a 3D reconstruction example that we have implemented in our laboratory.

3.2.2 System Assumptions for Edge Clouds

The definition of edge clouds varies from study to study, ranging from a smart traffic light that has some computing capability [1] to a small-scale data center [2]. In this study, we assume an edge cloud consists of multiple edge servers within radio access networks, e.g., eNodeBs, that are available for hosting computing tasks [82]. Different from traditional central clouds that are generally equipped with homogeneous and well-provisioned resources, edge clouds are opportunistic and heterogeneous by nature. An edge cloud is usually composed of nodes with varying computing capabilities (CPUs with different cores, GPUs, etc.), storage capacities (hard drives, memories, etc.), and network capacities.

We further assume there is no resource contention among different application processes on the same node. This assumption can be achieved by deploying edge applications in light containers such as Kubernetes [83] or NVIDIA docker [84] that supports GPU-level isolation.

In this work, we have implemented a Hetero-Edge testbed consisting of 7 edge servers. An edge server has one of the following CPU configurations: (1)Xeon E5-2630, 2.40GHz, 32 cores, (2)Xeon E5-2698, 2.30GHz, 64 cores, (3)Xeon W5590, 3.33GHz, 16 cores and (4)i7-3770, 3.40GHz, 64 cores. Their GPU configurations are the following: (1) no GPU, (2) Tesla K40 GPU, (3)Tesla K80 GPU and (4)Tesla C2050 GPU. Each computation node can have the following link bandwidth configurations: .5Gbps, 1Gbps, 2Gbps, 5Gbps, and 10Gbps.

3.2.3 Background on Apache Storm

In Hetero-Edge, we choose to adopt Apache Storm [85], a popular distributed real-time data stream processing framework, to support the distributed processing. Apache Storm has been deployed in various scenarios such as algorithmic trading, real-time video processing, distributed remote procedure call, etc. We choose Apache Storm because it offers the following advantages: (1) designed for pursuing ultra-low latency, (2) easily scale to

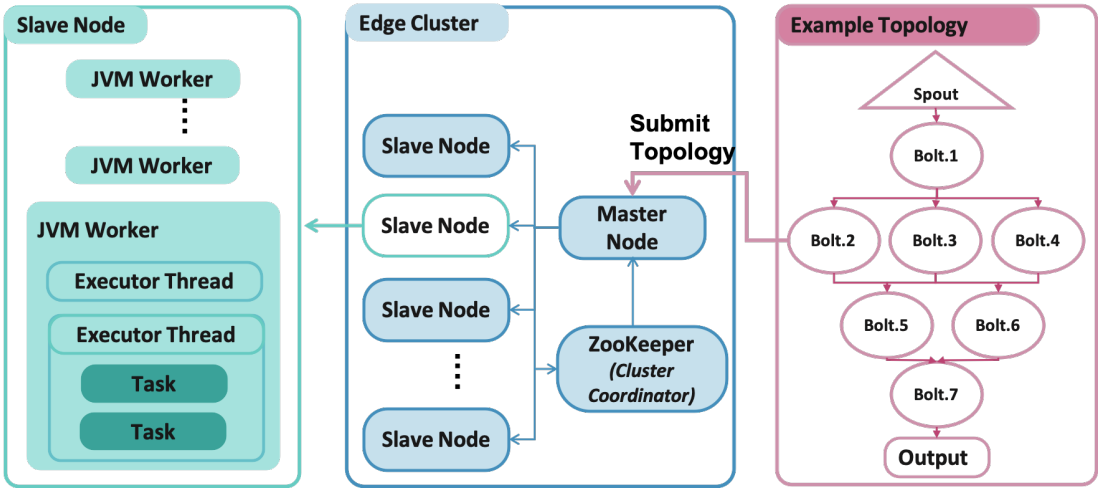


Fig. 3.2 Apache Storm Architecture and Example Topology. The rightmost figure shows an example topology that consists of a user-specified DAG. A user needs to submit this topology to the master node in a Storm cluster. Then, the master node distributes the tasks of the topology to the pool of its slave nodes who take the job of executing those tasks. The detailed system design of a slave node is shown in the leftmost figure

dynamically available resources, (3) no need to store any intermediate results (the main bottleneck of those distributed computing frameworks with the MapReduce design, e.g., Hadoop [86])

Here, we briefly overview its architecture and go over the core components that are relevant to our study. Apache Storm consists of a single *master* node and a pool of *slave* nodes. The master node is in charge of distributing tasks to the slave nodes while a slave node manages *worker* processes. Each worker process further manages *executor* threads, each of which executes a task in a given task graph.

In the logical layer, Apache Storm is composed of three core components: *Spout*, *Bolt* and *Topology*. A spout is usually the source of a data stream, a bolt is an intermediate processing function, and the topology (represented by a Directed Acyclic Graph (DAG)) steers the data flow according to the job logic. We present the overview of Apache Storm in Figure Figure 3.2.

When deploying an application on edge clouds with Apache Storm, we need to consider the following main issues:

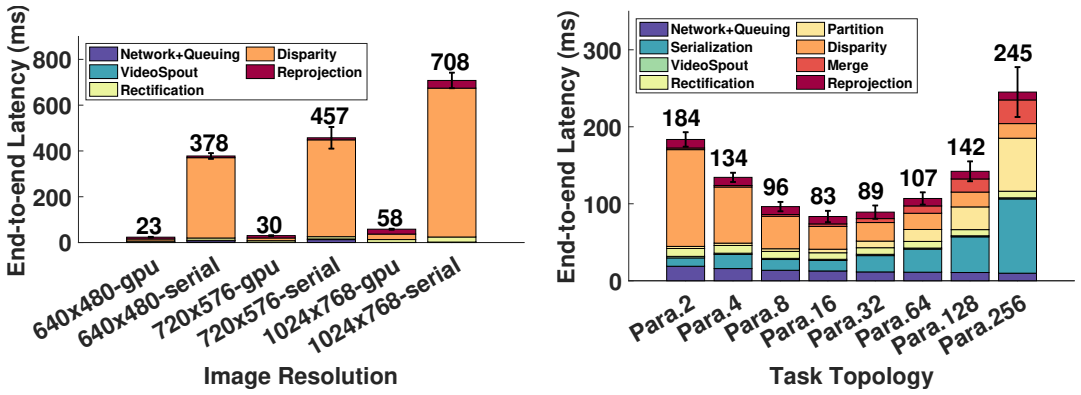
1. *Task topology construction*. In this step, we construct one or more suitable task topologies for the application, considering both data parallelism and task parallelism. We take into consideration the resource diversity when generating the task topology – we choose to have different task topologies for the same application under different resources. See Section subsection 3.3.2.

2. *Task scheduling*. When a topology is constructed, we assign each task bolt in the topology to a computation node based on certain scheduling principles. The default Storm task scheduler does not consider resource diversity and simply assigns nodes in a round-robin fashion. Such a schedule leads to long latency in heterogeneous edge cloud. In this study, we devise a *latency-aware* task scheduler that can considerably outperform the Storm default scheduler and the state of the art *resource-aware* scheduler [87]. See Section subsection 3.3.3.

3. *Stream grouping*. When a frame arrives at the system, this step considers how the output stream of a bolt (e.g., bolt 1 in Figure Figure 3.2) is partitioned among the next step data parallel bolts (e.g., bolts 2, 3 and 4 in Figure Figure 3.2). If the resource variation of these bolts is not considered, one of the data parallel bolts can become much slower than others, thus slowing down the entire processing. In this study, we devise a proportional partitioning scheme that can alleviate this problem. See Section subsection 3.3.4.

3.3 Detailed Hetero-Edge Design

In this section, we present the detailed design of Hetero-Edge. Hetero-Edge has many design details, and we specifically focus on those that can shorten the end-to-end application latency. Below, we use the 3D scene reconstruction application to drive our discussion. However, our design is not limited to this particular use case but is applicable to all the real-time edge vision applications that we describe in Section subsection 3.2.1.



(a) (b)

Fig. 3.3 (a) Latency breakdown of the 3D reconstruction application under different configurations; (b) Latency of different degree of data parallelism in the para-DAG with the resolution of 640x480. The 16-way para-DAG gives the lowest latency

3.3.1 Preparation: Bottleneck Analysis

Before presenting our design, we first break down the application into several functions – for the 3D reconstruction application, we have rectification, disparity calculation, and re-projection. We profile the processing latency of these functions with different image resolutions on Xeon E5-2360, and present the measured latency values in Figure Figure 3.3(a). We find that the disparity calculation function is the predominant bottleneck, which becomes even more pronounced as the image resolution goes up.

We next execute the disparity calculation function on Tesla K40 GPU and find its latency drops significantly. For example, 94% latency reduction with the resolution of 640x480 drops.

In our subsequent steps, we will use the above latency information to make scheduling decisions. Usually, it is a good practice to perform such a bottleneck analysis before trying to deploy an application. Fortunately, there are various tools we can use for this step. For example, we can use NVIDIA OPENACC [88] to identify the execution bottleneck as well as function dependency of an application.

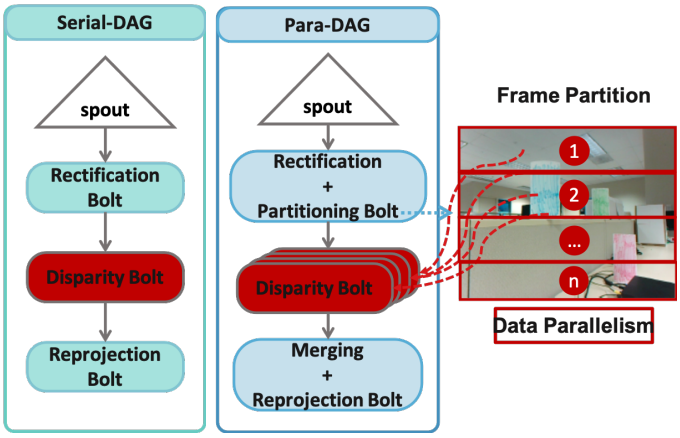
3.3.2 Task Topology Construction

Given the above function breakdown and the identified system bottleneck, we consider the following two practical topologies when with different available computing/networking resources and will evaluate these choices in Section subsection 3.4.2.

Serial Topology (serial-DAG): In a serial-DAG, we can either aggregate all functions into a single bolt for the benefit of introducing little inter-communication overheads or assign an individual function to different bolts for the merit of generating a pipelined flow to decrease task queuing time for an available processor, as shown in Figure Figure 3.4(a). Importantly, a serial-DAG is often more practical when we execute the bottleneck function on GPUs than on CPUs. In this case, the non-bottleneck functions can be scheduled either on the hosting CPU or even on a remote CPU.

Parallel Topology (para-DAG): Since the disparity step dominates the entire CPU processing latency, we next consider a topology that enables data parallelism to accelerate the computation, which we call a para-DAG (which is usually demanded when GPU is unavailable). With an n-way para-DAG, we partition the data set into n partitions and feed each partition into a data-parallel bolt that runs the same disparity calculation function. We illustrate this topology in Figure Figure 3.4(b).

Different partitioning strategies, e.g., range partition, hash partition or composite partition, can be adopted according to different algorithm designs of bottleneck functions. In this specific application, considering the disparity calculation function processes each row of the image independently, we partition the image in a row-major fashion. Say the original image has $M \times N$ pixels, and we partition the image into n ways. Then each partition has $\frac{M \times N}{n}$ pixels. Also, in order to guarantee each partition has enough pixels to generate the disparity map, we need to include the boundary rows in both partitions. As a result, additional pixels need to be included in each partition. In this case, the total number of pixels a partition thus becomes $(\frac{M \times N}{n} + M \times \frac{d}{2})$ where d is the searching block size defined in the block matching algorithm that calculates disparity. The resulting para-DAG topology



(a) (b)

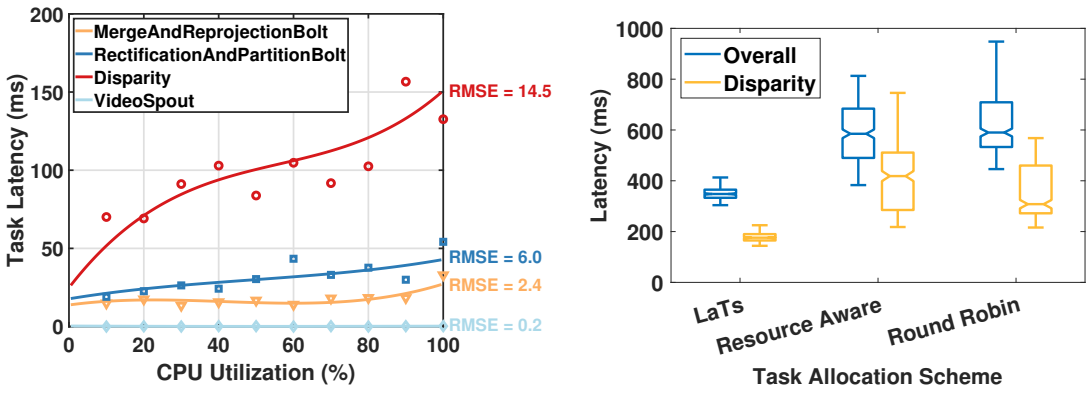
Fig. 3.4 We consider two task topologies: (1) serial-DAG whose bottleneck bolt (i.e., the disparity bolt) is usually scheduled on GPUs, and (2) para-DAG whose bolts are scheduled on CPUs

is shown in Figure Figure 3.4(b). Due to data partitioning, we need to introduce two more processing steps to the topology: partitioning and merging.

Considering each partition will combine more boundary rows that introduce additional computation/networking overheads (which is a usual case among any image partition algorithms), we further examine the topology to decide the suitable partitioning degree n. We choose the 640x480 image resolution at 1fps, and measure the overall latency as well as the latency for each component for different n values. The measured results are shown in Figure Figure 3.3(b). We find that the 16-way para-DAG gives the lowest latency, 83ms in our case.

Note we need to explore particular best value n for different applications.

In reality, if we need to support many more applications in the edge clouds, we can use tools such as those in [88] for automatic DAG partition and parallelization. Finally, we remark that data partitioning does not only apply to the para-DAG, but it should also be considered in generating the GPU code for the serial-DAG.



(a) (b)

Fig. 3.5 (a) Estimated processing latency vs CPU utilization. Using the measured latency values under different CPU utilization, we build a 3-order polynomial curve to estimate the processing latency under any utilization; (b)Comparing the latency of *LaTS*, *round-robin* and *resource-aware* task scheduler at a low frame rate of 1fps with high resource heterogeneity

3.3.3 Task Scheduling

After generating the two topologies for the application, we next enter the task scheduling phase and try to schedule both topologies on the available edge server nodes. That is, we need to assign each bolt in a topology onto a suitable edge server.

Apache Storm provides two task schedulers: the *round-robin* scheduler and the *resource-aware* scheduler [87]. The *round-robin* scheduler is the default Storm scheduler. It allocates bolts to computing nodes in a *round-robin* fashion, oblivious of the bolt resource demand and the available node resource. It may allocate, for example, a computation intensive task on a node that is short of CPU cycles, leading to long execution latency. To address the problem with the *round-robin* scheduler, the *resource-aware* scheduler selects a node with the most available CPU resources and fills it up before assigning any task to any other node. Here, users estimate the available CPU resources on each node as well as the requested CPU resource for each bolt at the compile time. Without a reliable estimation mechanism, it can either lead to resource waste (by overestimating the requested CPU) or result in resource contention (by overestimating the available CPU). Furthermore, neither *round-robin* nor *resource-aware* scheduler considers GPU scheduling.

Our proposed task scheduler has the following three main components: (1) a mechanism to estimate the performance and resource requirement of a task/bolt (we use these two terms interchangeably), (2) a tool to track the available resources within an edge cloud, and (3) a *latency-aware* task scheduling algorithm. Below we describe these components one by one.

Estimating a Bolt's Performance and Resource Demand

Before devising our task scheduling scheme, we first need to develop a mechanism to estimate a bolt's performance (i.e., processing latency) and resource demand (i.e., memory usage, network usage, GPU/CPU usage). Among these four items, a bolt's memory usage and network usage remain constant no matter where the bolt is executed, which we can capture through ThreadMXBean.getThreadAllocatedBytes(), and count the byte array length of output stream in Storm.

The other two items – a bolt's processing latency and processor usage (we consider both GPUs and CPUs here) – are not only determined by which server the bolt is executed on but also the load on the server. As such, in the profiling phase, we run each bolt on every edge server (including both CPUs and GPUs) at different processor utilization. We increase the processor utilization level from 0% to 100% with an increased interval of 10%. For each bolt-processor-utilization combination, we measure the latency (by recording the elapsed time through JAVA Timer API) and processor time (which is the amount of GPU/CPU time dedicated to this bolt process and can be captured through ThreadMXBean.getThreadCpuTime(), while GPU utilization can be read from nvidia-smi).

We can then feed the measured values into n-th order polynomial curves ($n = 3$ in this work because it provides the lowest predictive mean squared error) to derive an estimation model for each bolt. This model takes a particular processor's resource level and utilization as input, and gives an estimation of the bolt's processing latency and consumed processor utilization. Figure Figure 3.5 (a) plots the latency estimation models for the four bolts of

the 3D reconstruction application on the Xeon E5-2630 processor.

Finally, note that the estimation model in VideoStorm [5] does not consider the processor load, but we believe processor load is an important parameter to consider when estimating a bolt's latency and resource demand.

