Ph.D. DISSERTATION

Edge-Cloud Collaborative Platform for Live Video Analytics Applications

실시간 비디오 분석 응용을 위한 엣지-클라우드 협력적 플랫폼

FEBRUARY 2024

Department of Computer Science & Engineering
College of Engineering
Seoul National University

Juheon Yi

Edge-Cloud Collaborative Platform for Live Video Analytics Applications

Advisor Youngki Lee

Submitting a Ph.D. Dissertation of Engineering

December 2023

Department of Computer Science & Engineering
College of Engineering
Seoul National University

Juheon Yi

Confirming the Ph.D. Dissertation written by Juheon Yi

February 2024

Chair:	Byung-Gon Chun	(Seal)
Vice Chair:	Youngki Lee	(Seal)
Examiner:	Jae W. Lee	(Seal)
Examiner:	Sung-Ju Lee	(Seal)
Examiner:	Matthai Philipose	(Seal)

Abstract

Live video analytics enables various services including traffic monitoring, surveillance, person identification, and AR/MR. Despite the huge potential, enabling robust and efficient live video analytics apps is non-trivial. The core challenge lies in running the unique workload of analyzing the live video stream in real-time and seamlessly delivering the analysis results to the user for interaction on resource-constrained mobile devices. Such workload often requires a continuous and simultaneous execution of multiple Deep Neural Network (DNN) tasks on high-resolution video streams.

In this dissertation, we depict emerging live video analytics app scenarios and thoroughly characterize their workloads. We then analyze the technical challenges in supporting them, and introduce our research vision and systems to develop an end-to-end, edge-cloud cooperative platform for live video analytics apps.

We first design EagleEye, an AR system to identify missing person(s) in large, crowded urban spaces in real-time. Our key approach is *Content-Adaptive Parallel Execution*, which adapts the multi-DNN face identification pipeline depending on recognition difficulty (e.g., face resolution, pose) and cooperatively execute the workload at low latency using heterogeneous processors on mobile and cloud. To further innovate the performance of the state-of-the-art face identification techniques for LR faces, we also design a novel ICN and its training methodology that utilize the probes of the target to recover missing facial details in the LR faces for accurate recognition. Our results show that ICN significantly enhances LR face recognition accuracy (true positive by 78% with only 14% false positive), and EagleEye accelerates the latency by 9.07$\times$ with only 108 KBytes of data offloaded to the cloud.

We next design Pendulum, an end-to-end live video analytics system with *network-compute joint scheduling*. To overcome the limitations of single-stage scheduling systems for cloud offloading (e.g., bitrate adaptation for live video analytics, DNN schedul-

ing in edge/cloud server) in alternating resource bottleneck scenarios, we newly discover the tradeoff relationship between the video bitrate and DNN complexity. Leveraging this, we design an end-to-end system composed of (i) an efficient and scalable knob control mechanism, (ii) a lightweight tradeoff profiler, and (iii) a multi-user joint resource scheduler. Extensive evaluation on various datasets and state-of-the-art DNNs show that shows that Pendulum achieves up to 0.64 mIoU gain (from 0.17 to 0.81) and 1.29× higher throughput compared to state-of-the-art single-stage scheduling systems.

Finally, we design Heimdall, a mobile platform to support multi-DNN and rendering concurrency on mobile GPUs. To coordinate multi-DNN and rendering tasks, the *Preemption-Enabling DNN Analyzer* partitions the DNNs into smaller units (at operator-level) to enable fine-grained GPU time-sharing with minimal DNN inference latency overhead. Furthermore, the *Pseudo-Preemptive GPU Coordinator* flexibly prioritizes and schedules the multi-DNN and rendering tasks on GPU and CPU to satisfy the app requirements. Heimdall efficiently supports multiple MR app scenarios, enhancing the frame rate from 11.99 to 29.96 fps while reducing the worst-case DNN inference latency by up to ≈15 times compared to the baseline multi-threading approach.

keywords: Live video analytics, Edge-cloud cooperation, Mobile/edge AI

student number: 2020-39481

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Challenges

Live video analytics is an emerging class of applications (apps) which analyzes the live video streams from mobile devices (e.g., smartphones, AR glasses, CCTVs), delivers the analyzed results to the users, and enables real-time user interaction. It enables various services including traffic monitoring [1], surveillance [2], person identification [3], and AR/MR [4, 5] (refer to Chapter 2.1.1 for detailed scenarios).

Despite the potential, realizing robust and efficient live video analytics apps is highly challenging. The core challenge lies in supporting the unique workload of analyzing the live video stream in real-time and seamlessly delivering the analysis results to the user for interaction on resource-constrained mobile devices. Specifically, live video analytics app has the following computational requirements. First, it needs to accurately analyze the live video stream as well as the user behaviors (e.g., hand, gaze movement) for interaction, which often requires a continuous and simultaneous execution of multiple Deep Neural Networks (DNNs) on high-resolution video streams (see Table 2.1). Second, it should seamlessly synthesize and render the analysis results (e.g., bounding boxes and trajectories over the video frames, virtual objects) over the analyzed scenes for immersive user experiences. Finally, background DNN infer-

(a) Criminal chasing. (b) Immersive online shopping. (c) Augmented interactive workspace (source: [6]).

Figure 1.1: Live video analytics application scenarios.

ence computation and foreground UI rendering should be simultaneously performed in real-time under resource constraints.

Supporting such concurrent multi-DNN and rendering workload incurs the following technical challenges.

Large Data Size and Compute (Chapter 2.2.2). Each analysis task requires a repetitive execution of multiple DNNs over high-resolution videos. For example, identifying distant faces in criminal chasing scenario requires face detection on 1080p frame, and face recognition per each face. Running the task in low-latency is non-trivial for both on-device execution as well as cloud offloading: for example, it takes 8.6 and 3.4 seconds for a frame with 17 faces, mainly due to large number of DNN inferences and high network transmission latency due to large frame size.

Dynamic Resource and Workload Fluctuation (Chapter 2.2.3). The workload and resource availability independently fluctuate both within and across the network (video streaming) and compute (DNN inference) stages. Specifically, compute workload fluctuates depending on scene complexity (e.g., number of objects). Network bandwidth fluctuates due to user mobility and wireless channel status [7–9]. Available GPU utilization also fluctuates (independently of bandwidth) due to multi-user resource contention, causing inference latency slowdown [10]. The above factors fluctuate independently of each other, causing alternating resource bottleneck patterns (e.g., Pearson correlation coefficient -0.15, indicating a weak correlation).

Multi-Task Resource Contention (Chapter 2.2.4). Furthermore, running multiple DNN and rendering tasks on resource-constrained mobile devices (especially mobile GPUs) incurs severe resource contention. For example, running 4 DNN tasks and 1080p@30 fps video frame rendering tasks concurrently increases the inference latency from 59.93$\pm$3.68 to 1181$\pm$668, and drops the rendering frame rate to as low as 11.99 fps.

1.2 Research Statement and Design Goals

To tackle the aforementioned challenges, our dissertation aims to answer the following research question: *"How to collaboratively utilize edge and cloud to run the multi-DNN and rendering workload at high throughput and low latency?"* We aim to achieve the following design goals:

- **High Throughput and Low Latency.** We aim at end-to-end scheduling across the end users and analysis server for real-time video processing (e.g., 30 fps throughput) while satisfying the app-specified accuracy requirements. We also aim at soft real-time latency (e.g., $<$100 ms) so that the analysis result is delivered to users promptly for further actions (e.g., bounding box displayed on screen for officers to confirm).

- **End-to-End Optimization.** For robust performance under dynamic resource availability and workload fluctuation, we aim at end-to-end optimization across the edge device, network, and edge/cloud server. We also aim at cross-layer optimization across the application, framework, and OS stack.

- **High Scalability.** Our goal is to design a platform that is scalable across various input video sources (e.g., RGB, RGB-D, LiDAR), analysis tasks (e.g., detection, segmentation), and number of tasks/users.

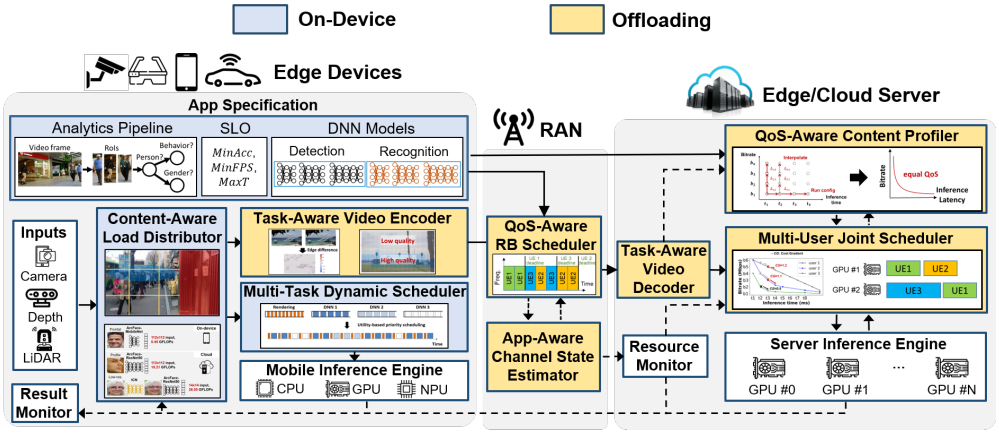


Figure 1.2: Edge-cloud cooperative platform architecture.

Table 1.1: Challenges and our solutions.

Challenges	**Approach**
C#1: Large data size and compute	**S#1:** Content-aware adaptation and cooperative execution (Chapter 4 - EagleEye)
C#2: Dynamic resource and workload fluctuation	**S#2:** Network-compute joint scheduling (Chapter 5 - Pendulum)
C#3: Multi-task resource contention	**S#3:** Multi-DNN/rendering concurrency on mobile GPUs (Chapter 6 - Heimdall)

- **Minimal App Modifications.** For generality, we aim to minimize app/task-specific model re-training or additional model preparation.

1.3 Proposed Platform Overview

Figure 1.2 shows the overall architecture of our edge-cloud collaborative platform for live video analytics apps. We realize our platform with three systems that tackles the aforementioned challenges as summarized in Table 1.1.

Operational Flow. Given the input source video stream (e.g., camera, LiDAR) and the app specification (video analytics pipeline, Service Level Objectives (SLOs), and DNN models), the platform collaboratively utilizes the mobile and the server inference

engines to run the workload. First, *Content-Aware Load Distributor* For the workload scheduled for offloading, the *Task-Aware Video Encoder* efficiently compresses the video with minimal task accuracy drop. The network (video streaming) and compute (DNN inference) stages of the offloading pipeline is scheduled jointly by the *QoS-Aware RB Scheduler* and the *Multi-User Joint Scheduler*, which jointly controls the video encoding bitrate and the inference DNN complexity depending on the network/compute resource availability and input scene content profiled by the *QoS-Aware Content Profiler*. Finally, the *Multi-Task Dynamic Scheduler* utilizes multiple mobile processors (e.g., CPU, GPU, NPU) to schedule the executions of multi-DNN and rendering tasks distributed for on-device inference.

1.4 Organization of the Dissertation

The rest of the dissertation is organized as follows. We characterize the workloads of future live video analytics applications, and analyze challenges in supporting the workload in Chapter 2. We then summarize prior works and their limitations in Chapter 3. Chapters 4–6 details our systems to realize our proposed edge-cloud cooperative platform. Finally, Chapter 7 summarizes discussion and future works.

Chapter 2

Motivational Studies

Our dissertation mainly focuses on MR as representative apps. We first conduct motivational studies to characterize the workloads of futuristic MR apps (Chapter 2.1.1), and analyze the core challenges in supporting the workload (Chapter 2.2).

2.1 Applications and Requirements

2.1.1 Application Scenarios

Criminal Chasing (Figure 1.1(a)). A police officer chasing a criminal in a crowded space (e.g., shopping mall) sweeps the mobile device to take a video of the area from distance. The mobile device processes the video stream to detect faces and find the matching one with the criminal. Specifically, it continuously runs face detection per scene and face recognition per each detected face. Detection results are seamlessly overlayed on top of the camera frames and rendered on screen to narrow down a specific area to search.

Immersive Online Shopping (Figure 1.1(b)). An online shopper wearing MR glasses positions a virtual couch in his room to see if the couch matches well before buying it. The MR glasses analyze the room by detecting its layout and furniture, and renders the couch in a suitable position. The user can also change the style of the couch (e.g.,

color, texture), as well as adjust the arrangement with his hand movements. This app requires i) running object detection and image segmentation simultaneously to analyze the room, ii) running hand tracking and image style transfer to recognize user's hand movements and adjust the style of the couch, and iii) rendering the virtual couch on the right spot seamlessly.

Augmented Interactive Workspace (Figure 1.1(c)). A student wearing MR glasses creates an interactive workspace by combining the physical textbooks and virtual documents. When he encounters a concept he does not understand, he commands the MR glasses to search for related documents on the web via hand gestures. The searched documents are augmented near the textbooks. Also, the note he makes on the textbooks is recognized and saved as a digital file in his device for future edits. This app runs hand tracking and text detection, while seamlessly rendering the virtual documents.

Other Multi-DNN MR Apps include MR emoji [11] (face detection + segmentation + style transfer) or surroundings monitoring for visual support [12] (object and face detection + pose estimation).

2.1.2 Workload Characterization

Real-time, Concurrent Multi-DNN Execution. The core of MR apps is accurately analyzing the physical world and user behavior to combine the virtual contents, which requires running multiple DNNs concurrently (see Tables 2.1 and 2.2 for examples). Also, such analysis needs to be continuously performed over a stream of images to seamlessly generate and overlay the virtual contents, especially in fast-changing scenes (e.g., criminal chasing). Moreover, DNNs need to run over high-resolution inputs for accurate analysis (e.g., recognizing small hand-writings or distant faces requires over 720p or 1080p frames [3, 13]). These characteristics are clearly distinguished from prior works [12, 14–16] that have mostly considered running a single DNN over simple scenes with a few main objects that can be analyzed with smaller resolution (e.g.,

Table 2.1: DNN and rendering requirements for the example MR app scenarios.

	Criminal chasing	Immersive online shopping	Augmented interactive workspace	MR emoji
Continuously executed DNNs (fps)	- Face detection [17] - Face recognition [18] ($<$ 1s per scene)	- Image segmentation [19] (1-5 fps) - Object detection [20] (1-5 fps) - Hand tracking [21] (1-10 fps)	- Text detection [13] (1-5 fps) - Hand tracking [21] (1-10 fps)	- Face detection [17] (1-10 fps) - Image segmentation [19] (1-10 fps)
Event-driven DNNs (response time)		- Image style transfer [22] ($<$ 0.1s)		- Image style transfer [22] ($<$ 0.1s)
Rendering (resolution, fps)	- Camera frames (1080p, 30 fps)[1] - Bounding boxes	- Couch (1440p, 60 fps)[2]	- Virtual documents (1440p, 60 fps)[2] - Handwriting updates	- Camera frames (1080p, 30 fps)[1] - Emoji/character mask

[1,2] Microsoft HoloLens 2 [23] can record 1080p videos at 30 fps, and display 1440p resolution at 60 Hz at maximum.

300×300).

Seamless Rendering on Top of Concurrent DNN Execution. MR apps need to seamlessly augment the virtual contents over the analyzed scenes for immersive user experiences. Such foreground rendering should be continuously performed in real-time in presence of the multi-DNN execution, causing serious contention on resource-constrained mobile GPUs.

Summary. Concurrent execution of multi-DNN and rendering necessitates a platform to prioritize and coordinate their execution on the mobile GPU. Careful coordination will become more important if an app requires audio tasks (e.g., voice command recognition, spatial audio generation) along with the vision tasks, or higher frame rate for more immersive user experience.

2.2 Challenges

2.2.1 Complexity of the State-of-the-art DNNs

One might think that multi-DNN execution on mobile devices is becoming less challenging due to the emergence of lightweight model architectures (e.g., MobileNet [25, 27]) and the increasing computing power of mobile GPUs. However, the challenge still exists. The main reason is that state-of-the-art DNNs do not employ the lightweight

Table 2.2: DNNs for the above MR apps. Inference time is measured on MACE over LG V50 (Adreno 640 GPU).

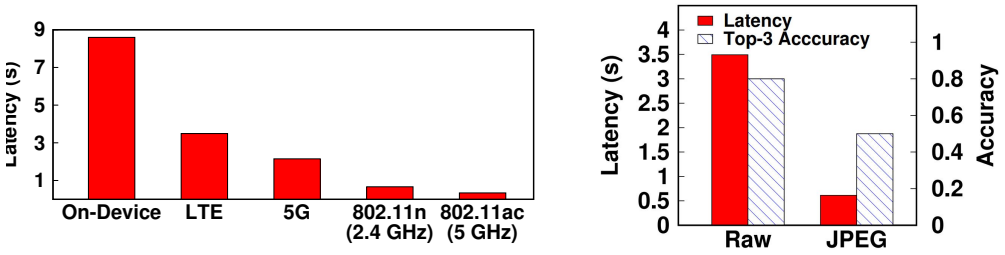
Task	Model	Input size	CPU/GPU ops	Inference time
Object detection	YOLO-v2 [20]	416×416×3	0/33	95 ms
Face detection	RetinaFace [17]	1,920×1,080×3	6/129	230 ms
Face recognition	ArcFace [18]	112×112×3	0/106	149 ms
Image segmentation	DeepLab-v3 [19]	513×513×3	0/101	207 ms
Image style transfer	StyleTransfer [22]	640×480×3	14/106	60 ms
Pose estimation	CPM [24]	192×192×3	0/187	14 ms
Hand tracking	PoseNet [21]	192×192×3	0/74	256 ms
Text detection	EAST [13]	384×384×3	8/117	214 ms

models directly, but enhance them with complex task-specific architectures to achieve higher accuracy.

Table 2.3 compares the complexity of state-of-the-art DNNs with their backbones in terms of floating-point operations (FLOPs) required for a single inference. The reported values are either from the original paper if available, or profiled with TensorFlow.Profiler.Profile() function. Overall, state-of-the-art DNNs require 5.76-10.75× FLOPs than their backbones, showing that the lightweight backbone is only a small part of the whole model. For instance, RetinaFace [17] detector employs feature pyramid [28] on top of MobileNet-v1 [25] to accurately detect tiny faces, whereas ArcFace [18] recognizer adds batch normalization layers on ResNet [26] and replaces 1×1 kernel to 3×3 for higher accuracy. Similar holds for DeepLab-v3 [19] (segmentation model), which adds multiple branches to the backbone MobileNet-v2 [27] to analyze the input image in various scales.

Table 2.3: Complexity comparison between state-of-the-art DNNs and backbones.

	State-of-the-art DNN		Backbone (input size scaled)	
Input size	Model	FLOPs	Model	FLOPs
1,920×1,080	RetinaFace [17]	9.54 G	MobileNet-v1-0.25 [25]	1.65 G
112×112	ArcFace [18]	10.13 G	ResNet [26]	0.95 G
513×513	DeepLab-v3 [19]	16.48 G	MobileNet-v2 [27]	1.54 G



(a) Offloading latency over various networks.

(b) Latency and accuracy of raw and JPEG-compressed frames.

Figure 2.1: Offloading latency of multi-DNN face identification pipeline.

2.2.2 Large Data Size and Compute

Each analysis task execution requires multiple DNN inferences over high-resolution videos. Running the multi-DNN workload in low-latency is both challenging for on-device execution and cloud offloading. Figure 2.1(a) compares the end-to-end latency on-device and offloading latency for multi-DNN face identification on 1080p frame (criminal chasing scenario). We use LG V50 (Qualcomm Adreno 640 GPU) and TensorFlow-Lite for on-device inference, and a desktop server with RTX 2080 Ti GPU. We use different wireless networks: outdoor LTE (11 Mbps) and 5G (45 Mbps), indoor 802.11n (92 Mbps) and 802.11ac (292 Mbps). First, on-device inference takes 8.6s to process a frame average 17 faces, mainly due to large number of DNN inferences. The offloading latency also remains above 0.3 s even for 802.11ac network, and increases to as high 3.4s in outdoor LTE, mainly due to the large data size (i.e., 6 MB 1080p image) to

be transferred over the network. While one may think that utilizing image compression (e.g., JPEG) can reduce the network transmission latency, it comes at the cost of DNN accuracy drop as shown in Figure 2.1(b), as such compression algorithms are mainly designed to minimize the impact on human cognition [29].

2.2.3 Dynamic Resource Availability and Workload

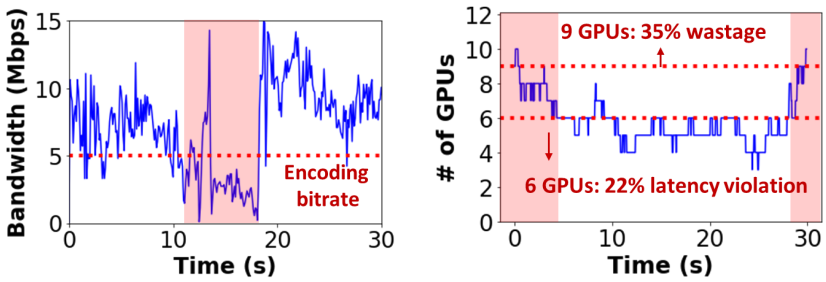
When offloading the video to the edge/cloud server, network/compute resource bottlenecks occur in a complex, alternating pattern over time, as workload and resource availability independently fluctuate both within and across the network and compute stages. This strongly motivates the need for joint scheduling. Specifically, network bottleneck occurs if the bandwidth is smaller than the video encoding bitrate. Compute bottleneck occurs if the DNN inference workload cannot run in real-time on available GPUs.[1]

Dynamic Scene Content and Workload. Video content is highly dynamic across time and location. This dynamically alters the usage of network and compute resources (i.e., video encoding bitrate and DNN inference latency) with low correlation. Specifically, bitrate is proportional to scene change speed (e.g., object or camera motion), whereas inference latency is proportional to the number of objects in the scene. Specifically, for common two-stage analysis tasks (e.g., face identification [3]) composed of (i) object detector and (ii) individual object analyzer, the total processing latency per frame is

$$T_{total} = T_{detect} + N_{object} \cdot T_{analyze}, \tag{2.1}$$

where T_{detect} and $T_{analyze}$ are the object detection and per-object analysis latencies and N_{object} is the number of objects. Figure 5.2 shows a scatter plot of how (bitrate, # of objects) are loosely correlated for different videos (MOT [31] and self-collected from YouTube, each dot is 1s chunk). For example, when a car with a dashcam is driving fast on a sparse highway, the video will yield a low bitrate and high inference

[1]This is a problem even for lightweight DNNs; YOLOv5s [30] latency on 1080p input is 26 ms on RTX 2080 Ti GPU; compute bottleneck occurs if available utilization drops below 78% for 30 fps video.



(a) Network bottleneck (bandwidth vs. bitrate).

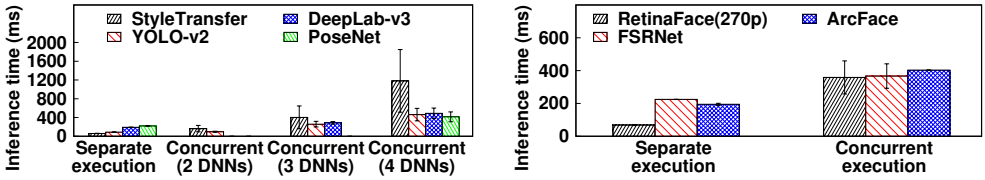
(b) Compute bottleneck (workload vs. available GPUs).

Figure 2.2: Example bottleneck timelines (colored red).

workload (blue dots). However, when it enters a crowded city road and moves slower, the trend will be the opposite (green dots).

Dynamic Resource Availability. Network bandwidth fluctuates due to user mobility and wireless channel status [7–9]. Available GPU utilization also fluctuates (independently of bandwidth) due to multi-user resource contention, causing inference latency slowdown [10].

Bottlenecks Occur Independently. The above factors fluctuate independently of each other, causing alternating resource bottleneck patterns. Figure 2.2 shows an example using a person analysis scenario [32] (composed of 2 recognition DNNs (EfficientNet-B4 [33]) per each person for behavior and gender classification) on MOT17-11 [31] video. First, Figure 2.2(a) shows an LTE bandwidth trace of MahiMahi [34] dataset. Network bottleneck occurs when bandwidth is smaller than the video bitrate (5 Mbps). Second, Figure 2.2(b) shows the same timeline in terms of the number of V100 GPUs required to run the DNN inference in real-time. Compute bottleneck occurs when the number of required GPUs is larger than the available GPUs (e.g., 6). Pearson correlation coefficient of the two bottleneck events is -0.15, indicating a weak correlation. This necessitates the need for joint scheduling of the two stages (i.e., when one stage is bottlenecked, we can use more resources on the other stage).



(a) MACE over LG V50 (immersive online shopping scenario).

(b) TF-Lite over Google Pixel 3 XL (criminal chasing scenario).

Figure 2.3: Multi-DNN GPU contention.

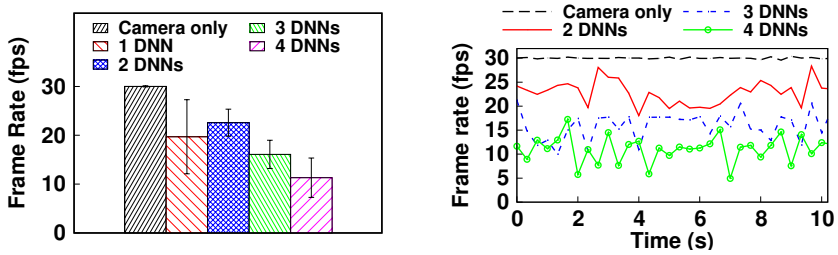
2.2.4 Multi-Task Resource Contention

Furthermore, supporting concurrent multi-DNN and redering task execution in resource-constrained mobile devices (especially mobile GPUs) is challenging due to the lack of framework and architecture support.

2.2.2.1 Multi-DNN GPU Contention

Existing mobile deep learning frameworks [14, 16, 35, 36] are mostly designed to run only a single DNN. The only way to run multiple DNNs concurrently is to launch multiple inference engine instances (e.g., TF-Lite's Interpreter, MACE's MaceEngine) on separate threads. However, multiple DNNs competing over limited mobile GPU resources incur severe contention, unexpectedly degrading the overall latency. More importantly, uncoordinated execution of multiple DNNs makes it difficult to guarantee performance for mission-critical tasks with stringent latency constraints.

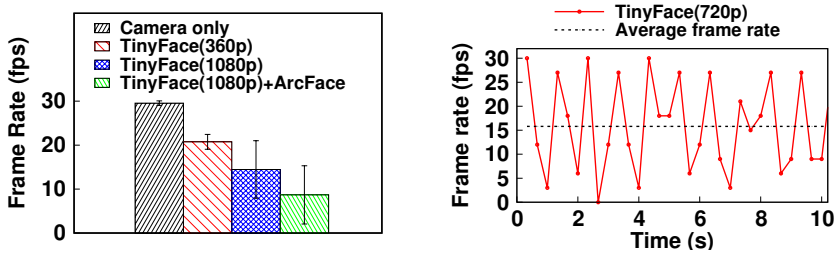
To evaluate the impact of multi-DNN GPU contention on latency, we run 4 DNNs in the immersive online shopping scenario in Table 2.2 on MACE over LG V50. Figure 2.3(a) shows that with more number of DNNs contending over the mobile GPU, the inference times increase significantly compared to when only a single DNN is running (denoted as Separate execution). More importantly, note that the individual DNN inference times are sufficient to satisfy the app requirements (i.e., the sum of the inference times of 4 the DNNs are 560.02 ms, indicating that they can run at $\approx$2 fps when



(a) Average frame rate.

(b) Frame rate over time.

Figure 2.4: Rendering-DNN GPU contention on MACE over LG V50 (immersive online shopping scenario).



(a) Average frame rate.

(b) Frame rate over time.

Figure 2.5: Rendering-DNN GPU contention on TF-Lite over Google Pixel 3 XL (criminal chasing scenario).

coordinated perfectly). However, the uncoordinated execution makes the performance of individual DNNs highly unstable (e.g., the latency of StyleTransfer increases from 59.93±3.68 to 1181±668 ms when 4 DNNs run concurrently), making it challenging to satisfy the latency requirement. We observe a similar trend in TF-Lite: Figure 2.3(b) shows that running 3 DNNs in the person identification pipeline developed in [3] incurs significant latency overhead.

2.2.2.2 Rendering-DNN GPU Contention

More importantly, existing frameworks only consider a single DNN running in an isolated environment (i.e., no other task contending over the mobile GPU), and are ill-suited for MR apps that require concurrent execution of rendering in presence of multiple DNNs. Figure 2.4 shows the 1080p camera frame rendering rate in presence

of multiple DNNs, with the same DNN setting as in Figure 2.3. Figure 2.4(a) shows that when multiple DNNs are running, rendering frame rate drops significantly due to the similar contention in Figure 6.1, becoming as low as 11.99 fps when all 4 DNNs are running. To make matters worse, GPU contention incurs frame rate heavily fluctuating over time as shown in Figure 2.4(b), significantly degrading perceived rendering quality to users. We observe a similar trend on TF-Lite when running TinyFace [37] detector and ArcFace [18] recognizer concurrently with the rendering task (Figure 2.5).

Chapter 3

Related Work

3.1 Live Video Analytics Applications

Live video analytics enables various useful apps including traffic monitoring [38], and AR/MR [3–5]. Gabriel [39] uses cloudlets for cognitive assistance. OverLay [40] and MARVEL [41] utilize cloud for location-based mobile AR services. A large body of work aimed to improve the practicality of video analytics systems, including adaptation [38], model merging [42], privacy protection [43], and continual learning [44,45]. In line with recent works, we characterize the workloads of futuristic live video analytics apps and design an edge-cloud collaborative platform to support the workload.

3.2 On-Device Systems

3.2.1 Mobile Deep Learning Frameworks

Although several frameworks have been developed from both industry [35, 36, 46, 47] and academia [14–16,48–53], they have been mostly focused on running a single DNN in an isolated environment (i.e., no other task contending over GPU). Few studies aimed at running multiple DNNs, but are limited to be applied for concurrent multi-DNN and rendering workload. DeepEye [12] and NestDNN [54] mainly focuses on

memory optimization. DeepEye [12] parallelizes fully connected layer parameter loading and convolutional layer computation but runs only a single DNN on GPU at each time. NestDNN [54] dynamically adapts model size considering available resources but does not consider the coordination of multi-DNN inferences. Lee et al. [55] and Mainstream [56] focus on sharing weights and computations between multiple DNNs.

3.2.2 On-Device Continuous Mobile Vision

Several studies have tackled the challenge of on-device deep learning by model compression [15, 49], inference speed acceleration [14, 16, 48, 52], and model size adaptation [50, 51]. However, existing systems mostly focused on running a single DNN on downsampled images (e.g., 300×300) to analyze one or a small number of large, primary object(s) in vicinity.

For multi-task concurrency support, there have been several studies aimed at enabling efficient GPU sharing on desktop/server GPUs, either by multiplexing multiple kernels temporally [57–59] or spatially [60–65]. Such techniques have been also applied for multi-DNN workloads [66–69]. However, they are ill-suited for mobile GPUs due to limited architecture support and memory bandwidth (see Chapter 6.3.1 for analysis).

3.3 Cloud Offloading Systems

3.3.1 Offloading for Continuous Mobile Vision

MCDNN [70] and DeepDecision [71] dynamically execute DNN on cloud or mobile based on available resources. VisualPrint [72] offloads extracted features rather than raw images to save bandwidth. Glimpse [73] tracks objects by offloading only trigger frames for detection and tracking them in the mobile. Liu *et al.* [74] pipeline network transmission and DNN inference to optimize latency. However, existing systems process the input image as a whole, either on mobile or cloud at a given time; such

approaches can result in significant latency in case of running complex multi-DNN pipeline.

3.3.2 Adaptive Bitrate for Live Video Analytics

A large body of works has designed adaptive bitrate techniques for live video analytics by controlling resolution [75], frame rate (frame filtering) [73,76–82], quantization [4], and a combination of all [83]. Other works have designed RoI filtering and streaming systems [84, 85] and super-resolution-enhanced streaming pipelines [86–88]. Quantization table optimization for DNNs [29, 89] has also been studied. However, they are designed for *network-only scheduling* and cannot scale to alternating resource bottleneck scenarios.

3.3.3 ML Serving in Edge/Cloud Server

Several works aimed at high-throughput inference serving on edge/cloud servers, with content-aware adaptation [38, 90], priority-aware scheduling [1, 54, 91], caching [92–96], or multi-edge workload balancing [32]. However, they are mostly *compute-only scheduling* (assume that videos arrive at the cloud without delay), lacking scalability in network bottlenecks. For example, VideoStorm [1] allocates CPU cores across users considering resource-quality tradeoffs (profiled offline), but cannot leverage additional network resources in compute bottleneck.

3.3.4 Edge-Cloud Collaborative Inference Systems

Several systems efficiently split the DNN inference workload across mobile/edge and cloud [10,97–99]. However, they mostly focus on image processing and lack consideration for videos (e.g., how to efficiently compress the intermediate inference features of consecutive frames). Furthermore, they mostly assume single task scenarios and lack consideration for multi-DNN and rendering concurrency support.

3.4 Tiny ML/Efficient Deep Learning

A large body of works aimed at DNN compression for resource-efficient deep learning in mobile/edge devices. They have leveraged various techniques including lightweight model architecture design [25, 30], weight pruning [100], quantization [101], combination of both [102, 103], hardware-aware model adaptation [104], and neural architecture search [105, 106]. Such optimization techniques can also be leveraged in our platform for resource-efficiency.

Chapter 4

EagleEye: AR-based Person Identification in Crowded Urban Spaces

4.1 Introduction

In this Chapter, we design EagleEye, a system for content-aware adaptation and edge-cloud collaborative execution in live video analytics. We take the AR person finding application as the representative multi-DNN live video analytics workload. Imagine a parent looking for her/his missing child in a highly crowded square. In many cases, a swarm of people in front of her/his eyes will quickly overload cognitive abilities; our motivational study shows that it takes ≈16 seconds to locate a person in a crowded scene (See Chapter 4.3 for details). An Augmented Reality (AR)-based service with smart glasses or a smartphone will be extremely helpful if it can capture the large crowd from distance and pinpoint the missing child in real-time (Figure 4.1). Despite recent advances in person identification techniques using various features such as face [18, 107, 108], gait [109, 110] or sound [111, 112], fast and accurate person identification in crowded urban spaces remains a highly challenging problem.

EagleEye is a AR-based system to identify missing person(s) in large, crowded urban spaces. It continuously captures the image stream of the place using commodity

Figure 4.1: Example usage scenario of EagleEye: parent finding a missing child. More examples in Chapter 4.2.

mobile cameras, identifies person(s) of interests, and shows where the target is in the scene in (soft) real-time. EagleEye not only shows a good example of future AR applications based on real-time analysis of complex scenes, but also characterizes the workload of future multi-DNN mobile deep learning systems.

Designing EagleEye involves critical technical challenges for both identification accuracy and latency.

• **Recognition Accuracy.** Compared to prior systems [113–115] that aim at identifying 1 or 2 faces in close vicinity (e.g., engaged in a conversation), the key challenge in building EagleEye is accurately detecting and recognizing distant small faces. In crowded spaces, individual faces often appear very small, with facial details blurred out. Recent Deep Neural Network (DNN)-based face recognition has shown remarkable progress in accurately identifying faces under various unconstrained settings [18, 116, 117] (e.g., variations in pose, occlusion, or illumination). However, the state-of-the-art techniques still fail to provide robust performance for Low-Resolution (LR) faces. Our study shows that Equal Error Rate, the value in the ROC curve where false acceptance and false rejection rates are identical, of the state-of-the-art DNN [18] grows from 9% to 27% when resolution drops from 112×112 to 14×14 (Chapter 4.3).

• **Identification Latency.** More importantly, it is challenging to analyze a crowded scene in (soft) real-time to allow users to sweep large spaces quickly. EagleEye im-

poses unique challenges compared to recent DNN-based continuous mobile vision systems [12,14–16,70,71,74]. Firstly, as shown in Figure 4.2, EagleEye requires running a series of complex DNNs multiple times for a single scene: face detection network once over a scene, our resolution enhancing network (introduced in Chapter 4.5.2) and face recognition network per each face. This is very different from prior systems that run a single DNN only once over a scene. Secondly, each DNN is highly complex to achieve high accuracy, incurring significant latency. Face detectors employ feature pyramid [28] which upsamples features in latter layers and adds up to earlier layers to detect small faces. Also, state-of-the-art recognizers are heavy ResNet-based. Finally, prior work mostly downsample the input frames (e.g., 300×300 [118]) to reduce complexity (this was possible as they analyze a small number of large, primary objects in vicinity). However, EagleEye should run the identification pipeline on high-resolution frames to detect a large number of distant faces that appear very small.

It is highly challenging to run a complex multi-DNN pipeline over high-resolution images in real-time. It is not even trivial to simply port state-of-the-art DNNs to mobile deep learning frameworks (e.g., TensorFlow-Lite) due to the limited number of supported operations. The challenge aggravates considering the execution latency. For instance, a lightweight MobileNet [25] can only process two 1080p frames per second on high-end mobile GPU (Table 4.1). Naive execution of EagleEye's entire pipeline takes 14 seconds for a scene with 30 faces (Figure 4.5). We can consider multithreading or offloading, but they are not also straightforward to apply. Multithreading degrades performance due to resource contention over limited mobile resources (e.g. GPU, CPU, memory). Also, 3G/LTE network with low bandwidth is likely the only wireless network available in crowded outdoor environments, making offloading non-trivial.

To tackle the challenges, we design and develop a suite of novel techniques and adopt them in EagleEye.

• **Identity Clarification Network.** We first design a novel end-to-end face iden-

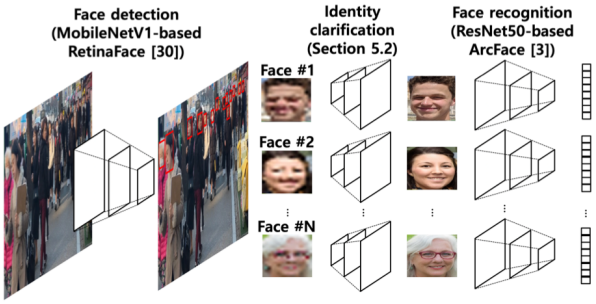


Figure 4.2: Multi-DNN face identification pipeline.

tification pipeline to identify small faces accurately. Our key idea is to add *Identity-Clarification Network (ICN)* on conventional 2-step pipeline (detection-recognition) to recover missing facial details in LR faces, thus resulting in a 3-step pipeline (detection-clarification-recognition as shown in Figure 4.2). ICN adopts a state-of-the-art image super-resolution network as the baseline and innovates it with specialized training loss functions to enhance LR faces for accurate recognition; note that prior super-resolution networks focus on generating perceptually natural images and fail to preserve identities, making them ill-suited for recognition [119] (See Chapter 4.5). Also, ICN enables identity-preserving reconstruction using reference images (*probes*) of the target, commonly available in our scenarios (e.g., photos of children provided by parents). We observe that the complexity of LR face recognition results from accepting positive identities rather than denying negative identities (see Chapter 4.5.2 for details). Thus, biasing ICN on the target improves LR face recognition accuracy with only a small increase in false positives. Overall, our ICN-enabled pipeline improves true positives by 78% with 14% false positives, against the 2-step identification pipeline.

• **Multi-DNN Execution Pipeline.** Our workload (i.e., running a series of DNNs multiple times on high-resolution images) requires a differentiated strategy to optimize the heavy computation. We develop a runtime system with *Content-Adaptive Parallel Execution* to run a multi-DNN face identification pipeline at low latency. The key idea

behind this approach is to divide the high-resolution image into multiple sub-regions and selectively enable different components in the pipeline, depending on the content. For instance, ICN is only applied to a region with LR faces while the entire pipeline is not executed for a background region with no faces. Furthermore, we exploit the spatial independence of face recognition workload (i.e., identifying faces in different sub-regions does not have dependency) to parallelize and pipeline the execution on heterogeneous processors on the mobile and cloud. Overall, our technique accelerates the latency by 9.07× with only 108 KBytes of data offloaded.

Our major contributions are summarized as follows:

- To the best of our knowledge, this is the first end-to-end mobile system that provides accurate and low-latency person identification in crowded urban spaces.
- We design a novel face identification pipeline capable of accurately identifying small faces in crowded spaces. By employing Identity Clarification Network to recover facial details of LR faces, we enhance true positives by 78% with 14% false positives.
- We design a runtime system to handle the unique workload of EagleEye (i.e., processing high-resolution images with multiple DNNs for complex scene analysis). We believe this will be an unexplored common workload for many mobile/wearable-based continuous vision applications. We utilize a suite of techniques to minimize the end-to-end latency to as low as 946 ms (9.07× faster than naive execution).
- We conduct extensive controlled and in-the-wild study (with real implementations and various datasets), validating the effectiveness of our proposed system.

4.2 Motivating Scenarios

Finding a Missing Child. In crowded squares or amusement parks, there are many cases where a parent loses track of her/his child. In such incidents, it is difficult to find the missing child with naked eyes since she/he becomes cognitively overloaded

to identify many people in vicinity. EagleEye can help the parent: by sweeping the mobile device to capture the space from distance, it can help quickly pinpoint possible faces and narrow down a specific area to search, so that the parent can find the child before the child moves to a different place. Similarly, police officers can use EagleEye to chase criminals in crowded malls, streets, squares, etc.

Children Counting in Field Trips. Teachers in kindergarten regularly take children out for field trips to catch educationally-depicting behaviors hardly captured in classroom settings. However, in reality, teachers spend most of the time counting children to make sure they are in place. EagleEye can be of extensive use to reduce the cognitive burden for the teachers so that they can focus on the original goal.

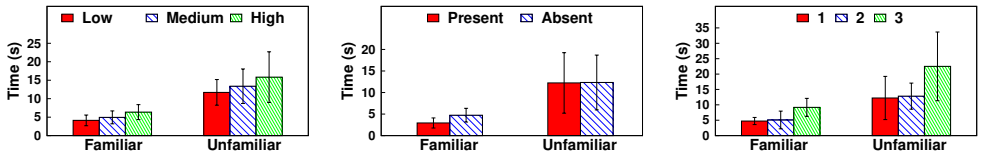
Social Services for Familiar Strangers. EagleEye can be used to build an interesting social service to connect people. For example, it can be used to identify familiar strangers (people whom we met in the past but do not remember the details) to help with interaction; a person attending a social event can use EagleEye to identify them and get an early heads-up before they are in close proximity to avoid embarrassing moments.

4.3 Preliminary Studies

To motivate EagleEye, we first conduct a few studies to verify (1) how quickly humans identify face(s) in crowded urban spaces and (2) whether it is feasible in terms of accuracy and speed to employ face recognition algorithms to aid humans' cognitive abilities.

4.3.1 How Fast Can Humans Identify Faces?

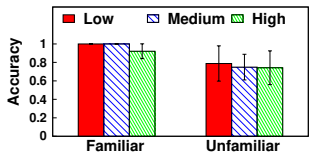
Prior studies report that it takes for humans about 700 ms to detect a face in a scene [120], and about 1 second to recognize the identity of a single face image [121]. We extend the experiments to study how long it takes to identify target(s) in crowded scenes. We



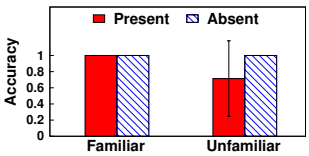
(a) Crowdedness (response time).

(b) Presence vs. absence (response time).

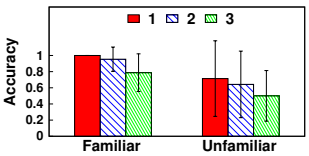
(c) Number of targets (response time).



(d) Crowdedness (accuracy).



(e) Presence vs. absence (accuracy).



(f) Number of targets (accuracy).

Figure 4.3: Human cognitive abilities on identifying faces in crowded scenes: response time and accuracy.

first recruit 6 college students (5 males and 1 female, age 24-28) as subjects for dataset collection, and take videos of them blending inside the crowd in various urban spaces including college campus, downtown streets, and subway stations. Next, we recruit 11 students (10 males and 1 female, age 24-32) who are of mutual acquaintances with the subjects (denoted as *Familiar*), and 14 other students (12 males and 2 females, age 20-26) who have never seen the subjects before (denoted as *Unfamiliar*).

In the experiments, the participants are seated in front of the screen with a similar setup as in [120]. Each participant is first shown faces of 1 to 3 target identities. Afterwards, a scene image (1080p resolution) is shown, in which target(s) may or may not exist. The participant clicks the location in the scene where she/he finds each target. Response time is measured as the duration between when the scene is displayed and when the participant finishes identifying all targets. The scenes are classified into three levels of crowdedness (examples are shown in Figure 4.16): i) *Low* (less than 10 people in close distance with face sizes at least 30×30 pixels), ii) *High* (more than 20 people with face sizes smaller than 14×14), and iii) *Medium* (between *Low* and *High*). Each participant is shown 5 scenes per each category (15 in total) and was asked to be

as precise as possible.

Figure 4.3 shows the response time/accuracy results. Our experimental results are summarized as follows (unless specified, the reported results are on *High* scenes):

- Overall, it takes 6.37 and 15.83 seconds on average to identify familiar and unfamiliar faces in crowded scenes, respectively, showing noticeable cognitive loads.
- It takes longer to identify unfamiliar faces than familiar ones.
- Not only does it take longer to identify a target in more crowded scenes, but the accuracy also drops (Figures 4.3(a) and (d)).
- Especially for the *Familiar* group, it takes longer to confirm the absence of target than presence. (Figures 4.3(b) and (e)). We observe that it is because when participants fail to locate the target in the scene, they start looking over again multiple times to confirm their decision.
- It takes longer to identify multiple targets, and accuracy drops as well (Figures 4.3(c) and (f)).

The above results clearly show the human's vulnerability to cognitive overload. While the study was designed as identifying the target person(s) in a scene image for controllability of the experiment, we conjecture that the cognitive overload will be greater in real-world settings where the scene does not fit into a single view.

4.3.2 How Accurate Can DNNs Identify Faces?

Faces in crowded spaces captured from a distance experience high variations in pose, occlusion, illumination, and resolution, making accurate recognition very challenging. While prior algorithms have achieved robust performance (e.g., over 90% accuracy) for the first three [18, 116, 117], the Low-Resolution (LR) face recognition problem has not been fully studied yet.

We conduct a study to analyze the difficulty of LR face recognition. We first train ResNet50 with ArcFace loss [18] on MS1M dataset [122], and test performance on 50 identities in VGGFace2 [123] testset (50 images per identity). Figure 4.4 shows that

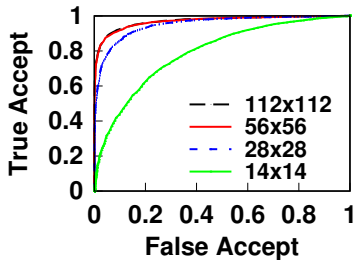


Figure 4.4: Face verification accuracy.

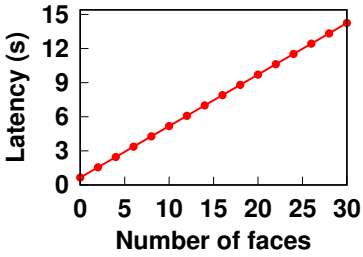
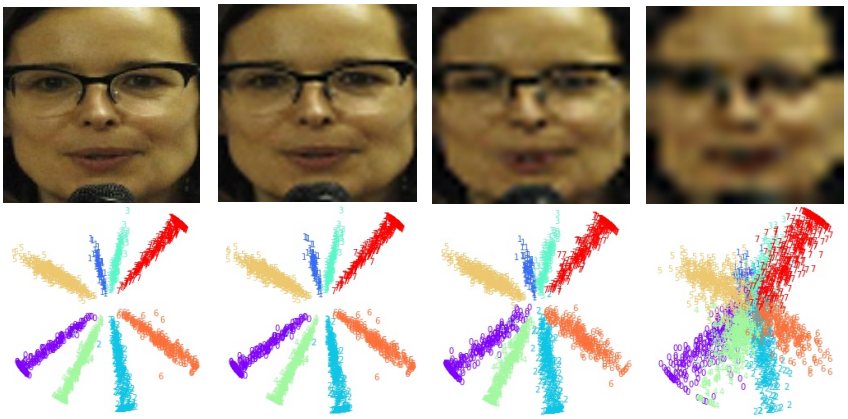


Figure 4.5: Latency of face identification pipeline.



(a) 112×112. (b) 56×56. (c) 28×28. (d) 14×14.

Figure 4.6: Feature map visualization for varying resolutions (points with same color represents same identity).

verification (determining whether two faces match or not) accuracy drops significantly as resolution decreases. Equal Error Rate (EER), the value in the ROC curve where false acceptance and false rejection rate are identical, grows as high as 0.27 when the resolution is 14×14.

For further analysis, we run a small study with 8 identities in VGGFace2 [123] testset. We train ResNet50 [26] with 2-dimensional output features using SphereFace loss [107]. Figure 4.6 visualizes the trained features for varying resolutions, where the points with the same color represent the same identity. We observe that when the resolution is high (e.g., 112×112), features for each identity form non-overlapping

Table 4.1: Inference time of DNNs with TensorFlow-Lite running on LG V50 (Qualcomm Adreno 640 GPU).

	Model	
Input size	MobileNetV1 [25] (Classification)	YOLO-v2 [20] (Detection)
224×224	24 ms	357 ms
640×360	55 ms	1,477 ms
1,280×720	209 ms	5,009 ms
1,920×1,080	452 ms	9,367 ms

Table 4.2: Complexity and latency of component DNNs. FLOPs are measured with *tf.profiler.profile()* function.

Task	Model	FLOPs	Inference time
Face detection	RetinaFace [17] (MobileNetV1-based)	9.54 G	648 ms per 1080p image
Identity clarification	Ours (Chapter 4.5.2)	15.84 G	166 ms per 14×14 face
Face recognition	ArcFace [18] (ResNet50-based)	10.21 G	287 ms per 112×112 face

sharp clusters. However, as resolution drops, clusters become wider and start to overlap with each other, becoming indistinguishable.

4.3.3 How Fast Can DNNs Identify Faces?

Conventional face identification pipelines operate in a 2-step manner (i.e., face detection on the image and face recognition on each detected face sequentially). In our scenarios, both steps require significant computation. First, the detection network should run on a high-resolution frame to detect distant faces that appear very small. In such settings, providing real-time performance is challenging; Table 4.1 shows that YOLOv2 [20], one of the fastest networks that can be used for face detection, takes

more than 9 seconds to process a 1080p frame. Second, recognition latency increases proportionally to the number of faces, which can be very large in crowded scenes. Figure 4.5 shows that naively running the state-of-the-art multi-DNN face identification pipeline composed of DNNs summarized in Table 4.2 [1] takes more than 14 seconds to process a scene with 30 faces even on a high-end LG V50 with Qualcomm Adreno 640 GPU.

4.3.4 Summary

In crowded spaces, humans become cognitively overloaded, clearly necessitating the need for a system to aid their abilities. However, DNN-based face recognition algorithms cannot be applied directly as they fail to identify LR faces accurately, and naive execution incurs significant latency.

4.4 EagleEye: System Overview

4.4.1 Design Considerations

High Recognition Accuracy. Our primary objective is to design a face identification pipeline capable of accurately identifying target(s) in crowded spaces, even when he/she appears very small.

Soft Real-Time Performance. While enabling an accurate face identification pipeline, our goal is to provide soft real-time performance (e.g., 1 fps) for application usability. We aim to devise techniques to optimize various latency components in the end-to-end system while incurring a minimum loss in recognition accuracy.

Use of Commodity Mobile Camera. We aim at achieving high accuracy using frames captured by cameras of commodity smartphones or wearable glasses (e.g., 1080p

[1] These are the state-of-the-art not only in terms of accuracy but also in terms of complexity. For face detectors, comparable networks are heavy VGG16 [124] or ResNet101 [37]-based. Recent face recognizers are based on 64-layered ResNet [107, 108].

frames at 30 fps [125]). If cameras with higher resolution or optical zoom-in are available, our approach can help cover a more extensive search area.

Minimal Use of Offloading. In our common use cases (i.e., a moving user in crowded outdoor environments), we assume that the availability of edge servers and Wi-Fi connectivity are limited. For robust performance, we aim to minimize the amount of data offloaded to the cloud and run most of the computation on local.

4.4.2 Operational Flow

Figure 4.7 shows the nutshell operation of EagleEye: given a crowded scene image, we adaptively process each region with different pipelines depending on the content. For background regions, we do not run any DNN. For non-background regions, we run face detection and adaptively select the latter part of the pipeline to process each detected face based on different variations: i) large, frontal faces (which are very easy to recognize) are processed with a lightweight recognition network, ii) large, profile faces (whose resolutions are sufficient but pose variations make recognition difficult) are processed with a heavy recognition network, and iii) small faces are first processed with *Identity Clarification Network)* (which enhances resolution of LR faces for accurate recognition) and then with heavy recognition network. Finally, exploiting the spatial independence of the task, we process each region and face in parallel on heterogeneous processors on mobile and cloud.

Figure 4.8 shows the operational flow of EagleEye. We employ *Content-Adaptive Parallel Execution* to run the complex multi-DNN face identification pipeline at low latency using heterogeneous processors on mobile and cloud. Given an input frame, *Spatial Pipelining* first divides it into spatial blocks, so that each block can be processed in a pipelined and parallel manner. Afterwards, *Edge-Based Background Filtering* rules out background blocks with edge intensity lower than a threshold. For the remaining blocks, we detect faces on the mobile CPU. Each detected face is scheduled to a different pipeline by *Variation-Adaptive Face recognition*. Large, frontal faces are

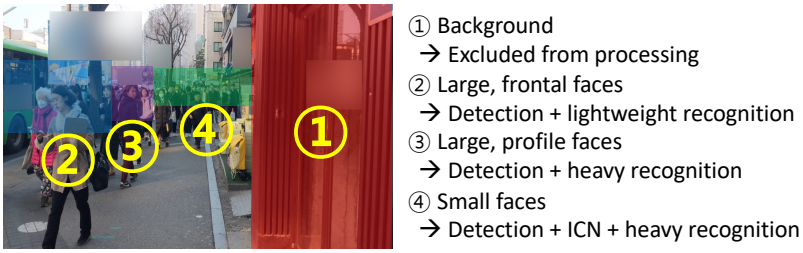


Figure 4.7: Operation of EagleEye in a nutshell.

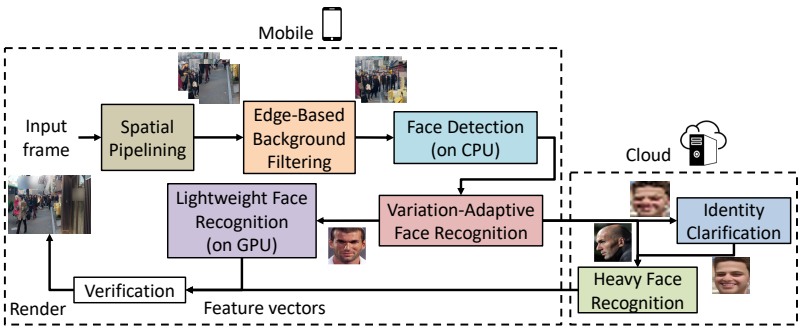


Figure 4.8: EagleEye system overview.

processed by lightweight recognition network running on mobile GPU. The rest is offloaded to the cloud, where large, profile faces are processed by heavy recognition network, and small faces are processed by ICN and then by heavy recognition network.

4.5 Identity Clarification-Enabled Face Identification Pipeline

In this section, we detail our novel 3-step face identification pipeline. It operates as shown in Figure 4.2: i) detect faces in the scene, ii) enhance each LR face with ICN, and iii) extract feature vectors for each face with recognition network.