Real-time Edge Resource Monitoring

We periodically collect the available resources for each edge server. For this purpose, the following actions are performed: (1) collecting the port bandwidth of a node using the *iPerf/scp* utility; (2) collecting CPU frequency and utilization of a node using the *lscpu* utility; and (3) collecting the current memory usage of a JVM worker from the Storm's daemon. (4) collecting GPU utilization of a node using the *nvidia-smi* utility.

Proposed Heuristic: Latency-Aware Task Scheduling (LaTS)

As mentioned earlier, we construct two task topologies for each application: a serial-DAG and a para-DAG. In the task scheduling phase, we need to consider both topologies.

A serial-DAG only makes sense if we schedule the bottleneck bolt on a GPU; otherwise, the latency will be too long. The other non-bottleneck bolts will be scheduled on CPUs because the functions associated with these bolts usually receive much lower speedup ratios by switching to GPU.

Below, we first describe our CPU scheduling strategy – how we schedule a list of bolts to CPUs – and we then briefly describe our GPU scheduling strategy which essentially uses the same technique as CPU scheduling. The CPU scheduling part is needed when we schedule non-bottleneck bolts for a serial-DAG as well as when we schedule all the bolts for a para-DAG. Given a pool of bolts to be scheduled on the CPUs (we refer to them as CPU bolts below), we rank them in the descending order of the required CPU time – we estimate their required CPU time on the same processor. Then we schedule the bolt from the bolt list one by one.

For a given bolt and an edge node, we perform the following two types of latency estimation. First, we measure the node's CPU utilization and use the latency estimation model as shown in Figure Figure 3.5 (a) to estimate the processing latency. Next, we measure the node's available bandwidth and use the bolt's output streaming size to estimate the network transmission latency. The total latency for this bolt-node combination is then the sum of these two types of latency values. In this way, we can estimate the total latency of this bolt on every edge node in the system.

We assign the bolt to the node that gives the minimal latency. After the assignment, we also need to decrease the available resources on that node by removing the amount of resources consumed by this bolt (processor utilization, memory and bandwidth).

We repeat the above process until we finish scheduling all the CPU bolts.

We next describe how we schedule the bottleneck bolt in a serial-DAG to the appropriate GPU. The idea is very similar to the above CPU scheduling. We choose the GPU that gives the shortest overall latency (we estimate both processing latency and networking latency here) to host the bottleneck bolt.

We refer to the above task scheduling algorithm as *LaTS*. Ideally, when a new user stream connects to the edge cloud, we need to run the algorithm on both topologies and choose the assignment that has the lowest latency. Note that when the edge cloud becomes larger or when many users are connected to the edge cloud, it may be too costly to exactly follow this procedure. In this case, we need to develop faster heuristics to perform task scheduling. We describe and evaluate such heuristics in Sections subsection 3.4.2 and subsection 3.4.3 (check Figure Figure 3.10).

3.3.4 Stream Grouping

Apache storm provides the flexibility of specifying how to steer a bolt's output stream to the connecting bolt(s), i.e., streaming grouping. This decision becomes particularly important when the connecting bolts run in the data parallel mode – they run the same task

function but on their own partition of the data set. For example, in our para-DAG shown in Figure Figure 3.4 (b), we can specify which disparity bolts we choose to use and how to partition the stream among the chosen disparity bolts. This decision can vary from frame to frame.

A good stream grouping algorithm can take into consideration the resource variation among the bolts and then partition the data to ensure these data-parallel bolts finish at about the same time. If one such bolt is scheduled on a node with fewer resources and incurs a much longer latency than its peers, then it becomes a straggler [11] and slows down the entire processing. In general, their succeeding bolt has to wait for all the data-parallel bolts to finish before it can start processing. We note that our LaTS task scheduling algorithm is already effective in avoiding stragglers because it tries to place each bolt on the node that yields the shortest latency. However, when the edge servers have vastly different resource levels, stragglers cannot be avoided by the task scheduling phase alone. Clever stream grouping techniques thus become critically important in avoiding stragglers.

Proportional Partition Stream Grouping (Pro-Par): The default stream grouping algorithm in Apache Storm partitions the data set equally among the bolts and cannot avoid stragglers if the bolts have varying computing capabilities. In Hetero-Edge, we propose to adopt a proportional partitioning method, *Pro-Par* in short. In Pro-Par, we periodically estimate the computing capacity of those nodes that host the data parallel bolts and then partition the stream in such a fashion that a node's partition is proportional to its computing capacity.

By default, we equally partition the stream among all the data parallel bolts and measure their processing latency periodically. When the gap between the fastest bolt and the slowest bolt exceeds a certain threshold, we trigger Pro-Par. Specifically, let us suppose each disparity bolt's latency is $\{t_1, ..., t_m\}$ where m is the degree of data parallelism.

We adopt the min-max normalization method to remap the m different latency values to the range (0,1). We estimate the computing capability of each node c_i as $\frac{1}{m \times t_i}$, and the overall computing capability C provided by m nodes is $\sum_{i=1}^{m} c_i$. Then we partition the

stream proportionally to each node's computing capacity and transmit the resulting partition to each bolt.

3.4 Measurements and Experiments

We have implemented the proposed techniques (described in Section section 3.3) on the Hetero-Edge testbed. In this section, we present our evaluation effort in detail. Our evaluation has the following three components: (1) evaluating the proposed schemes using an edge cloud that does not have GPUs, (2) evaluating the proposed schemes using an edge cloud with GPUs, (3) putting everything together and taking a close look at the Hetero-Edge run-time dynamics.

3.4.1 Evaluation of Hetero-Edge with only CPU and Network Heterogeneity

LaTS Better Handling Resource Heterogeneity: We compare three task schedulers – i.e., *round-robin*, *resource-aware*, and *LaTS* – in terms of the ability to handle CPU and network diversity. The image resolution is 1440x1080. We consider a very low frame rate, 1fps, so that we can focus on the latency alone. Here, we consider seven edge servers (Xeon E5-2630, 2.4GHz) with different CPU and network resources. Five nodes have 10Gpbs network links and no other processes. Two nodes have 1Gbps links and their CPUs are already partially occupied (available CPU utilization 10%-40%). We find that LaTS performs the best: 40.0% shorter than round-robin, and 41% shorter than *resource-aware* as shown in Figure Figure 3.3 (b). In addition, we use the yellow bars to show the latency distributions of the 16 disparity bolts that are in data parallel mode. We find that LaTS leads to lower average latency for the 16 disparity bolts and a much lower gap between the fastest and slowest bolts, which is the key to minimizing the overall latency. Note that we have also tried other edge cloud configurations and have observed similar trends. As a result, we believe that LaTS can better address CPU and network diversity among edge servers and lead to a shorter end-to-end latency. It also does a better job preventing stragglers.

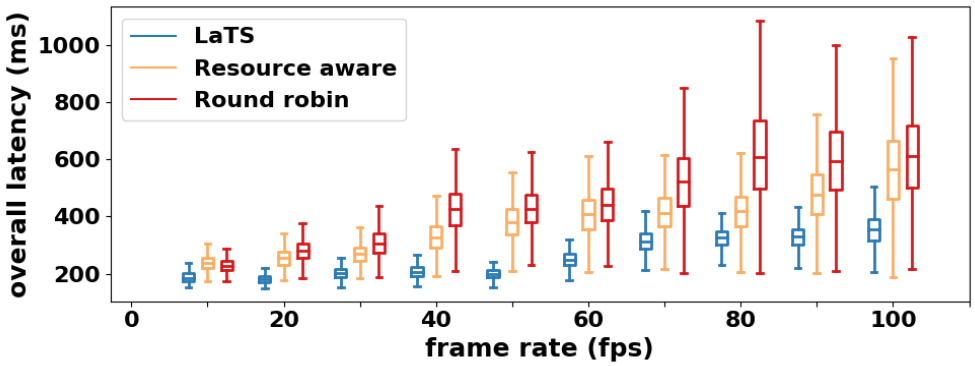


Fig. 3.6 Comparing the latency of *LaTS*, *round-robin* and *resource-aware* at high system load by increasing the single stream's frame rate

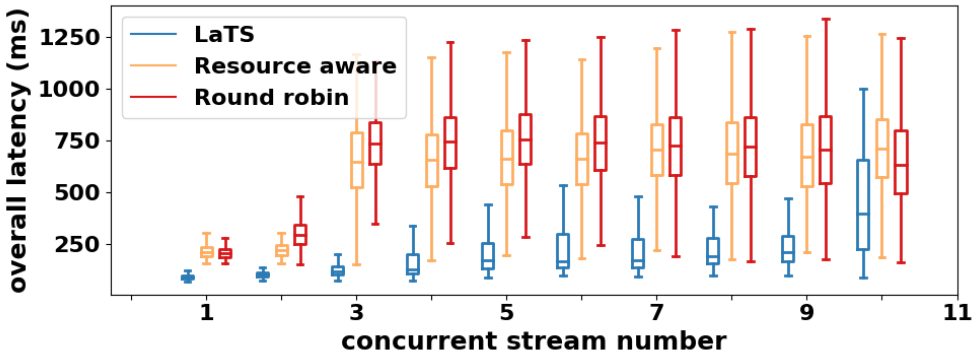


Fig. 3.7 Comparing the latency of *LaTS*, *round-robin* and *resource-aware* at high system load by increasing the number of concurrent streams

LaTS Better Handling High System Load: Next, we look at how these three schemes handle a higher load.

In the first set of experiments, we have 4 Xeon E5-2630 (2.4GHz) edge nodes, two with 10Gbps links and 100% CPU available, one with 1Gbps links and 100% CPU available, and one with 10Gbps links and 20% CPU available. We have a single stream and increase the stream's frame rate, from 10 to 150fps, with an increase of 10fps. The image resolution is 640x480. The results are shown in Figure Figure 3.6. We find that LaTS continuously outperforms the other two schedulers by more than 25%.

In the second set of experiments, we fully load Hetero-Edge with the setting as the resource heterogeneity experiment in Figure Figure 3.3 (b). We fix each stream's frame rate as 30fps and increase the number of concurrent streams, from 1 to 10, with an increase of

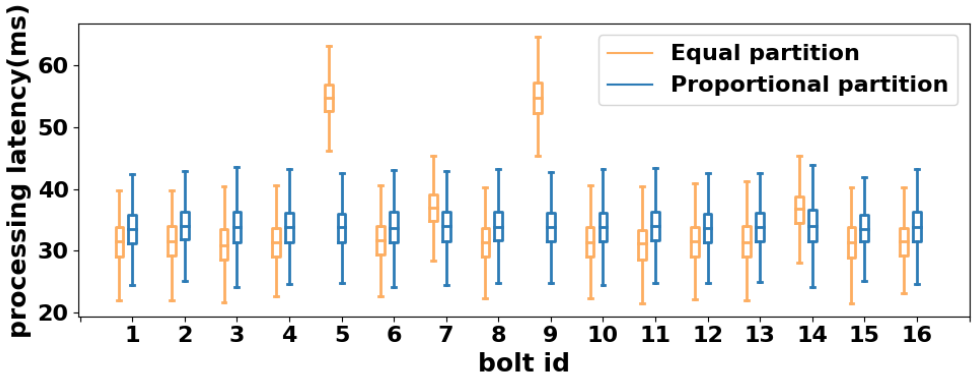


Fig. 3.8 Comparing the latency distribution of data parallel bolts with Pro-Par and equal partition stream grouping

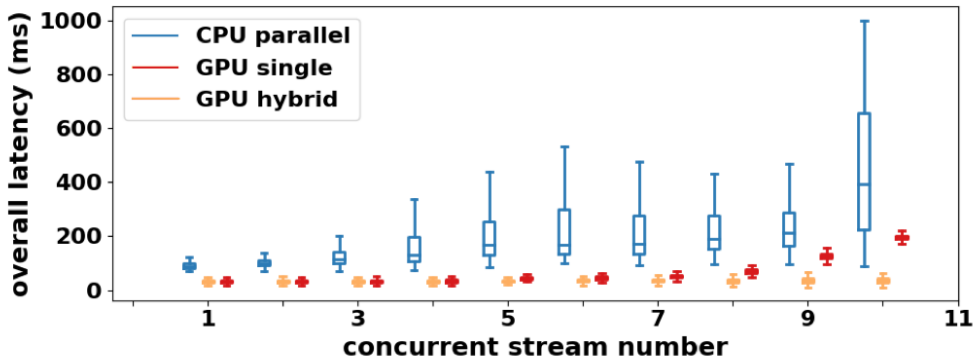


Fig. 3.9 Comparing the latency of CPU Parallel, GPU Single and GPU Hybrid when we have utilized both GPUs and CPUs

1 stream. The results are shown in Figure Figure 3.7. We find that LaTS can support up to 9 concurrent streams without the latency significantly going up. On average, its latency is 66% lower compared to *round-robin* and 61% to *resource-aware*.

Pro-Par Better Handling Stragglers: Next we evaluate the effectiveness of Pro-Par in mitigating the stragglers. In this set of experiments, we target a case wherein the straggler bolts continue to slow down because their nodes have insufficient resources (which is already the best effort by LaTS). The equal partition stream grouping leads to two straggler bolts (5, 9 in Figure Figure 3.8) that require 41% more time compared to other bolts. With our proportional partition stream grouping, it balances the workload based on the computing capability of each bolt and therefore effectively avoids stragglers. Its slowest bolt is 36% faster than the equal partition steam grouping scheme.

3.4.2 Evaluation of Hetero-Edge with GPU, CPU and Network Heterogeneity

In this subsection, we consider edge clouds that have heterogeneous CPUs, networks and GPUs. Since Storm does not consider GPU by default, we only focus on our own schemes in this subsection. We compare the latency results of the following three scheduling strategies: (1) *CPU Parallel*: para-DAG bolts running on CPUs, (2) *GPU Single*: serial-DAG bolts running on the same node (with the bottleneck bolt running on the GPU while non-bottleneck bolts running on the host CPU) and (3) *GPU Hybrid*: the bottleneck bolt of a serial-DAG running on one node's GPU while non-bottleneck bolts run on other nodes' CPUs. Please note that these three schemes are special cases of our LaTS scheme. By understanding which of these three strategies is faster and when to use which, we can greatly speed up LaTS as we do not need to search every possible combination.

We increase the number of streams from 1 to 10, with the frame rate for each stream to be 30fps. Figure Figure 3.9 shows the comparison results. As expected, CPU Parallel gives much longer latency than the two GPU solutions. GPU Hybrid performs better than GPU Single when the concurrent stream number is more than 8 due to the over-utilization of the host CPUs. When the stream number is less than 8, two GPU schemes perform similarly as a result of the low communication overhead.

3.4.3 Supporting real-time edge vision applications through Hetero-Edge

After evaluating each technique in different settings, we finally put together everything and evaluate whether our Hetero-Edge platform can effectively support the intended real-time edge vision applications (3D construction in our example). Suppose we provide 3D reconstruction services to nearby mobile users with our edge cloud (that involves all the nodes in our testbed, including both GPUs and CPUs). Each interested user connects to our edge cloud and starts a stream; after a certain number of frames, the user ends the stream. In our experiments, we use the following synthetic workload: each user stream has a frame rate of 30fps and a video resolution of 640x480; each user initiates a session of 1 minutes

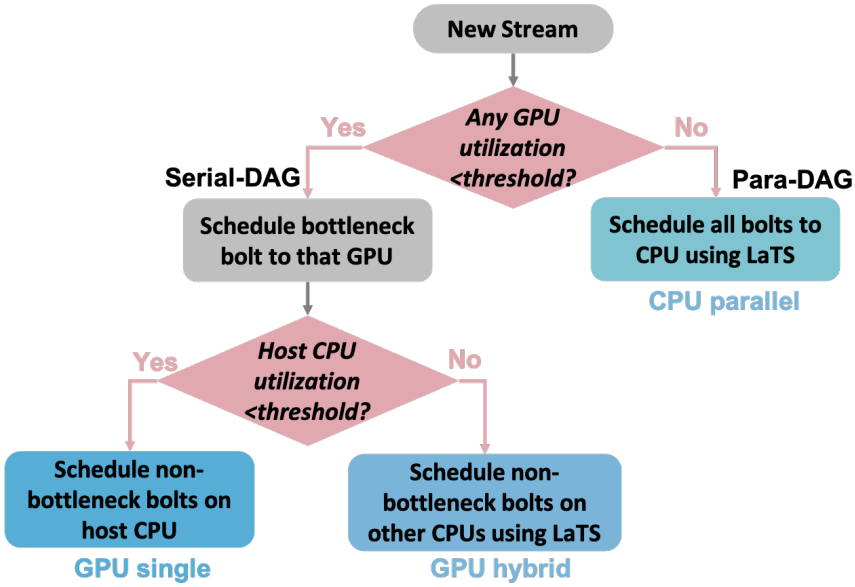


Fig. 3.10 When a new stream arrives at the system, we follow this flow to find out which scheme we are going to use to schedule this stream: GPU Single, GPU Hybrid or CPU Parallel. This flow is faster than exactly going through the LaTS scheduler

and leaves the system; the user session arrival process follows a Poisson distribution with an average arrival rate of 20.

Suggested by the results reported in Figure Figure 3.9, our edge server adopts the policy described in Figure Figure 3.10. Following this policy, when the GPUs and their host CPUs are rather empty, we choose GPU Single to schedule streams. Slowly, the CPUs on those nodes that have GPUs will become busy, and we can switch to GPU Hybrid to schedule the arriving streams. Finally, when all the GPUs get busy, we resort to CPU Parallel to serve the subsequent streams.