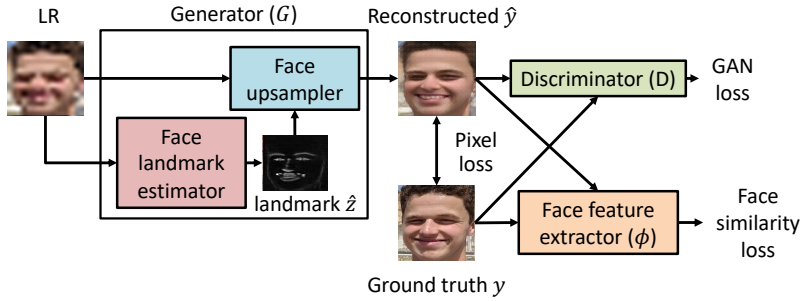


Figure 4.9: Identity Clarification Network: overview.

4.5.1 Face Detection

The first step of our pipeline is face detection. The detection network should be accurate in detecting small faces, since faces missed in this step would lose the chance of being identified at all. At the same time, it should be lightweight so that it can run in (soft) real-time. We experiment various state-of-the-art DNNs and select RetinaFace detector [17] with MobileNetV1 [25] backbone for the following reasons: i) it adopts context module which has been proven very effective in detecting small faces [124, 126], and ii) it is the fastest among the state-of-the-art group due to its lightweight backbone network (others are heavy VGG16-based [124] or ResNet101-based [37]).

4.5.2 Identity Clarification Network

LR faces lack details crucial for identification. To enhance recognition accuracy, we design *ICN*, which enhances the resolution of LR faces using Generative Adversarial Network (GAN). As conventional GANs reconstruct faces with significant distortion from the original identity (Figure 4.11), we adapt GAN to reconstruct identity-preserving faces by using various loss functions, as well as a specialized training methodology (*Identity-Specific Fine-Tuning*).

Network Architecture. Figure 4.9 shows the overview of ICN. For generator G, we

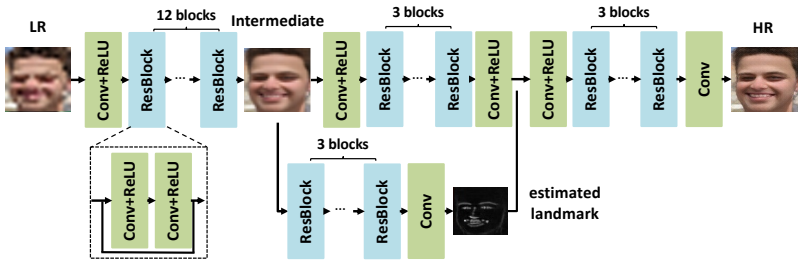


Figure 4.10: Generator network architecture.

adopt Residual block [26]-based architecture similar to FSRNet [127] as shown in Figure 4.10, which has shown high reconstruction performance. Furthermore, we employ anti-aliasing convolutional and pooling layers [128] to improve robustness to pixel misalignment in face detection and cropping process. We employ various additional networks and loss functions to train ICN to preserve identity as follows.

Following the convention in super-resolution [129, 130], the generator is trained to minimize the pixel-wise L2 loss between the reconstructed face and the ground truth,

$$L_{pixel} = \frac{1}{HW} \sum_{i=1}^{H} \sum_{j=1}^{W} \left(\|y_{i,j} - \tilde{y}_{i,j}\|^2 + \|y_{i,j} - \hat{y}_{i,j}\|^2 \right), \tag{4.1}$$

where H, W are height and width, $\tilde{y}$ and $\hat{y}$ are the intermediate and final High-Resolution (HR) face in Figure 4.10, respectively, and y is the ground truth.

As reconstructing HR faces is very challenging, recent studies have shown that employing a facial landmark estimation network to guide the reconstruction process yields superior performance [127, 131]. We adopt the approach to estimate facial landmarks from the intermediate HR face instead of directly from the LR face. The facial landmark estimation network is trained to minimize the MSE between estimated and ground truth landmarks,

$$L_{landmark} = \frac{1}{N} \sum_{n=1}^{N} \sum_{i,j} \|z_{i,j}^n - \hat{z}_{i,j}^n\|^2, \tag{4.2}$$

where $\hat{z}_{i,j}^n$ is the estimated heatmap of the n-th landmark at pixel (i, j) and z is the ground truth.

Figure 4.11: GANs reconstruct realistic faces, but fail to preserve the face identity.

Recent studies have shown that GAN [132] plays an important role in reconstructing realistic images. We employ WGAN-GP [133] for improved training stability, whose loss is defined as:

$$L_{GAN} = -D(\hat{y}) = -D\left(G\left(x\right)\right), \tag{4.3}$$

where $G(x)$ denotes the HR face reconstructed by the generator, and D denotes the discriminator that classifies whether the reconstructed face looks real or not, which is trained by minimizing the following loss function (refer to the original paper [133] for details),

$$L_{Discriminator} = D(\hat{y}) - D(y) + \lambda\left(\|\nabla_{\hat{x}} D\left(\hat{x}\right)\|_2 - 1\right)^2. \tag{4.4}$$

We also enforce the reconstructed face to have similar features with the ground truth by minimizing the face similarity loss

$$L_{face} = \frac{1}{d}\|\psi\left(y\right) - \psi\left(\hat{y}\right)\|^2, \tag{4.5}$$

where $\psi(\cdot)$ denotes d-dimensional feature vector extracted by the VGG16 network trained on ImageNet [134].

Finally, the above loss functions are combined as a weighted sum and minimized in the training process,

$$L_{total} = L_{pixel} + 50 \cdot L_{landmark} + 0.1 \cdot L_{GAN} + 0.001 \cdot L_{face}. \tag{4.6}$$

Identity-Specific Fine-Tuning. Baseline ICN aims to adapt conventional GANs to overcome their limitation (i.e., reconstructing perceptually realistic faces at the cost

of significant distortion from the ground truth). However, we notice that it still often reconstructs faces with distorted identity from the original. Accordingly, we need another step to employ ICN for our purpose of accurate recognition.

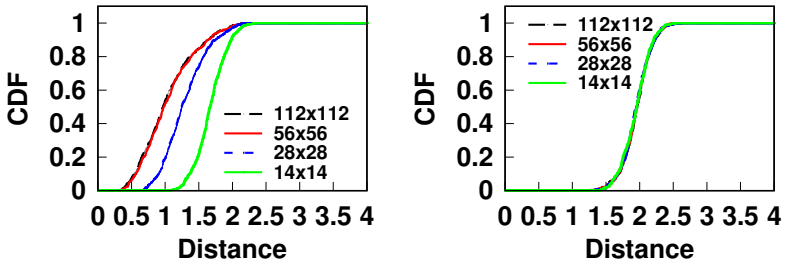
Before introducing our approach, we further dig deeper into the LR face recognition problem. Figure 4.12 shows that as resolution decreases, L2 distance between features of faces with the same identity increases significantly, whereas those of different identities remain identical. In other words, the difficulty of LR face recognition comes from the hardship of accepting positive pair of faces, rather than denying negative pairs. Therefore, LR face recognition accuracy can be enhanced if we can bring back the features of faces with the same identity close to each other.

To this end, we develop *Identity-Specific Fine-Tuning* to re-train ICN with reference images (*probes*) of the target, which is commonly available in our target scenarios (e.g., photos of children provided by parents). Such re-training process enables ICN to instill the facial details of the target into the input LR face, thus making it easier to recognize when a LR face of target identity is captured. While such biasing may also increase false positives caused by LR faces that do not match the target identity pulled towards the probes, we observe that such cases only occur for ones that are very close to the target in feature space, thus yielding gain in true positives outweighing false positives (78% vs. 14% as shown in Chapter 4.8.3).

Probe Requirements. To fine-tune the ICN to instill facial details of the target, Identity-Specific Fine-Tuning requires probe images with rich facial details. As an initial study we collect the probes with high-resolution, and leave detailed analysis of the impact of the composition of probes (e.g., pose or occlusion) as future work.

Data Augmentation. To diversify the probes as well as boost robustness to various real-world degradation, we also utilize the following augmentation techniques:

- **Illumination.** Change value (V) component in HSV color space.
- **Blur.** Apply Gaussian blur with varying kernel sizes.
- **Noise.** Add Gaussian noise with varying variance.



(a) Same identity pair.

(b) Different identity pair.

Figure 4.12: CDF of face distances for varying resolutions.

- **Flip.** Apply horizontal flip.
- **Downsampling.** Resize with different downsampling kernels. (e.g., bicubic, nearest neighbor).

Scalability. Finally, the overhead of fine-tuning the baseline ICN pre-trained on a large-scale face dataset to a specific target identity is not significant (e.g., takes about 20 minutes on a single NVIDIA GTX 2080Ti GPU). Thus, we expect it can be flexibly re-trained at deployment as the target changes.

4.5.3 Face Recognition and Service Provision

At the final stage, state-of-the-art ResNet50-based ArcFace [18] runs on each face to extract 512-dimensional feature vector, which is compared to that of the target probes. Those with distance below the threshold are highlighted on the screen so that the user can take further actions. To compensate for possible motion between the image capture and output rendering (about 1 second as our evaluation shows), we can employ motion tracking to shift the bounding boxes using approaches used in prior detection systems [73, 74].

4.6 Real-Time Multi-DNN Execution

In this section, we detail our runtime system to execute the multi-DNN face identification pipeline at low latency. We start with workload characterization by identifying the

sources of latency, followed by our proposed *Content-Adaptive Parallel Execution*.

4.6.1 Workload Characterization

Sequential Execution of Multiple DNNs. Identifying target person(s) in a crowded scene requires a sequential execution of multiple complex DNNs (i.e., face detection, identity clarification, and recognition) whose individual complexities are summarized in Table 4.2.

High-Resolution Input. Conventional object detection networks downsample the input images to reduce complexity (e.g., 416×416 [20] or 300×300 [118]). However, in our case, the input image size should be retained large (e.g., 1080p), so that small faces have enough pixels to be detected. As the complexity of DNN inference grows proportionally to the image size, latency becomes significant when processing such high-resolution images.

Repetitive Execution for Each Face. ICN and recognition network must repeatedly run for each face detected by the face detection network. The latency increases proportionally to the number of faces in the scene, which becomes significant in crowded spaces.

4.6.2 Content-Adaptive Parallel Execution

4.6.2.1 Optimization Strategies

Content-Adaptive Pipeline Selection. We adaptively process each region of the image with different pipelines depending on the content. This helps optimize the latency incurred when processing a large number of faces, while maintaining high recognition accuracy.

Spatial Independence and Parallelism. Identifying faces in different regions of the image is spatially independent. Furthermore, recognizing each detected face can be executed simultaneously. To take full advantage of such opportunities for parallelism,

(a) Raw frame. (b) Edges. (c) Filtered.

Figure 4.13: Edge-based background filtering.

we divide the image into spatial blocks and process them in a pipelined and parallel manner using heterogeneous processors on mobile and cloud. This helps optimizing the latency of multi-DNN execution on high-resolution images.

4.6.2.2 Content-Adaptive Pipeline Selection

We develop techniques to optimize the latency of complex multi-DNN face identification pipeline execution while maintaining high accuracy. Specifically, *Edge-Based Background Filtering* rules out background regions where faces do not exist at all. *Variation-Adaptive Face Recognition* selects different recognition pipelines depending on recognition difficulty.

Edge-Based Background Filtering. Running face detection on regions where faces do not exist at all (e.g., background) is a wasteful computation. To mitigate the problem, we use edges in the image to rule out such regions before running the identification pipeline. Specifically, given a frame as shown in Figure 4.13(a), we detect edges as in Figure 4.13(b), filter out blocks with edge intensity below a threshold as depicted in Figure 4.13(c), and run face detection only on the remaining blocks. Note that edge detectors are extremely lightweight, especially considering that we can even detect edges on downsampled images. For example, the time complexity of Canny edge detector [135] for $H \times W$ frame is $O(HW \cdot \log(HW))$, and it runs in less than 2 ms for 360p frame on LG V50. Thus, its overhead is minimal even when the edge detection is not effective for some scenes having full of objects and no background regions.

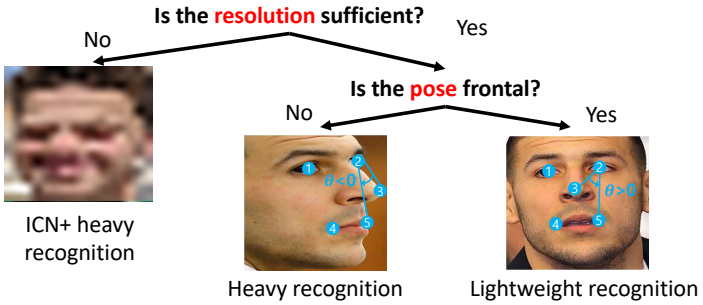


Figure 4.14: Variation-Adaptive Face Recognition.

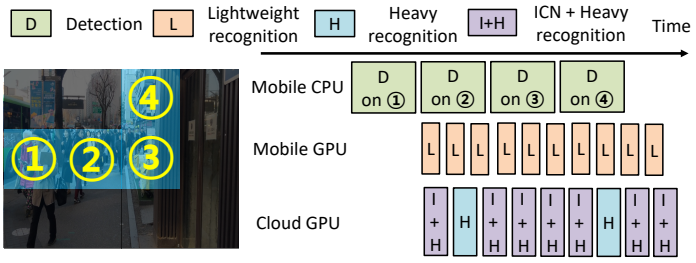


Figure 4.15: Spatial Pipelining on heterogeneous processors.

Variation-Adaptive Face Recognition. State-of-the-art recognition networks are designed very complex (e.g., heavy ResNet backbone with a large number of batch normalization layers) to accurately identify faces even under high variations in pose, illumination, etc. However, employing such heavy networks for faces in ideal conditions is an overkill. For example, MobileFaceNet [136] and ResNet50-based ArcFace [18] achieve comparable accuracy on LFW [137] dataset composed of large, frontal faces (98.9% vs. 99.3%), whereas inference time differs by more than 20× (14 ms vs. 287 ms). Therefore, we aim to optimize latency by adaptively processing each face depending on its variation (i.e., recognition difficulty).

Figure 4.14 depicts our Variation-Adaptive Face Recognition, which utilizes the size of bounding box and 5 face landmarks detected by RetinaFace [17] detector. First, small faces are processed by ICN and then by ResNet50-based ArcFace [18].

Algorithm 1 Combined operational flow of EagleEye

1: **while** application is running **do**
2: $Result \leftarrow \{\}$
3: $Frame \leftarrow acquireFrameFromCamera()$
4: $Edges \leftarrow EdgeDetector(Frame)$
5: $NonBackground \leftarrow BackgroundFilter(Edges)$ $Block$ in $NonBackground$
6: $Faces \leftarrow FaceDetection(Block)$ $face$ in $Faces$
7: $Result \leftarrow$Result$\cup AdaptiveFaceRecognition(face)$
8: Render $Result$ on screen

For large faces, we estimate the pose using the detected landmarks; for example, if the angle between the line connected by points $(2, 3)$ and $(2, 5)$ measured in counterclockwise direction is negative, we can tell that the face is looking to the right. As faces with pose variations are difficult to accurately identify, they are also processed by ResNet50-based ArcFace (ICN is not needed here as resolution is already sufficient). The remaining faces (large and frontal) which are easy to identify are processed by MobileFaceNet [136].

4.6.2.3 Execution Planning

We optimize latency of multi-DNN face identification pipeline by scheduling each component DNN execution to the most suitable processor on mobile and cloud.

Offloading Decision. As our target scenarios assume crowded outdoor environments with congested 3G/LTE network, offloading high-resolution images for detection is impractical; instead, we offload only the detected faces. Specifically, LR faces are suitable for offloading, as their data sizes are very small (e.g., 14×14 pixels) whereas the required computation (i.e., ICN and heavy recognition) incurs significant latency on mobile (e.g., 166+287 ms). We also offload large, profile faces, and leave only the large, frontal faces to be processed by lightweight recognition on mobile.

Mobile Processor Mapping. The mobile needs to run both detection and lightweight recognition. However, simply multithreading the execution on GPU does not help op-

timize latency, as mobile GPUs lack preemptive multitasking support. Therefore, we utilize heterogeneous processors (CPU and GPU) to parallelize the execution. As dynamically switching the mapping over time is challenging due to high latency overhead of loading DNN on mobile GPUs (e.g., 2 seconds for 118 MB ResNet50-based ArcFace [18] on LG V50 with TensorFlow-Lite), we statically run detection on CPU and recognition on GPU considering the following aspects:

- **Memory I/O.** Running face detection on GPU requires high-resolution images loaded onto GPU memory, and output feature maps from different stages in the feature pyramid (whose size is proportional to the input image size) copied back to CPU to be post-processed to bounding boxes. Considering memory overhead, it is more suitable to run face recognition on GPU whose input/output are small-sized faces and 1D feature vectors.
- **Inference time.** Besides, we observe that the inference speed slowdown of RetinaFace detector running on CPU is 1.22× (648 vs. 793 ms), whereas it is 2.07× for MobileNetV1-based ArcFace recognizer (14 vs. 29 ms). Therefore, running detection on CPU and recognition on GPU is more feasible to optimize overall latency, especially when the number of faces is large.

4.6.2.4 Spatial Pipelining

To further optimize the latency, we exploit the spatial independence of the workload by processing each image sub-block in a pipelined and parallel manner. As depicted in Figure 4.15, given non-background blocks in a scene, we detect faces in one block on mobile CPU, while simultaneously processing faces detected in another block on mobile and cloud GPU.

Note that we need to divide the image into blocks in an overlapping manner with padding, so as to prevent faces from being split across different blocks (and thereby failing to be detected). While fine dividing increases the chance of higher parallelism, it also increases the computational overhead due to padding. Based on our empirical

(a) Low. (b) Medium. (c) High.

Figure 4.16: In-the-wild dataset examples.

Table 4.3: Average and standard deviation of the composition of each face type in the test dataset.

	Low	Medium	High
Large frontal	3.00±2.62	3.85±2.11	5.20±3.73
Large profile	1.00±0.76	1.50±1.49	2.8±1.78
Low-resolution	3.07±1.75	5.45±2.50	8.87±3.64
Total	7.07±1.79	11.10±3.74	16.87±4.78

evaluation on such tradeoff in Chapter 4.8.4, we divide an image into 4x4 blocks.

4.6.2.5 Putting Things Together

Algorithm 1 summarizes the combined operational flow. Upon acquiring a frame from the camera, we detect edges (line 4) and filter out background (line 5). For non-background blocks (line 5), we run face detector on CPU (line 6) and process each face adaptively in mobile or cloud GPU (lines 6–7) in a pipelined and parallel manner. Finally, the recognition result is rendered on the screen.

4.7 Implementation

Mobile. We implement the mobile side of EagleEye on two commodity smartphones running on Android 9.0.0: LG V50 with Qualcomm Snapdragon 855 and Adreno 640 GPU and Google Pixel 3 XL with Qualcomm Snapdragon 845 and Adreno 630 GPU. Unless stated otherwise, we report evaluation results on LG V50. RetinaFace [17]

and MobileFaceNet [136] are implemented using TensorFlow 1.12.0 and converted to TensorFlow-Lite for mobile deployment. Image processing functions (edge detection, face cropping) are implemented using OpenCV Android SDK 3.4.3. The mobile device is connected to the server via a TCP connection.

Cloud. We implement the cloud side of EagleEye on a desktop PC running on Ubuntu 16.04 OS, equipped with Intel Core i7-8700 3.2 GHz CPU and an NVIDIA RTX 2080 Ti GPU (11 GB RAM). We implement most of the cloud-side functions in Python 3.5.2 and utilize Numba [138], a Just-In-Time (JIT) compiler for Python, to accelerate the performance comparable to C/C++. ICN and ResNet50-based ArcFace [18] are implemented using TensorFlow 1.12.0.

4.8 Evaluation

4.8.1 Experiment Setup

DNN Training. We train our face detector on WIDER Face [139] train dataset. Also, we train our face recognizers (both the light and heavy models) on MS1M [122] dataset. ICN is trained on FFHQ dataset [140]. As FFHQ dataset does not contain face landmark labels, we employ state-of-the-art network [141] to estimate face landmarks and use them as ground truth labels.

Datasets. We evaluate EagleEye with two different datasets: *single faces* and *crowded scenes*. For single faces, we collect 50 identities in VGGFace2 [123] testset, with 50 samples per each identity. For the scenes, we use in-the-wild images (mostly containing faces of a single ethnicity group) collected and classified depending on crowdedness (i.e., *Low*, *Medium*, and *High*) as described in Chapter 4.3.1 (examples are shown in Figure 4.16). The detailed composition of the faces in the scene dataset are summarized in Table 4.3. We also categorize the dataset depending on whether the target is present or not. Furthermore, we also collect scene images from WIDER Face [139] test dataset, which contains diverse ethnicity groups (15 images per each crowdedness

category).

Evaluation Protocols and Metrics. We evaluate the performance of EagleEye with the following evaluation protocols and metrics:

• **Latency:** the time interval between the start and the end of the pipeline execution, measured on mobile.

• **Equal Error Rate (EER):** the value in the ROC curve where the false acceptance and false rejection rates are identical.

• **True Positive (TP) & False Positive (FP):** the rate in which the test faces are correctly/wrongly accepted as the target, respectively, given a fixed threshold.

• **Top-K Accuracy:** the percentage of images in which the distance between the target face and the probe is within the top K-th among all faces in the scene (applies for scenes with the target present). This can also be interpreted as recall for a single target.

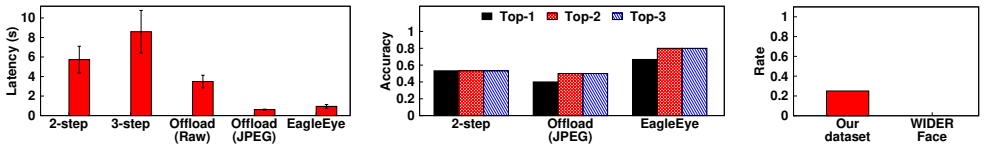
• **False Alarm:** the percentage of images in which the system falsely detects that the target is present in the scene (applies for scenes with the target absent).

Comparison Schemes. We compare the performance of EagleEye with the following comparison schemes:

• **2-step baseline** runs the conventional 2-step identification pipeline (MobileNetV1-based RetinaFace and ResNet50-based ArcFace) all on the mobile sequentially.

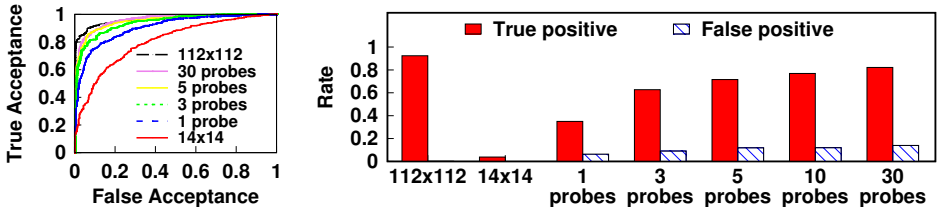
• **3-step baseline** runs our proposed 3-step identification pipeline (MobileNetV1-based RetinaFace, ICN, and ResNet50-based ArcFace) all on the mobile sequentially.

• **Full offload** fully offloads the image to the cloud over LTE and runs the 3-step identification pipeline. The image is sent either raw or after JPEG compression. Note: we run this experiment under a normal LTE performance ($\approx$11 Mbps), and it is likely that the performance of full offloading could be worse than what we report in crowded outdoor environments.



(a) End-to-end latency. (b) Top-K Accuracy. (c) False Alarm increase.

Figure 4.17: EagleEye performance overview.



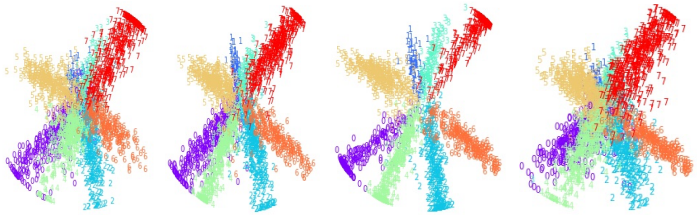
(a) Ideal case. (b) Our scenario.

Figure 4.18: Performance of Identity Clarification Network.

4.8.2 Performance Overview

We first evaluate the overall performance of EagleEye compared with alternatives for *High* scenes. Figure 4.17 shows the results. Firstly, as shown in Figure 4.17(a), EagleEye outperforms the latency of the 3-step baseline by 9.07× (with only 108 KBytes of data offloaded to the cloud). Also, it shows the highest Top-K accuracy (80% of Top-2 accuracy vs. 53% for the 2-step baseline) at the reasonable increase of false alarms (Figure 4.17(b) and (c)). A reason for the increase of the false alarm is that our dataset contains the faces of the same ethnicity group, increasing the chance of similar-looking identities with the target. For the WIDER Face dataset which contains more diverse ethnicity groups, we did not observe any false alarm increase. Note that the accuracy and false alarms are better with *Medium* and *Low* scenes, as shown in Figure 4.25.

Interestingly, while fully offloading JPEG-compressed images achieves the smallest latency, we observe that its Top-2 accuracy drops to 50% as shown in Figure 4.17(b), as compression artifacts hinder reconstruction performance of ICN and recognition network. We could apply video compression (e.g., H.264) to minimize latency more, but it would further degrade performance as it adopts motion vector-based inter-frame encoding, incurring additional distortion in the faces. As compression artifact reduc-



(a) 14×14. (b) Baseline ICN. (c) Ideally fine-tuned. (d) Fine-tuned to identity #6 (orange).

Figure 4.19: Feature map visualization for ICN.

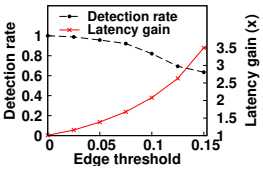
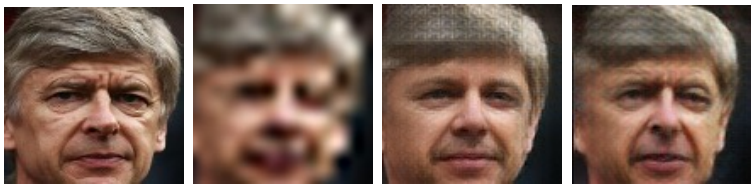


Figure 4.20: Background filtering.

tion is a challenging problem, recent attempts have been made to design specialized DNNs for it [142, 143]. Thus, we conjecture that solving this issue will not be trivial and leave detailed investigation as future work.

4.8.3 Identity Clarification Network

We evaluate the performance of ICN with a varying number of probes used for Identity-Specific Fine-Tuning. Figure 4.18 shows the results for (a) ideal cases (ICN trained for individual faces) and (b) our scenarios (ICN trained with a target identity), respectively. For the ideal case, ICN recovers the accuracy of 14×14 faces similar to 112×112 with about 5 probes only. For our scenarios, as the number of probes increases, ICN injects more facial details of the target to the input LR face, significantly increasing the chance to identify the target with a relatively small increase in the FP. Figure 4.18(b) shows that the gain in TP (78%) outweighs that of FP (14%). We further analyze the reasons for accuracy improvement using a simple example with the 8 identities (the same setting as in Chapter 4.3.2). From the 14×14 LR faces whose features severely overlap with each other (Figure 4.19(a)), the baseline ICN (without fine-tuning) clusters each identity's features more tightly, but some overlapping regions still remain (Figure 4.19(b)). When enhancing each LR face with ICN fine-tuned with corresponding probes, we observe each feature cluster is separated even more clearly (Figure 4.19(c)). In the case of applying ICN fine-tuned to target identity #6 (orange samples), Fig-



(a) 112×112. (b) 14×14. (c) Baseline. (d) Fine-tuned

Figure 4.21: Reconstruction example of ICN.

ure 4.19(d) shows that the samples corresponding to the target are grouped to form a tight cluster. While other identity groups are pulled towards the target, the cases where the pulled samples overlap with those of the target (false positive) are not dominant.

Finally, Figure 4.21 shows the face reconstruction examples of ICN. Baseline ICN reconstructs a face quite similar to the ground truth but lacks some fine attributes (e.g., wrinkles) in the ground truth face. Identity-Specific Fine-tuning enables the ICN to instill such details in the reconstructed face, thus enabling accurate recognition.

4.8.4 Content-Adaptive Parallel Execution

4.8.4.1 Edge-Based Background Filtering

Next, we evaluate the performance of our Edge-Based Background Filtering method. Figure 4.20 shows the detection rate and latency gain as we increase the edge intensity threshold. Higher threshold results in higher latency gain, but at the cost of loss in detection rate. We observe threshold between 0.05 and 0.08 balances the tradeoff, and we empirically set it as 0.08 which achieves 1.76× latency gain with 8.7% loss in detection rate. Figure 4.22 shows an example of image blocks being filtered for different thresholds (covered in black in Figure 4.22(c)–(e)). With a higher threshold, blocks containing large faces starts to get ruled out. The tradeoff can be more aggressively made if our system can only focus on identifying distant, small faces while relying on users to recognize large, closer faces.

(a) Raw frame. (b) Detected edges. (c) 59% left. (d) 30% left. (e) 8% left.

Figure 4.22: Example operation of Edge-Based Background Filtering.

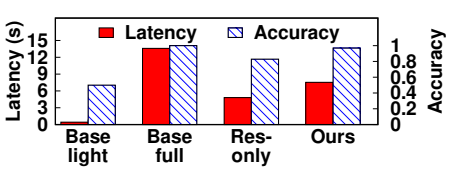
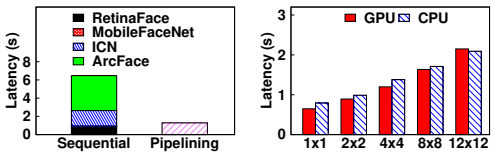


Figure 4.23: Performance of Variation-Adaptive Face Recognition.



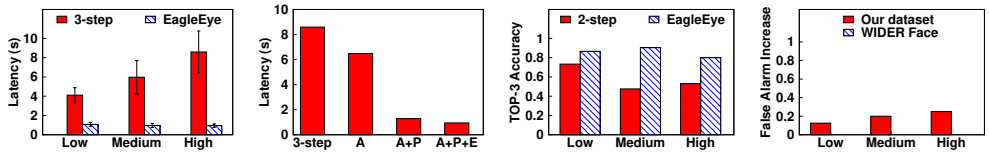
(a) End-to-end latency. (b) Detection latency.

Figure 4.24: Spatial Pipelining performance.

4.8.4.2 Variation-Adaptive Face Recognition

To evaluate the effectiveness of Variation-Adaptive Face Recognition, we synthesize a group of faces, which contains 10 samples per each case classified in Figure 4.14. We compare our technique (adapting the recognition pipeline based on pose and resolution) with the following baselines: (i) running a lightweight recognizer (MobileFaceNet [136]) on all faces (denoted as *Base light*), (ii) running ICN and a heavy recognizer (ResNet50-based ArcFace [18]) on all faces (denoted as *Base full*), (iii) adaptively applying the lightweight and heavy recognizers based on the resolution only (denoted as *Res-only*). We did not apply our parallel and pipelined execution for this experiment so that only the relative comparisons are meaningful.

Figure 4.23 shows that our approach achieves comparable accuracy with *Base full*, while reducing the latency by 1.80×. On the contrary, *Base light* and *Base full* suffer from low accuracy and significantly high latency, respectively. The *Res-only* yields fairly high accuracy gain with small latency overhead, but the accuracy remains lower than *Base full* as large profile faces processed by light MobileFaceNet results in inaccurate decisions.



(a) Latency. (b) Latency breakdown. (c) Top-3 Accuracy. (d) False alarm increase.

Figure 4.25: End-to-end latency for varying crowdedness.

4.8.4.3 Spatial Pipelining

Figure 4.24(a) shows the performance of Spatial Pipelining on *High* scenes. Our pipelining yields 5.03× acceleration compared to the baseline that runs face detection and processes faces with Variation-Adaptive Face Recognition sequentially using the mobile GPU (denoted as *Sequential*).

We further analyze the effect of the number of blocks to parallelize. Figure 4.24(b) shows the latency of face detector with varying number of blocks. We need to divide the image in an overlapping manner to prevent faces split across blocks, which increases computational overhead due to repetitive face detection on the overlapping regions. Thus, the larger the number of blocks, the higher the latency overhead. Considering the tradeoff between such cost and gain for parallelism, we divide the image into 4×4 blocks by default.

4.8.5 Performance for Varying Crowdedness

Figure 4.25(a) shows the end-to-end latency comparison of 3-step baseline and EagleEye. The latency of EagleEye remains similar regardless of crowdedness, mainly because we pipeline and parallelize the execution on mobile and cloud. However, the latency of 3-step increases with more crowded scenes since recognition latency increases proportionally to the number of faces. Accordingly, we conjecture that the latency gain will be greater as crowdedness increases even more. Furthermore, current bottleneck remains at the face detection stage, and we expect that the latency will be further reduced as face detectors become more optimized.

Figure 4.25(b) shows the latency breakdown on *High* scenes for gradually adding

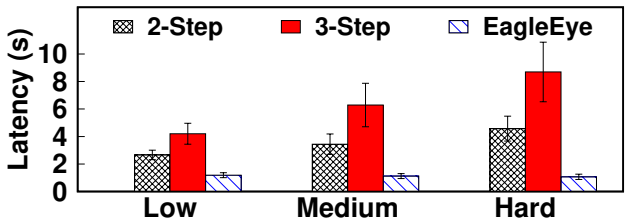


Figure 4.26: Latency evaluation on Google Pixel 3 XL.

on the components of EagleEye: Variation-Adaptive Face Recognition (A), Spatial Pipelining (P), and Edge-Based Background Filtering (E). Combining each component yields a synergetic gain, achieving 9.07× acceleration compared to the 3-step baseline.

Finally, Figure 4.25(c) shows the Top-3 accuracy and false alarm increase of EagleEye compared to the 2-step baseline. Overall, EagleEye yields 27.6% accuracy gain, with accuracy above 80% even for *High* scenes. Figure 4.25(d) shows that at the cost of such accuracy gain, EagleEye results in 19.1% increased false alarm. Such increase is due mainly to the fact that our dataset contains the people with the same ethnicity, and we observe no increase in false alarm in case of WIDER Face dataset.

4.8.6 Performance on Other Mobile Devices

Lastly, we evaluate the end-to-end latency on Google Pixel 3 XL to validate the performance of EagleEye on other mobile devices. The inference times of MobileNetV1-based RetinaFace, ICN, ResNet50-based ArcFace, and MobileFaceNet are 918, 225, 193, 18 ms, respectively. Figure 4.26 shows that the latency performance of EagleEye and gain compared to 3-step baseline are similar (8.14× for *Hard* scenes) to previous results, indicating that EagleEye shows consistent performance on other devices.

Chapter 5

Pendulum: Network-Compute Joint Scheduling for Scalable Live Video Analytics

5.1 Introduction

In this Chapter, we design Pendulum, an end-to-end system for robust and efficient cloud offloading-based live video analytics. Live video analytics pipeline is composed of two stages: (i) *Network stage*: video streaming over network, and (ii) *Compute stage*: real-time Deep Neural Network (DNN) inference on edge/cloud server. The key to achieving high accuracy and throughput is to flexibly adapt the pipeline depending on dynamic workload and resource availability, affected by scene complexity [1, 32, 38], network bandwidth [83], and server contention [10]. Here, new important challenges arise from complex patterns of alternating resource bottlenecks across the network and compute stages. For an example of face identification application, we observe the video bitrate (changed by channel status and contention) and required DNN inferences (changed by the number and pose of faces in the scene) have a weak Pearson correlation coefficient of -0.15 (Chapter 2.2.3), causing frequent alternation between bandwidth and GPU bottlenecks, necessitating the need for joint adaptation of the two stages.

Prior live video analytics systems support adaptation to fluctuating resources, but they are limited to *single-stage scheduling* (network [4, 73, 76, 83, 85, 89] or compute [10, 38]).[1] Their goal is to minimize the target resource usage (either network or compute) with minimal accuracy drop, assuming that the other resource is sufficiently provisioned. They have severe limitations in alternating resource bottleneck scenarios (Chapter 5.2.2). First, they suffer from throughput/accuracy degradation when the other stage is bottlenecked. Also, they likely require sufficient provisioning of untargeted resources, causing unnecessary costs and resource wastage.

To overcome the limitations of prior works, Pendulum features a *network-compute joint scheduling* mechanism (Chapter 5.3.1). When a stage is bottlenecked, we not only reduce the bottlenecked resource usage but also leverage the surplus resource in the remaining stage for accuracy compensation. It widens the adaptation space (e.g., further reducing the minimum bitrate satisfying the accuracy requirement), achieving a more stable and higher throughput/accuracy over single-stage scheduling. It also helps utilize network and compute resources in a balanced way, preventing unnecessary resource over-provisioning (for instance, a cloudlet [144] scenario in Chapter 5.2.1). Joint scheduling is generalizable and can be integrated into many prior single-stage systems (Chapter 5.3.3).

To design our mechanism, we newly discover the tradeoff relationship between video bitrate and DNN complexity (i.e., heavier DNN can compensate the accuracy drop from low bitrate and vice versa). Heavier DNNs (with more layers and filters) are capable of capturing more complex features [33], thus achieving high accuracy in low-bitrate videos. A key factor is the receptive field size, proportional to the number of layers [145, 146]; larger receptive field enables a contextual understanding of the scene (e.g., detects the object not just by looking at it but also the objects around it). We

[1] We refer to the single-stage scheduling as *controlling the network or compute resources independently*, albeit it indirectly affects the remaining stage (e.g., adapting video frame rate or resolution changes DNN inference latency or number of inferences).

quantitatively verify this over two tasks (object detection and semantic segmentation) and various DNNs (Chapter 5.3.2).

Joint scheduling incurs non-trivial challenges.

• **Scheduling Knob Selection.** Joint scheduling requires a careful selection of control knobs considering inter-dependency between network and compute stages. Prior single-stage systems consider different knobs: (i) frame rate, resolution, quantization for network stage, and (ii) DNN backbone for feature extraction (e.g., heavy ResNet, lightweight MobileNet), weight quantization, and input enhancement (e.g., super-resolution) for compute stage. However, it is non-trivial to control them concurrently as controlling a single knob affects not just the corresponding stage but also the other, undesirably altering the accuracy and resource demand. In addition, some knobs require considerable preparation effort (e.g., model re-training), limiting their scalability.

• **Lightweight Profiling.** Joint scheduling requires continuous tracking of three-way tradeoffs among network, compute, and accuracy, which imposes a severe challenge to the scheduling system design. The main difficulty comes from dynamic changes in the tradeoff patterns depending on the video content (e.g., object speed, lighting condition) (Chapter 5.5.2). Due to the black-box nature of DNN inference, it is inevitable to profile them by running the DNNs over 2D (bitrate, DNN) configuration space. For example, a full search of 5 bitrates$\times$7 DNNs takes 3.4s on an RTX 2080 Ti GPU.

• **Scheduling Policies and Algorithms.** New scheduling policies and algorithms are necessary to fully utilize the joint scheduling mechanism. In particular, allocating both network and compute resources across multiple users to (i) resolve resource contention and (ii) minimize total cost is a multi-dimensional knapsack scheduling problem, known as NP-hard (Chapter 5.6.2).

We address challenges with following components.

• **Joint Scheduling Mechanism (Chapter 5.5).** We first identify (Quantization Parameter (QP), backbone) pair as the most suitable knobs for joint scheduling (Chap-

ter 5.5.1), considering (i) control independency (i.e., affecting the target stage only), and (ii) preparation overhead. First, QP is easy to adjust in widely-used video codecs and it controls bitrate independently of the compute stage; reducing frame rate or resolution implicitly reduces the compute resource usage (through the number of DNN inferences and latency), complicating the control of compute resources in joint scheduling. Second, backbone scaling involves a minimum task- and model-specific efforts (e.g., loss function design) compared to others.

Next, we develop a resource-efficient tradeoff profiler with two techniques (Chapter 5.5.2). First, we utilize an ensemble of lightweight features that efficiently captures scene change (e.g., object motion, lighting condition change) to trigger profiling only when the tradeoff curve changes. Second, we develop a novel *weighted multi-knob accuracy interpolation* technique to efficiently profile the 2D config space by measuring the accuracy of a few (QP, backbone) pairs and estimating the accuracy of the rest. Our key insight is that the accuracy gain from increasing one knob value saturates as the remaining knob value becomes higher. We minimize the profiling overhead by up to 93.9% (1.5% of serving cost) and 40% compared to fixed-period full search profiling and state-of-the-art Chameleon [38] with negligible accuracy drop.

• **Multi-User Joint Scheduler (Chapter 5.6).** We design an *Iterative Max Cost Gradient algorithm* to obtain an approximate solution with minimal computational overhead (Chapter 5.6.3). Specifically, it (i) finds the user-wise optimal configs and (ii) iteratively adjusts each user's config with the *maximum cost gradient* (i.e., adjusting the user with the maximum expected decrease in bottleneck resource usage by increasing a unit usage of the other resource). Our algorithm obtains a near-optimal solution with efficient $O(M \cdot N)$ time complexity.

Our key contributions are summarized as follows:

• To our knowledge, Pendulum is the first live video analytics system with network-compute joint scheduling.

• We design an end-to-end system for joint scheduling, composed of (i) an efficient

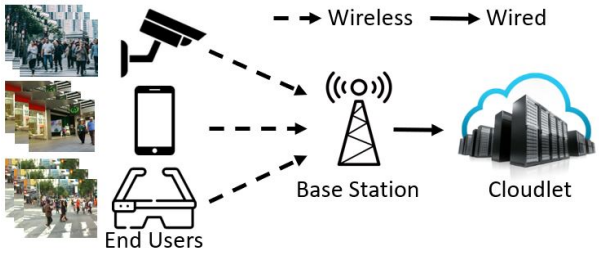


Figure 5.1: Scenario: cloudlet-based city monitoring.

and scalable knob control mechanism, (ii) a lightweight tradeoff profiler, and (iii) a multi-user joint resource scheduler.

- We conduct extensive evaluation on various datasets and state-of-the-art DNNs. Pendulum achieves up to 0.64 mIoU gain (from 0.17 to 0.81) and 1.29× higher throughput compared to state-of-the-art single-stage scheduling systems.

5.2 Motivation

5.2.1 Target Scenarios and System Goals

Scenario. We aim to apply our joint scheduling mechanism for a wide range of video analytics applications, especially in Multi-Access Edge Computing (MEC) scenarios illustrated in Figure 5.1. For instance, a police agency deploys CCTVs and officers with AR glasses/smartphones in a city for monitoring [32] (e.g., criminal chasing, jaywalking detection, traffic monitoring).[2] It connects the cameras through a shared radio access network (e.g., private 5G [147, 148]) for video streaming, and deploys a cloudlet (shared edge GPU server) [144] next to the base station for processing.

System Goals. We aim to achieve the following goals.

- **Real-time Processing.** We aim at end-to-end scheduling across end users and analysis server for real-time video processing (e.g., 30 fps throughput) while satisfying the app-specified accuracy requirement.

[2]We are collaborating with multiple Asian companies for deployment of our system for face-recognition-based real-time airport check-in and other MEC scenarios.

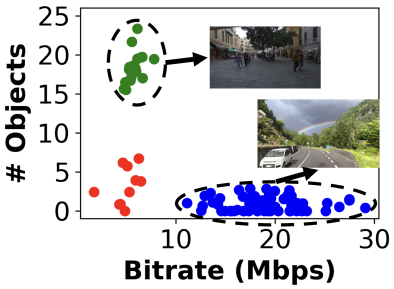


Figure 5.2: Bitrate and workload (# objects) for different scenes.

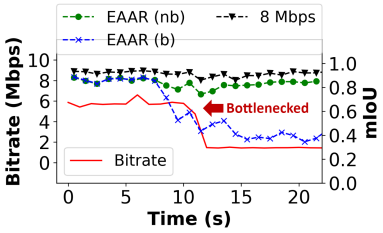


Figure 5.3: Bitrate and mIoU of network-only scheduling (b: bottleneck, nb: no bottleneck).

• **Soft Real-Time Latency**. We aim at soft real-time latency (e.g., <100 ms) so that the analysis result is delivered to users promptly for further actions (e.g., bounding box displayed on screen for officers to confirm).

• **Cost Minimization**. While achieving real-time throughput, our goal is to minimize the system operational cost. Specifically, network and compute costs vary depending on service providers (e.g., 5G 1 Mbps streaming: $0.36 [149] to $0.54 [150] per hour, V100 GPU: $0.74 [151] to $0.91 [152] per hour). We aim to efficiently schedule network/compute resource usage to avoid bottlenecks while also avoiding over-provisioning.

• **Minimal App Modifications**. For generality, we aim at minimal modifications on the app's analytics pipeline (e.g., video codec, DNN model). Especially, we aim to avoid app/task-specific model re-training or additional model preparation.

5.2.2 Limitations of Single-Stage Scheduling

Limited Adaptation Space. It is challenging to simultaneously achieve high accuracy and throughput by controlling only a single stage of the pipeline. Figure 5.3 shows an example operation timeline (video encoding bitrate and accuracy (mean Intersection over Union, mIoU)) of EAAR [4] (network-only scheduling with dynamic RoI encoding and motion vector-based frame skipping) on a MOT17 [31]-04 video. EAAR effectively optimizes the video bitrate to ≈5.68 Mbps with minimal mIoU drop com-

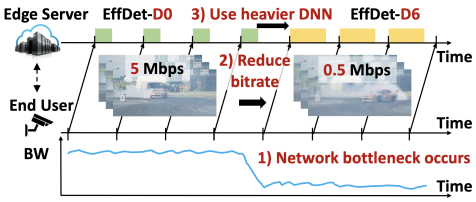


Figure 5.4: Joint scheduling example for network bottleneck scenario.

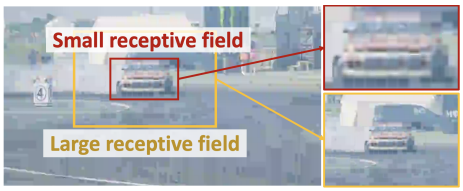


Figure 5.5: Illustration of the impact of the receptive field.

pared to 8 Mbps encoding (EAAR (nb)). However, when the network bandwidth drops below its capability, throughput/accuracy drop is inevitable. For example, when the bandwidth is throttled to 2 Mbps at t=10s (using tc [153]), reducing the encoding bitrate accordingly incurs a significant mIoU drop (EAAR (b)).

Difficulty of Resource Planning. It is also challenging to plan the right amount of resource usage for the remaining stage to avoid the resource bottleneck or wastage from under/over-provisioning. For example, with the same setting as in Figure 2.2(b), using 6 GPUs incurs a 22% latency violation rate, while using 9 GPUs incurs 35% resource wastage. Elastic resource provisioning incurs higher operational costs or may not even be possible for edge server scenarios.

5.3 Approach

5.3.1 Key Idea: Joint Scheduling

Our approach is to jointly schedule the network and compute resources to achieve both high accuracy and throughput. Figure 5.6 shows a comparison of single-stage and joint scheduling. Network-only scheduling (which only controls the video bitrate with a fixed GPU utilization) can only reduce the bitrate up to b_1 without violating the app accuracy requirement (ACC_{th}). If network bandwidth drops below b_1, network bottleneck occurs, dropping the throughput. However, joint scheduling utilizes additional compute resources (up to the available budget) to further reduce the bitrate to as low as b_2, thus being more robust to bandwidth drop. Figure 5.4 shows an exam-

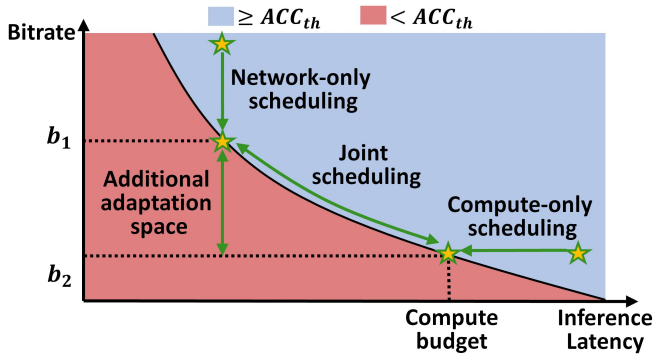


Figure 5.6: Single-stage vs. joint scheduling comparison.

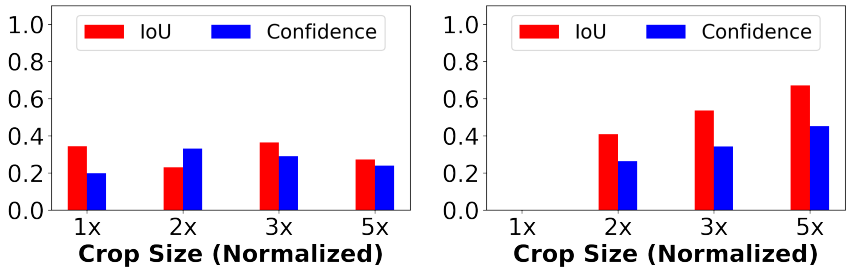
ple operation timeline for the network bottleneck scenario. At the start when network bandwidth is sufficient, we stream the video at a high bitrate (e.g., 5 Mbps) and run the EfficientDet-D0 [106] object detector. When the bandwidth drops, we reduce the bitrate accordingly to 0.5 Mbps so that the video can be streamed in real-time. To compensate the accuracy drop from reduced video quality, we leverage additional compute resources by running the heavier EfficientDet-D6. Similarly, we reduce the DNN complexity and increase the bitrate in the case of compute bottleneck.

5.3.2 Why is Joint Scheduling Possible?

We analyze why there is a tradeoff relationship between video bitrate and DNN complexity. Specifically, *"Why can heavier backbone compensate the accuracy drop from low bitrate (and vice versa)?"* Heavy DNN with a large number of layers and filters can capture diverse complex features [33]. Especially, large receptive field size (i.e., how large an area a DNN analyzes to detect objects) [145] helps achieve high accuracy in low-bitrate videos. The receptive field of a DNN is proportional to the number of layers (e.g., $O(\sqrt{N})$ [146]); a DNN with a large receptive field size can analyze the scene context with a wider view. Figure 5.5 shows an example. For low-bitrate videos, it is highly challenging to recognize a car just by looking at it. However, with a large receptive field, we can analyze the scene context (e.g., road lane, nearby car) and easily classify the blurry object as a car.

(a) 1× GT box. (b) 2× GT box. (c) 3× GT box.

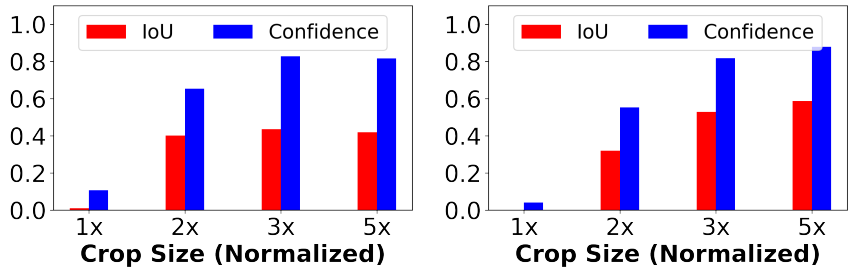
Figure 5.7: Example detection results (box and confidence) for different crop sizes.



(a) EfficientDet-D0. (b) EfficientDet-D6.

Figure 5.8: Detection accuracy in low-bitrate video.

We quantitatively verify this on two widely-used tasks (object detection and semantic segmentation) using DAVIS17 [154] video. First, we run EfficientDet [106] detector with two backbones (D0 and D6 with 49 and 134 layers) on image patches cropped from the low-bitrate video around the ground truth bounding box (to emulate the effect of the receptive field by restricting the maximum area a DNN can analyze) (Figure 5.7). Figure 5.8(a) shows that EfficientDet-D0's accuracy remains low, regardless of the crop size. However, Figure 5.8(b) shows that EfficientDet-D6's accuracy constantly improves with larger crop size, indicating that it leverages its large receptive field for accurate detection. We observe a similar trend for the semantic segmentation: Figure 5.9 shows the accuracy for the same video using the FPN [155] with ResNet-18 and ResNet-101 backbones. Our observation is also well aligned with the findings in the ML field showing that the models with more layers and filters achieve higher accuracy and can compensate for lower input resolution [33].



(a) FPN-ResNet-18.

(b) FPN-ResNet-101.

Figure 5.9: Segmentation accuracy in low-bitrate video.

Table 5.1: Applicability of joint scheduling in state-of-the-art single-stage scheduling systems.

System	Mechanism	Control knob	Possible joint scheduling knob
DDS [85]	Two-path streaming (probe + feedback)	Low/high video/RoI bitrate	DNN complexity
EAAR [4]	RoI encoding (object/background)		
AWStream [83]	Content-aware bitrate adaptation	Video bitrate	
Glimpse [73] Reducto [76]	Frame filtering	Filter threshold	Tracker complexity
Chameleon [83]	Content-aware DNN adaptation	DNN complexity	Video bitrate
SPINN [10]	DNN inference early exit	Early exit layer	

5.3.3 Generality of Joint Scheduling

Joint scheduling is orthogonal to prior single-stage scheduling systems and other optimization techniques (e.g., RoI encoding and frame filtering). In this light, it can be generally integrated into many prior systems. Table 5.1 shows the control knob and mechanisms of state-of-the-art single-stage scheduling systems along with the possible joint scheduling knobs we suggest. For example, DDS [85] controls two bitrates to which the probe and feedback streams are encoded (e.g., 2 and 6 Mbps). In case of the network bottleneck, along with adjusting the two bitrates (e.g. to 1 and 3 Mbps),

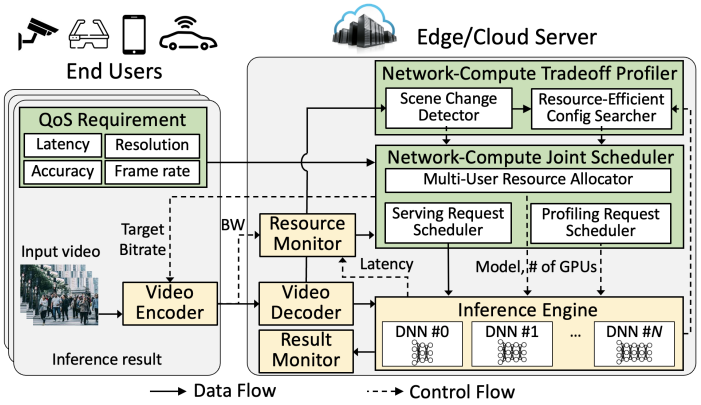


Figure 5.10: Pendulum system architecture.

we can use a heavier DNN to compensate for the accuracy drop. We also quantitatively show the benefits of joint scheduling on existing systems (DDS and EAAR) in Chapter 5.7.3.

5.4 System Overview

Figure 5.10 shows the system architecture of Pendulum.

Data Flow. End users specify their app QoS requirements (e.g., latency, accuracy) and stream their live video encoded at the target bitrate specified by the server. Upon receiving and decoding the frames, the server runs the target DNN inference and aggregates the results.

Control Flow. Server monitors resource availability (users' network bandwidths, DNN inference latencies, and GPU utilization), analyzes the network-compute tradeoff, and schedules the resources across multiple users. Specifically, network and compute resources are joint scheduled by controlling the (QP, backbone) pair (Chapter 5.5.1). To efficiently track the tradeoff, the profiler (i) dynamically triggers profiling only under significant scene change and (ii) minimizes the number of configs to search (Chapter 5.5.2). Finally, the multi-user joint scheduler finds a joint resource allocation to minimize the overall resource cost and resolve contention (Chapter 5.6).