We run our service for 2 hours, and report important run time parameters in Figure Figure 3.11. From top to bottom, we have (1) the number of connected streams, (2) the number of GPU Single, GPU Hybrid, and CPU Parallel streams, (3) GPU utilization, (4) CPU utilization of those nodes that have GPUs, (5) CPU utilization of those nodes that do not have GPUs, and (6) the histogram of the end-to-end per frame latency.

We would like to highlight the following observations from the results. Firstly, the average per-frame latency is 32ms, which we believe is satisfying for many real-time edge vision applications. Secondly, our platform can effectively schedule streams across highly

heterogeneous computation nodes – on average, we have the most GPU Single streams and the least CPU parallel streams. Thirdly, we find that the GPUs have lower utilization than their host CPUs because these CPUs spend more time processing the non-bottleneck bolts than GPU processing the bottleneck bolts. As a result, when host GPUs become fully utilized, GPU Hybrid becomes useful.

Lessons Learned: By offering the edge service for 2 hours, we have learned a few lessons regarding application deployment on edge servers. The most valuable lesson we have learned is that it is important to include GPUs in an edge cloud. It can help significantly reduce the per-frame latency. However, we need to pay extra attention to GPU scheduling as well as coordinating GPUs and CPUs to finish one job efficiently. Finally, optimizing CPU scheduling is also very important, such as carefully matching the task demand with resource availability and matching the workload level with resource availability.

3.5 Related Work

In this section, we briefly discuss related work in execution acceleration by edge cloud computing, popular distributed and parallel computing platforms, and relevant task allocation algorithms.

3.5.1 Execution Acceleration via Edge Cloud Computing

Many works in the rising Edge Computing field have been proposed to tackle challenges in systems, models, and applications as it introduces a new computing and communication paradigm. Yang et al. [89] propose to dynamically partition data stream between mobile and cloud server to minimize processing latency by a centralized genetic algorithm. Chaufournier et al. [90] use multi-path TCP to accelerate edge cloud service migration to reduce the network latency when mobile users move away. Pang et al. [91] consider a latency-motivated cooperative task computing framework for selection of edge clouds to provision edge services. Bahreini et al. [92] design an online heuristic algorithm that

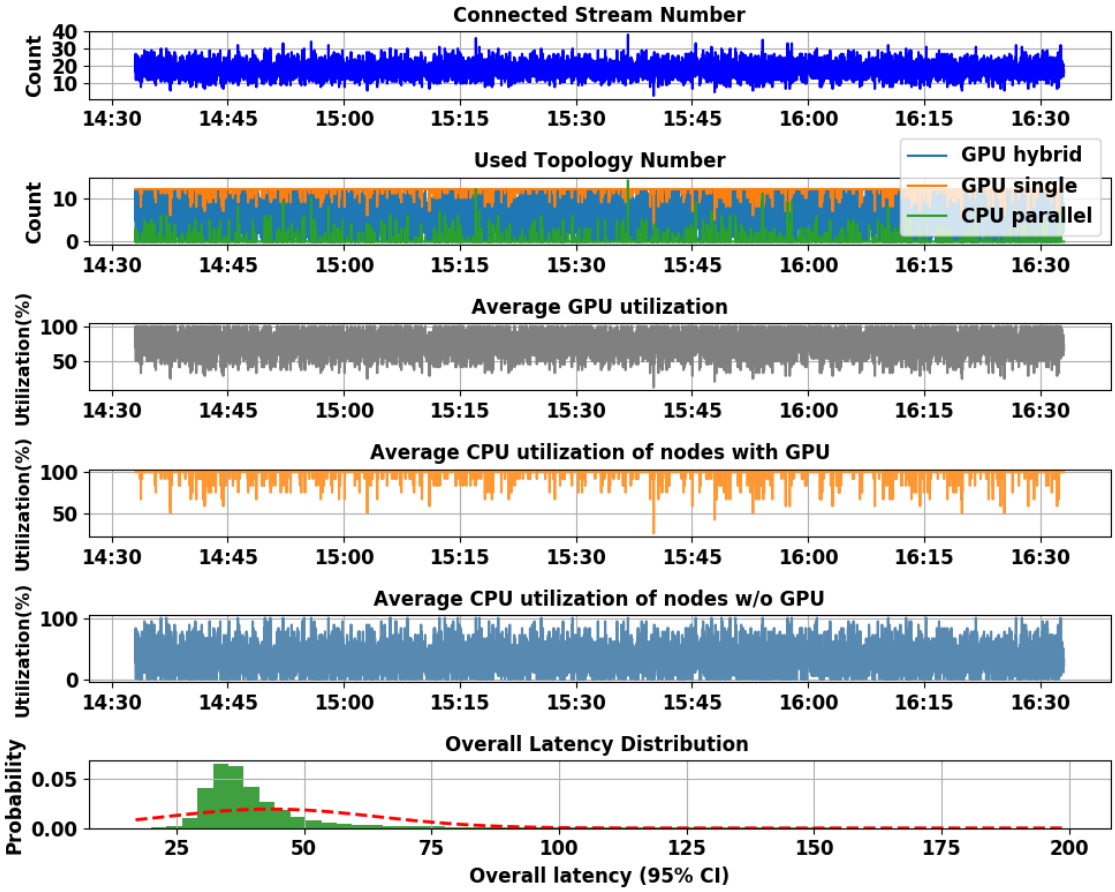


Fig. 3.11 Important run time parameters for our 3D reconstruction edge cloud in a 2-hour duration

efficiently places application tasks in edge clouds to minimize execution time. Zhang et al. [75] propose an edge-based VR gaming architecture where edge clouds perform heavy frame rendering tasks to reduce end-to-end latency significantly. FemtoCloud [93], P^3-Mobile [94] explore idle mobile devices to configure a compute cluster and provisions cloud services at the edge. Liu et al. [95, 96] design and implement high quality and low latency VR and AR system leveraging the support of edge clouds. Users can leverage this mobile cluster to perform parallel programming to accelerate computation speed. Those works focus on optimizing the communication pattern in the systems to achieve lower latency, but they fail to provide a practical execution platform and a resource orchestration mechanism which are aware of resource heterogeneity at the edge.

3.5.2 Task Allocation Algorithms

Efficiently assigning tasks of an application to proper processors is critical to achieve high performance in a heterogeneous computing environment [97]. Task allocation, as an NP-complete problem, has been extensively studied and many heuristics solutions have been proposed according to diverse optimization goals [98]. In terms of Storm platform specific task allocation schemes, several major schedulers have been proposed. T-storm [99] and the work in [100] proposed a traffic-aware task allocation that tries to minimize inter-node and inter-process traffic. R-storm [87] introduced a resource-aware task allocation that intends to increase overall throughput by maximizing resource utilization. Although the above schedulers showed performance improvement over the default Round-Robin mechanism, they failed short in achieving lower latency in the context of edge computing which required proactive available resource estimation and task profiling. The state of art of stragglers mitigation is to introduce speculative execution that waits to observe the progress of the tasks of a job and launches duplicates of those tasks that are slower [11]. This approach, however, is usually applied in cloud computing where computing resource is much more abundant to utilize.

3.6 Concluding Remarks and Future Directions

In this paper, we develop a *latency-aware* edge resource orchestration platform based on Apache Storm. The platform aims to support real-time responses to edge applications that are computation intensive. The main contribution of our platform stems from a set of *latency-aware* task scheduling schemes. By deploying the proposed platform on a group of edge servers with heterogeneous CPU, GPU and networking resources, we show that we are able to support real-time edge vision applications, rending the per frame latency around 32ms. Our study shows that edge cloud computing is indeed a promising platform to support emerging edge applications. Moving forward, we will continue to investigate how to further drive down the latency, e.g., distributing the bottleneck bolt across multiple GPUs. We

will also investigate how to efficiently integrate our edge cloud computing platform with traditional central clouds to support applications that need to utilize both modes. Another future work topic is that of understanding the impact of access network bandwidth on edge resource assignment and scheduling.

CHAPTER 4
PARALLEL OFFLOADING TO FURTHER ACCELERATE COMPUTER VISION JOBS

4.1 Introduction

In the past few years, we have witnessed the rapid development of Deep Neural Networks (DNNs), due to the fast-growing computation power and data availability [101]. Thanks to these advancements, mobile applications, particularly mobile vision applications, enjoy a performance boost in various vision-related tasks such as photo beautification, object detection and recognition, and reality augmentation [102]. However, to achieve state-of-the-art performance, DNN models (e.g., [103, 104]) usually have complicated structures with numerous parameters, hence a high demand in computation and storage. As a result, it is challenging to run full-size DNN models on mobile devices, even running into heat dissipation issues. Meanwhile, mobile deep vision applications are often interactive and require fast or even real-time [1] responses. Examples include adversarial point cloud generation [105] that reconstructs 3D scenes for intuitive surrounding interpretation and video object semantic segmentation [106] that facilitates personal activity recognition. In these cases, it is hard, if not impossible, to satisfy the applications' latency requirements due to the limited processing capacity on mobile devices.

To this end, researchers have spent a great deal of effort to improve the performance of mobile deep vision applications. On the one hand, various techniques have been developed to make DNN models smaller to reduce the computation load, e.g., weight and branch pruning and sharing [107, 108], tensor quantization [109, 110], knowledge distillation [111], and network architecture search [112]. However, these techniques often lead to compromised model accuracy due to the fundamental trade-off between model size and

[1] Frame rate required for real-time processing is application dependent.

model accuracy [113]. On the other hand, people have proposed to increase the computing resources by using massive accelerators, such as GPU, FPGA [114] and ASIC [115]. Nevertheless, due to the fundamental limits of size and power, mobile devices still fall short to meet the requirements of target applications.

To solve these challenges, several offloading approaches have been proposed [96, 116, 117, 118, 119, 120, 121]. By offloading the intensive model inference to a powerful edge server, for example, AWS Wavelength [122], the inference latency can be significantly reduced. With the high bandwidth and low latency provided by the emerging 5G networks [123], offloading is promising to provide a good user experience for mobile deep vision applications. However, existing offloading methods are insufficient in two aspects.

Firstly, most existing solutions use low-resolution images through the entire pipeline, which makes the inference task lightweight, but lose the opportunity to leverage the rich content of high resolution (e.g., 2K or 4K) images/frames. Taking advantage of such rich information is important for applications such as video surveillance for crowded scenes [124], real-time Autopilot system [125], and online high-resolution image segmentation [126]. Secondly, most existing methods only consider offloading tasks between a single pair of server and client, assuming that no competing clients or extra edge resources available. In practice, a single edge server is equipped with costly hardware, for example, Intel Xeon Scalable Processors with Intel Deep Learning Boost [127] or NVIDIA EGX A100 [128], which are typically shared by multiple clients (i.e., multi-tenant environment). Moreover, the heterogeneous resource demands of applications running on edge servers [129] and highly dynamic workloads by mobile users [130] lead to resource fragmentation. If the fragmentation cannot be efficiently utilized, it may produce significant resource waste across edge servers.

To this point, in order to meet the latency requirements of deep mobile vision applications with heterogeneous edge computing resources, *it is advantageous to offload smaller inference tasks in parallel to multiple edge servers*. This mechanism can benefit many real-

world deep vision tasks, including multi-people keypoint detection for AR applications and multi-object tracking for autonomous driving tasks [131], where objects can be distributed to different servers for parallel task processing. Meanwhile, offloading to multiple servers imposes several challenges. Firstly, it requires the client to effectively partition the inference job into multiple pieces while maintaining the inference accuracy. In the case of keypoint detection or instance segmentation, simply partitioning a frame into several slices may split a single instance into multiple slices, therefore, dramatically decreasing the model accuracy. Secondly, the system needs to be aware of available computation resources on each server and dynamically develops the frame partitioning solution, so that it can ensure no server in the parallel offloading procedure to become the bottleneck. Finally, such a system should have a general framework design that is independent of its host deep vision applications.

To address the aforementioned challenges, we propose and design **ELF**[2], a framework to accelerate high-resolution mobile deep vision offloading in heterogeneous client and edge server environment, by distributing the computation to available edge servers adaptively. ELF adopts three novel techniques to enable both low latency and high quality of service. To eliminate the accuracy degradation caused by the frame partitioning, we first propose a content-aware frame partitioning method. It is promoted by a fast recurrent region proposal prediction algorithm with an attention-based LSTM network that predicts the content distribution of a video frame. Additionally, we design a region proposal indexing algorithm to keep track of the motion across frames and a low resolution compensation solution to handle new objects when first appear. Both work jointly to help understand frame contents more accurately. Finally, ELF adopts lightweight approaches to estimate the resource capacity of each server and dynamically creates frame partitions based on the resource demands to achieve load balance. Overall, ELF is designed as a plug-and-play extension to the existing deep vision networks and requires minimal modifications at the application level. We have implemented ELF on commercial off-the-shelf servers and four

[2]Elf is a small creature in stories usually described as smart, agile, and has magic power

mobile platforms in Linux and Android OS, supporting Python, C++, and Java deep vision applications. We will make our code open source and available at GitHub.

The main contributions of this paper are as follows.

- To the best of our knowledge, we are the first to propose a high-resolution mobile deep vision task acceleration system that offloads the computation to multiple servers to minimize the end-to-end latency.
- To perform the computation offloading from mobile to server while simultaneously considering image content, computation cost, and server resource availability, we propose a set of techniques including recurrent region proposal prediction, and region proposal centric video frame partitioning and offloading, and region proposal computation cost estimation.
- We have built a prototype system with comprehensive experiments to demonstrate that our ELF system can be integrated with 10 state-of-the-art deep vision models and reduce the end-to-end latency by parallel offloading, up to 4.85×, with using 52.6% less bandwidth on 4 edge servers while keeping the accuracy sacrifice within 1%.
- We have learned valuable lessons of the relations between inference latency and the model design. Such lessons will help the vision community to better design models to benefit more from parallel offloading.

4.2 Motivation and Challenges

Target Applications. In this paper, we target those applications that employ state-of-the-art convolutional neural network (CNN) models to conduct a variety of challenging computer-vision tasks from images or videos. Examples include image segmentation, multi-object classification, multi-person pose estimation, and many others. In general, those applications take an input image or video frame which is often of high resolution, e.g., 1920×1080,

containing multiple objects, and perform a two-step processing task. First, they use CNN networks to extract feature maps from the input and generate region proposals (RPs) for every object. Each RP is a candidate region where an object of interest – for example, a cat or child – may appear. Second, they use a CNN network to evaluate each RP and output the fine-grained result such as the classified object type or the key body points of a person. These state-of-the-art CNN models are usually highly computation intensive and run at a low frame rate, e.g., from 0.5 to 10 frames per second (fps) even on a high-end GPU (e.g. NVIDIA TITIAN 1080Ti) [132, 103, 133].

Limitations of Existing Task-Offloading Approaches. Offloading the inference tasks of CNNs onto an edge server is a promising approach to realizing the target applications on mobile devices [96, 134]. However, these existing task-offloading approaches are limited in two critical aspects. First, they only support task offloading to just one server, assuming that the server has sufficient resources to finish the offloaded task in time. However, a costly offloading server, for example, Intel Xeon Scalable Processors with Intel Deep Learning Boost [127] or NVIDIA EGX A100 [128], is usually shared by multiple clients and thus may not have sufficient resources to run a task. To demonstrate it, we profiled the computing latency of ResNet50 [135]. Each client runs on NVIDIA Jetson Nano [136] with 802.11.ax and the server runs the model inference on an NVIDIA TITIAN V GPU. The computing latency goes up in a linear pattern from 25.9 ms to 162.2 ms when changing the number of concurrent clients from 1 to 4. To handle the latency burst, Amazon SageMaker [137] adopts Kubeflow Pipelines to orchestrate and dynamically configure the traffic running on each server. However, this solution cannot handle resource fragmentation and may waste the computing cycles.

Another limitation of existing solutions is that they use low-resolution (e.g., 384×288 [138]) images or videos to make the inference task lightweight. However, cameras on today's mobile devices typically capture with a much higher resolution such as 2K and 4K. Such a big gap causes two problems. On one hand, those existing low-resolution solutions

(a) Equal partitioning (b) Ideal partitioning

Fig. 4.1 Examples of video frame partitioning. The simple partitioning method in (a) may split pixels of the same object into multiple parts and yield poor inference results. We can achieve much better partitioning using ELF, close to the ideal partitioning shown in (b)

fail to leverage the rich information of high-resolution images and videos to enable advanced applications such as various video analytics, for example, smart intersection [139]. Existing studies have already shown running object recognition related tasks on high-resolution images can largely increase the detection accuracy [140] . On the other hand, supporting high resolutions requires more computations and further undermines the assumption that one server can provide sufficient resources for the entire application. Our measurement results show that the inference latency of MaskRCNN [132] running on Jetson TX2 [141] boosts by 25%, 50% and 300% with increasing the image resolution from 224×224 to 1K, 2K and 4K, respectively, making the offloading harder.

To address the limitations of the existing work and the high resource demands of the target applications, in this paper, we design ELF, a lightweight system to accelerate high-resolution mobile deep vision applications through parallel task offloading to multiple servers.