5.5 Joint Scheduling Mechanism

Pendulum's joint scheduling mechanism is composed of three components: (i) selection of control knobs for video bitrate and DNN inference latency (Chapter 5.5.1), (ii) a runtime resource-efficient profiler that tracks the Pareto-optimal tradeoff configs with minimal overhead (Chapter 5.5.2), and (iii) a resource availability estimator to narrow down candidate configs that satisfy the accuracy requirement and resource budget for runtime scheduling (Chapter 5.5.3). The joint scheduler (Chapter 5.6) is built on top of these components.

5.5.1 Joint Scheduling Knob Selection

Several knobs have been studied in prior single-stage systems. However, it is non-trivial to control them jointly, as controlling one knob may affect not just the corresponding stage but the other as well, unexpectedly altering the accuracy and resource demands. We identify *(QP, backbone)* pair as the suitable joint scheduling knobs with our two design criteria: (i) *Control Independency* (i.e., independently affect the network and compute stages), and (ii) *Knob Preparation Overhead* (i.e., require minimal model re-training or additional model preparation).

5.5.1.1 Knobs for Video Bitrate

Bitrate is determined by the following three knobs:

$$Bitrate \propto Frame\,rate \times Resolution \times Quantization. \tag{5.1}$$

Several works designed efficient knob control methods for bitrate optimization (e.g., frame rate [73, 76–80], resolution [38, 75, 156], quantization [4], or a combination of all [83]). While all knobs can be used for joint scheduling, we choose pixel quantization as it does not affect the resource usage of the compute stage and incurs the smallest overhead on the compute stage.

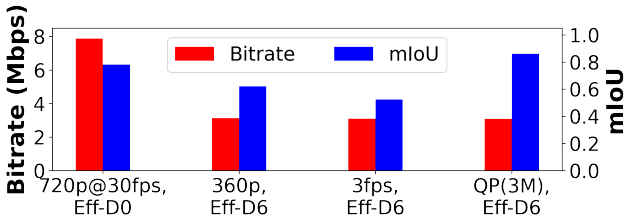


Figure 5.11: Joint scheduling performance comparison for different video bitrate knobs (resolution vs. fps vs. QP).

• **Frame Rate.** Frame skipping and tracking-based interpolation enable joint scheduling. However, it is non-trivial to prepare an accurate tracking model and the tracking accuracy tends to drop significantly with fast-changing scenes [157].

• **Resolution.** Heavier DNNs accurately detects small objects in low-resolution videos, making the resolution a plausible knob. The resolution, however, makes the joint scheduling more complicated since it implicitly affects the computing stage by changing the inference latency and accuracy even when the same DNN is used.

• **Pixel Quantization.** Quantization enables joint scheduling as analyzed in Chapter 5.3.2. Quantization controls bitrate without affecting the compute stage (as input size remains the same), as opposed to the other knobs. We control the Quantization Parameter (QP) commonly exposed in video codecs (e.g., H.264/265).

Quantitative Comparison. Figure 5.11 compares the joint scheduling performance of different bitrate knobs on BDD dataset [158] (we observe a similar trend on other datasets as well). When trying to reduce a 720p@30fps video's bitrate from 8 Mbps (leftmost bars) to 3 Mbps and compensate the accuracy by increasing the backbone from EfficientDet-D0 to D6, QP control (rightmost bars) achieves 0.24 and 0.34 higher mIoU than resolution and fps, respectively. Resolution and fps result in sub-optimal bitrate-accuracy tradeoff, as reducing them reduces the inference latency or the number of inferences, negating the effect of joint scheduling.

5.5.1.2 Knobs for DNN Inference

Inference latency is determined by the following knobs:

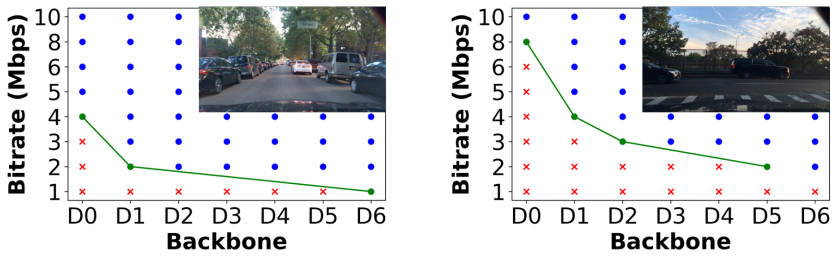
$$Latency \propto InputEnhancement + Backbone \times Quantization. \tag{5.2}$$

Input enhancement is the preprocessing step (e.g., super-resolution, compression artifact removal) to improve the frame quality before DNN inference. Quantization is applied to the DNN weights (e.g., float16, int8) to reduce the computation at the cost of a small accuracy drop. Backbone refers to the feature extractor of a DNN: it is characterized by the architecture, number of layers, and channels (e.g., ResNet-50, MobileNet-0.25). While these knobs can be adopted for joint scheduling, We choose the backbone scaling as the most suitable knob with the consideration of scalability across multiple tasks.

• **Input Enhancement.** Utilizing input enhancement as a scheduling knob limits the scalability to different tasks, as it requires (i) task-specific model re-design [3], and (ii) frequent online fine-tuning (e.g., super-resolution performance drops on unseen contents [88]).

• **Weight Quantization.** Quantization can serve as a joint scheduling knob but are practically limited at the moment as it requires task-specific tuning efforts (e.g., value clipping [159], loss function design [160]) and hardware support. Also, it currently offers only a limited number of config choices (e.g., float32, float16, int8).

• **Backbone.** DNNs for various tasks commonly support diverse backbones while many backbones with accuracy-latency tradeoff are readily available (e.g., ResNet-18/34/50/101/152, EfficientNet-B0 to B7) [17, 30, 33, 106, 161]. With the minimal model preparation cost and task supportability, we adopt backbone scaling as the control knob for DNN inference.



(a) Static scene, good lighting.

(b) Dynamic scene, poor lighting.

Figure 5.12: Tradeoff curves for different scenes. Blue/red points: configs above/below the accuracy requirement, green curve: Pareto-optimal configs.

5.5.2 Network-Compute Tradeoff Profiler

5.5.2.1 Goals and Challenges

Goal. The profiler's goal is to analyze the Pareto-optimal tradeoff configs of the user's live video stream. We define that a (bitrate, inference latency) config (b, t) is Pareto-optimal if (b, t) satisfies the accuracy constraint and no other (b', t) and (b, t') exists s.t. $b' < b$, $t' < t$.

Challenges. Profiling incurs two key challenges.

• **Fast-Changing Tradeoff Curve.** The tradeoff curve should be constantly profiled at runtime, as it frequently changes over time depending on the scene content. Figure 5.12 shows an example on a BDD [158] video for 8 bitrates and 7 EfficientDet [106] backbones. In a static scene (the car is not moving) with good lighting conditions (Figure 5.12(a)), it is easy to detect objects from a low-bitrate video with lightweight backbones. Thus, the Pareto-optimal configs have small b, t values. In contrast, the values become larger for dynamic scene (the car is moving) with poor lighting conditions (Figure 5.12(b)). The changes occur fast; for BDD [158] dashcam videos, the cost-optimal YOLOv5 [30] backbone for 4 Mbps encoding bitrate changes every 4.4s on average.

• **High Profiling Overhead.** Each tradeoff profiling event requires a 2D space search (bitrate and backbone), involving repetitive DNN inference over a frame en-

coded in multiple bitrates. Frequent profiling (e.g., fixed-interval, periodic [38]) incurs significant resource overhead. For example, exploring all search space composed of 7 EfficientDet backbones (D0-D6) and 5 QPs over a single frame takes 3.4 seconds on RTX 2080 Ti GPU. Triggering this every 2 seconds incurs ≈50% overhead compared to 30 fps inference serving).

5.5.2.2 Dynamic Profiler Overview

Lightweight scene change detector (Chapter 5.5.2) uses lightweight feature ensemble to trigger profiling only when significant scene change is detected and the Pareto-optimal configs are expected to have changed. For each triggered frame, *Intra-frame profiler* (Chapter 5.5.2) uses a novel *weighted multi-knob accuracy interpolation* to avoid exhaustive full search of the 2D config space.

5.5.2.3 Lightweight Scene Change Detector

Two categories of features can be utilized for scene change detection: heavy-but-accurate high-level (e.g., SIFT [162], SURF [163]) and lightweight-but-noisy low-level. We adopt a lightweight feature ensemble approach for lightweight scene change detection. We choose the following features that complementarily capture diverse aspects of scene change (camera and object motion, lighting condition). They have a high correlation with tradeoff curve changes, allowing an effective skipping of unnecessary profiling (Chapter 5.7.6).

- **Camera Motion: Average Motion Vector [4].** We detect scene change if the sum of average motion vector magnitudes (obtained from video codec without overhead) from reference to current frames is over th_1.
- **Object Motion: Bounding Box Drift.** We detect scene change if the mIoU between the serving inference results of reference and current frames is below th_2.

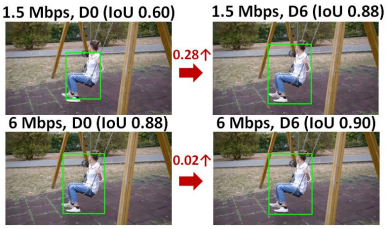


Figure 5.13: Motivation for weighted multi-knob accuracy interpolation.

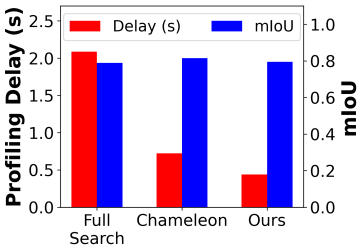


Figure 5.14: Profiler performance comparison.

• **Lighting Condition Change: Color Histogram Difference [14].** We detect scene change if the chi-square distance of the color histograms between the reference and current frames is over th_3.

We detect scene change if two or more conditions are satisfied. We use MOT [31] and BDD [158] datasets to empirically fit the thresholds th_1, th_2, th_3 as 0.5.

5.5.2.4 Intra-Frame Profiler

To avoid exhaustive full search of multi-dimensional profiling space, Chameleon [38] uses a linear multi-knob accuracy interpolation technique. It is based on knob independence assumption: accuracy gain from increasing one knob value is independent of the remaining knobs. Thus, it profiles each axis with fixed remaining-stage values and interpolates the rest. We improve this as follows.

Weighted Multi-Knob Accuracy Interpolation. Figure 5.13 illustrates our key insight: IoU increase when increasing the backbone from D0 to D6 saturates as the bitrate is higher (as D0 backbone already finds the object accurately). To take this into account, weighted multi-knob interpolation works as follows. Assume that we profiled the backbone axis with bitrate fixed to b_0. We interpolate the accuracy gain from increasing the backbone from D_j to $D_{j+\Delta}$ with bitrate b_i as

$$\begin{aligned} ACC(b_i, D_{j+\Delta}) - ACC(b_i, D_j) = w(b_i - b_0)\times \\ (ACC(b_0, D_{j+\Delta}) - ACC(b_0, D_j)) \quad \text{for} \;\; 0 \leq i \leq N. \end{aligned} \tag{5.3}$$

where $ACC(b_i, D_j)$ is the accuracy of the config (b_i, D_j), and $w(b_i - b_0)$ is the weight factor; note that for Chameleon [38], $w(b_i - b_0) = 1$. We model $w(b_i - b_0)$ as a linear function with $w(0) = 1$ and $w(b_N - b_0) = 0.3$.

Single-Knob Accuracy Interpolation. Within the single-axis, we search only half the configs (e.g., even indices) and linearly interpolate the rest. We use golden config output (i.e., heaviest backbone on highest bitrate) as ground truth, similar to prior works [38, 76, 83, 85].

Evaluation. Figure 5.14 shows the effectiveness of our profiler: when profiling 4×7 configs (1,2,4,8 Mbps and EfficientDet D0-D6) on BDD dataset, our profiler reduces the overhead by 79% and 40% compared to full search and Chameleon [38] without accuracy drop.

5.5.3 Resource Availability Estimator

The profiler provides the joint scheduler (Chapter 5.6) with plausible configs that satisfy the resource budget and app accuracy requirement. For this, the resource availability estimator continuously monitors resource usage and budget for both network and compute stages. For network stage, we estimate the bandwidth using the received packet sizes and timestamps with Exponentially Weighted Moving Average (EWMA) filtering. Configs with bitrate below the bandwidth are marked as plausible. For compute stage, the DNN inference latencies of all candidate backbones are first profiled offline (stored as a look-up table) and updated online upon each inference completion using EWMA filtering. Configs with latency that enable real-time processing are marked as plausible (e.g., with a single GPU, <33 ms latency is required for 30 fps video).

5.5.4 Other Design Considerations

Batching and Multithreading. For precise inference latency control (which is crucial for resource availability estimation and scheduling) and real-time latency guarantee

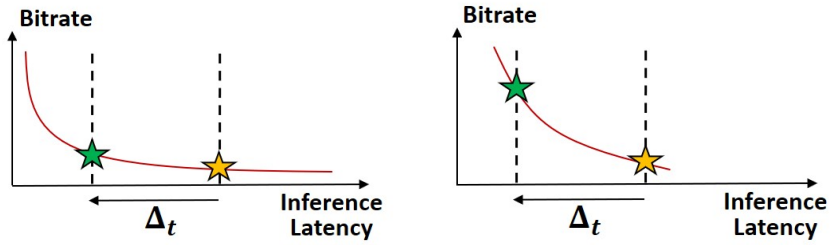
(Chapter 5.2.1), we do not use batching nor multi-threading; multi-threading achieves only 25% throughput improvement at the cost of $\approx$100$\times$ tail latency [164].

Exception Handling. Pendulum may not find any configs to schedule in two cases. The first is when a user has no accuracy-satisfying config under resource budget. In such a case, we lower the bottlenecked resource usage (for real-time processing) and increase the remaining stage usage to maximum (for best-effort accuracy). The second is when both stages are bottlenecked (rarely happens as analyzed in Chapter 2.2.3). In such a case, Pendulum notifies the user of the achievable accuracy under current resources for further actions (e.g., lower accuracy requirement or secure more resources).

Pipelined and Parallel Re-Encoding. The live video stream should be re-encoded into multiple bitrates for tradeoff profiling. To minimize the re-encoding delay, we (i) parallelize the multi-bitrate re-encoding process, as well as (ii) pipeline it with the live video decoding process. Specifically, the server spawns multiple video codecs for re-encoding (one for each target bitrate) per user upon his connection. When the server receives each live video frame, it feeds the frame to the re-encoding codecs so that the frame is re-encoded into multiple bitrates in parallel while the next live frame is received.

Parallel Profiling Inference. Each tradeoff profiling event involves multiple DNN inferences (across multiple bitrates and backbones). The profiler runs the inferences over multiple GPUs in parallel to minimize the delay.

Multi-GPU Load Balancing. The scheduler balances the inference workload across GPUs to avoid cases where some GPU(s) are overloaded (e.g., with heavy backbones) and cause long processing latency and idle time on other GPUs. Specifically, the scheduler keeps track of the inference request queue of the GPUs, and schedules the new request to the GPU whose expected finish time is the shortest.



(a) User A's tradeoff.

(b) User B's tradeoff.

Figure 5.15: Motivation of multi-user joint scheduling: additional bandwidth required to compensate Δ_t inference latency differs depending on the user's tradeoff curve.

5.6 Multi-User Joint Scheduling

5.6.1 Overview

Our goal is to jointly allocate resources across users so that (i) resource constraints are satisfied, and (ii) overall resource cost is minimized. While multi-user resource scheduling has been widely studied in the context of DNN inference serving at cloud servers [1, 91], joint scheduling of the network and compute resources has rarely been considered. Figure 5.15 motivates the importance of good joint scheduling. Assume two users (with different tradeoff curves) use the configs marked as yellow stars. Then, compute bottleneck occurs, and the total inference latency should be reduced by Δ_t. Adjusting User B's bitrate (for accuracy compensation) requires a significantly more increase than adjusting User A's, likely resulting in inefficient overall solutions.

Finding the optimal resource allocation across multiple users is challenging, mainly due to the large search space. Specifically, the allocation problem involves an exploration of $O(M^N)$ options for N users (e.g., 10s-100s), each with M configs (e.g., 49 for 7 candidate backbones and bitrates, respectively). Also, different tradeoff curves of individual users and scenes make it non-trivial to narrow down the solution space efficiently. To achieve our goal, we formulate a cost minimization problem (Chapter 5.6.2) and design a greedy scheduling algorithm to find an approximate solution (Chapter 5.6.3).

Algorithm 2 Iterative Max Cost Gradient Algorithm

Inputs: Accuracy-satisfying configs $C_i = \{(b_{i,j}, t_{i.j})\}$ for users $1, ..., N$, resource budgets $B_{network}, B_{compute}$.

Output: Cost-minimizing resource allocation $S = \{j_1, ..., j_N\}$

1: $S \leftarrow \{0, ..., 0\}$ i in $1, ..., N$ Chapter tate $S[i] \leftarrow GetCostOptimalConfig(C_i)$
DetectBottleneck(S, $B_{network}$, $B_{compute}$) == $true$

2: $j \leftarrow FindMaxCostGradientUser(C, S)$

3: $S[j] \leftarrow AdjustConfigByStep(C[j], S[j])$

4: return S

5.6.2 Scheduling Problem Formulation

User i $(= 1, ..., N)$ has K_i accuracy-satisfying configs $\{C_{i,j} = (b_{i,j}, t_{i,j})\}(j = 1, ..., K_i)$. User i processes f_i fps video and runs n_i inferences per frame (i.e., total inference time per second = $f_i \cdot n_i \cdot t_{i,j}$). User i experiences BW_i Mbps bandwidth, and the server has N_{GPU} GPUs, with maximum utilization time t_{th}. The scheduler periodically finds the cost-minimizing allocation $J^* = \{j_1^*, ..., j_N^*\}$ is obtained by solving

$$\begin{aligned} &\min_J \left(Cost_{Net}\left(\sum_i b_{i,j}\right) + Cost_{Comp}\left(\sum_i t_{i,j}\right)\right) \\ &\text{s.t. } \sum_i f_i \cdot n_i \cdot t_{i,j} \leq t_{th} \cdot N_{GPU}, \;\; b_{i,j} \leq BW_i \;\; \forall i = 1, ..., N \end{aligned} \tag{5.4}$$

The first constraint enforces that the total inference times do not exceed the compute budget. The second constraint enforces that each user's selected bitrate does not exceed his current network bandwidth so as to avoid excessive network transmission delay.[3] Note that we only model the serving cost as the tradeoff profiling cost is negligible; they may temporarily affect per-frame latency, but the impact on average throughput is negligible (e.g., 1.5% of serving cost as shown in Chapter 5.7.6). We model the cost functions using linear billing model [165, 166].

[3]We currently consider BW_i as the given value (e.g., allocated by the MAC scheduler at the base station and estimated by the resource monitor separately). In case the operator has control over the RAN/Core, BW_i can also be dynamically allocated across users.

5.6.3 Scheduling Algorithm

The formulation falls into the multi-dimensional knapsack problem, which is known to be NP-hard. We design a *Iterative Max Cost Gradient Algorithm* which (i) first finds the user-wise optimal configs and (ii) incrementally adjusts the allocation until bottleneck is resolved. This heuristic effectively finds a near-optimal solution with the search space reduced from $O(M^N)$ to $O(M \cdot N)$.

Algorithm 2 describes the operation of the Iterative Max Cost Gradient algorithm. The scheduler takes the most up-to-date profiled accuracy-satisfying tradeoff configs $\{C_{i,j}\}$ for each user i along with the resource budgets $B_{network}$ and $B_{compute}$ as input, and outputs the cost-minimizing resource allocation S. First, it finds user-wise cost-optimal configs (lines 1–1). Then, it checks if the selected configs exceed the resource budgets. If a bottleneck is detected (line 1), it iteratively adjusts the config of the user with maximum cost gradient by a step until the bottleneck is resolved (lines 2–3). The cost gradient is defined as how much the bottleneck stage's resource cost can be reduced by increasing the remaining stage's cost. For example, if the network is bottlenecked, User i (with currently selected config j)'s cost gradient CG_i is

$$CG_i = \left| \frac{C_{network} \cdot (b_{i,j+1} - b_{i,j})}{C_{compute} \cdot (t_{i,j+1} - t_{i,j})} \right|, \tag{5.5}$$

assuming $\{(b_{i,j}, t_{i,j})\}$ is sorted ascending order of t.

Figure 5.16 shows an example for three users. Assume that the user-wise cost-optimal configs are black crosses. If the bitrate sum exceeds the network budget, the scheduler first reduces the bitrate of User 2 who has the highest cost gradient (1.5) (*Iter* #1). The scheduler next chooses User 2 whose cost gradient is the largest (*Iter* #2). The process repeats until the bottleneck is resolved.

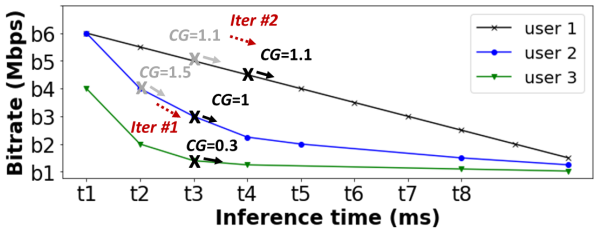


Figure 5.16: Iterative Max Cost Gradient algorithm operation example (2 iterations, CG: cost gradient).

Table 5.2: Datasets for evaluation.

	Camera	Content	Scene Change	# Videos
MOT [31]	CCTV, Handheld	Streets	Moderate	5
BDD [158]	Dashcam	City road	Fast	6
Game (self-collected)	Dashcam	Racing game	Very fast	10

5.7 Evaluation

5.7.1 Setup

System Implementation. We use TensorFlow 2.6.2 C api + CppFlow [167] and PyTorch 1.10.1 C++ api for DNN inference. We use Secure Reliable Transport (SRT) [168] for live video streaming. We use FFMpeg 4.1.9 and CUDA-accelerated H.264 for video encoding and OpenCV 4.4.0 for image processing. The server is implemented on a Supermicro SuperServer 4029GP-TRT2 with Intel Xeon Gold 5128 CPU and 8× RTX 2080 Ti GPUs (shared for serving and profiling).

Datasets. We use three datasets with different scene change speeds (Table 5.2): five 30fps videos (02, 04, 09, 10, 11) for MOT [31] (moderate), 6 videos from BDD [158] (fast), and self-collected racing game videos from YouTube (very fast). All videos are scaled to 720p. For videos without ground truth labels, we use golden config (heaviest backbone, highest bitrate) outputs.

Tasks and DNNs. We use two DNN tasks: object detection and semantic segmenta-

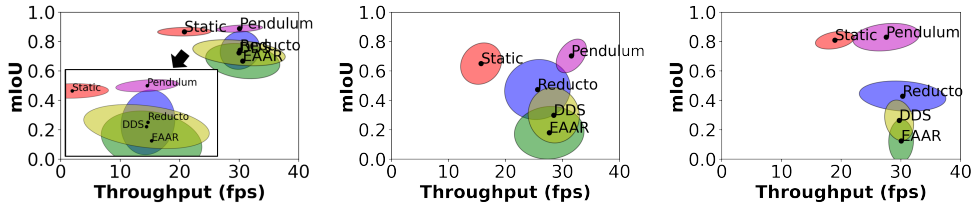
tion. For detection, we use YOLO-v5 [30] with 5 backbones (n/s/m/l/x), and EfficientDet with 7 backbones (D0-D6) [106]. For segmentation, we train FPN [169] with 7 EfficientNet backbones (B0-B6) [33] on BDD [158]. All backbones can be loaded on a single GPU (e.g., 6.5 GB for 7 EfficientDet backbones, 10.4 GB for 7 FPN backbones). Unless stated otherwise, we report performance using EfficientDet.

Bottleneck Generation. We use Linux tc [153] to shape the network bandwidth and incur a network bottleneck. We linearly scale the DNN inference latency by a slowdown factor (by injecting delay after inference similar to previous work [10]) to emulate the compute bottleneck.

Single-Stage Baselines: Static uses a fixed (bitrate, backbone). **DDS [85]** uses two-path streaming (low-quality probe frame + high-quality feedback for regions with low-confidence inference results). **EAAR [4]** uses dynamic RoI encoding (high quality only for regions where objects existed in the previous frame) and motion vector-based frame filtering. **Reducto [76]** uses pixel/edge/area feature difference-based frame filtering (feature type and bitrate-satisfying threshold are offline profiled per each task and dataset). **Backbone Adaptation (BA)** only adapts the DNN backbone (based on our profiler in Chapter 5.5) and uses a fixed bitrate. This is equivalent to single-knob Chameleon [38]. We also show joint scheduling gain in DDS and EAAR (Chapter 5.7.3).

Multi-Stage Baseline: Pendulum**-Decoupled** is a network-compute-decoupled joint scheduler. Upon network bottleneck, all users reduce their bitrates at a same scale until bottleneck is resolved. Then, the server compensates the accuracy drop by choosing the cost-optimal backbones that satisfy their accuracy requirement.

Metrics: Throughput (fps) is the number of frames processed per each 1s window. Albeit it can fluctuate due to processing latency jitter, the average should match the input frame rate (e.g., 30 fps) to avoid frame drop.



(a) MOT (moderate speed). (b) BDD (fast speed). (c) Game (very fast speed).

Figure 5.17: Throughput-accuracy comparison in network bottleneck scenario.

5.7.2 End-to-End Improvement

Throughput-Accuracy Performance. Figure 5.17 compares the throughput-accuracy of Pendulum against baselines across three datasets. Ellipses show the 1-σ range of the results. We throttle the network bandwidth from 20 to 3 Mbps after 4 seconds after the streaming start. Static, which uses a fixed (4 Mbps, EfficientDet-D1), suffers from throughput drop due to network bottleneck. Overall, Pendulum consistently achieves $\approx$30 fps throughput and higher accuracy compared to baselines: up to 0.64 mIoU gain (Game, Pendulum: 0.81 vs. EAAR: 0.17). The performance of baselines varies depending on the dataset. For MOT with moderate scene changes, all baselines effectively optimize the bitrate to below 3 Mbps (e.g., EAAR: 2.98 Mbps), achieving comparable accuracy to Pendulum. However, for BDD and Game with faster scene change, the accuracy drops significantly, especially for EAAR and DDS. For EAAR, object region from previous frame becomes highly stale in fast-changing scenes. For DDS, fast-changing video encoded in a low-bitrate probe stream (e.g., 1 Mbps) suffers from severe frame quality drop, resulting in inaccurate DNN inference and feedback frame request. Consequently, they both end up streaming the entire frame in low-quality. Pendulum achieves higher mIoU even when DDS and EAAR always use the heaviest D6 backbone (e.g., 0.71 vs. 0.49, 0.48 in BDD).

Frame-wise Latency. Pendulum also achieves low frame-wise latency. Figure 5.18 compares the frame-wise latency of DDS and Pendulum. Pendulum yields $<$100 ms latency, much smaller than DDS involving two frame transmissions and DNN inferences per frame.

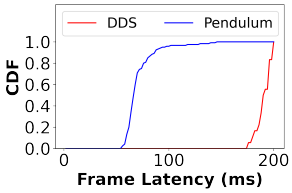
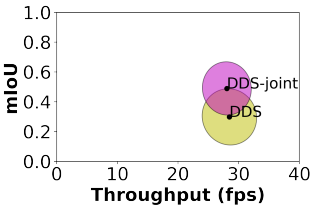
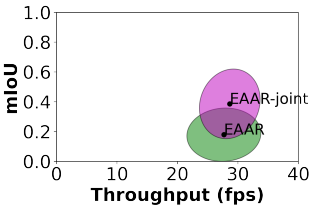


Figure 5.18: Frame-wise latency comparison.

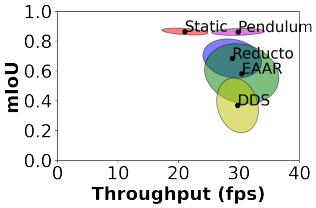


(a) DDS vs. DDS-joint, BDD.

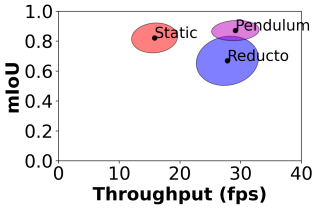


(b) EAAR vs. EAAR-joint, BDD.

Figure 5.19: Joint scheduling on state-of-the-art systems.



(a) YOLOv5 (detection), MOT.



(b) FPN (segmentation), BDD.

Figure 5.20: Performance across various tasks & DNNs.

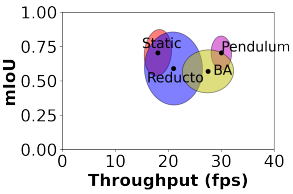


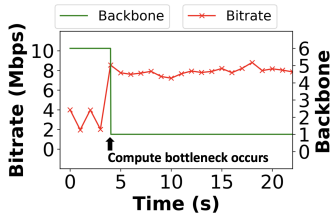
Figure 5.21: Performance in compute bottleneck (BDD).

5.7.3 Joint Scheduling on SOTA Systems

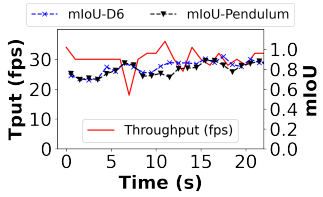
We next evaluate joint scheduling gain on state-of-the-art network-only scheduling systems: DDS and EAAR. When network bottleneck occurs (same as Chapter 5.7.2), DDS-joint and EAAR-joint reduce the bitrates accordingly and increase the DNN backbone from EfficientDet-D1 to D6 (increment can be optimized by profiling). Figures 5.19(a) and (b) show the results on the BDD dataset. Joint scheduling achieves 0.17 and 0.20 higher mIoU than baseline DDS and EAAR, respectively, showing the generality of joint scheduling.

5.7.4 Performance on Other Models & Tasks

YOLOv5 (Detection). We repeat the same experiment as in Figure 5.17(a), but with YOLOv5 backbones. We observe a similar trend: Pendulum achieves 0.17, 0.32 and 0.52 higher mIoU than Reducto, EAAR, and DDS, respectively. Note that DDS's mIoU is lower than when using the EfficientDet backbones, as the lightweight YOLOv5



(a) Bitrate and backbone.



(b) Throughput and mIoU.

Figure 5.22: Pendulum operation in compute bottleneck.

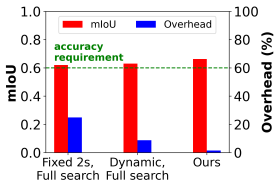


Figure 5.23: Profiler performance breakdown.

backbone yields less accurate feedback regions on low-bitrate videos.

FPN (Segmentation). Figure 5.20(b) shows that Pendulum achieves similar performance for FPN (e.g., ≈30 fps throughput with 0.19 higher mIoU than Reducto).

5.7.5 Performance in Compute Bottleneck

Throughput-Accuracy. Figure 5.21 shows the performance of Pendulum in compute bottleneck scenario on MOT. We increase the inference latency slowdown factor from 1× to 2× 4 seconds after the app starts. Static (2 Mbps, EfficientDet-D6) suffers from throughput drop. Compared to Reducto and BA which only reduce the DNN workload (either by reducing the number of frames or backbone), Pendulum effectively increases the bitrate (from 1.98 to 5.42 Mbps) resulting in 0.12 higher mIoU and 1.29× higher throughput.

Operation Timeline. Figure 5.22 shows an example operation timeline of Pendulum in compute bottleneck scenario for a BDD video. Figure 5.22(a) shows the bitrate and backbone over time. At the start, Pendulum uses (2 Mbps, EfficientDet-D6) config; bitrate peaks every 2 seconds due to periodic high-bitrate frames for profiling (Chapter 5.5.2). After 4s when compute bottleneck occurs, Pendulum reduces the backbone to D1 and increases the bitrate to 8 Mbps (according to the profiling results). Figure 5.22(b) shows the throughput and mIoU in the same timeline. Pendulum shortly suffers from throughput drop when bottleneck occurs, but quickly recovers from it while retaining the same accuracy by using D6 backbone.

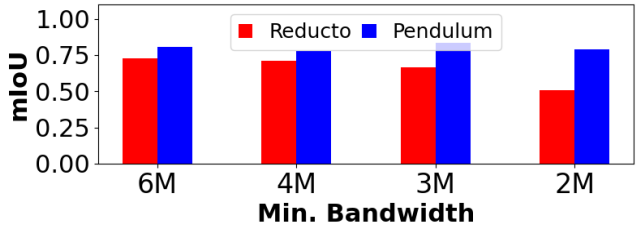
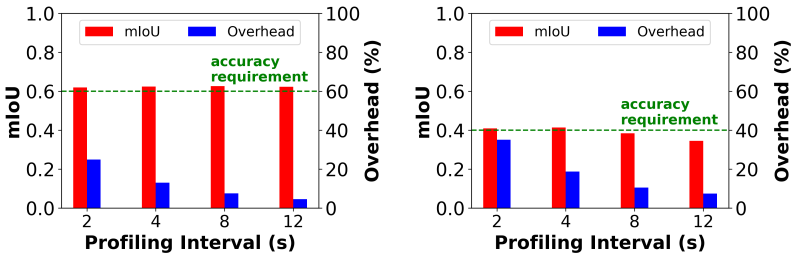


Figure 5.24: Performance under bandwidth fluctuation.



(a) MOT. (b) BDD.

Figure 5.25: Impact of profiling interval on performance.

5.7.6 System Microbenchmarks

5.8.6.1 Performance Under Bandwidth Fluctuation

Figure 5.24 shows the performance of Reducto and Pendulum under bandwidth fluctuation scenarios (e.g., mobile devices). For real-time throughput, both systems conservatively reduce the bitrate to minimum observed bandwidth. While Reducto's accuracy quickly drops as bottleneck becomes more severe, Pendulum robustly retains the accuracy by increasing the backbone.

5.8.6.2 Resource-Efficient Tradeoff Profiler

Performance Breakdown. Figure 5.23 shows the performance breakdown of the profiler in terms of profiling overhead (ratio between the total costs for profiling and serving) and mIoU. We use 5 YOLOv5 backbones (n/s/m/l/x) and 4 bitrates (1, 2, 4, 8 Mbps); a full search over all configs takes 1s. Fixed interval, full search profiling on 2 frames per every 2s window incurs 24.9% overhead. Scene change-based dynamic pro-

filing reduces the overhead to 8.8%. Finally, leveraging the knob independence (which reduces the number of searches to 8), reduces the overhead to 1.5% (93.9% smaller than the fixed-interval, full search) with negligible mIoU drop.

Scene Change Detector Accuracy. Figure 5.25 shows the impact of profiling interval on serving accuracy and overhead across two datasets. The results show that the interval should change according to the scene change speed. In MOT with static scenes (Figure 5.25 (a)), triggering the profiling intermittently at every 12s saves 18% overhead without accuracy drop. However, for BDD with more dynamic scenes (Figure 5.25 (b)), frequent profiling is necessary (e.g., every 4s). For the two datasets, our scene change detector accurately triggers profiling at every 9.09s and 2.21s on average, respectively.

5.8.6.3 Multi-User Joint Resource Scheduler

Figure 5.26 compares the performance of various schedulers. We assume 10 users with randomly sampled tradeoff curves from MOT and BDD (each with 4-6 Pareto-optimal configs, bitrate range in [1,10] Mbps and EfficientDet-D0-D6 backbones). We assume a network bottleneck scenario, where the total bitrate sum should not exceed 20 Mbps. Unit network and compute costs are set as $0.36 and $0.74 (Chapter 5.2.1). *Per-User Optimal* chooses user-wise cost-optimal configs independently, resulting in severe network bottleneck. Compared to the full search optimal solution, our Iterative Max Cost Gradient algorithm achieves comparable performance by exploring only 0.004% searches. It also reduces the total cost by 25% compared to Pendulum-Decoupled.

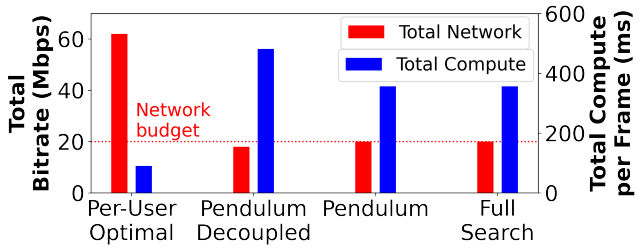


Figure 5.26: Multi-user scheduling performance.

Chapter 6

Heimdall: Mobile GPU Coordination Platform for AR Applications

6.1 Introduction

In this Chapter, we design Heimdall, a mobile GPU coordination platform to support concurrent multi-DNN and rendering tasks for AR apps. Heimdall newly designs and implements a *Pseudo-Preemptive mobile GPU coordinator* to enable highly flexible coordination among multi-DNN and rendering tasks. Heimdall is distinguished from prior work in that i) it coordinates latency-sensitive foreground rendering tasks along with background DNN tasks to achieve stable rendering performance of ≈30 fps, and ii) it addresses resource contention among multiple DNNs to meet their latency requirements.

Designing Heimdall involves the following challenges:

• **Multi-DNN GPU Contention.** Compared to prior mobile deep learning frameworks [14,16,35,36] that have mostly been designed for running a single DNN, emerging AR apps require concurrent multi-DNN execution (Chapter 2.1.2). Not only are the individual state-of-the-art DNNs very complex to run in real-time (Chapter 2.2.1), running multiple DNNs concurrently incurs severe contention over limited mobile GPU

resources, degrading overall performance. For example, our study shows that running 3 to 4 different DNNs commonly required in AR apps (e.g., object detection, image segmentation, hand tracking) concurrently on Google TensorFlow-Lite (TF-Lite) [35] and Xiaomi MACE [36] over high-end Adreno 640 GPU incurs as high as 19.7× slowdown (Chapter 2.2.4). Although several recent studies aimed at running multiple DNNs concurrently on mobile [3, 12, 54], they have mostly focused on memory optimization [12,54] or cloud offloading [3]; multi-DNN GPU contention remains unsolved.

• **Rendering-DNN GPU Contention.** More importantly, prior works only consider a DNN running in an isolated environment where no other task is contending over the GPU. When running rendering in parallel with DNNs, GPU contention degrades and fluctuates the frame rate, degrading user experience (e.g., drops from 30 to 11.99 fps when 4 DNNs run in background (Chapter 2.2.4)).

There have been studies to schedule concurrent tasks on desktop/server GPUs [57–59,66,67,69,170,171], either with *parallel execution* by dividing GPU cores (e.g., using NVIDIA Hyper-Q [60]) with hardware architectural support, or with *time-sharing* through preemption (e.g., using CUDA stream prioritization). However, mobile GPUs do not provide architectural support for parallel execution, while fine-grained preemption is not easy as well due to high context switch costs caused by large state size and limited memory bandwidth (Chapter 6.3.1). Even with architecture evolution, the need for an app-aware coordinator to dynamically prioritize and allocate resources between multiple DNNs persists (Chapter 7.2.3). We can also consider cloud offloading, but it is not trivial to employ it in outdoor scenarios where network latency is unstable.

To tackle the challenges, we design a *Pseudo-Preemption* mechanism to support flexible scheduling of concurrent multi-DNN and rendering tasks on mobile GPU. We take the *time-sharing* approach as a baseline, and enable context switches only when a semantic unit of the DNN or rendering task is complete. This does not incur additional memory access cost, which is the core difficulty in applying conventional

preemption (triggered by periodic hardware interrupt regardless of the app context) for mobile GPUs. Accordingly, it allows the multi-DNN and rendering tasks to time-share the GPU at a fine-grained scale with minimal scheduling overhead. With this new capability, we flexibly prioritize and run the tasks on the GPU to meet the latency requirements of the AR app. Our approach can also be useful for the emerging neural processors (e.g., NPUs or TPUs), as preempting hard-wired matrix multiplications is complicated and context switch overhead can be more costly due to larger state sizes (Chapter 7.2.3).

To implement *Pseudo-Preemption* mechanism, Heimdall incorporates the following components:

• **Preemption-Enabling DNN Analyzer.** The key in realizing *Pseudo-Preemption* is breaking down the bulky DNNs into small schedulable units. Our *Preemption-Enabling DNN Analyzer* measures the execution times of DNN and rendering tasks on the target mobile device and partitions the DNNs into the units of scheduling to enable fine-grained GPU time-sharing with minimal scheduling overhead. We notice that the execution time of individual DNN operator (op) is sufficiently small (e.g., $<$5 ms for 89.8% of ops). Exploiting this, the analyzer groups several consecutive ops as a scheduling unit which can fit between the two consecutive rendering events. As rendering latencies are often very small (e.g., 2.7 ms for rendering a 1080p camera frame), each task is used as the scheduling unit. Note that existing frameworks run the entire bulky DNN inference all at once (e.g., Interpreter.Run() in TF-Lite [172], MaceEngine.Run() in MACE [36]), limiting multi-DNN and rendering tasks to share the mobile GPU at a very coarse-grained scale.

• **Pseudo-Preemptive GPU Coordinator.** We design a GPU coordinator that schedules the DNN and rendering tasks on GPU and CPU. It can employ various scheduling policies based on multiple factors: profiled latencies, scene variations, and app/user-specified latency requirements. As the base scheduling policy, the coordinator assigns the top priority to the rendering tasks and executes them at the target frame rate

(e.g., 30 fps) to guarantee the usability of the app. Between the rendering events, the coordinator decides the priority between multiple DNNs and determine which chunk of DNN ops (grouped by the analyzer) to run on the GPU. It also decides whether to offload some DNNs to the CPU in case there is a high level of contention on the GPU. Note that existing frameworks provide no means to prioritize a certain task over others, making it difficult to guarantee performance under contention.

Our major contributions are summarized as follows:

- To our knowledge, this is the first mobile GPU coordination platform for emerging AR apps that require concurrent multi-DNN and rendering execution. We believe our platform can be an important cornerstone to support many emerging AR apps.
- We design a *Pseudo-Preemption* mechanism to overcome the limitations of mobile GPUs for supporting concurrency. With the mechanism, Heimdall enhances the frame rate from ≈12 to ≈30 fps while reducing the worst-case DNN inference latency by up to ≈15 times compared to the baseline multi-threading method.
- We implement Heimdall on MACE [36], an OpenCL-based mobile deep learning framework, and conduct an extensive evaluation with 8 state-of-the-art DNNs (see Table2.2) and various mobile GPUs (i.e., recent Adreno series) to verify the effectiveness.

6.2 Analysis on GPU Contention

The workload of upcoming AR apps is unique in that it runs multiple compute-intensive DNNs simultaneously while seamlessly rendering the virtual contents. However, existing mobile deep learning frameworks lack support for multi-DNN and rendering concurrent execution, and severe GPU contention incurs significant performance degradation for both DNN and rendering tasks.

Algorithm 3 shows the OpenCL-based DNN inference flow in MACE.[1] Upon the

[1]The logic is implemented in SerialNet.Run() function, while TF-Lite is implemented similarly using OpenGL/OpenCL.

Algorithm 3 OpenCL-based DNN inference in MACE

1: **for** $Operator$ in $Graph$ **do**
2: $TargetDevice \leftarrow Operator.GetTargetDevice()$ $TargetDevice == GPU$
3: $Kernel \leftarrow Operator.GetKernel()$
4: $clCommandQueue.enqueueNDRangeKernel(Kernel)$ $TargetDevice == CPU$
5: $clCommandQueue.finish()$
6: $Operator.RunOnCPU()$

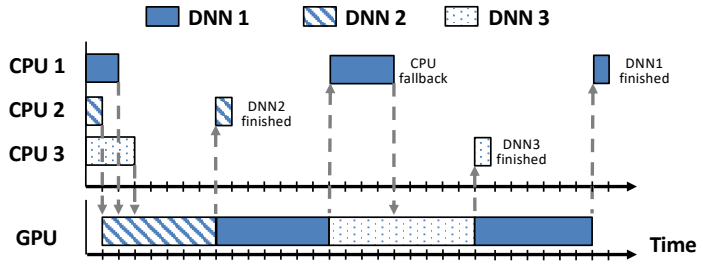


Figure 6.1: Multi-DNN GPU contention example.

inference start, the framework executes a series of operators (ops) constituting the DNN. Per each op, the framework first identifies if it is executed on GPU or CPU (lines 1–2). A GPU op is executed by enqueueing its kernel to the command queue to be executed by the GPU driver (lines 2–4). As enqueueNDRangeKernel() function is an asynchronous call, consecutive GPU ops are enqueued in short intervals (few μs) and executed in batches by the driver to enhance GPU utilization. However, when a CPU op is encountered, it can be executed only after the previously enqueued GPU ops are finished and the result is available to the CPU via CPU/GPU synchronization (lines 4–6).

Figure 6.1 illustrates an example 3-DNN GPU contention scenario that can occur in the above inference process. Each thread on different CPU cores first runs the input preprocessing and enqueues the DNN inference to the GPU. At this step the first contention occurs; DNN#1 and #3 cannot access the GPU until the already running DNN#2 is finished. After DNN#2 finishes, DNN#1 takes control over the GPU and

runs its inference. However, let's assume that some ops in DNN#1 are not supported by the GPU backend of the framework and needs to be executed on the CPU (Table 2.2 shows how frequently this occurs for different DNNs; more details are in Chapter 6.6.2). In such a case, DNN#1 encounters another contention: even when the CPU op execution is finished, it cannot access the GPU until the already running DNN#3 finishes. As a result, the inference latency of DNN#1 is significantly delayed.

The above contention becomes more severe with more number of DNNs concurrently running. Furthermore, DNNs with more CPU fallback ops suffer more from contention, as they lose access over the GPU at every CPU op execution. For example, in Figure 2.3(a), StyleTransfer [22] containing 14 CPU ops suffers the most latency overhead compared to other DNNs that contain no CPU ops.

6.3 Heimdall System Overview

6.3.1 Approach

The core challenge in supporting concurrency on mobile GPU lies in the lack of support for parallelization or preemption. As analyzed in Chapter 6.2, mobile GPU can run only a single task at a given time, making it hard to provide stable performance when multiple tasks are running. Existing mobile deep learning frameworks, however, fail to consider such limitations, and are ill-suited for AR workloads in two aspects: i) they run the entire bulky DNN inference all at once (e.g., by Interpreter.Run() in TF-Lite, MaceEngine.Run() in MACE), limiting multi-DNN and rendering tasks to share the GPU at a very coarse-grained scale (Table 2.2), and ii) they provide no means to prioritize a certain task over others, making it challenging to guarantee performance under contention.

6.4.1.1 Why Not Apply Desktop GPU Scheduling?

One possible approach is to implement parallelization or preemption in mobile GPUs. Although there have been many studies to support multitask scheduling on desktop/server-grade GPUs [57–59,69,170,171], they are either designed for CUDA-enabled NVIDIA GPUs (which are unsupported in mobile devices) or require hardware modifications (e.g., memory hierarchy [65]), making it difficult to apply for commodity mobile GPUs. Also, adopting similar ideas is not straightforward due to the following limitations of mobile GPUs.

Limited Architecture Support. Several studies focused on spatially sharing the GPU to run multiple kernels in parallel, either by partitioning the computing resources [69, 170] (e.g., starting from Kepler architecture [173] released in 2012, NVIDIA GPUs can be parallelized in units of Streaming Multiprocessors using Hyper-Q [60]) or fusing parallelizable kernels with compiler techniques [171, 174]. However, such techniques are unsupported in mobile GPUs architecturally at the moment.

Limited Memory Bandwidth. Other studies aimed at time-sharing the GPU by fine-grained context switching [57–59], as well as enabling high-priority tasks to preempt the GPU even when others are running [69] (e.g., by using CUDA stream prioritization). However, frequent context switching incurs high memory overhead due to large state size, which is burdensome for mobile GPUs with limited memory bandwidth. For example, ARM Mali-G76 GPU in Samsung Galaxy S10 (Exynos 9820) has 26.82 GB/s memory bandwidth shared with the CPU, which is 23$\times$ smaller than that of NVIDIA RTX 2080Ti (i.e., 616 GB/s). Each context switch requires 120 MB memory transfer (=20 cores$\times$24 execution lanes/core$\times$64 registers/lane$\times$32 bits), which incurs at least 4.36 ms latency even when assuming the GPU fully utilizes the shared memory bandwidth. While recent Qualcomm GPUs (Adreno 630 and above) support preemption [175] (which can be utilized by setting different context priorities in OpenCL), we observed that each context switch (both between rendering–DNN and DNN–DNN) in-

curs 2–3 ms overhead on LG V50 with Adreno 640 GPU, aside from the fact that the priority scheduling is possible only at a coarse-grained scale (i.e., low, medium, and high). Such memory overhead would be burdensome in the multi-DNN and rendering AR workload, where context switch should occur at a 30 fps (or higher) scale.

6.4.1.2 Our Approach: Pseudo-Preemption

To tackle the challenges, we design a *Pseudo-Preemption* mechanism to coordinate multi-DNN and rendering tasks. As parallelization is unsupported in mobile GPUs, we take the *time-sharing* approach as a baseline. To mimic the effect of preemption while avoiding the burdensome context switch memory overhead, we divide the DNN and rendering tasks into smaller chunks (i.e., scheduling units) and switch between them only when each task chunk is finished, enabling multi-DNN and rendering tasks to time-share the GPU at a fine-grained scale. A possible downside of our approach is that fragmenting the GPU tasks may incur latency overhead, as the GPU driver would lose the chance to batch more tasks to enhance GPU utilization. However, such overhead can be minimized as we can flexibly adjust the scheduling unit size to balance time-sharing granularity and latency overhead (e.g., 89.8% of the DNN ops run within 5 ms, and rendering latencies are typically small).

6.3.2 Design Considerations

Commodity Mobile Device Support. Our goal is to support a wide range of commodity mobile devices by requiring no modification to existing hardware or GPU drivers. We focus on using mobile GPU and CPU in this work, and plan to add NPU/TPU support when the hardware and APIs are more widely supported. We also leave cloud/edge offloading out of our scope, as it introduces latency issues in outdoor mobile scenarios.

Guarantee Stable Rendering Performance. Our main goal is to enable seamless rendering even in the presence of multi-DNN execution. We aim to minimize the frame rate drop and fluctuation due to GPU contention, which harms the user experience.

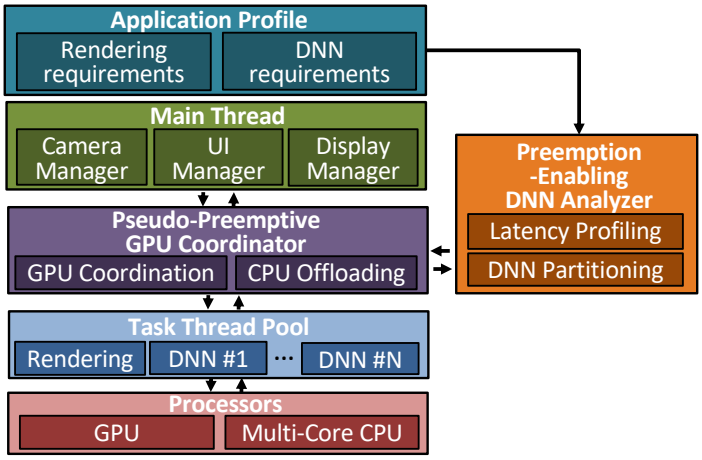


Figure 6.2: System Architecture of Heimdall.

Coordinate Multi-DNN Execution. While guaranteeing seamless rendering, we aim to coordinate multiple DNNs to satisfy the app requirements with minimal inference latency overhead.

No Loss of Model Accuracy. Our goal is to incur no accuracy loss for each DNN inference. We leave runtime model adaptation for latency-accuracy tradeoff (e.g., via pruning [54]) to future work.

Transparency. Finally, we aim to design a system that minimizes the extra efforts required for the app developers to use our platform.

6.3.3 System Architecture

Figure 6.2 depicts the overall architecture of Heimdall. Given the app profile (rendering frame rate and resolution, DNNs to run and latency constraints), *Preemption-Enabling DNN Analyzer* first profiles the information necessary to determine the scheduling units to enable the *Pseudo-Preemption* mechanism. First, it profiles the rendering and DNN inference latencies on the target AR device to determine how much time the DNNs can occupy the GPU between the rendering events (Chapter 6.4.2). Second, it partitions the DNNs into chunks (scheduling unit) that can fit between the rendering events with minimal inference latency overhead (Chapter 6.4.3).

At runtime, *Pseudo-Preemptive GPU Coordinator* takes multi-DNN and rendering tasks from the main thread (that controls the camera, UI, and display), and coordinates their execution to satisfy the app requirements. Specifically, it first defines a *utility function* to compare which DNN is more important to run at a given time based on the inference latency and scene contents (Chapter 6.5.2), and coordinates their execution on GPU, as well as dynamically offload some DNNs to the CPU to reduce the GPU contention (Chapter 6.5.3).

6.4 Preemption-Enabling DNN Analyzer

6.4.1 Overview

What Should We Analyze? The goals of the analyzer are i) profile rendering and DNN inference latencies on the target device (which varies depending on the mobile SoC and GPU) to let the coordinator get a grasp on how it can dynamically schedule their execution, and ii) partition the bulky DNNs into chunks (i.e., the units of scheduling), to enable fine-grained GPU coordination and guarantee rendering performance.

Static Profiling vs. Dynamic Profiling? The app requires to run multi-DNN, rendering, and other tasks (e.g., pre/postprocessing for the DNN inference, camera) simultaneously, which may fluctuate the execution times of each task at runtime. However, as mobile GPUs do not support preemption (i.e., a task cannot be interrupted once started), the execution times on GPU remain stable regardless of the presence of other tasks. Thus, offline profiling and DNN partitioning approach is feasible for GPU. However, the execution times of DNNs on CPUs may fluctuate due to resource contention; Figure 6.5 shows that the inference times on CPU increase and fluctuate when the camera is running in background. Thus, CPU execution times need to be continuously tracked at runtime.

How Fine Should We Partition the DNNs? Inference times of DNNs typically exceed multiple rendering intervals as shown in Table 2.2. At the op-level, however, the

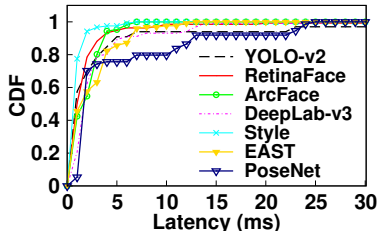


Figure 6.3: Operator-level latency distribution.

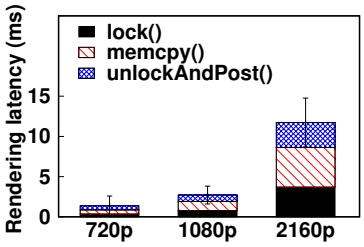


Figure 6.4: Camera frame rendering latency.

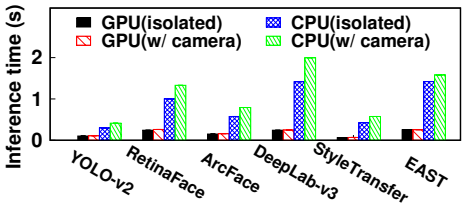


Figure 6.5: DNN inference latency with and without camera.

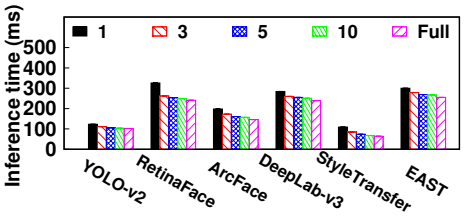


Figure 6.6: DNN inference latencies for varying partition sizes.

execution times remain small enough, making fine-grained partitioning feasible to fit in between the rendering events. For example, for the 7 DNNs in Table 2.2 whose inference latencies are over 33 ms, Figure 6.3 shows that on average 89.8% of the ops run within 5 ms on Google Pixel 3 XL. Therefore, it suffices to partition the DNNs at the op-level and not below (e.g., convolution filter-level). However, note that dividing the DNN too finely also has its downside: it incurs higher latency overhead as the GPU driver loses the chance to batch more consecutive ops to enhance GPU utilization.