Design Challenges. There are several key challenges in designing the ELF system. The first challenge lies in how to partition the computation. Broadly speaking, there are two approaches, model-parallel and data-parallel. Model parallelism, i.e., splitting a large model into multiple subsets of layers and running them on multiple servers, generates the large intermediate outputs from convolution layers which would lead to high communication overhead among servers [142]. For example, cracking open the DNN black-box [117]

demonstrates that ResNet152 [135] produces the outputs with 19-4500× larger than the compressed input video. In this work, we explore data-parallelism by partitioning an input frame and offloading each frame partition to a different server. However, as shown in Figure Figure 4.1(a), the simple equal partitioning may not work because 1) offloading a partition containing parts of an object may significantly reduce the model accuracy and 2) offloading a partition containing no objects may lead to excessive waste. Instead, we need to develop a smart video frame partitioning scheme to generate the ideal image partitioning shown in Figure Figure 4.1(b). The second challenge is how to distribute the tasks to multiple servers to minimize the total model inference latency. Ideally, all the servers should finish their tasks at the same time. However, that is hard to achieve because multiple dynamic factors must be considered together: the number of objects in the input images, the resource demand of processing each object, the number of servers and their available resources. Furthermore, another challenge in ELF is to minimize the workload of the resource-limited mobile device. In particular, the video frame partitioning is the step before offloading, running on the mobile device, and thus must be efficient and lightweight.

4.3 Overview and Design Guidelines

ELF intends to address the above challenges by adopting the following steps:

a) **Recurrent region proposal prediction**. On the mobile end, whenever a new video frame arrives, ELF first predicts its region proposals based on the list of region proposals in history frames. The prediction results include each region proposal's coordinates. Here, a *region proposal* (RP) refers to a group of pixels containing at least one object of interest, e.g., a vehicle or a person.

b) **Frame partitioning and offloading**. Given the list of predicted RPs, ELF partitions the frame into "RP boxes". All the RP boxes collectively cover all the RP pixels while trying to remove unnecessary background pixels. An RP must be entirely included in at least one RP box. ELF then offloads these partitions to suitable edge servers for

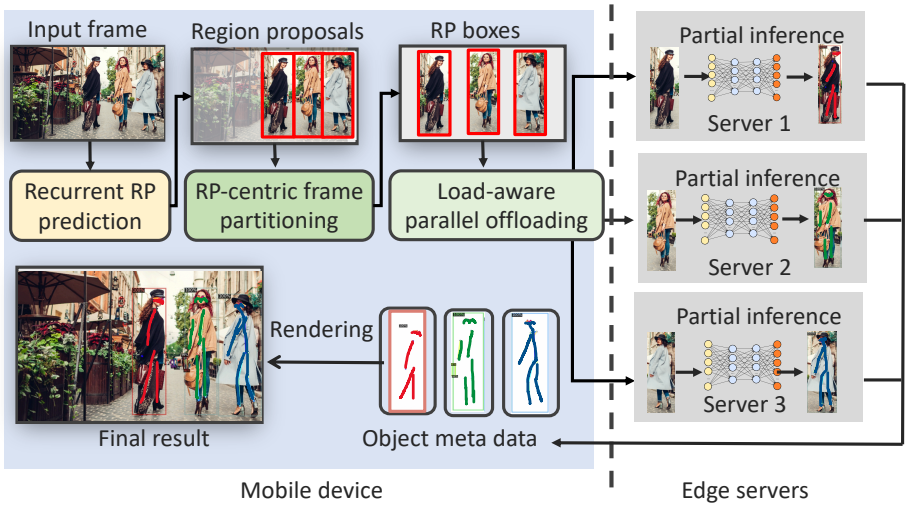


Fig. 4.2 ELF system architecture. We explain the architecture using a multi-person pose estimation example with three edge servers

processing, taking into consideration the partition's computation cost and server resource availability.

c) **Partial inference and result integration**. Taking the offloaded partition as input, the edge servers run the application-specific CNN networks to yield partial inference results. These partial results are finally integrated at the mobile side to form the final result.

The above workflow is illustrated in Figure Figure 4.2. While the third step is natural and easy to do, the first two steps call for careful designs to achieve the goals of ELF. In the rest of this section, we discuss the design guidelines of these two key components of ELF, focusing on how they are designed to address the challenge described in Section section 4.2.

4.3.1 Recurrent Region Proposal Prediction

We adopt the following guidelines to devise the recurrent RP prediction algorithm: 1) the algorithm is lightweight; 2) the algorithm can effectively learn the motion model of the objects/RPs from history frames; and 3) the algorithm pays more attention to more recent frames. Here, a well-designed algorithm can accurately predict the RP distribution and help minimize the impact of the frame partitioning upon the deep vision applications' model accuracy. Following the guidelines above, we devise an attention-based Long Short-

Term Memory (LSTM) network for recurrent RP prediction. Note that the main-stream RP prediction/tracking algorithms require large CNN models [143]. Instead, our approach efficiently utilizes the historical RP inference results and converts the computing-intensive image regression problem to a light-weight time series regression problem. As part of the prediction algorithm, we also develop an RP indexing algorithm that keeps track of the motion across frames. Finally, we also propose a *Low Resolution Compensation* scheme to handle new objects when they first appear.

4.3.2 RP-Centric Video Frame Partitioning and Offloading

Partitioning a video frame allows ELF to offload each partition to a different edge server for parallel processing. Ideally, a well-designed frame partitioning scheme should show a negligible overhead in the operation of partitioning and merging. Keeping these goals in mind, we design an RP-centric approach with the following guidelines.

Content awareness. The partitioning algorithm should be aware of the number of and locations of RPs in a frame and be inclusive. As such, the algorithm is supposed to have each RP completely included in at least one frame partition. Otherwise, the object contained in this RP may be missed.

Computation cost awareness. Depending upon the objects contained in each partition, partitions have different computation costs. For example, it is usually more challenging to identify multiple overlapping vehicles with similar colors than identifying a single vehicle. The algorithm should thus take into consideration this cost heterogeneity to achieve load balancing among the servers.

Resource awareness. After partitioning, the algorithm next matches these partitions to a set of edge servers. Unlike central clouds, edge cloud servers exhibit heterogeneous computing/storage/networking resources due to the distributed nature and high user mobility [116]. This makes the matching problem even more challenging. A poor match may result in job stragglers that complete much slower than their peers and thus significantly increases the

overall latency.

4.4 Fast Recurrent Region Proposal Prediction

When a new frame arrives, ELF predicts the coordinates of all the RPs in the frame, based on the RPs in the previous frames. In this section, we present three components that are key to achieve fast and effective RP prediction: an attention-based Long Short-Term Memory (LSTM) prediction network, a region proposal indexing algorithm and a low-resolution frame compensation scheme. We choose to use attention-based LSTM for its powerful capabilities of learning rich spatial and temporal features from a series of frames.

4.4.1 Problem Definition and Objective

As the objective of training the LSTM network is to acquire accurate RP predictions, the objective of the network (i.e., the loss function in our prediction network) can be expressed as:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{R}}_i^t, \boldsymbol{R}_i^t) &= \textstyle\sum_i \left[(\hat{x}_i^t - x_i^t)^2 + (\hat{y}_i^t - y_i^t)^2 + (\hat{a}_i^t - a_i^t)^2 \right] \\ s.t. \quad \hat{\boldsymbol{R}}_i^t &= f(\boldsymbol{R}_i^{t-N}, \boldsymbol{R}_i^{t-N+1}, ..., \boldsymbol{R}_i^{t-1}), \end{aligned} \tag{4.1}$$

where the vector $\boldsymbol{R}_i^t$ denotes the i-th RP at frame t, and $\boldsymbol{\theta}$ is the model parameters. Here, $\hat{\boldsymbol{R}}_i^t$ is a vector of 4 attributes: $[x_{\text{tl}}, y_{\text{tl}}, x_{\text{br}}, y_{\text{br}}]$, which are the x, y coordinates of its top-left and bottom-right corners, respectively. Further, N is the number of previous frames used in the prediction network $f(\cdot)$. In the rest of this section, we explain our algorithmic effort in minimizing the prediction error as calculated in Equation (4.1).

4.4.2 Attention-Based LSTM Network

Below we present the details of our attention-based LSTM RP prediction network.

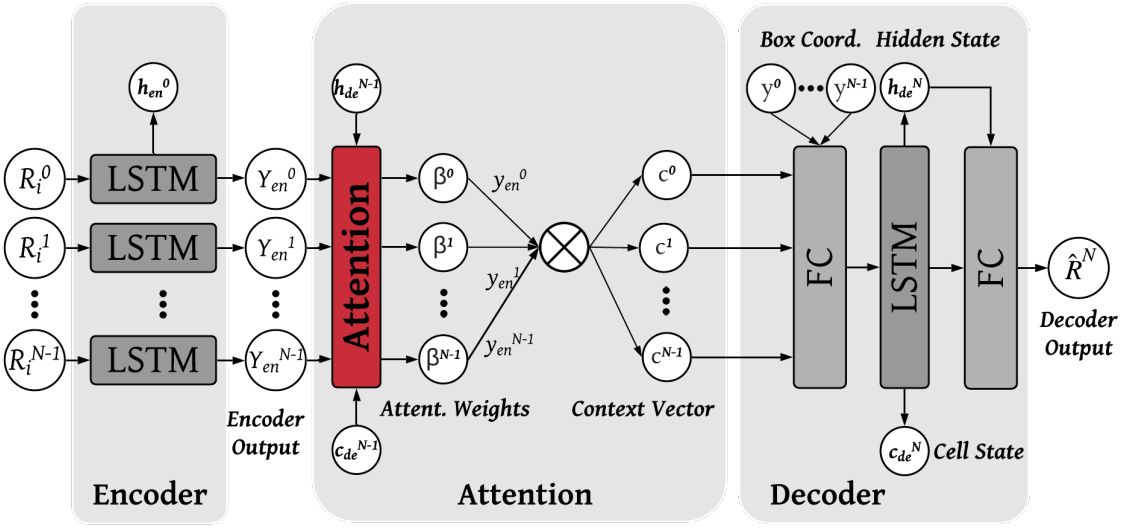


Fig. 4.3 Our attention-based LSTM network

Network structure

Recently, attention-based RNN models [144, 145] have shown their effectiveness on predicting time series data. In this work, we adapt a dual-stage attention-based RNN model (DA-RNN [146]), and develop a compact attention-based LSTM network for RP predictions. Our model consists of three modules – an encoder module, an attention module and a decoder module, as shown in Figure Figure 4.3.

Encoder: To predict the i-th RP in the current frame, the encoder takes the spatial and temporal information (i.e., the RP's locations in history frames) of the i-th RP from N past frames $\boldsymbol{R}_i^t \in \mathbb{R}^{5\times 1}$ as input, and encodes them into the feature map $\{\boldsymbol{Y}_{\text{en}}^t\}, t \in \{0, ..., N-1\}$. This encoding is conducted by a two-layer LSTM [147], which can be modeled as:

$$\boldsymbol{Y}_{\text{en}}^t = f_{\text{en}}(\boldsymbol{Y}_{\text{en}}^{t-1}, \boldsymbol{R}^t), \tag{4.2}$$

where $f_{\text{en}}(\cdot,\cdot)$ denotes the LSTM computation.

Attention: Subsequently, we adopt an attention module which is a fully-connected layer to select the most relevant encoded feature. The first step is to generate the attention weight

Algorithm 1 Region Proposal Indexing

Input: RP $\boldsymbol{R}_i^{t-1} = [x_{\text{tl},i}^{t-1}, y_{\text{tl},i}^{t-1}, x_{\text{br},i}^{t-1}, y_{\text{br},i}^{t-1}]$ for object i in frame $t-1$, where $i \in [0, 1, ..., m^{t-1}]$ and m^{t-1} is number of objects in frame $t-1$. Label set is $\boldsymbol{L}$.

Output: For frame at t, assign an index to each region proposal $\boldsymbol{R}^t$.

{Step-1. Initialization:}

if $t < N$ **then** ▷ label with a consistent index

  $\boldsymbol{R}_i^t[5] \leftarrow l_i^t,\ l_i^t \in \boldsymbol{L}; \quad \forall i \in [0, 1, ..., m^{t-1}]$

end if

{Step-2. Measure distance and area:}

for $i := 1$ to m^{t-1} **do**

  for $k := 1$ to m^t **do**

    $D_{i,k}^{\text{x}} \leftarrow |(x_{\text{tl},i}^{t-1} + x_{\text{br},i}^{t-1}) - (x_{\text{tl},k}^{t} + x_{\text{br},k}^{t})|/2$ ▷ x-axis.

    $D_{i,k}^{\text{y}} \leftarrow |(y_{\text{tl},i}^{t-1} + y_{\text{br},i}^{t-1}) - (y_{\text{tl},k}^{t} + y_{\text{br},k}^{t})|/2$ ▷ y-axis.

    $A_{i,k} \leftarrow |1 - \frac{(x_{\text{br},i}^{t-1} - x_{\text{tl},i}^{t-1})(y_{\text{br},i}^{t-1} - y_{\text{tl},i}^{t-1})}{(x_{\text{br},k}^{t} - x_{\text{tl},k}^{t})(y_{\text{br},k}^{t} - y_{\text{tl},k}^{t})}|$ ▷ Area

  end for

end for

{Step-3. Match and label:}

for $k := 1$ to m^t **do**

  $\hat{i} \leftarrow \arg\min_i \ \{A_{i,k}\}_{i=0}^{m^{t-1}}; \quad \text{s.t. } D_{i,k}^{\text{x}} < 0.02,\ D_{i,k}^{\text{y}} < 0.02,\ A_{i,k} < 0.2$

  if $\hat{i}$ is not $None$ **then**

    $\boldsymbol{R}_k^t[5] = \boldsymbol{R}_{\hat{i}}^{t-1}[5]$ ▷ label with matched RP

  else

    $\boldsymbol{R}_k^t[5] = l_k^t,\ l_k^t \in \boldsymbol{L}$ ▷ new label for unmatched

  end if

end for

$\boldsymbol{\beta}$:

$$l^t = \boldsymbol{W}_2 \tanh(\boldsymbol{W}_1 [\boldsymbol{Y}_{\text{en}}; c_{\text{de}}^{N-1}; h_{\text{de}}^{N-1}]) \tag{4.3}$$

$$\beta^t = \frac{\exp(l_i^t)}{\sum_{j=T-N-1}^{T} \exp(l_j^t)} \tag{4.4}$$

where $[\boldsymbol{Y}_{\text{en}}; c_{\text{de}}^{N-1}; h_{\text{de}}^{N-1}]$ is a concatenation of the encoder output $\boldsymbol{Y}_{\text{en}}$, decoder cell state vector c_{de}^{N-1} and decoder hidden state vector h_{de}^{N-1}. $\boldsymbol{W}_1$ and $\boldsymbol{W}_2$ are the weights to be optimized. Thereafter, the context vector can be computed as:

$$\boldsymbol{c}^t = \sum_{j=0}^{N-1} \beta_j^t \boldsymbol{Y}_{\text{en}} \tag{4.5}$$

which captures the contributions of different encoder outputs.

Decoder: The decoder module processes the context vector through a fully connected layer, an LSTM model and a fully-connected regressor.

4.4.3 RP Indexing

To precisely predict a region proposal, we need to collect historical data, which provides necessary information such as motion model and trajectories. However, many vision applications, such as those discussed in Section section 4.2, commonly output object labels in random order. Thus it is hard to match and track region proposals across frames. For example, let us look at the example illustrated in Figure Figure 4.4, where the same objects (and RPs) in consecutive frames have different labels. We use such an indexing algorithm instead of a vision-based matching algorithm because the latter could introduce significant overhead [148].

To address this issue, we devise a light consistent RP indexing algorithm. From the very first video frame, ELF assigns a unique index to each region proposal. In each upcoming frame, ELF matches each RP with the corresponding index that was assigned earlier. Note that, if an RP includes a brand-new object that was not seen before, a new index will be automatically assigned. Here, we match the RPs across frames based on a combination of *RP position shift* and *RP area shift*. The *RP position shift* measures the change of the center point along the x-/y-axis between the current frame and the previous frame, as specified by Lines 6 and 7 in Algorithm Algorithm 1. A larger value indicates a bigger spatial shift and thus a lower matching probability. The *RP area shift* measures the amount of area change between the RPs in two adjacent frames, as specified by Line 8 in Algorithm Algorithm 1. A lower value indicates a higher matching probability. In our work, when the x/y RP position shift are both under 0.02 and the area shift ratio is under 0.2, we declare a match. We select the thresholds because they help to generate the lowest prediction loss in the evaluation. The sum of RP position shift and RP area shift will be taken as an additional metric when

there exist multiple RPs simultaneously satisfying the above threshold requirement.

RP Expansion. Another challenge in RP prediction lies in the possibility that the predicted RP bounding box may not cover all the pixels of an object due to motion. For example, as shown in Figure Figure 4.5, the predicted bounding box excludes the person's hands, which will affect the object detection performed on the edge server. To address this challenge, we carefully expand the bounding box by $p\%$. The downside of this scheme is the increased data transmission and computation. We conduct a trade-off study in Section subsection 4.7.3. Here, we can adopt different strategies to determine how much we would like to expand. First, we could vary the value of p based upon the corresponding RP position shift amount. The more the shift, the more we can expand. Second, we can use the prediction confidence level. If the model shows higher confidence upon the current RP prediction, we assign a lower expansion ratio.

4.4.4 Handling New Objects When First Appear

The above attention LSTM-based prediction can deal with only the objects that already occurred in the previous frame, but not new ones never seen before. In this subsection, we discuss how to handle the new objects when they appear for the first time in a frame.

To handle new objects, we propose a *low resolution compensation* (LRC) scheme with a balanced trade-off between computation overhead and new-object detection accuracy. Importantly, while inference with the down-sampled frame cannot produce fine-grained outputs that are required by the applications, such as object masks or key body points, we find that inference with down-sampled frames can still detect the presence of objects. Figure Figure 4.15 and Figure Figure 4.14 validate this observation. To reduce the computation overhead, LRC down-samples a high-resolution video frame by a max-pooling operation. Then ELF offloads the resized video frame, along with the partitions from regular sized partitions, to edge servers to run application-specific models, which usually consist of an object detection component. Based on the inference results, ELF can roughly locate the new

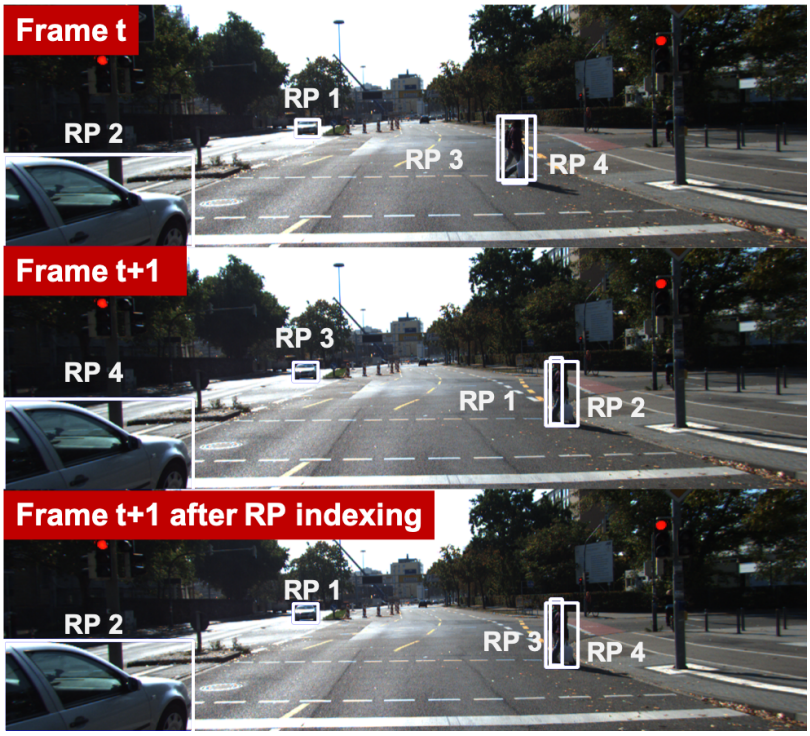


Fig. 4.4 An example result for RP indexing

objects in the frame.

Please note that here we use the same application-specified deep learning neural networks in the LRC module even though it may lead to a higher computation overhead than some lightweight networks. In this way, we do not compromise the new object detection accuracy. Meanwhile, ELF runs LRC once per n frames to reduce such an overhead. n is a hyperparameter, indicating the trade-off between computation cost and at most n-frame delay to realize new objects.

4.5 RP-Centric Video Frame Partitioning and Offloading

Based on the RP predictions, ELF partitions a frame into multiple pieces, focusing on regions of interest while removing unnecessary dummy background pixels. Video frame partitioning plays a dominant role in minimizing the offloading traffic and balancing workloads across edge servers.

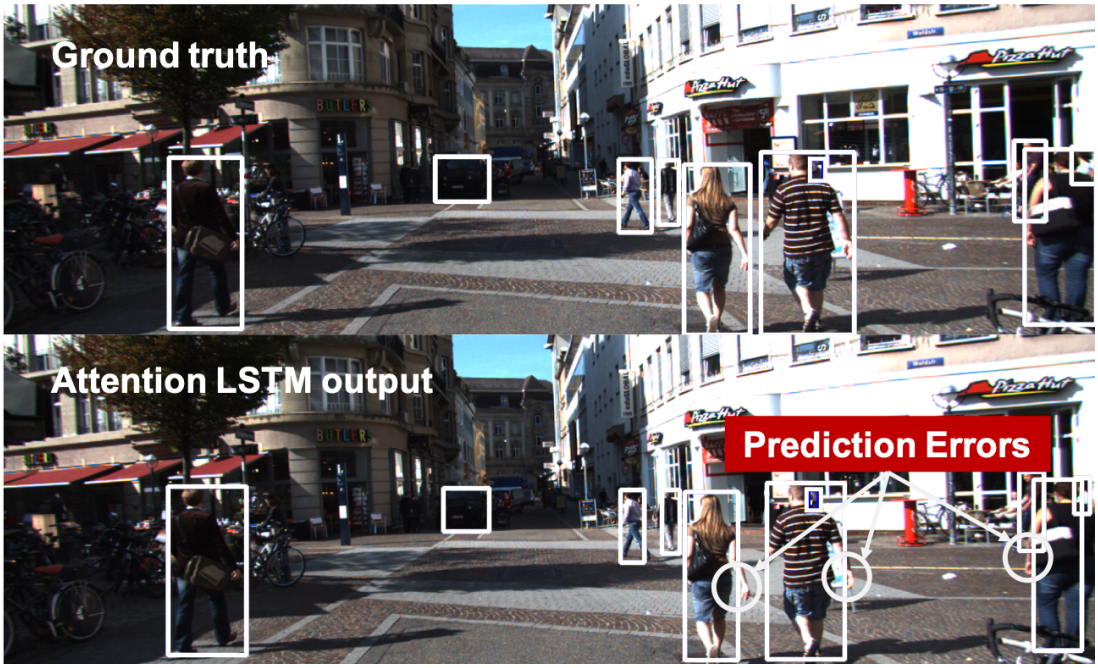


Fig. 4.5 An example prediction error. Part of the objects are outside of the predicted RP bounding box

4.5.1 Problem Statement

ELF takes the following items as input: (i) video frame F_t at time t, (ii) the list of RP predictions in which $\boldsymbol{R}_i^t$ denotes the i-th RP in frame F_t, with $i \in [1, ..., M]$ and M as the total number of RPs, and (iii) the available resource capacity, with p_j^t denoting the available resource capacity of the j-th server ($j \in [1, ..., N]$) at time t. Based on the input, ELF packs the M RP processing tasks and one LRC task into N' offloading tasks ($N' \leq N$), and offload each task onto an edge server.

The overall objective of the partitioning and the offloading process is to minimize the completion time of the offloading tasks that are distributed across N' edge servers. In other words, minimizing the completion time of the task which has the longest execution time among all the tasks. Please note that we assume that the mobile only has access to a *limited* number of servers and that we try to make full use of these servers to minimize the application's completion time.

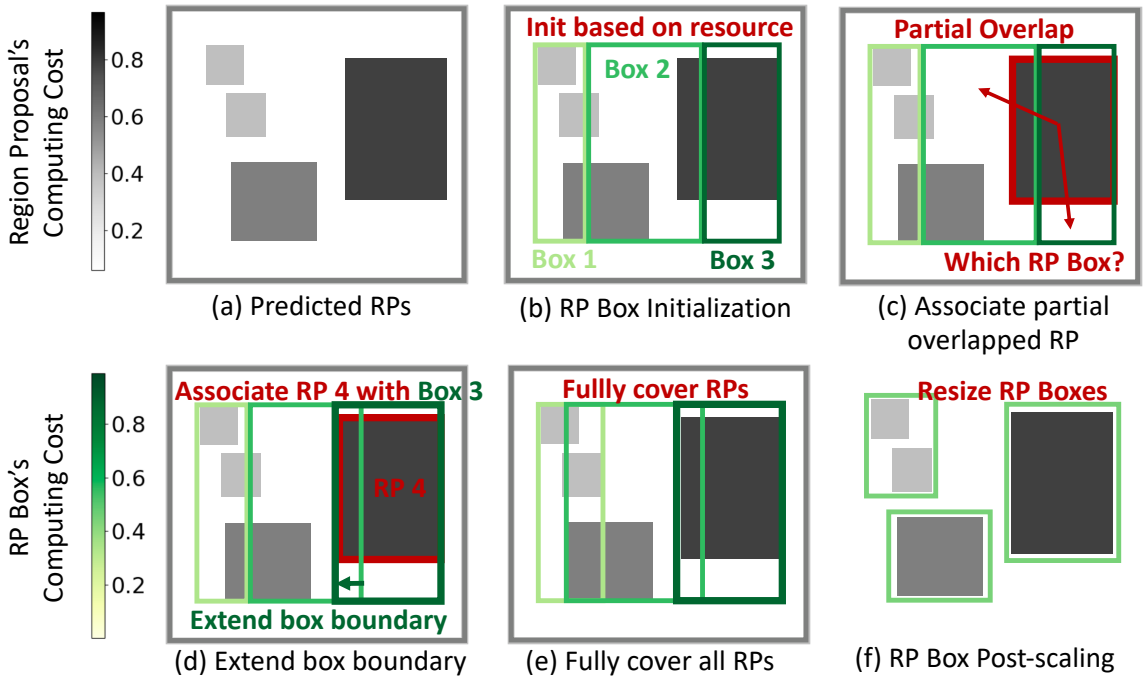


Fig. 4.6 RP-centric frame partitioning pipeline with an example frame

The optimization objective can be written as:

$$\begin{aligned} &\min \quad \max(\{T_k^t\}) \quad k \in [1, ..., N'], \\ &s.t. \;\; T_k^t = T_{\text{rps},k}^t + T_{\text{lrc},k}^t \cdot \mathbf{1}_{(t \bmod n=0)} \cdot \mathbf{1}_{(\arg\max\{p^t\}=k)}, \\ &T_{\text{rps},k}^t \approx \frac{C_{\text{rps},k}^t}{p_k^t}, \;\; T_{\text{lrc},k}^t \approx \frac{C_{\text{lrc},k}^t}{p_k^t} \end{aligned} \tag{4.6}$$

where T_k^t denotes the completion time on the k-th server[3] at time-t. T_k^t consists of two completion-time terms, $T_{\text{rps},k}^t$ and $T_{\text{rps},k}^t$, for RPs and LRC respectively. $\mathbf{1}_{\text{condition}}$ returns 1 if and only if the condition meets, otherwise returns 0. Further, $C_{\text{rps},k}^t$ and $C_{\text{lrc},k}^t$ are the computing cost of RP box and LRC offloading to server k, which will be described in Equation (4.8).

Below we will describe our scheduling algorithm that tries to satisfy this objective.

[3]We re-index the server with $k \in [1, ..., N']$ instead of $j \in [1, ..., N]$, owing to the aforementioned task packing.

4.5.2 Why Not Directly Schedule Each Individual RP Task?

After predicting the list of RPs, a straightforward scheduling approach is to cut out all the RPs and individually schedule each RP processing task onto edge servers. In this case, if the number of RPs is larger than the number of available servers, we may need to schedule multiple small RP tasks onto one server. While this sounds intuitive in many domains, it may not work the best for vision tasks due to the potential fragmentation problem. First, the execution time of r (r is a small number such as 2 and 3) small RP (e.g., $< 5\%$ size of the original image) tasks is not much less than r times of the execution time of running a single r-fold RP task. For example, Figure Figure 4.20 shows that a frame with a size of 1%, 4%, 9% takes 35.19ms, 37.9ms, and 44.24ms, respectively, for instance segmentation inference. This is because in a CNN model, except for the part extracting feature maps and converting anchors to RPs, the rest of the network is usually the same regardless of the size of the input. Second, it is hard to determine a good cropping strategy. On one hand, the precise cut-out individual RPs will lead to poor detection inference accuracy due to the lack of necessary background pixels; on the other hand, if we leave large padding around the RPs, then the total offloaded data will be too large to be efficient. Third, too many cropping operations generate high memory copy overheads which may likely become problematic on mobile devices.

4.5.3 RP Box Based Partitioning and Offloading

RP Box Initialization. Given the above observations, ELF proposes an RP scheduling method that is more content- and resource-aware than the above naive counterpart. The key data structure here is what we call RP boxes. Compared to a single RP, an RP-box is larger and consists of one or more nearby RPs, as illustrated in Figure Figure 4.6 (f) with 4 RPs and 3 RP boxes. The number of offloading tasks is determined by the number of available edge servers. Each offloading task consists of either an LRC task, or an RP-box processing task, or both. Scheduling an RP box instead of individual RPs, we can avoid the

fragmentation problems mentioned above.

Before partitioning a frame, ELF first crops the area with all the RPs and horizontally partitions it into N segments (N is #available servers), where each segment corresponds to an initial RP box. The i-th RP box will be offloaded to the i-th server and we adjust the size of the RP box to control the load on each server. Here, we first need to explain how the LRC task scheduling interferes with the RP box scheduling. Note that, the LRC task is only available every n frames ($\mathbf{1}_{(t \bmod n=0)}$ in Equation (4.6)). At the LRC round, we partition the cropped image into $(N-1)$ segments and have $(N-1)$ RP boxes accordingly. We reserve one server for the LRC task[4]. Regardless of the number of RP boxes, the scheduling algorithm works the same – during the LRC round, we treat the LRC task the same as another RP box processing task. Below, without the loss of generality, we assume there are N RP boxes.

The size of each RP box is initialized to be proportional to the available resource of the corresponding server, as depicted in Figure Figure 4.6(b). This initialization can help achieve load balancing.

RP Association. Thereafter, we associate each RP with an RP box as follows. For each RP, ELF evaluates its spatial relationship with all the RP boxes. For a pair of RP r and box b, their spatial relationship falls into one of the three cases:

- *Inclusion*. In this case, r is completely included in b and we conveniently associate them.

- *Partial-overlap*. In this case, r intersects with b. At the same time, it has a partial-overlap relationship with at least one other box as well. Here, we choose to associate with the RP box that has the most overlap with the RP. If there is a tie, we choose the RP box with a larger gap between the server resource capacity and the computation costs of the RPs that are already associated.

- *No-overlap*. In this case, r is not associated with b.

[4]Special care needs to be taken with the configuration of a total of 1 or 2 edge servers, and we discuss how we handle these two special cases in Section subsection 4.7.4

Elf applies the association steps to all the RPs.

RP Box Adjustment. After all the RPs have got associated with a box, Elf resizes each RP box such that it can fully cover all the RPs that are associated with it. Please see Figures Figure 4.6 (e) and (f). After this adjustment, the computation cost of some RP boxes may drastically increase compared to the initialization stage and thus break the intended load balancing. To avoid this, we examine those RP boxes whose cost increase exceeds a pre-defined threshold (we discuss how to estimate an RP box's computation cost in Section subsection 4.5.4). For these boxes, the associated RPs are sorted ascendingly w.r.t the computation cost. We try to re-associate the first RP on the list (the one with the lowest cost) to the neighboring box who has enough computation capacity to hold this RP. After each re-association, the two boxes need to adjust their sizes accordingly and estimate the new computation cost. we repeat this re-association process as far as the load distribution is becoming more even. We stop this process if the re-association results in even higher load imbalance.

Here, we formally evaluate the load-balanced situation by:

$$\Theta = \mathrm{Var}(\{T_k^t\}) \tag{4.7}$$

where Θ denotes the variance of the estimated execution time of all the tasks. A smaller Θ denotes a more balanced partitioning and offloading. We can calculate T_k^t by the following Equation (4.6) where $C^t_{\mathrm{rps},k}$ and C^t_{lrc} can be found as:

$$C^t_{\mathrm{rps},k} = \sum_v \{C^t_{\mathrm{rp},v}\},\ C^t_{\mathrm{lrc}} = \alpha \cdot (\sum_{k=1}^{M} C^t_{\mathrm{rps},k}) \tag{4.8}$$

where α is the LRC down-sample ratio.

Offloading. Finally, Elf offloads each RP box and the LRC task (if available in that round) to the corresponding edge server and executes the application-specific models in a

data-parallelism fashion.

4.5.4 Estimating Server Capacity and RP Computation Cost

In this subsection, we describe how ELF estimates the server resource capacity and each RP's computation cost.

ELF considers two ways of estimating a server's resource capacity. The first approach is through passive profiling. It calculates server m's current end-to-end latency as the average latency over the last n (default value of 7) offloading requests that are served by m. Then the resource capacity is defined as $1/T_m$. The second approach is through proactive profiling. ELF periodically queries the server for its GPU utilization.

ELF also considers two ways of estimating an RP's computation cost. The first approach is based on the RP's area, assuming the cost is linearly proportional to the RP area. The second approach is through Spatially Adaptive Computation Time (SACT) [149]. Here, we briefly explain how to borrow the concept of SACT to estimate computing cost of RPs. SACT is a model optimization that early stops partial convolutional operations by evaluating the confidence upon the outputs from intermediate layers. Overall, SACT indicates how much computation has been applied with each pixel of a raw frame input. ELF can accordingly estimate the cost of an RP at the pixel level. To adopt this approach, we need to slightly modify the backbone network as instructed in [149].

We adopt the passive resource profiling and RP area-based estimation in the implementation as they are more friendly to ELF's users and require less system maintenance efforts. We will deliver other options in the future work.

4.6 System Implementation

We implement a prototype system of ELF in Pytorch1.4 and Python3.6 for easy integration with most Python-based deep learning networks. We wrote the performance-critical functions in C++ to achieve low latency execution. In total, our implementation consists

of about 3,710 lines of codes. Our implementation is developed on Ubuntu16.04. We integrate ZeroMQ4.3.2 [152], an asynchronous messaging library that is widely adopted in distributed and concurrent systems, for high-performance multi-server offloading. We use NVIDIA docker [153] to run offloading tasks on edge servers. We also wrap nvJPEG [154] with Pybind11 for efficient hardware-based image/video encoding on mobile devices.