6.4.2 Latency Profiling

Rendering Latency. Given the target rendering frame rate (f) and resolution, the analyzer first measures the rendering latency, T_{render}. This determines how much time the DNNs can occupy the GPU between rendering events (i.e., $\frac{1}{f} - T_{render}$). For example, rendering 1080p frames on Adreno 640 GPU in LG V50 takes 2.7 ms (Figure 6.4), leaving 30.6 ms for DNNs when the frame rate is 30 fps.

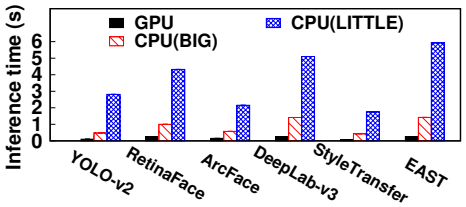


Figure 6.7: Example DNN latency profiling result on Google Pixel 3 XL.

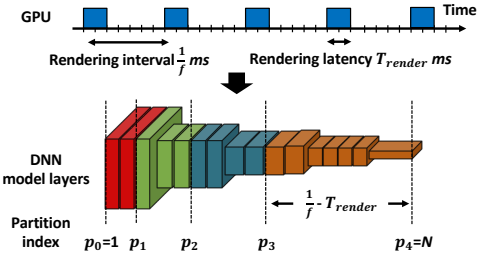


Figure 6.8: Operation of DNN partitioning.

DNN Latency. Secondly, the analyzer measures the DNN inference latencies on the target GPU and CPU. Figure 6.7 shows an example of the profiled results on different processors (i.e., the GPU and CPU cores in the ARM big.LITTLE architecture) in LG V50.[2] The analyzer also measures the inference latencies of DNNs running on CPU at runtime to track variations due to CPU resource contention.

6.4.3 DNN Partitioning

Basic Operation. Figure 6.8 shows the operation of DNN partitioning. Given a DNN D composed of N ops, let $T(D_{i,j})$ denote the execution time of a subgraph from i-th to j-th op. Our goal is to determine a set of K indices $\{p_1 = 1, p_2, p_3, ..., p_K = N\}$ that partition the DNN in a way such that each partition execution time lies within the rendering interval,

$$T(D_{p_i,p_{i+1}}) \leq \frac{1}{f} - T_{render} \quad 1 \leq i \leq K-1. \tag{6.1}$$

Although there are multiple solutions that satisfy the constraints, dividing the model too finely (e.g., running only one or two ops at a time) incurs higher scheduling overhead, as the GPU driver loses the chance to batch more consecutive ops to enhance GPU utilization: Figure 6.6 shows that executing only a single op at a time incurs 13 to 70% latency overhead compared to running the entire model at once. Thus, the

[2]We currently assume that each DNN uses only a single CPU core, and leave multi-core CPU execution to future work.

analyzer minimizes K by grouping as many consecutive ops as possible without exceeding the rendering interval. This is done as follows: i) starting from the first op of the model, incrementally increase the op index i until the latency of executing op 1 to i exceeds the rendering interval, ii) group op 1 to $i-1$ as the first partition, and iii) start from op i and repeat the process until reaching the final op.

Relaxation. The main drawback of our approach is that undesirable GPU idle time occurs when a partition execution time is shorter than the rendering interval (especially at the end of the model where there are not enough ops left). To alleviate the issue, we relax the constraint in Equation (6.1) and allow the partition execution time to exceed the rendering interval by a small margin (e.g., 5 ms), so that more ops can be packed to maximize GPU utilization.

6.5 Pseudo-Preemptive GPU Coordinator

6.5.1 Overview

Where Does the Coordinator Operate? The coordinator should take into account the rendering and DNN requirements of the app, and coordinate their execution (in the units of scheduling determined by the analyzer) considering the task priorities. With this requirement, we embed the coordinator in app-level deep learning framework, rather than the OS or the device driver layer where the workloads are highly abstracted.

Operational Flow. The coordinator assigns the top priority to the rendering task and executes it at the target frame rate. We take this design decision as degradation or fluctuation in the rendering frame rate immediately affects the usability of AR apps. It is possible to change the scheduling policy to make rendering and DNN tasks to have the same priority in case rendering is less important.

The coordinator takes in the DNN inference requests from the main thread via admission control, so that the inference of a DNN is enqueued only after its previous inference has finished. When a DNN inference is enqueued, the latest camera frame is

fed as input after either resizing it to the model input size or cropping the sub-region depending on the task. The scheduling event is triggered after every rendering event to decide the priority between DNNs and determine which DNN chunk (partitioned by the analyzer) to run on the GPU until the next rendering event. To achieve the goal, we define a *utility function* that characterizes the priority of a DNN and formulate a scheduling problem that enables fine-grained GPU time-sharing between multiple DNNs to satisfy the app requirements. It also decides whether to offload some DNNs to the CPU in case the GPU contention level is too high.

6.5.2 Utility Function

To schedule multiple DNNs, we need a formal way to compare which DNN is more important to run at a given time. For this purpose, we define a *utility function* for each DNN. The utility of a DNN D_i whose k-th inference is enqueued by the main thread at $t^i_{start,k}$ is modeled as a weighted sum of the two terms,

$$U_{D_i}(t) = L_{D_i}(t, t^i_{start,k}) + \alpha \cdot C_{D_i}(t^i_{start,k}, t^i_{start,k-1}), \tag{6.2}$$

where $L(t, t_{start})$ is the *latency utility* that measures the freshness of the inference, $C_{D_i}(t^i_{start,k}, t^i_{start,k-1})$ is the *content variation utility* that captures how rapidly the scene content has changed from the last DNN inference, and α is the scaling factor (empirically set as 0.01 in our current implementation).

6.6.2.1 Latency Utility

The latency utility of the DNN D_i is calculated as,

$$L_{D_i}(t, t^i_{start,k}) = L^0_{D_i} - \left(\beta_i \cdot (t - t^i_{start,k})^{\gamma_i}\right)^2. \tag{6.3}$$

The latency utility is modeled as a concave function so that it decreases more rapidly over time to prevent the coordinator from delaying the execution too long. Three parameters can be configured to set the priorities between DNNs. β_i controls the proportion of the GPU time each DNN can occupy (e.g., setting β_i to 1 for all DNNs will enable equal sharing). $L^0_{D_i}$ and γ_i controls the priority among DNNs; a DNN with

higher $L^0_{D_i}$ and γ_i will have higher initial utility but decrease more rapidly, so that the coordinator can allow it to preempt the GPU more frequently before its utility drops.

6.6.2.2 Content Variation Utility

The content variation utility D_i is computed as the difference between the input frames of the consecutive inferences at $t^i_{start,k}$ and $t^i_{start,k-1}$. Normally, this can be done by calculating the structural similarity (SSIM) [176] between the two frames. However, this is infeasible in mobile devices due to high computational complexity. Alternatively, we take the approach in [177] and compute the difference between the Y values (luminance) Y^k of the two frames (which has a high correlation with the SSIM and requires only $O(N)$ computations),

$$C_{D_i}(t^i_{start,k}, t^i_{start,k-1}) = \sum_{h=1}^{H} \sum_{w=1}^{W} |Y^k_{h,w} - Y^{k-1}_{h,w}|, \tag{6.4}$$

where H, W is the height and width of the frame.

6.5.3 Scheduling Problem and Policy

Given the DNNs and their utilities, the coordinator schedules their execution to maximize the overall performance (defined as a policy). Specifically, the coordinator operates in a two-step manner: i) schedule DNNs to efficiently share the GPU, and ii) determine whether to offload some DNNs to the CPU to resolve contention.

6.6.3.1 GPU Coordination Policy

Among many possible policies, we define two common GPU coordination policies, following a similar approach in [54]. Assume that N DNNs $D_1, ..., D_N$ are running on GPU, with latency constraints $t_{1,max}$, ..., $t_{M,max}$ (which are set appropriately depending on the app scenario). The two policies are formulated as follows.

MaxMinUtility policy tries to maximize the utility of a DNN that is currently experiencing the lowest utility. This is done by solving,

$$\begin{aligned} &\min_{i} U_{D_i}(t). \\ \text{s.t.} \quad &t^i_{end,k} - t^i_{start,k} \leq t_{i,max} \end{aligned} \tag{6.5}$$

Under the MaxMinUtility policy, the coordinator tries to fairly allocate GPU resources to balance performance across multiple DNNs. We expect this policy to be useful in AR apps mostly consisted of continuously executed DNNs that need to share the GPU fairly (e.g., augmented interactive workspace scenario in Table 2.1).

MaxTotalUtility policy tries to maximize the overall sum of utilities of the DNNs. This is done by solving,

$$\begin{aligned} &\max_{i} \quad \textstyle\sum_{t=1}^{N} U_{D_i}(t). \\ \text{s.t.} \quad &t^i_{end,k} - t^i_{start,k} \leq t_{i,max} \end{aligned} \tag{6.6}$$

Under the MaxTotalUtility policy, the coordinator favors a DNN with higher utility (i.e., allow it to preempt the GPU more frequently) and runs the remaining DNNs at the minimum without violating their deadline. This policy will be useful in case an AR app requires to run high-priority event-driven DNNs at low response time (e.g., immersive online shopping scenario in Table 2.1).

6.6.3.2 Opportunistic CPU Offloading

As the app runs more DNNs in parallel, the computational complexity may exceed the mobile GPU capabilities. In such a case, GPU contention would degrade the overall utilities of the DNNs, possibly making it impossible to satisfy the app requirements. The coordinator periodically determines if some DNNs should be offloaded to the CPU to reduce the GPU contention level.

Let $P_1, P_2, ..., P_N$ denote the processor (GPU or CPU) the N DNNs are running on. The processor mapping is determined by solving the following problem,

$$\max_{P_1,P_2,...,P_N} \quad \textstyle\sum_{t=1}^{N} U_{D_i,P_i}(t), \tag{6.7}$$

where $U_{D_i,P_i}(t)$ denotes the utility of D_i running on processor P_i (affected by the inference time on P_i, which is profiled by the analyzer). As changing the target processor (i.e., allocating memory for the model weights and feature maps) incurs around 50 ms latency in MACE, we reconfigure the mapping at every 1-second interval.

6.5.4 Greedy Scheduling Algorithm

Solving the above scheduling problem is computationally difficult, as well as infeasible to plan offline (as the solution varies depending on scene contents). Thus, we solve it in a greedy manner to obtain an approximate solution.

GPU Coordination. For each scheduling event, the coordinator first checks how many partitions are left to execute for each DNN. Based on the profiled latencies of the remaining partitions, the coordinator checks if the inference can finish within the time left before its deadline; in case a DNN is not expected to finish within the deadline, the coordinator runs it immediately. If otherwise, the coordinator determines which DNN to execute based on their current utility values. Specifically, the MaxMinUtility policy selects a DNN with the current lowest utility. The MaxTotalUtility policy iteratively computes the expected sum of utilities at the current scheduling event assuming that a specific DNN chunk is executed, and selects the chunk which maximizes the sum (without consideration of the future). Specifically, the utility sum is estimated by adding the latency delay equal to the scheduling interval to the latency utility of the DNNs that are not chosen, so as to reflect the additional latency delay due to the execution of another DNN.

CPU Offloading. Among the DNNs running on GPU, the coordinator picks the DNN experiencing the highest latency and offloads it to CPU if the profiled CPU inference time is (1+m)$\times$ smaller than the current latency on GPU (m is a positive margin to avoid ping-pong effect between CPU and GPU); per each scheduling event, only one DNN is offloaded to the CPU. If no DNN is offloaded, the coordinator also checks whether it should bring a DNN on CPU back to GPU. Similarly, a DNN is reloaded

to GPU if its inference time on CPU is (1+m)$\times$ larger than its last inference time on GPU.

6.6 Additional Optimizations

The end-to-end inference pipeline for every DNN involves several steps that need to be executed on the CPU: i) preprocessing the input image before the inference, ii) postprocessing the inference output to an adequate form, and iii) ops in the model that are unsupported by the GPU backend of the mobile deep learning framework and needed to be executed on CPU. Granting GPU access to a DNN that currently needs to run such steps incurs unwanted GPU idle time, slowing down the overall inference latency. This becomes especially significant when processing high-resolution complex scene images. For example, RetinaFace [17] detector with inference pipeline shown in Figure 6.9 spends 106 out of 287 ms total inference time on CPU to process a 1080p image with 20 faces. To enhance GPU utilization, we parallelize the following components.

6.6.1 Preprocessing and postprocessing

Before enqueueing a DNN inference to the task queue for the Pseudo-Preemptive GPU Coordinator to schedule, we run the following steps in parallel with other DNN inference running on the GPU), so that the DNN can fully occupy the GPU when given the access from the coordinator.

Preprocessing. The preprocessing steps involve resizing the input frame (RGB byte array) to the DNN's input size, converting it to float array, and scaling the pixel values (e.g., from [0,255] to [-1,1]).

Postprocessing. The postprocessing steps involve converting the inference output to task-specific forms. For example, face detection requires converting the output feature map to bounding boxes and performing non-maximum suppression to filter out

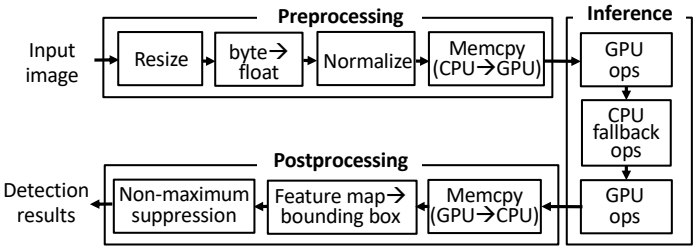


Figure 6.9: End-to-end DNN inference pipeline example for RetinaFace detector.

redundant ones.

6.6.2 CPU Fallback Operators

GPU backend of a mobile deep learning framework typically supports only a limited number of ops (i.e., a subset of the ops supported in the cloud framework). In case an op is unsupported by the GPU backend, it falls back to CPU for execution. We identify the CPU fallback op indexes of a DNN at the profiling stage and run them in parallel with other DNNs at runtime. Note that CPU fallback occurs frequently, especially for complex state-of-the-art DNNs. For example, TF-Lite does not support tf.image.resize() required in feature pyramid network [28], which most state-of-the-art object detectors rely on for detecting small objects. Similarly, MACE does not support common ops such as tf.crop(), tf.stack().

6.7 Implementation

We implement Heimdall by extending MACE [36], an OpenCL-based mobile deep learning framework, to partially run a subset of the ops in the DNN at a time by modifying MaceEngine.Run() (and underlying functions) to MaceEngine.RunPartial(startIdx, endIdx). We use OpenCV Android SDK 3.4.3 for camera and image processing. We evaluate Heimdall on two commodity smartphones: LG V50 (Qualcomm Snapdragon 855 SoC, Adreno 640 GPU) running on Android 10.0.0 and 9.0.0, and Google Pixel 3 XL (Snapdragon 845 SoC, Adreno 630 GPU) running on Android 9.0.0. We also used

two different vendor-provided OpenCL libraries obtained from LG V50 and Google Pixel 2 ROMs. We achieved consistent results across different settings, and report the best results on LG V50.

We choose the DNNs with sufficient model accuracy for the evaluation, implement and port them on MACE (the list is summarized in Table 2.2). We implement RetinaFace [17], ArcFace [18], EAST [13], PoseNet [21] using TensorFlow 1.12.0. For MobileNet-v1 [25], CPM [24], and StyleTransfer [22], we use the models provided in the MACE model zoo [178]. For DeepLab-v3 [19] and YOLO-v2 [20], we use the pre-trained models from the original authors.

6.8 Evaluation

6.8.1 Experiment Setup

Scenarios. We evaluate Heimdall for 3 scenarios in Table 2.1 with the DNNs in Table 2.2: immersive online shopping, augmented interactive workspace, and AR emoji.

Evaluation Metrics.

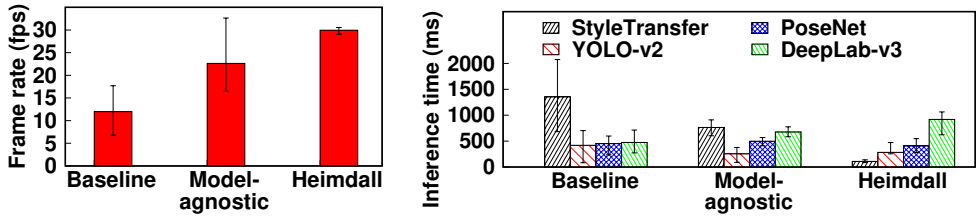
• **Rendering Frame Rate:** the number of frames rendered on the screen, measured every 1/3 seconds.

• **Inference Latency:** the time interval between when the DNN inference is enqueued to the coordinator (after preprocessing), and when the last op of the model is executed. While we omitted pre/postprocessing latency to evaluate only the GPU contention coordination performance, end-to-end latency can also be enhanced as we parallelize such steps as well (Chapter 6.6).

Comparison Schemes.

• **Baseline MACE** creates multiple MaceEngine instances (one per each DNN) in separate threads and runs multi-DNN and rendering tasks in parallel without any coordination.

• **Model-Agnostic DNN Partitioning** executes 5 ops of a DNN at a time (regardless



(a) Rendering frame rate.

(b) DNN inference latency.

Figure 6.10: Performance overview of Heimdall on LG V50.

of the model or rendering requirements). This is supported in MACE to enhance UI responsiveness by preventing DNNs from occupying the GPU for too long, implemented by invoking cl::Event.wait() after 5 clEnqueueNDRangeKernel() calls.

6.8.2 Performance Overview

We first evaluate Heimdall with the MaxTotalUtility policy on immersive online shopping scenario compared with alternatives. The app requirements are set to render frames at 30 fps, run segmentation (DeepLab-v3) and hand tracking (PoseNet) at 1 and 2 fps, respectively. Image style transfer (StyleTransfer) is set to have higher priority than others to satisfy the low response time requirement.

Figure 6.10(a) shows the rendering performance, where the error bar denotes the minimum and maximum frame rates. Heimdall supports a stable 29.96 fps rendering performance, whereas the baseline suffers from low and fluctuating frame rate (6.82-17.70 fps, 11.99 on average). While the model-agnostic partitioning slightly enhances the frame rate, it still suffers from fluctuation due to the uncoordinated execution of DNNs and rendering.

Figure 6.10(b) shows the DNN latency results, where the error bar denotes the minimum and maximum inference latencies. Overall, Heimdall efficiently coordinates the DNNs to satisfy the app requirements: StyleTransfer, PoseNet, and DeepLab-v3 run at 109, 409, 919 ms on average, respectively (maximum 139, 548, 1064 ms), while the worst-case inference latency of StyleTransfer is also reduced by 14.92× (from 2074 to 139 ms). This is achieved by i) giving preemptive access to StyleTransfer, ii) running DeepLab-v3 at the minimum and PoseNet more frequently to satisfy the latency con-

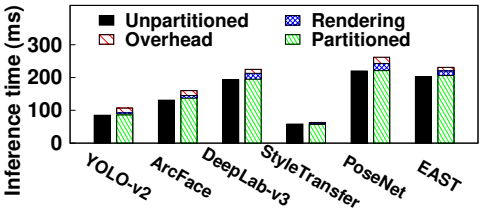
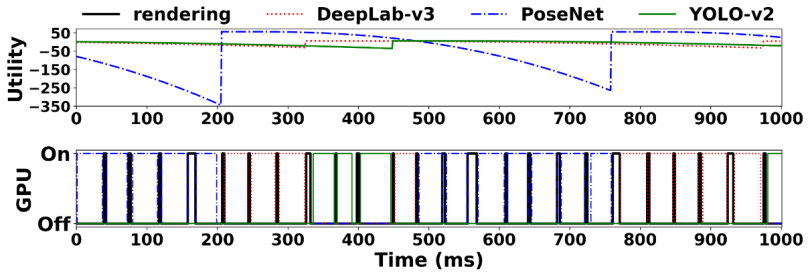


Figure 6.11: DNN partitioning overhead.

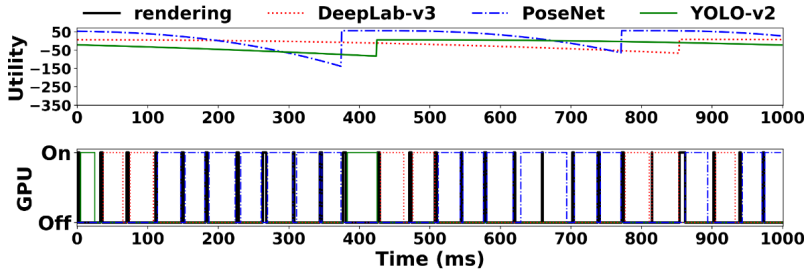
straints of both tasks, and iii) offloading YOLO-v2 to CPU to reduce GPU contention level (which also benefits YOLO-v2). Baseline and model-agnostic partitioning that cannot support such coordination fail to satisfy the app requirements, especially for StyleTransfer which is more vulnerable to GPU contention due to several CPU fallback ops as analyzed in Chapter 6.2.

6.8.3 DNN Partitioning/Coordination Overhead

Next, we evaluate the DNN partitioning and coordination overhead on inference latency when executed with 1080p camera frame rendering at 30 fps. Figure 6.11 shows that the total GPU latency of the partitioned DNN chunks remain almost identical to unpartitioned inference latency, as Preemption-Enabling DNN Analyzer tries to pack as many ops as possible. The remaining overhead other than the rendering latency includes multiple factors, including the GPU idle time due to DNN chunks that do not perfectly fit into the rendering interval, scheduling algorithm solver, and logging process for the evaluation (this is negligible on runtime). Most importantly, our current implementation is limited to coordinating multiple DNN inferences on CPU (due to fallback or offloading) on different cores; other tasks (e.g., camera, pre/postprocessing steps) may interfere and cause latency overhead. We plan to handle the issue in our future work for further optimization.



(a) MaxMinUtility.



(b) MaxTotalUtility.

Figure 6.12: Performance comparison of GPU coordination policies.

6.8.4 Pseudo-Preemptive GPU Coordinator

GPU Coordination Policy. Figure 6.12 shows how the 3 DNNs in the immersive online shopping scenario are coordinated (i.e., utility over time and GPU occupancy) on the GPU under two policies in Chapter 6.5.3. Figure 6.12(a) shows that the MaxMinUtility policy executes a DNN with the currently lowest utility and enables a fair resource allocation between the 3 DNNs. Figure 6.12(b) shows that MaxTotalUtility policy favors PoseNet which has higher priority than others (i.e., higher $L^0_{D_i}$ and γ_i in Equation (6.3), meaning that the utility is higher when the inference is enqueued but decays rapidly over time) to maximize the total utility. As a result, the utility of PoseNet remains higher than that under the MaxMinUtility policy.

Opportunistic CPU Offloading. Next, we incorporate the opportunistic CPU offloading in the same setting as in Figure 6.12(a). Figure 6.13 shows the GPU/CPU occupancy and utility over time for the 3 DNNs. When CPU offloading is triggered at around t=1600 ms, YOLO-v2 (which had the least priority and thus had been exe-

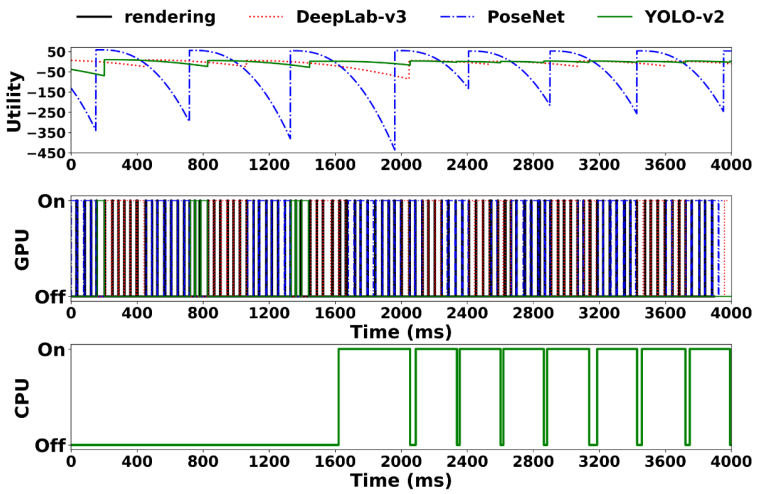
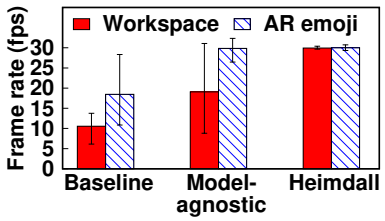


Figure 6.13: Opportunistic CPU offloading performance.

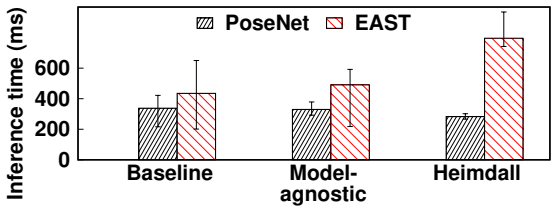
cuted sporadically) is offloaded to CPU. This benefits the other two DNNs on GPU as the contention level decreases (notice that the utility of PoseNet becomes higher after CPU offloading), while YOLO-v2 also benefits as it experiences faster inference latency as compared to when it was contending with the other two DNNs on GPU.

6.8.5 Performance for Various App Scenarios

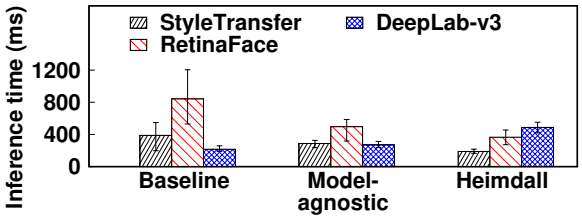
Figure 6.14 shows the performance of Heimdall on two different scenarios: augmented interactive workspace and AR emoji. Overall, we observe consistent results. Figure 6.14(a) shows that Heimdall enables higher and stable rendering frame rate. Figure 6.14(b) shows that for the interactive workspace scenario, Heimdall coordinates the two DNNs by offloading the text detection (EAST) to the CPU so that the hand tracking (PoseNet) can run more frequently on the GPU. However, the latency gain is not as high as expected due to the scheduling overhead caused by multiple concurrent CPU tasks. Finally, Figure 6.14(c) shows that for the AR emoji scenario, Heimdall prioritizes StyleTransfer to guarantee low inference latency, while balancing the latencies between RetinaFace and DeepLab-v3.



(a) Rendering frame rate.



(b) DNN inference latency (interactive workspace).



(c) DNN inference latency (AR emoji).

Figure 6.14: Performance of Heimdall for other AR app scenarios.

Table 6.1: Face detection and person segmentation accuracy (IoU) for the AR emoji scenario.

Baseline		Model-agnostic		Heimdall	
Bounding box	Mask	Bounding box	Mask	Bounding box	Mask
0.52±0.12	0.93±0.02	0.57±0.12	0.92±0.02	0.63±0.11	0.90±0.03

6.8.6 DNN Accuracy

We evaluate the impact of Heimdall on DNN accuracy for the AR emoji scenario. For repeatable evaluation, we sample 5 videos of a single talking person from the 300-VW dataset [179]. As the dataset does not provide the face bounding box and person segmentation mask labels, we run our DNNs on every frame and use the results as ground truth to be compared with the runtime detection results. Table 6.1 shows the detection accuracy in terms of mean Intersection over Union (IoU). For baseline multi-threading, face detection accuracy remains low, as RetinaFace (with several CPU fallback ops) runs at only ≈1 fps due to contention with DeepLab-v3 (Figure 6.14(c)).

While model-agnostic partitioning alleviates the issue, it cannot coordinate the two DNNs. With Heimdall, we can flexibly run RetinaFace more frequently (≈3 fps) to improve the face detection accuracy at the cost of relatively smaller loss in the segmentation accuracy. Note that the performance gain came from utilizing the app-specific content characteristics (i.e., the face moves more rapidly than the body). For other app scenarios, we can similarly take into account the target scene content characteristics to coordinate multiple DNNs and improve the overall accuracy.