ELF is designed and implemented as a general acceleration framework for diverse mobile deep vision jobs. We aim to support existing applications with minimal modifications of applications. The required modifications only focus on the data-processing part where a DNN is executed over an input frame and returns a prediction result. Here, we assume that an application can split the data-processing part from the rest of it. Thus, the other parts and the internal logic of applications remain the same. Specifically, in our implementation, ELF interacts with its host deep learning model, given most are written in Python, through the following two APIs:

```
1. def cat(instance_lists:  List["Instances"]) -> "Instances",

2. def extract(instance :  Instances) -> "List["RP"]".
```

ELF employs the first API to aggregate the partial inference results, and the second API to extract RPs from the data structure of partial inference results to be used as the prediction for the next frame. With these two APIs, ELF can hide its internal details and provides a high-level API for applications:

Table 4.1 Comparisons of end-to-end latency (ms) and inference accuracy (AP) in three deep vision applications: instance segmentation [132], object classification [150], and key-point detection [151]. For *SO* and ELF, the end-to-end latency is further decomposed into {Frame en/de-code, ELF functions, average server processing, network transmission, parallel task synchronization}

Deep Vision Applications	Latency (ms)				Accuracy (AP)			
	TX2	Nano	*SO*	**ELF**	TX2	Nano	*SO*	ELF
Instance Segmentation	1,070	2,107	226×{13.4, 0, 78.9, 7.5, 0}%	**104**×{6.4, 7.9, 67.2, 4.1, 14.2}%	0.803	0.803	0.803	0.799
Object Classification	1,141	2,160	87.3×{8.7, 0, 89.7, 1.4, 0}%	**66.8**×{2.4, 8.7, 79.7, 1.3, 7.7}%	0.672	0.672	0.672	0.671
Key-point Detection	2,452	4,705	252×{8.9, 0, 90.4, 0.6, 0}%	**137**×{5.2, 7.1, 75.7, 2.8, 9.0}%	0.661	0.661	0.661	0.654

```
3. def run(img:  numpy.array) -> "Instances",
```

This API can make the inference function as same as the one running locally, while ELF can run multi-way offloading and feed the merged results to applications. Moreover, if an application is written in Java or C++ and calls a Python inference model inside, we can further wrap ELF with Java Native Interface (JNI) or Pybind. ELF will support this extension in future work. We believe that ELF requires a reasonably small effort from developers for the benefit of significantly reduced latency.

Furthermore, we discuss the placement of ELF functions. We argue that ELF's function should run on mobile devices but not edge servers for two reasons: 1) finishing both functions locally enables to offload less than 50% data as redundant background pixels will be removed; 2) all the ELF functions only take 5-7ms on mobile, and thus the offloading benefit will be trivial considering much fewer data to ship.

Finally, we point out that ELF does require its applications to be able to process frames with different resolutions which is the fundamental condition underlying the frame partitioning in ELF. However, deep learning networks can easily meet this requirement by adopting either an ROI-pooling layer [155], or a spatial pyramid pooling layer [156] to crop and pool feature maps into a fixed size. Adding these two layers can render a network to be able to handle varied frame resolution, and has been widely adopted in many existing studies [151, 157, 158, 159].

4.7 Performance Evaluation

We successfully integrate ELF with ten state-of-the-art deep learning networks and thoroughly evaluate ELF in the following three typical applications: instance segmentation, multi-object classification and multi-person pose estimation. Our results show that ELF can accelerate the inference up to 4.85× and 3.80× on average with using 52.6% less bandwidth on 4 edge servers while keeping the inference accuracy sacrifice within 1%.

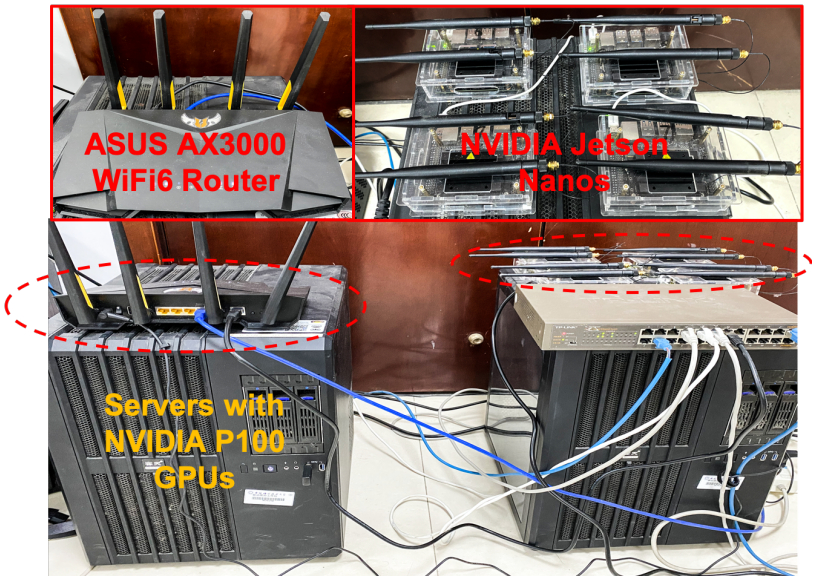


Fig. 4.7 The hardware platform

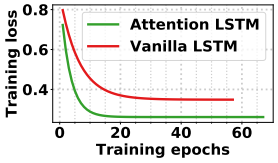


Fig. 4.8 Training loss of RP Prediction in the first 60 epochs

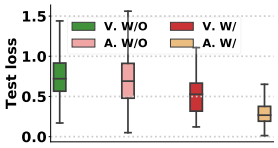


Fig. 4.9 Test loss of RP Prediction w/- and w/o RP indexing

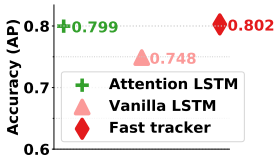


Fig. 4.10 Inference accuracy for RP prediction algorithms

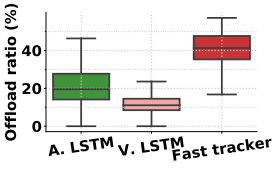


Fig. 4.11 Offload ratio for three RP prediction algorithms

4.7.1 Experiment Setup

Mobile Platforms: We use four mobile platforms: Google Pixel4 (Qualcomm Snapdragon 855 chip consisting of eight Kryo 485 cores, an Adreno 640 GPU and a Hexagon 690 DSP), (2) Nexus 6P (Snapdragon 810 chip with four ARM Cortex-A57 cores and four ARM Cortex-A53 cores, an Adreno 430 GPU), (3) Jetson Nano [136] (Quad-core ARM Cortex-A57 MPCore CPU, NVIDIA Maxwell GPU with 128 CUDA cores), and (4) Jetson TX2 [141] (Dual-Core NVIDIA Denver 2 64-Bit + Quad-Core ARM Cortex-A57 MPCore CPU, NVIDIA Pascal GPU with 256 CUDA cores). The evaluation results with Jetson TX2 have been reported if not explicitly stated otherwise study the performance difference of mobile devices.

Edge Servers: We use up to 4 edge servers. Each server runs Ubuntu 16.04 and has one

NVIDIA Tesla P100 GPU (3,584 CUDA Cores), Intel Xeon CPU (E5-2640 v4, 2.40GHz).

Networks: We use WiFi6 (802.11.ax, ASUS-AX3000, 690Mbps) to connect the mobile platforms and edge servers. Based on the WiFi network, we also use the Linux traffic shaping to emulate a Verizon LTE (120Mbps) link using the parameters given by a recent Verizon network study [160]. Moreover, we randomly set the available bandwidth of each server in 70% to 100% to introduce the network heterogeneity. The emulated LTE network has been used if not explicitly stated otherwise study the network impacts. Figure Figure 4.7 shows our experimental platform.

CNN Models and Datasets : We consider ten state-of-the-art models: CascadeRCNN [161], DynamicRCNN [162], FasterRCNN [103], FCOS [163], FoveaBox [164], FreeAnchor [165], FSAF [166], MaskRCNN [132], NasFPN [167], and RetinaNet [150]. Also, we use MOTS dataset [168] for instance segmentation, KITTI dataset [169] for multi-object classification, PoseTrack [170] dataset for pose estimation. MaskRCNN has been adopted if not explicitly stated otherwise study the model difference.

Comparison with Existing Offloading Work: Please note that the existing offloading algorithms are either model parallelism [117, 171] or to filter offloading data [118, 119, 120, 121]. They focus on offloading work to a single server and are complementary to our work.

CNN Networks and Benchmarks: We use MaskRCNN [132] and MOTS dataset [168] for instance segmentation, RetinaNet [150] and KITTI dataset [169] for multi-object classification, DensePose [151] and PoseTrack [170] dataset for multi-person pose estimation.

4.7.2 Evaluation of ELF System

We conduct a set of experiments to show the overall performance of the whole ELF system. We evaluate and report the inference accuracy and end-to-end latency in the following 4 settings: (1) *TX2*, a baseline running the application on Jetson TX2, (2) *Nano*, a baseline running the application on Jetson Nano, (3) *SO*, a baseline of existing offloading strategy

that offloads the CNN inference to a single edge server, and (4) ELF-3, our approach of partitioning the frame and offloading the partial inferences to three edge servers (one running an LRC task and the other two running an RP box each). We conduct the experiments using all three applications with the emulated Verizon LTE network.

First, let us look at the performance with the instance segmentation application. Table Table 4.1 reports the overall execution latency and accuracy in all the settings. The results show that running such a demanding application on the mobile platforms yields unbearably long delays of 1,070ms and 2,107ms on *TX2* and *Nano*, respectively. Offloading the inference task can greatly shorten the latency. While *SO* reduces the latency to 226ms, ELF-3 can further bring down the latency from to only 104ms, i.e., a 53.98% latency reduction compared to *SO*. Compared to using the entire frame for inference as in *SO*, ELF-3 achieves almost the same accuracy – 0.799 vs 0.803 – a very small drop of less than 0.5%. Such an accuracy drop is because 1) performing LRC once every 3 frames may sometimes miss new objects and tiny objects, although it rarely happens, and 2) ELF removes the background pixels not covered by the RP boxes. However, we believe this small accuracy drop is acceptable, especially considering the significant latency reduction of 53.98%.

We observe a similar trend with the other two applications. The results of multi-object detection show that while keeping the accuracy almost the same – 0.672 for *SO* and 0.671 for ELF-3 — we can enjoy a 23.48% latency reduction, from 87.3ms to 66.8ms. The results of the key point detection show that ELF-3 can achieve a 45.44% latency reduction compared to *SO* with a small accuracy drop, from 0.661 to 0.654.

All applications and both metrics considered, we believe ***ELF* provides a very viable approach to support mobile deep vision by parallel offloading**. In the following evaluation, we use instance segmentation as our driving example to evaluate the effectiveness of the key components of ELF.

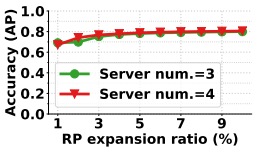


Fig. 4.12 Inference accuracy vs RP expansion ratio

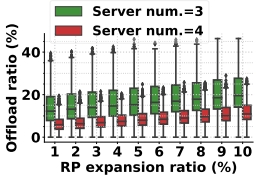


Fig. 4.13 Offload ratio vs RP expansion ratio

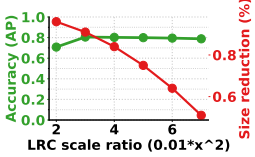


Fig. 4.14 Inference accuracy vs LRC ratio

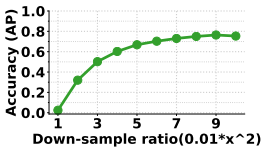


Fig. 4.15 Inference accuracy vs down-sample ratio

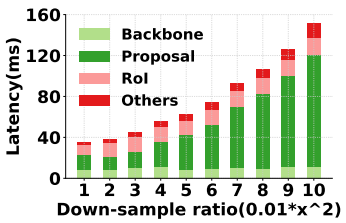


Fig. 4.16 Instance segmentation processing latency vs down-sample ratio

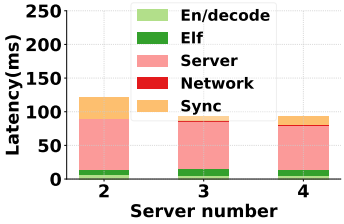


Fig. 4.17 End-to-end latency vs GPU numbers

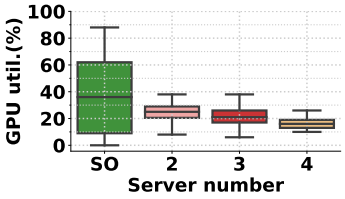


Fig. 4.18 Average GPU utilization vs GPU numbers

4.7.3 Evaluation of RP Prediction

Next, we evaluate the ELF RP prediction module. In our evaluation, we trained the attention LSTM network as follows. Two readily available video datasets, CityScapes [172] and KITTI [173] that contain object labels for each frame, were used in the training. Using 60% of the dataset as training data, we applied the RP indexing algorithm to maintain a consistent order of the region proposals. Finally, we train the network using the Adam optimizer [174] with an initial learning rate of 1e-3 to minimize the loss function $\mathcal{L}(\cdot,\cdot)$ defined in Equation (4.1).

The Effectiveness of Attention LSTM: We show the training loss curve (defined in Equation (4.1)) in the first 60 epochs in Figure Figure 4.8 and the loss on the test dataset in Figure Figure 4.9. The trend shows that our attention LSTM outperforms vanilla LSTM, by reducing the test loss from 0.51 to 0.25. Further, we report the impact of different prediction algorithms on the inference accuracy while keeping other modules the same. Here, we also consider Fast tracker [175], which predicts the current RPs as the RPs in the previous video

frame, but with a scale up of 200%, as the baseline. Figure Figure 4.10 and Figure Figure 4.11 show the inference accuracy and the offload ratio (defined as the ratio of the total offloading traffic, with respect to the offloading traffic in *SO*). The vanilla LSTM predictor has the lowest offloading traffic, with an 11% offload ratio, but at a considerable inference accuracy downgrade compared to our attention LSTM, 0.748 vs. 0.799. Meanwhile, fast tracker shows a slightly higher inference accuracy compared to us, 0.802 vs 0.799, but the offloading traffic almost doubles. All things considered, our attention LSTM could achieve a good inference accuracy with a reasonably low offloading traffic.

The Impact of RP Indexing: As part of our RP prediction, we also report the impact of RP indexing. Figure Figure 4.9 shows the test loss of two prediction algorithms with and without RP indexing. With RP indexing, vanilla LSTM reduces the loss from 0.71 to 0.51 and attention LSTM reduces the loss from 0.7 to 0.25, demonstrating the effectiveness of RP indexing.

The Impact of RP Expansion Ratio: Moreover, the RP expansion ratio trades off the application accuracy with the average offloading traffic volume, i.e., a larger ratio leads to a higher application accuracy at the cost of more offloading data. Figures Figure 4.12 and Figure 4.13 show the inference accuracy and offload ratio with different expansion ratios. After increasing the RP expansion ratio to 4% or higher, the accuracy stays at 0.799, the highest ELF can achieve. However, when ELF has the ratio under 4%, we observe an accuracy downgrade. For example, the accuracy is 0.70 and 0.75 when the expansion ratio is 1% and 2%, respectively. Also, we identify the same pattern with ELF-3 and ELF-4. On the other hand, with an RP extension ratio of 4%, the offload ratio is 7% for ELF-3 and 13% for ELF-4. Overall, 4% is a good RP extension ratio for ELF.

The Impact of LRC Parameters: We then show LRC can efficiently detect new objects when first appear. Figure Figure 4.15 presents the accuracy using the down-sampled frames for inference with a down-sampling scale x increasing from 1 to 10, with the resulting frame size from $0.01 \times x^2$. At the scale of 8, the accuracy degrades to 0.75, and at the scale of 4,

the accuracy goes down to 0.62. The result indicates that using low-resolution frames alone hurts application accuracy. Figure Figure 4.14 reports the inference accuracy by offloading frame partitions and LRC with different down-sample ratios. When the LRC ratio has been increased to 16%, the accuracy keeps at 0.799 (the SO solution achieves 0.803). When the LRC ratio has been increased to 16%, the accuracy keeps at 0.799 (the SO solution achieves 0.803).

4.7.4 Evaluation of RP-Centric Partitioning and Offloading

Next, we evaluate the frame partitioning and offloading module. We first describe the end-to-end latency when different numbers of servers are available for ELF with ten state-of-the-art deep learning networks. Here, we assume each server has only a single GPU available for the mobile application. A special case to consider, if there is only a GPU, ELF will adopt a single RP box that covers all the RPs but removes the surrounding background pixels and stack with the LRC task. KITTI dataset has been resized to the resolution 2560×1980 to study the high-resolution scenario in the section.

When there are two servers available, ELF offloads the LRC task and one of the RP boxes to one server and the other RP box to the second server. Compared to *SO*, ELF-2 reduces the latency by 2.80× on average, up to 3.63× . When ELF has three or more servers, it uses one server for LRC, and one RP box each on the other servers. We measure a latency reduction of 2.94× on average, up to 3.71× with ELF-3 , 3.80× on average, up to 4.85× with ELF-4, and 4.18× on average, up to 5.43× with ELF-5 respectively. The results demonstrate that ELF work with the arbitrary number of available edge servers and it outperforms the *SO* even with a single server.

Key Observations: We observe that the latency with different server numbers highly depends on the size of the RP boxes shipped to each edge server. With the frame partitioning algorithm, the *maximal* size of RP box compared to the raw frame as 51.7%, 23.7%, 23.7%, 15.7%, and 11.6%, the computing bottleneck in that offloading round, with the server

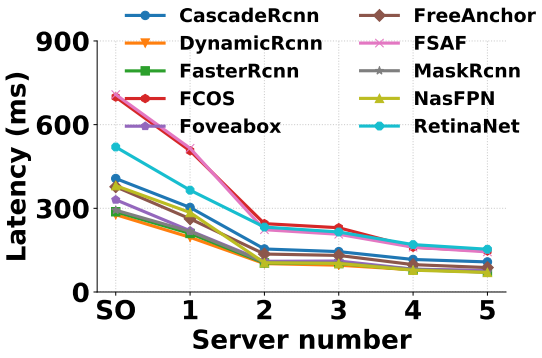


Fig. 4.19 End-to-end latency vs server numbers

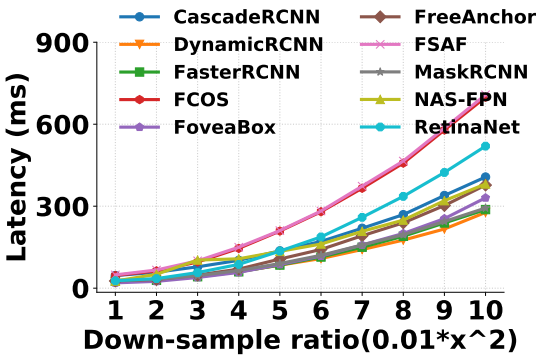


Fig. 4.20 Processing latency vs down-sample ratio

number from 1 to 5, respectively. Please note that ELF-2 and ELF-3 have the same RP box size because both adopt 2 RP boxes but the later assigns the LRC task on the third server. Accordingly, ELF reduces bandwidth usage by 48.3%, 52.6%, 52.6%, 52.9%, and 53.6%. Importantly, another observation is that the model inference time strongly relates to the input size. Figure Figure 4.20 shows the inference latency at the server running ten state-of-the-art models with down-sample ratio $0.01 \cdot x^2$ where x is from 1 to 10. Here, the down-sample ratios of 49%, 25%, 16%, and 9% share a rough correspondence to the RP box size with the server numbers of 1, 2 (3), 4, 5. This observation is the underlying reason why ELF can significantly reduce the inference latency by having each server inferring part of the frame.

Lessons Learned: Moreover, we identify the inference time shows distinct sensitivity among different deep vision models. First, the models, for example, FCOS [163], with more, even fully, convolutional operations present a stronger correlation between frame resolution and inference latency. Second, two-stage models, for example, RCNN series [103, 132, 161, 162], usually generate the same number of Regions of Interest (ROI) independent of the input resolution and then ship each of them down the pipeline. The second stage thus costs the same time. Overall, the lessons we have learned regarding how to design models to benefit more from parallel offloading are: 1) one-stage models with more convolutional operations are preferred, 2) two-stage models can dynamically adjust the number of ROI based on the frame resolution as a higher resolution input potentially involves more objects.

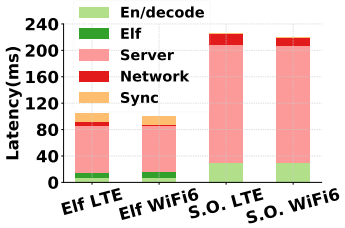


Fig. 4.21 End-to-end latency vs network conditions

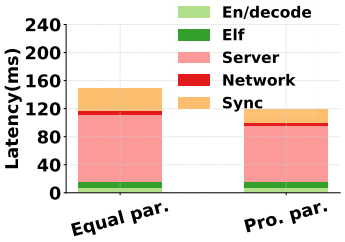


Fig. 4.22 End-to-end latency vs RP box partitioning schemes

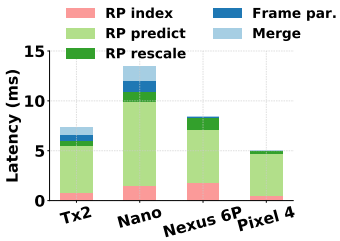


Fig. 4.23 System overheads of ELF functions

Finally, we show the average GPU utilization under different configurations in Figure Figure 4.18. In the case of *SO*, the average GPU utilization is 37% and the 95_{th} percentile is 82%. In the case of ELF-3, the average GPU utilization is 21% and the 95_{th} percentile is 31%. We note that using 3 GPUs in the case of ELF-3 shows lower per GPU utilization than *SO*. In average, ELF-3 only consumes 1.7× GPU utilization in total running with 3 GPUs, than *SO* to finish a single request. Moreover, a lower per GPU utilization allows ELF to have more chance to efficiently utilize those resource fragmentation and thus improve the total GPU utilization of edge servers.

4.7.5 Dealing with Dynamic Network Condition and GPU Utilization

We investigate how well ELF deals with dynamic network conditions and available GPU utilization. First, we compare ELF-3 and *SO* with Verizon LTE (120Mbps) and WiFi6 (690Mbps) networks. Figure Figure 4.21 shows that when switching from WiFi6 to LTE, the network latency of ELF increases from 1.4ms to 6.5ms and that of *SO* increases from 10.1ms to 17.1ms. ELF is less sensitive to the network bandwidth because it offloads much less data than *SO* as shown in Figure Figure 4.13. Next, we increase the GPU utilization of one server to 70% and compare our resource-aware RP box allocation method with a resource-agnostic equal RP box allocation method. Figure Figure 4.22 shows that our method results in a latency of 119ms while the other method has a latency of 149ms.

4.7.6 ELF System Overhead on Mobile Side

ELF incurs a small amount of overhead on the mobile side. Figure Figure 4.23 shows the latency of five ELF functions. Jetson TX2 and Nano are evaluated with the Python implementation and take 7ms, and 13.6ms in total. Nexus 6P and Pixel 4 are evaluated with C++ by Java native interface and cost 7.8ms and 4.8ms in total. The incurred system overhead is sufficiently low to deliver its parallel offloading functions for the significant latency reduction. Moreover, RP prediction costs 70%+ of the total time as the attention LSTM model is implemented in Python and exported to C++ with TorchScript. We will rewrite the prediction model with TensorRT [176], a C++ library that facilitates high-performance inference, in the future work to minimize the RP prediction latency.

4.8 Related Work

Video Analytics Optimizations. AWS Wavelength [122] moves Amazon Web Services to Verizon's 5G edge computing platforms to deliver low latency video services. Intel and NVIDIA contribute Intel Xeon Scalable Processors with Intel Deep Learning Boost [127] and NVIDIA EGX A100 [128] to enable real-time AI processing at the edge. Pano [177] proposes a quality-adaptation scheme that balances user-perceived video quality and video encoding efficiency. Chameleon [178] dynamically selects the best configurations for existing NN-based video analytics to save computing resources by up to 10×. VideoStorm [5] and AWStream [179] adapt the configuration to balance accuracy and processing delay.

Offloading Deep Neural Networks. Cracking open the DNN [117], Neurosurgeon [171], DeepThings [180] partition then distribute the layers of deep learning networks over edge servers and mobile/IoT devices (model parallelism) to reduce the inference latency. FilterForward [118], Reducto [119], Glimpse [120] and Vigil [121] perform selective data offloading based on the feature type, filtering threshold, query accuracy, and video content to minimize the running latency. EdgeAssisted [96] uses a motion vector based object tracking to adaptively offload those regions of interests. Frugal Following [181] dynamically

tracks objects and only runs a DNN with significantly changes. DDS [182] continuously sends a low-quality video stream to the server that runs the DNN to determine where to re-send with higher quality to increase the inference accuracy. These works only support single-server offloading. Instead, ELF is designed to accelerate high-resolution vision tasks through distributed offloading of multiple servers.

Accelerating Model Inferences. Branch pruning and sharing [107, 183, 184, 185] remove redundant or less critical parameters [108] to trade-off the model complexity with inference latency. Tensor quantization [109] uses fewer bits to represent parameters for model compression. DeepCache [186] caches and reuses the result of convolutional operations to reduce the repeated computation. Simultaneously handle tasks with a single model through multi-task learning [187] for less computation. Furthermore, massive accelerators, e.g., GPU [141, 136], FPGA [114], and ASIC [115, 188], are designed to perform model inference in a high-throughput and low-latency fashion. All these works are complementary to ours.

4.9 Conclusion and Future Work

We designed and implemented ELF, an acceleration framework for mobile deep learning networks. ELF can partition a video frame into multiple pieces and offload them simultaneously to edge servers for parallel computing. The main contribution of the framework stems from its recurrent region proposal prediction and content-aware video frame partitioning algorithms. Our study shows that ELF is promising to minimize the end-to-end latency of emerging mobile deep vision applications. Moving forward, we will continue to investigate how to further drive down the latency, e.g., integrating the data-parallelism approach of ELF with those model-parallelism solutions. We will also investigate the impact of access network bandwidth on ELF task assignment and mapping. Another future work topic is the efficient model design for deep vision applications to better benefit from parallel offloading. Finally, we will study how to efficiently orchestrate heterogeneous edge resources to

minimize the AI processing latency.

CHAPTER 5
CONCLUSION

In conclusion, this dissertation investigates deployment of real-time edge applications in a distributed fashion, and the associated orchestration of networking and computing resources necessary to achieve low latency.

We first propose the EC+ architecture that seamlessly distributes the required processing across user devices, edge clouds, and the central cloud to achieve ultra low-latency responses, frequent refreshing, and a large number of concurrent players. To complement this architecture, we also investigated a game service placement algorithm which aims to maximize the gaming performance for all the players by dynamically placing their services on those edge clouds that can lead to the best performance.

Second, we develop a *latency-aware* edge resource orchestration platform based on Apache Storm. The platform aims to support real-time responses to edge cloud assisted applications that are computation intensive. The main contribution of our platform stems from a set of *latency-aware* task scheduling schemesbased on DAGs (directed acyclic graphs). By deploying the proposed platform on a group of edge servers with heterogeneous CPU, GPU and networking resources, we show that we can support real-time edge vision applications, achieving the per frame latency around 32ms for an example AR application.

Finally, we designed and implemented ELF, an acceleration framework for mobile deep learning networks. ELF can partition a video frame into multiple pieces and offload them simultaneously to edge servers for parallel computing. The main contribution of the designed framework stems from its accurate region proposal prediction and resource-aware video frame partitioning algorithms. Our study shows that ELF is a promising framework for reducing the end-to-end latency of emerging mobile deep vision applications. Moving forward, we will continue to investigate how to further drive down the latency, e.g.,

developing a CPU/GPU hybrid pipeline to schedule ELF functions. We will also investigate the impact of access network bandwidth on ELF task assignment and mapping.

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, ACM, 2012, pp. 13–16.

[2] Y.-Y. Shih, W.-H. Chung, A.-C. Pang, T.-C. Chiu, and H.-Y. Wei, "Enabling low-latency applications in fog-radio access networks," *IEEE Network*, vol. 31, no. 1, pp. 52–58, 2017.

[3] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, ACM, 2014, pp. 68–81.

[4] W. Zhang, Y. Hu, Y. Zhang, and D. Raychaudhuri, "Segue: Quality of service aware edge cloud service migration," in *CloudCom*, 2016.

[5] 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 *NSDI*, vol. 9, 2017, p. 1.

[6] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "Lavea: Latency-aware video analytics on edge computing platform," in *SEC*, ACM, 2017, p. 15.

[7] K. Ha, Y. Abe, T. Eiszler, Z. Chen, W. Hu, B. Amos, R. Upadhyaya, P. Pillai, and M. Satyanarayanan, "You can teach elephants to dance: Agile vm handoff for edge computing," in *SEC*, ACM, 2017, p. 12.

[8] W. Cerroni and F. Callegati, "Live migration of virtual network functions in cloud-based edge networks," in *ICC*, IEEE, 2014, pp. 2963–2968.

[9] J. Carmack, *John Carmack's Delivers Some Home Truths on Latency*, http://oculusrift-blog.com/john-carmacks-message-of-latency.

[10] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, *et al.*, "Storm twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, ACM, 2014, pp. 147–156.

[11] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones.," in *NSDI*, vol. 13, 2013, pp. 185–198.

[12] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "The brewing storm in cloud gaming: A measurement study on cloud to end-user latency," in *NetGames*, 2012.

[13] NVIDIA, *Geforce NOW*, https://www.nvidia.co.uk/shield/games/.

[14] SONY, *PlayStation NOW*, https://www.playstation.com/en-us/explore/playstationnow/.

[15] C.-Y. Huang, K.-T. Chen, D.-Y. Chen, H.-J. Hsu, and C.-H. Hsu, "GamingAnywhere: The First Open Source Cloud Gaming System," *TOMM*, p. 10, 2014.

[16] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Network*, vol. 27, no. 4, pp. 16–21, 2013.

[17] Nvidia, *Geforce NOW System Requirements*, https://shield.nvidia.com/support/geforce-now/system-requirements/2.

[18] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[19] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *MCC*, 2012.

[20] T. Taleb and A. Ksentini, "Follow me cloud: Interworking federated clouds and distributed mobile networks," *IEEE Network*, pp. 12–19, 2013.

[21] HTC, *Vive's immersive room-scale technology*, https://www.vive.com/us/.

[22] Google, *Google Cardboard brings immersive experiences to everyone in a simple and affordable way.* https://vr.google.com/cardboard/get-cardboard/.

[23] S. M. LaValle, *VIRTUAL REALITY*. Cambridge University Press, 2015.

[24] Lookingglass, *Virtual reality mmorpg*, https://lookingglass.services/virtual-reality-mmorpg/.

[25] J. Lee, *7 Games You Can Mod to Add VR Support Right Now*, http://www.makeuseof.com/tag/mods-provide-vr-support/.

[26] Kalydo, *Kalydo*, https://en.wikipedia.org/wiki/Kalydo/.

[27] Spawnapp, *Spawnapp*, http://spawnapp.com//.

[28] NVIDIA, *Nvidia grid*, http://www.nvidia.com/object/nvidia-grid.html13.

[29] Y. Zhang, P. Qu, J. Cihang, and W. Zheng, "A cloud gaming system based on user-level virtualization and its resource scheduling," *TPDS*, pp. 1239–1252, 2016.

[30] A. Banitalebi-Dehkordi, M. Azimi, M. T. Pourazad, and P. Nasiopoulos, "Compression of high dynamic range video using the hevc and h. 264/avc standards," in *QShine*, 2014.

[31] 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 *MobiSys*, 2015.

[32] 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.

[33] F. M. Bartucca, C. D. Jeffries, R. V. Slager, and N. C. Strole, *Detecting network instability*, US Patent 6,918,067, Jul. 2005.

[34] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande, "Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game," in *NetGames*, 2004.

[35] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *MCC*, 2013.

[36] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "Where's the bear?: Automating wildlife image processing using iot and edge cloud systems," in *IoTDI*, 2017.

[37] A. Bharambe, J. Pang, and S. Seshan, "Colyseus: A Distributed Architecture for Online Multiplayer Games," in *NSDI*, 2006.

[38] N. E. Baughman and B. N. Levine, "Cheat-proof playout for centralized and distributed online games," in *INFOCOM*, 2001.

[39] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "A hybrid edge-cloud architecture for reducing on-demand gaming latency," *Multimedia Systems*, pp. 503–519, 2014.

[40] Y. Lin and H. Shen, "Cloudfog: Leveraging fog to extend cloud gaming for thin-client mmog with high quality of experience," *TPDS*, pp. 431–445, 2016.

[41] H.-J. Hong, D.-Y. Chen, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu, "Qoe-aware virtual machine placement for cloud games," in *NetGames*, 2013.

[42] M. Steiner, B. G. Gaglianello, V. Gurbani, V. Hilt, W. D. Roome, M. Scharf, and T. Voith, "Network-aware service placement in a distributed cloud environment," *SIGCOMM CCR*, pp. 73–74, 2012.

[43] J. T. Piao and J. Yan, "A network-aware virtual machine placement and migration approach in cloud computing," in *GCC*, 2010.

[44] S.-H. Lim, J.-S. Huh, Y. Kim, and C. R. Das, "Migration, assignment, and scheduling of jobs in virtualized environment," *Migration*, vol. 40, p. 45, 2011.

[45] C. Ghribi, M. Hadji, and D. Zeghlache, "Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms," in *CCGrid*, 2013.

[46] F. Douglis and J. Ousterhout, "Transparent process migration: Design alternatives and the sprite implementation," *Software: Practice and Experience*, vol. 21, no. 8, pp. 757–785, 1991.

[47] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *NSDI*, 2005.

[48] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of virtual machine live migration in clouds: A performance evaluation," in *CLOUD*, 2009.

[49] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," in *CLUSTER*, 2009.

[50] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan, "Adaptive vm handoff across cloudlets," Technical Report CMU-CS-15-113, CMU School of Computer Science, Tech. Rep., 2015.

[51] J. Chen, M. Arumaithurai, X. Fu, and K. K. Ramakrishnan, "G-COPSS: A Content Centric Communication Infrastructure for Gaming," in *ICDCS*, 2012.

[52] M. Claypool, D. Finkel, A. Grant, and M. Solano, "Thin to win?: Network performance analysis of the onlive thin client game system," in *NetGames*, 2012.

[53] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, pp. 40–45, 2006.

[54] Wikipedia, *Actions per minute*, https://en.wikipedia.org/wiki/Actions_per_minute.

[55] C. F. Perez-Monte, M. D. Perez, S. Rizzi, F. Piccoli, and C. Luciano, "Modelling frame losses in a parallel alternate frame rendering system with a computational best-effort scheme," *Computers & Graphics*, vol. 60, pp. 76–82, 2016.

[56] D. L. Poole and A. K. Mackworth, *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[57] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein, "Dynamic service placement in geographically distributed clouds," *JSAC*, pp. 762–772, 2013.

[58] S. Wang, R. Urgaonkar, T. He, K. Chan, M. Zafer, and K. Leung, "Dynamic service placement for mobile micro-clouds with predicted future costs," *TPDS*, 2016.

[59] A. Ksentini, T. Taleb, and M. Chen, "A markov decision process-based service migration procedure for follow me cloud," in *ICC*, 2014.

[60] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-clouds," *Performance Evaluation*, pp. 205–228, 2015.

[61] P. Deshpande, A. Kashyap, C. Sung, and S. R. Das, "Predictive methods for improved vehicular wifi access," in *MobiSys*, 2009.

[62] A. J. Nicholson and B. D. Noble, "Breadcrumbs: Forecasting mobile connectivity," in *MobiCom*, 2008.

[63] V. A. Siris and D. Kalyvas, "Enhancing mobile data offloading with mobility prediction and prefetching," *SIGMOBILE CCR*, pp. 22–29, 2013.

[64] F. Zhang, C. Xu, Y. Zhang, K. Ramakrishnan, S. Mukherjee, R. Yates, and T. Nguyen, "Edgebuffer: Caching and prefetching content at the edge in the mobility-first future internet architecture," in *WoWMoM*, 2015.

[65] H. Tian, D. Wu, J. He, Y. Xu, and M. Chen, "On achieving cost-effective adaptive cloud gaming in geo-distributed data centers," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, pp. 2064–2077, 2015.

[66] S. Grizan, D. Chu, A. Wolman, and R. Wattenhofer, "Djay: Enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit gpu load balancing," in *SoCC*, 2015.

[67] D. Wu, Z. Xue, and J. He, "Icloudaccess: Cost-effective streaming of video games from the cloud with low latency," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1405–1416, 2014.

[68] T. N. B. Duong, X. Li, R. S. M. Goh, X. Tang, and W. Cai, "Qos-aware revenue-cost optimization for latency-sensitive services in iaas clouds," in *DS-RT*, 2012.

[69] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, *CRAWDAD dataset epfl/mobility (v. 2009-02-24)*, Downloaded from http://crawdad.org/epfl/mobility/20090224, Feb. 2009.

[70] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *WCNC*, 2005.

[71] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*, ACM, 2011, pp. 301–314.

[72] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[73] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through vm migration and transmission power control," *TOC*, vol. 66, no. 5, pp. 810–819, 2017.

[74] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *TON*, vol. 25, no. 3, pp. 1818–1831, 2017.

[75] 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*, ACM, 2017, p. 8.

[76] W. Zhang, Y. Hu, Y. Zhang, and D. Raychaudhuri, "Segue: Quality of service aware edge cloud service migration," in *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*, IEEE, 2016, pp. 344–351.

[77] J. Wang, B. Amos, A. Das, P. Pillai, N. Sadeh, and M. Satyanarayanan, "A scalable and privacy-aware iot service for live video analytics," in *MMSys*, ACM, 2017, pp. 38–49.

[78] Q. Shan, B. Curless, Y. Furukawa, C. Hernandez, and S. M. Seitz, "Occluding contours for multi-view stereo," in *CVPR*, 2014, pp. 4002–4009.

[79] O. Hilliges, D. Kim, S. Izadi, M. Weiss, and A. Wilson, "Holodesk: Direct 3d interactions with a situated see-through display," in *SIGCHI*, ACM, 2012, pp. 2421–2430.

[80] J. Kim and C. Park, "End-to-end ego lane estimation based on sequential transfer learning for self-driving cars," in *CVPR, 2017*, IEEE, 2017, pp. 1194–1202.

[81] N. Bernini, M. Bertozzi, L. Castangia, M. Patander, and M. Sabbatelli, "Real-time obstacle detection using stereo vision for autonomous ground vehicles: A survey," in *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference on*, IEEE, 2014, pp. 873–878.

[82] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, and Y. Zhang, "Mobile edge cloud system: Architectures, challenges, and approaches," *IEEE Systems Journal*, no. 99, pp. 1–14, 2017.

[83] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[84] *NVIDIA Docker*, https://github.com/NVIDIA/nvidia-docker.

[85] *Apache Storm*, http://storm.apache.org/.

[86] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, Ieee, 2010, pp. 1–10.

[87] B. Peng, M. Hosseini, Z. Hong, R. Farivar, and R. Campbell, "R-storm: Resource-aware scheduling in storm," in *Proceedings of the 16th Annual Middleware Conference*, ACM, 2015, pp. 149–161.

[88] *NVIDIA OPENACC*, https://developer.nvidia.com/openacc.

[89] L. Yang, J. Cao, Y. Yuan, T. Li, A. Han, and A. Chan, "A framework for partitioning and execution of data stream applications in mobile cloud computing," *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 23–32, 2013.

[90] L. Chaufournier, P. Sharma, F. Le, E. Nahum, P. Shenoy, and D. Towsley, "Fast transparent virtual machine migration in distributed edge clouds," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ACM, 2017, p. 10.

[91] A.-C. Pang, W.-H. Chung, T.-C. Chiu, and J. Zhang, "Latency-driven cooperative task computing in multi-user fog-radio access networks," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, IEEE, 2017, pp. 615–624.

[92] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, ACM, 2017, p. 5.

[93] K. Habak, M. Ammar, K. A. Harras, and E. Zegura, "Femto clouds: Leveraging mobile devices to provide cloud service at the edge," in *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, IEEE, 2015, pp. 9–16.

[94] J. Silva, D. Silva, E. R. Marques, L. Lopes, and F. Silva, "P3-mobile: Parallel computing for mobile edge-clouds," in *Proceedings of the 4th Workshop on CrossCloud Infrastructures & Platforms*, ACM, 2017, p. 5.

[95] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser, "Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering," in *MobiSys*, ACM, 2018, pp. 68–80.

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

[97] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.

[98] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 45, 2015, pp. 393–406.

[99] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, IEEE, 2014, pp. 535–544.

[100] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive online scheduling in storm," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ACM, 2013, pp. 207–218.

[101] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[102] M. Xu, J. Liu, Y. Liu, F. X. Lin, Y. Liu, and X. Liu, "A first look at deep learning apps on smartphones," in *The World Wide Web Conference*, 2019, pp. 2125–2136.

[103] 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.

[104] M. Teichmann, M. Weber, M. Zoellner, R. Cipolla, and R. Urtasun, "Multinet: Real-time joint semantic reasoning for autonomous driving," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, IEEE, 2018, pp. 1013–1020.

[105] C. Xiang, C. R. Qi, and B. Li, "Generating 3d adversarial point clouds," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 9136–9144.

[106] S. Xu, D. Liu, L. Bao, W. Liu, and P. Zhou, "Mhp-vos: Multiple hypotheses propagation for video object segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 314–323.

[107] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1389–1397.

[108] B. Fang, X. Zeng, and M. Zhang, "Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ACM, 2018, pp. 115–127.

[109] 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.

[110] Z. He and D. Fan, "Simultaneously optimizing weight and quantizer of ternary neural network using truncated gaussian approximation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 11 438–11 446.

[111] J. Yim, D. Joo, J. Bae, and J. Kim, "A gift from knowledge distillation: Fast optimization, network minimization and transfer learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 4133–4141.

[112] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[113] T. Lee, Z. Lin, S. Pushp, C. Li, Y. Liu, Y. Lee, C. Xu, F. Xu, L. Zhang, and J. Song, "Occlumency: Privacy-preserving remote deep-learning inference using sgx," in *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking, MobiCom 2019, October 21-25, 2019, Los Cabos, Mexico*, ACM, 2019.

[114] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 161–170.

[115] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.

[116] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, IEEE, 2019.

[117] J. Emmons, S. Fouladi, G. Ananthanarayanan, S. Venkataraman, S. Savarese, and K. Winstein, "Cracking open the dnn black-box: Video analytics with dnns across the camera-cloud boundary," in *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges*, 2019, pp. 27–32.

[118] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. R. Dulloor, "Scaling video analytics on constrained edge nodes," *arXiv preprint arXiv:1905.13536*, 2019.

[119] Y. Li, A. Padmanabhan, P. Zhao, Y. Wang, G. H. Xu, and R. Netravali, "Reducto: On-camera filtering for resource-efficient real-time video analytics," in *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 2020, pp. 359–376.

[120] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan, "Glimpse: A programmable early-discard camera architecture for continuous mobile vision," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 292–305.

[121] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," *MobiCom, ACM*, 2015.

[122] *Aws wavelength: Bring aws services to the edge of the verizon 5g network.* https://enterprise.verizon.com/business/learn/edge-computing/.

[123] A. Narayanan, E. Ramadan, J. Carpenter, Q. Liu, Y. Liu, F. Qian, and Z.-L. Zhang, "A first look at commercial 5g performance on smartphones," in *Proceedings of The Web Conference 2020*, 2020, pp. 894–905.

[124] S. Zhou, W. Shen, D. Zeng, M. Fang, Y. Wei, and Z. Zhang, "Spatial–temporal convolutional neural networks for anomaly detection and localization in crowded scenes," *Signal Processing: Image Communication*, vol. 47, pp. 358–368, 2016.

[125] N. Tijtgat, W. Van Ranst, T. Goedeme, B. Volckaert, and F. De Turck, "Embedded real-time object detection for a uav warning system," in *The IEEE International Conference on Computer Vision (ICCV) Workshops*, Oct. 2017.

[126] H. Zhao, X. Qi, X. Shen, J. Shi, and J. Jia, "Icnet for real-time semantic segmentation on high-resolution images," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 405–420.

[127] *Intel xeon scalable processors*, https://www.intel.com/content/www/us/en/products/processors/xeon/scalable.html.

[128] *Nvidia egx a100: Delivering real-time ai processing and enhanced security at the edge*, https://www.nvidia.com/en-us/data-center/products/egx-a100/.

[129] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[130] L. Peterson, T. Anderson, S. Katti, N. McKeown, G. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay, and A. Vahdat, "Democratizing the network edge," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 2, pp. 31–36, 2019.

[131] S. Yang, E. Bailey, Z. Yang, J. Ostrometzky, G. Zussman, I. Seskar, and Z. Kostic, "Cosmos smart intersection: Edge compute and communications for bird's eye object tracking," in *Proc. 4th International Workshop on Smart Edge Computing and Networking (SmartEdge'20)*, 2020.

[132] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[133] H.-S. Fang, S. Xie, Y.-W. Tai, and C. Lu, "Rmpe: Regional multi-person pose estimation," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 2334–2343.

[134] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, IEEE, 2018, pp. 1421–1429.

[135] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[136] *Nvidia jetson nano, the ai platform for autonomous everything*, https://www.nvidia.com/jetson-nano.

[137] *Amazon sagemaker: Machine learning for every developer and data scientist*, https://aws.amazon.com/sagemaker/.

[138] K. Sun, B. Xiao, D. Liu, and J. Wang, "Deep high-resolution representation learning for human pose estimation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[139] D. Raychaudhuri, I. Seskar, G. Zussman, T. Korakis, D. Kilper, T. Chen, J. Kolodziejski, M. Sherman, Z. Kostic, X. Gu, *et al.*, "Challenge: Cosmos: A city-scale programmable testbed for experimentation with advanced wireless," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–13.

[140] 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 *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7310–7311.

[141] *Nvidia jetson tx2, the fastest, most power-efficient embedded ai computing device*, https://developer.nvidia.com/embedded/jetson-tx2.

[142] M. Wang, C.-c. Huang, and J. Li, "Supporting very large models using automatic dataflow graph partitioning," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ACM, 2019, p. 26.

[143] P. Voigtlaender, M. Krause, A. Osep, J. Luiten, B. B. G. Sekar, A. Geiger, and B. Leibe, "Mots: Multi-object tracking and segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 7942–7951.

[144] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *2016 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, IEEE, 2016, pp. 4945–4949.

[145] Y. Wang, M. Huang, X. Zhu, and L. Zhao, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615.

[146] Y. Qin, D. Song, H. Chen, W. Cheng, G. Jiang, and G. Cottrell, "A dual-stage attention-based recurrent neural network for time series prediction," *arXiv preprint arXiv:1704.02971*, 2017.

[147] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with lstm," 1999.

[148] Q. Wang, L. Zhang, L. Bertinetto, W. Hu, and P. H. Torr, "Fast online object tracking and segmentation: A unifying approach," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2019, pp. 1328–1338.

[149] M. Figurnov, M. D. Collins, Y. Zhu, L. Zhang, J. Huang, D. Vetrov, and R. Salakhutdinov, "Spatially adaptive computation time for residual networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 1039–1048.

[150] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[151] R. Alp Güler, N. Neverova, and I. Kokkinos, "Densepose: Dense human pose estimation in the wild," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7297–7306.

[152] P. Hintjens, *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.", 2013.

[153] *Build and run docker containers leveraging nvidia gpus*, https://github.com/NVIDIA/nvidia-docker.

[154] *Nvidia gpu-accelerated jpeg encoder and decoder*, https://developer.nvidia.com/nvjpeg.

[155] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[156] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial pyramid pooling in deep convolutional networks for visual recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 9, pp. 1904–1916, 2015.

[157] G. Rogez, P. Weinzaepfel, and C. Schmid, "Lcr-net++: Multi-person 2d and 3d pose detection in natural images," *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[158] Z. Gao, L.-S. Gao, H. Zhang, Z. Cheng, and R. Hong, "Deep spatial pyramid features collaborative reconstruction for partial person reid," in *Proceedings of the 27th ACM International Conference on Multimedia*, 2019, pp. 1879–1887.

[159] X. Cheng, P. Wang, and R. Yang, "Learning depth with convolutional spatial propagation network," *IEEE transactions on pattern analysis and machine intelligence*, 2019.

[160] A. Narayanan, J. Carpenter, E. Ramadan, Q. Liu, Y. Liu, F. Qian, and Z.-L. Zhang, "A first measurement study of commercial mmwave 5g performance on smartphones," *arXiv preprint arXiv:1909.07532*, 2019.

[161] Z. Cai and N. Vasconcelos, "Cascade r-cnn: Delving into high quality object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6154–6162.

[162] H. Zhang, H. Chang, B. Ma, N. Wang, and X. Chen, "Dynamic r-cnn: Towards high quality object detection via dynamic training," *arXiv preprint arXiv:2004.06002*, 2020.

[163] Z. Tian, C. Shen, H. Chen, and T. He, "Fcos: Fully convolutional one-stage object detection," in *Proceedings of the IEEE international conference on computer vision*, 2019, pp. 9627–9636.

[164] T. Kong, F. Sun, H. Liu, Y. Jiang, L. Li, and J. Shi, "Foveabox: Beyound anchor-based object detection," *IEEE Transactions on Image Processing*, vol. 29, pp. 7389–7398, 2020.

[165] X. Zhang, F. Wan, C. Liu, R. Ji, and Q. Ye, "Freeanchor: Learning to match anchors for visual object detection," in *Advances in Neural Information Processing Systems*, 2019, pp. 147–155.

[166] C. Zhu, Y. He, and M. Savvides, "Feature selective anchor-free module for single-shot object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 840–849.

[167] G. Ghiasi, T.-Y. Lin, and Q. V. Le, "Nas-fpn: Learning scalable feature pyramid architecture for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 7036–7045.

[168] P. Voigtlaender, M. Krause, A. Osep, J. Luiten, B. B. G. Sekar, A. Geiger, and B. Leibe, "Mots: Multi-object tracking and segmentation," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[169] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[170] M. Andriluka, U. Iqbal, E. Ensafutdinov, L. Pishchulin, A. Milan, J. Gall, and S. B., "PoseTrack: A benchmark for human pose estimation and tracking," in *CVPR*, 2018.

[171] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 615–629, 2017.

[172] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[173] M. Menze and A. Geiger, "Object scene flow for autonomous vehicles," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[174] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *ICLR*, 2015.

[175] D. Held, S. Thrun, and S. Savarese, "Learning to track at 100 fps with deep regression networks," in *European Conference on Computer Vision*, Springer, 2016, pp. 749–765.

[176] *Nvidia tensorrt programmable inference accelerator*, https://developer.nvidia.com/tensorrt.

[177] Y. Guan, C. Zheng, X. Zhang, Z. Guo, and J. Jiang, "Pano: Optimizing 360 video streaming with a better understanding of quality perception," in *Proceedings of the ACM Special Interest Group on Data Communication*, 2019, pp. 394–407.

[178] J. Jiang, G. Ananthanarayanan, P. Bodik, S. Sen, and I. Stoica, "Chameleon: Scalable adaptation of video analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 253–266.

[179] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 236–252.

[180] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[181] K. Apicharttrisorn, X. Ran, J. Chen, S. V. Krishnamurthy, and A. K. Roy-Chowdhury, "Frugal following: Power thrifty object detection and tracking for mobile augmented reality," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 96–109.

[182] K. Du, A. Pervaiz, X. Yuan, A. Chowdhery, Q. Zhang, H. Hoffmann, and J. Jiang, "Server-driven video streaming for deep learning inference," in *Proceedings of the*

Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, 2020, pp. 557–570.

[183] A. Veit and S. Belongie, "Convolutional networks with adaptive inference graphs," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 3–18.

[184] 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*, 2018, pp. 389–400.

[185] S. Jiang, Z. Ma, X. Zeng, C. Xu, M. Zhang, C. Zhang, and Y. Liu, "Scylla: Qoe-aware continuous mobile vision with fpga-based dynamic deep neural network reconfiguration," in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 1369–1378.

[186] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu, "Deepcache: Principled cache for mobile deep vision," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, ACM, 2018, pp. 129–144.

[187] M. Long, H. Zhu, J. Wang, and M. I. Jordan, "Unsupervised domain adaptation with residual transfer networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 136–144.

[188] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2016, pp. 243–254.