6.8.7 Energy Consumption Overhead

Finally, we report the impact of Heimdall on energy consumption. We use Qualcomm Snapdragon Profiler [180] to measure the system-level energy consumption. For all the three evaluated app scenarios, baseline multi-threading consumes 4.8–5.1 W, mostly coming from the ≈100% GPU utilization which is known to be the dominant source of mobile SoC energy consumption [181] (capturing 1080p camera frames and rendering them on screen without any DNN running consumes 1.9–2.3 W). Similarly, the GPU utilization in Heimdall remains ≈100% and consumes 5.1-5.2 W. The slight increase in the energy consumption comes from the additional CPU tasks coming from the increased frame rate and the scheduling overhead of the Pseudo-Preemptive GPU coordinator.

Chapter 7

Conclusion

7.1 Summary

In this dissertation, we depicted emerging live video analytics app scenarios and characterized their workload. We then analyzed the technical challenges in realizing them, and introduced our research vision and systems to develop an end-to-end edge-cloud cooperative platform to support the workload.

We first designed EagleEye, a wearable camera-based system to identify missing person(s) in large, crowded urban spaces in real-time. To further innovate the performance of the state-of-the-art face identification techniques on LR face recognition, we designed a novel ICN and a training methodology that utilize the probes of the target to recover missing facial details in the LR faces for accurate recognition. We also develop Content-Adaptive Parallel Execution to run the complex multi-DNN face identification pipeline at low latency using heterogeneous processors on mobile and cloud. Our results show that ICN significantly enhances LR face recognition accuracy (true positive by 78% with only 14% false positive), and EagleEye accelerates the latency by 9.07$\times$ with only 108 KBytes of data offloaded to the cloud.

We next designed Pendulum, an end-to-end live video analytics system with network-compute joint scheduling. To overcome the limitations of single-stage scheduling sys-

tems in alternating resource bottleneck scenarios, we newly discover the tradeoff relationship between the video bitrate and DNN complexity. Leveraging this, we design an end-to-end system composed of (i) an efficient and scalable knob control mechanism, (ii) a lightweight tradeoff profiler, and (iii) a multi-user joint resource scheduler. Extensive evaluation shows that Pendulum achieves up to 0.64 mIoU gain (from 0.17 to 0.81) and 1.29× higher throughput compared to state-of-the-art single-stage scheduling systems.

Finally, we designed Heimdall, a mobile GPU coordination platform for emerging AR apps. To coordinate multi-DNN and rendering tasks, the Preemption-Enabling DNN Analyzer partitions the DNN into smaller units to enable fine-grained GPU time-sharing with minimal DNN inference latency overhead. Furthermore, the Pseudo-Preemptive GPU Coordinator flexibly prioritizes and schedules the multi-DNN and rendering tasks on GPU and CPU to satisfy the app requirements. Heimdall efficiently supports multiple AR app scenarios, enhancing the frame rate from 11.99 to 29.96 fps while reducing the worst-case DNN inference latency by up to ≈15 times compared to the baseline multi-threading approach.

7.2 Discussion

7.2.1 Extension to Other Workloads

The workload of many future multi-DNN-enabled live video analytics applications is similar to EagleEye in that they require running a series of complex DNNs repetitively to detect objects in a high-resolution scene image and analyze each identified instance (e.g., text identification, pedestrian identification, etc.). For such applications, our Content-Adaptive Parallel Execution can be generally adapted to enhance performance by applying different pipeline depending on the content and parallelizing the execution over heterogeneous processors on mobile and cloud.

7.2.2 Robustness to Wider Network and System Environments

Knob Choices Under Other System Goals. We choose the knobs mainly to maximize control independency and minimize knob preparation overhead; other knob choices can also be feasible under different goals. For example, to minimize GPU memory overhead for edge device deployment, quantization can be considered with some model preparation efforts. When handling a wide range of network bandwidths including extreme bottleneck (e.g., < 1 Mbps for LPWAN-based disaster monitoring [182]), enhancement can be an adequate knob.

Joint Scheduling in Edge-Cloud Collaborative Inference Systems. Joint scheduling can be extended to a collaborative inference context, i.e., across (i) on-device, (ii) network, and (iii) cloud stages. For example, in case of a compute bottleneck, the mobile device can partially run the DNN inference workload and offload the remaining to the cloud (e.g., partial RoIs [3,97] or DNN intermediate features [10,98,99]), which can be jointly scheduled at the network and the cloud stages.

7.2.3 Impact of Hardware Evolution

7.2.1.1 Mobile GPU Evolution

Even when mobile GPUs evolve similar to desktop GPUs, the need for an app-aware coordination platform to dynamically schedule multiple tasks to satisfy the AR app requirements will persist.

Parallelization. With the architecture support, we can consider porting desktop GPU computing platforms (e.g., recent CUDA for ARM server platforms [183]) and spatially partitioning the GPU to run multi-DNN and rendering tasks concurrently. However, due to a limited number of computing cores and power of mobile GPUs (e.g., RTX 2080Ti: 13.45 TFLOPs vs. Adreno 640: 954 GFLOPs), static partitioning would be limited in running multiple compute-intensive DNNs. Instead, a coordinator should dynamically allocate resources at runtime; when an inference request for a heavy DNN

with high priority is enqueued, the coordinator should allocate more number of partitioned resources dynamically to minimize response time.

Preemption. With fine-grained, near-zero overhead preemption support (e.g., NVIDIA Pascal GPUs [184] support instruction-level preemption at 0.1 ms scale [185]), we can consider employing prior multi-DNN scheduling for desktop GPUs [67,69]. However, prior works mostly assume that the task priorities are fixed in advance, whereas in AR apps they can be dynamic depending on the scene contents (e.g., in the surroundings monitoring scenario, face detection would need to run more frequently than object detection in case there are many people). Therefore, a coordinator would be needed to dynamically adjust priorities at runtime for app usability.

7.2.2.2 Emergence of NPUs/TPUs

Recently, neural processors are being embedded in mobile devices (e.g., Google Pixel 4 edge TPU [186], Huawei Kirin NPU [187]). Such processors maximize computing power by packing a large number of cores specialized for DNN inference. For example, Google TPUs employ 128×128 systolic array-based matrix units (MXUs), which accelerate matrix multiplication by hard-wired calculation without memory access. We envision that our Pseudo-Preemption mechanism can also be useful in coordinating multiple tasks on such neural processors, as i) it is challenging to preempt the hard-wired MXUs, and ii) context switch overhead on bandwidth-limited mobile SoCs can be more costly due to larger state sizes than GPUs.

7.3 Future Works

7.3.1 Joint Scheduling Extension to App-RAN Cross-Layer Control

Pendulum currently takes the network-as-a-black-box approach (i.e., app-level bandwidth estimation and bitrate control). We plan to extend joint scheduling to app-RAN (Radio Access Network) cross layer control. Specifically, we expect the following

scheduling gains if the platform operator has control over both the RAN (e.g., private 5G [147, 148]) and the cloud server. *i) Better Scheduling Accuracy and Cost Reduction.* The RAN can provide a more accurate estimate of the network bandwidth to the cloud server based on the monitored channel status at the physical layer (e.g., uplink SINR) [188]. The RAN can also dynamically allocate bandwidth (Resource Blocks, RBs) across users considering their video content and network-compute tradeoffs (e.g., higher bandwidth to users with more dynamic scenes) to reduce overall cost. For example, our preliminary experiment with real-world video traces from BDD and MOT datasets shows that app-aware RB scheduling reduces the overall compute cost by up to 52% compared to app-agnostic equal RB scheduling. *ii) Per-Frame Latency Guarantee.* By jointly scheduling the RB transmission order as well as the GPU inference order, we can improve per-frame latency predictability and latency SLO satisfaction ratio. For example, in case a user's frame transmission is unexpectedly delayed due to SINR fluctuation, we can prioritize his DNN inference at the cloud server to compensate for the delay and meet the latency deadline. We plan to integrate such cross-layer control in our future work.

7.3.2 System Support for 3D Point Cloud Videos

Our current platform is mostly designed for high-resolution RGB videos. We plan to extend our platform support for more sensor modalities (e.g., RGB-D, LiDAR) for diverse live video analytics applications. As an initial attempt, we are designing an end-to-end live video analytics platform for 3D point cloud videos. 3D point cloud analysis add a new dimension of depth perception, which is crucial for various live video analytics apps (e.g., autonomous driving, indoor robot navigation). Especially, processing the 3D point cloud yields higher accuracy than 2D projection with lower number of FLOPs (e.g., 5% higher accuracy with 7$\times$ small MACs [189]). However, streaming and analyzing 3D point cloud is challenging due to its large data size. For example, our preliminary study shows that the end-to-end Octree-based streaming and

3D object detection pipeline takes ≈800 ms for 110 K points LiDAR frame. Fast and efficient 3D point cloud is also non-trivial, mainly due to its high data sparsity. Despite the challenges, we identified a key opportunity for optimization: objects only compose ≈10% for typical 3D point cloud frames. We are currently developing a system that efficiently selects among multiple sampling features (e.g., edge filtering, resolution scaling, temporal tracking) to efficiently sample high-saliency points (and thus the streaming and analysis latency) from the input point cloud without accuracy drop.

Bibliography

[1] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 377–392.

[2] X. Liu, P. Ghosh, O. Ulutan, B. Manjunath, K. Chan, and R. Govindan, "Caesar: cross-camera complex activity recognition," in *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, 2019, pp. 232–244.

[3] J. Yi, S. Choi, and Y. Lee, "EagleEye: Wearable camera-based person identification in crowded urban spaces," in *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking*. ACM, 2020.

[4] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[5] J. Yi and Y. Lee, "Heimdall: mobile gpu coordination platform for augmented reality applications," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–14.

[6] Z. Li, M. Annett, K. Hinckley, K. Singh, and D. Wigdor, "HoloDoc: Enabling mixed reality workspaces that harness physical and digital content," in *Proceed-*

ings of the 2019 CHI Conference on Human Factors in Computing Systems, 2019, pp. 1–14.

[7] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 197–210.

[8] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 325–338.

[9] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012, pp. 97–108.

[10] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane, "Spinn: synergistic progressive inference of neural networks over device and cloud," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, 2020, pp. 1–15.

[11] "Samsung Galaxy S9 AR Emoji," https://www.sammobile.com/news/galaxy-s9-ar-emoji-explained-how-to-create-and-use-them/. Accessed: 25 Mar. 2020.

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

[13] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "EAST: an efficient and accurate scene text detector," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2017, pp. 5551–5560.

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

[15] X. Zeng, K. Cao, and M. Zhang, "MobileDeepPill: A small-footprint mobile deep learning system for recognizing unconstrained pill images," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 56–67.

[16] 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.

[17] J. Deng, J. Guo, Y. Zhou, J. Yu, I. Kotsia, and S. Zafeiriou, "RetinaFace: Single-stage dense face localisation in the wild," *arXiv preprint arXiv:1905.00641*, 2019.

[18] J. Deng, J. Guo, N. Xue, and S. Zafeiriou, "ArcFace: Additive angular margin loss for deep face recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4690–4699.

[19] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam, "Encoder-decoder with atrous separable convolution for semantic image segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 801–818.

[20] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *IEEE CVPR*, 2017.

[21] C. Zimmermann and T. Brox, "Learning to estimate 3d hand pose from single rgb images," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 4903–4911.

[22] L. Engstrom, "Fast style transfer," https://github.com/lengstrom/fast-style-transfer/, 2016.

[23] "Microsoft HoloLens 2," https://www.microsoft.com/en-us/hololens/. Accessed: 25 Mar. 2020.

[24] S.-E. Wei, V. Ramakrishna, T. Kanade, and Y. Sheikh, "Convolutional pose machines," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4724–4732.

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

[26] 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.

[27] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[28] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 2117–2125.

[29] X. Xie and K.-H. Kim, "Source compression with bounded dnn perception loss for iot edge computer vision," in *The 25th Annual International Conference on Mobile Computing and Networking*, 2019, pp. 1–16.

[30] "Ultralytics YOLO-v5," https://github.com/ultralytics/yolov5. Accessed: 15 Feb. 2023.

[31] L. Leal-Taixé, A. Milan, I. Reid, S. Roth, and K. Schindler, "Motchallenge 2015: Towards a benchmark for multi-target tracking," *arXiv preprint arXiv:1504.01942*, 2015.

[32] X. Zeng, B. Fang, H. Shen, and M. Zhang, "Distream: scaling live video analytics with workload-adaptive distributed edge intelligence," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 409–421.

[33] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[34] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate Record-and-Replay for HTTP," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 417–429.

[35] "TensorFlow-Lite on GPU for Mobile," https://www.tensorflow.org/lite/performance/gpu_advanced. Accessed: 15 Dec. 2019.

[36] "XiaoMi Mobile AI Compute Engine (MACE)," https://github.com/XiaoMi/mace. Accessed: 25 Mar. 2020.

[37] P. Hu and D. Ramanan, "Finding tiny faces," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 951–959.

[38] 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.

[39] 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.

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

[41] K. Chen, T. Li, H.-S. Kim, D. E. Culler, and R. H. Katz, "MARVEL: Enabling mobile augmented reality with low energy and low latency," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2018, pp. 292–304.

[42] A. Padmanabhan, N. Agarwal, A. Iyer, G. Ananthanarayanan, Y. Shu, N. Karianakis, G. H. Xu, and R. Netravali, "Gemel: Model merging for memory-efficient, real-time video analytics at the edge," in *USENIX NSDI*, April 2023.

[43] F. Cangialosi, N. Agarwal, V. Arun, S. Narayana, A. Sarwate, and R. Netravali, "Privid: Practical,{Privacy-Preserving} video analytics queries," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 209–228.

[44] R. Bhardwaj, Z. Xia, G. Ananthanarayanan, J. Jiang, Y. Shu, N. Karianakis, K. Hsieh, P. Bahl, and I. Stoica, "Ekya: Continuous learning of video analytics models on edge compute servers," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 119–135.

[45] K. Mehrdad, G. Ananthanarayanan, K. Hsieh, J. J. , R. N. , Y. Shu, M. Alizadeh, and V. Bahl, "Recl: Responsive resource-efficient continuous learning for video analytics," in *USENIX NSDI*, April 2023.

[46] "Alibaba Mobile Neural Network (MNN)," https://github.com/alibaba/MNN. Accessed: 25 Mar. 2020.

[47] "Qualcomm Neural Processing SDK for AI," https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk. Accessed: 25 Mar. 2020.

[48] S. Bhattacharya and N. D. Lane, "Sparsifying deep learning layers for constrained resource inference on wearables," in *Proc. ACM SenSys*, 2016.

[49] N. D. Lane, P. Georgiev, and L. Qendro, "DeepEar: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 283–294.

[50] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2018, pp. 278–291.

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

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

[53] R. Lee, S. I. Venieris, L. Dudziak, S. Bhattacharya, and N. D. Lane, "MobiSR: Efficient on-device super-resolution through heterogeneous mobile processors," in *The 25th Annual International Conference on Mobile Computing and Networking*. ACM, 2019, p. 54.

[54] 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.

[55] S. Lee and S. Nirjon, "Fast and scalable in-memory deep multitask learning via neural weight virtualization," in *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, 2020, pp. 175–190.

[56] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D. G. Andersen, and G. R. Ganger, "Mainstream: Dynamic stem-sharing for multi-tenant video processing," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, 2018, pp. 29–42.

[57] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 193–204.

[58] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 593–606, 2015.

[59] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 358–369.

[60] "NVIDIA Hyper-Q," http://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf. Accessed: 25 Mar. 2020.

[61] "NVIDIA GPU virtualization," https://www.nvidia.com/ko-kr/data-center/graphics-cards-for-virtualization/. Accessed: 25 Mar. 2020.

[62] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 407–418.

[63] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU resource utilization through alternative thread block scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 260–271.

[64] B. Wu, G. Chen, D. Li, X. Shen, and J. Vetter, "Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015, pp. 119–130.

[65] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu, "MASK: Redesigning the GPU memory hierarchy to support multi-application concurrency," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 503–518.

[66] Z. Fang, D. Hong, and R. K. Gupta, "Serving deep neural networks at the cloud edge for vision applications on mobile platforms," in *Proceedings of the 10th ACM Multimedia Systems Conference*, 2019, pp. 36–47.

[67] H. Zhou, S. Bateni, and C. Liu, "S^3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *2018 IEEE Real-Time and*

Embedded Technology and Applications Symposium (RTAS). IEEE, 2018, pp. 190–201.

[68] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2019, pp. 305–317.

[69] Y. Xiang and H. Kim, "Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference," in *IEEE RTSS*, 2019.

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

[71] 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.

[72] P. Jain, J. Manweiler, and R. Roy Choudhury, "Low bandwidth offload for mobile ar," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. ACM, 2016, pp. 237–251.

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

[74] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM, 2019.

[75] J. Hu, A. Shearer, S. Rajagopalan, and R. LiKamWa, "Banner: An image sensor reconfiguration framework for seamless resolution-based tradeoffs," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2019, pp. 236–248.

[76] 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.

[77] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu, "Focus: Querying large video datasets with low latency and low cost," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 269–286.

[78] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, "Noscope: optimizing neural network queries over video at scale," *arXiv preprint arXiv:1703.02529*, 2017.

[79] C. Canel, T. Kim, G. Zhou, C. Li, H. Lim, D. G. Andersen, M. Kaminsky, and S. Dulloor, "Scaling video analytics on constrained edge nodes," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 406–417, 2019.

[80] T. Zhang, A. Chowdhery, P. Bahl, K. Jamieson, and S. Banerjee, "The design and implementation of a wireless video surveillance system," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, 2015, pp. 426–438.

[81] C. Pakha, A. Chowdhery, and J. Jiang, "Reinventing video streaming for distributed vision analytics," in *10th USENIX workshop on hot topics in cloud computing (HotCloud 18)*, 2018.

[82] M. Xu, T. Xu, Y. Liu, and F. X. Lin, "Video analytics with zero-streaming cameras," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 459–472.

[83] 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.

[84] W. Zhang, Z. He, L. Liu, Z. Jia, Y. Liu, M. Gruteser, D. Raychaudhuri, and Y. Zhang, "Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, 2021, pp. 201–214.

[85] 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.

[86] Y. Wang, W. Wang, J. Zhang, J. Jiang, and K. Chen, "Bridging the edge-cloud barrier for real-time advanced vision analytics," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[87] H. Yeo, Y. Jung, J. Kim, J. Shin, and D. Han, "Neural adaptive content-aware internet video delivery," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 645–661.

[88] J. Kim, Y. Jung, H. Yeo, J. Ye, and D. Han, "Neural-enhanced live streaming: Improving live video ingest via online learning," 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. 107–125.

[89] K. Du, Q. Zhang, A. Arapin, H. Wang, Z. Xia, and J. Jiang, "Accmpeg: Optimizing video encoding for accurate video analytics," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 450–466, 2022.

[90] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 1–17.

[91] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.

[92] R. LiKamWa and L. Zhong, "Starfish: Efficient concurrency support for computer vision applications," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 213–226.

[93] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "$pretzel$: Opening the black box of machine learning prediction serving systems," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 611–626.

[94] H. Guo, S. Yao, Z. Yang, Q. Zhou, and K. Nahrstedt, "Crossroi: cross-camera region of interest optimization for efficient real time video analytics at scale," in

Proceedings of the 12th ACM Multimedia Systems Conference, 2021, pp. 186–199.

[95] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, P. Bahl, and J. Gonzalez, "Spatula: Efficient cross-camera video analytics on large camera networks," in *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2020, pp. 110–124.

[96] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.

[97] J. Yi, S. Kim, J. Kim, and S. Choi, "Supremo: Cloud-assisted low-latency super-resolution in mobile devices," *IEEE Transactions on Mobile Computing*, 2020.

[98] 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.

[99] S. Yao, J. Li, D. Liu, T. Wang, S. Liu, H. Shao, and T. Abdelzaher, "Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency," in *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*, 2020, pp. 476–488.

[100] L. Yu and W. Xiang, "X-pruner: explainable pruning for vision transformers," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 24 355–24 363.

[101] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "Smoothquant: Accurate and efficient post-training quantization for large language models," in

International Conference on Machine Learning. PMLR, 2023, pp. 38 087–38 099.

[102] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[103] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[104] P. Guo, B. Hu, and W. Hu, "Mistify: Automating {DNN} model porting for {On-Device} inference at the edge," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 705–719.

[105] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.

[106] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 10 781–10 790.

[107] W. Liu, Y. Wen, Z. Yu, M. Li, B. Raj, and L. Song, "SphereFace: Deep hypersphere embedding for face recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 212–220.

[108] H. Wang, Y. Wang, Z. Zhou, X. Ji, D. Gong, J. Zhou, Z. Li, and W. Liu, "CosFace: Large margin cosine loss for deep face recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5265–5274.

[109] J. Han and B. Bhanu, "Individual recognition using gait energy image," *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 2, pp. 316–322, 2005.

[110] H. Wang, X. Bao, R. Roy Choudhury, and S. Nelakuditi, "Visually fingerprinting humans without face recognition," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2015, pp. 345–358.

[111] J. Chauhan, Y. Hu, S. Seneviratne, A. Misra, A. Seneviratne, and Y. Lee, "BreathPrint: Breathing acoustics-based user authentication," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 278–291.

[112] K. R. Farrell, R. J. Mammone, and K. T. Assaleh, "Speaker recognition using neural networks and conventional classifiers," *IEEE Transactions on speech and audio processing*, vol. 2, no. 1, pp. 194–205, 1994.

[113] Y. Zhao, S. Wu, L. Reynolds, and S. Azenkot, "A face recognition application for people with visual impairments: Understanding use beyond the lab," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, p. 215.

[114] S. Panchanathan, S. Chakraborty, and T. McDaniel, "Social interaction assistant: a person-centered approach to enrich social interactions for individuals with visual impairments," *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 5, pp. 942–951, 2016.

[115] L. B. Neto, F. Grijalva, V. R. M. L. Maike, L. C. Martini, D. Florencio, M. C. C. Baranauskas, A. Rocha, and S. Goldenstein, "A kinect-based wearable face recognition system to aid visually impaired users," *IEEE Transactions on Human-Machine Systems*, vol. 47, no. 1, pp. 52–64, 2016.

[116] L. He, H. Li, Q. Zhang, and Z. Sun, "Dynamic feature learning for partial face recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 7054–7063.

[117] J. Lezama, Q. Qiu, and G. Sapiro, "Not afraid of the dark: NIR-VIS face recognition via cross-spectral hallucination and low-rank embedding," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 6628–6637.

[118] "TensorFlow-Lite Object Detection Demo," https://www.tensorflow.org/lite/models/object_detection/overview. 15 Dec. 2019.

[119] P. Li, L. Prieto, D. Mery, and P. J. Flynn, "On low-resolution face recognition in the wild: Comparisons and new techniques," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 8, pp. 2000–2012, 2019.

[120] M. B. Lewis and A. J. Edmonds, "Face detection: Mapping human performance," *Perception*, vol. 32, no. 8, pp. 903–920, 2003.

[121] M. Kampf, I. Nachson, and H. Babkoff, "A serial test of the laterality of familiar face recognition," *Brain and cognition*, vol. 50, no. 1, pp. 35–50, 2002.

[122] Y. Guo, L. Zhang, Y. Hu, X. He, and J. Gao, "MS-Celeb-1M: A dataset and benchmark for large-scale face recognition," in *European Conference on Computer Vision*. Springer, 2016, pp. 87–102.

[123] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman, "VGGFace2: A dataset for recognising faces across pose and age," in *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*. IEEE, 2018, pp. 67–74.

[124] X. Tang, D. K. Du, Z. He, and J. Liu, "Pyramidbox: A context-assisted single shot face detector," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 797–813.

[125] "EyeSight Rapter AR Glass," https://everysight.com/about-raptor/. Accessed: 15 Dec. 2019.

[126] M. Najibi, P. Samangouei, R. Chellappa, and L. S. Davis, "SSH: Single stage headless face detector," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 4875–4884.

[127] Y. Chen, Y. Tai, X. Liu, C. Shen, and J. Yang, "FSRNet: End-to-end learning face super-resolution with facial priors," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2492–2501.

[128] R. Zhang, "Making convolutional networks shift-invariant again," *International Conference on Machine Learning (ICML)*, 2019.

[129] B. Lim, S. Son, H. Kim, S. Nah, and K. Mu Lee, "Enhanced deep residual networks for single image super-resolution," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 136–144.

[130] N. Ahn, B. Kang, and K.-A. Sohn, "Fast, accurate, and lightweight super-resolution with cascading residual network," in *Proc. ECCV*, 2018.

[131] A. Bulat and G. Tzimiropoulos, "Super-FAN: Integrated facial landmark localization and super-resolution of real-world low resolution faces in arbitrary poses with gans," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 109–117.

[132] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[133] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville, "Improved training of wasserstein gans," in *Advances in Neural Information Processing Systems*, 2017, pp. 5767–5777.

[134] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. IEEE, 2009, pp. 248–255.

[135] J. Canny, "A computational approach to edge detection," in *Readings in computer vision*. Elsevier, 1987, pp. 184–203.

[136] S. Chen, Y. Liu, X. Gao, and Z. Han, "MobileFaceNets: Efficient CNNs for accurate real-time face verification on mobile devices," in *Chinese Conference on Biometric Recognition*. Springer, 2018, pp. 428–438.

[137] G. B. Huang, M. Ramesh, T. Berg, and E. Learned-Miller, "Labeled faces in the wild: A database for studying face recognition in unconstrained environments," University of Massachusetts, Amherst, Tech. Rep. 07-49, October 2007.

[138] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, 2015, p. 7.

[139] S. Yang, P. Luo, C.-C. Loy, and X. Tang, "WIDER FACE: A face detection benchmark," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 5525–5533.

[140] T. Karras, S. Laine, and T. Aila, "A style-based generator architecture for generative adversarial networks," *arXiv preprint arXiv:1812.04948*, 2018.

[141] A. Bulat and G. Tzimiropoulos, "How far are we from solving the 2d & 3d face alignment problem?(and a dataset of 230,000 3d facial landmarks)," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 1021–1030.

[142] J. Guo and H. Chao, "One-to-many network for visually pleasing compression artifacts reduction," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 3038–3047.

[143] G. Lu, W. Ouyang, D. Xu, X. Zhang, Z. Gao, and M.-T. Sun, "Deep kalman filtering network for video compression artifact reduction," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 568–584.

[144] 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, pp. 14–23, 2009.

[145] "Computing Receptive Fields of Convolutional Neural Networks," https://distill.pub/2019/computing-receptive-fields/. Accessed: 15 Feb. 2023.

[146] W. Luo, Y. Li, R. Urtasun, and R. Zemel, "Understanding the effective receptive field in deep convolutional neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[147] A. Aijaz, "Private 5g: The future of industrial wireless," *IEEE Industrial Electronics Magazine*, vol. 14, no. 4, pp. 136–145, 2020.

[148] M. Wen, Q. Li, K. J. Kim, D. López-Pérez, O. A. Dobre, H. V. Poor, P. Popovski, and T. A. Tsiftsis, "Private 5g networks: concepts, architectures, and research landscape," *IEEE Journal of Selected Topics in Signal Processing*, vol. 16, no. 1, pp. 7–25, 2021.

[149] "AT&T Cell Phone Plans," https://www.att.com/5g/consumer/. Accessed: 15 Feb. 2023.

[150] "T-Mobile Cell Phone Plans," https://www.t-mobile.com/cell-phone-plans. Accessed: 15 Feb. 2023.

[151] "Google Cloud Pricing," https://cloud.google.com/compute/gpus-pricing. Accessed: 15 Feb. 2023.

[152] "Amazon EC2 Pricing," https://aws.amazon.com/ko/ec2/pricing/on-demand/. Accessed: 15 Feb. 2023.

[153] "Linux traffic control," https://man7.org/linux/man-pages/man8/tc.8.html. Accessed: 15 Feb. 2023.

[154] J. Pont-Tuset, F. Perazzi, S. Caelles, P. Arbeláez, A. Sorkine-Hornung, and L. Van Gool, "The 2017 davis challenge on video object segmentation," *arXiv preprint arXiv:1704.00675*, 2017.

[155] "A Unified Architecture for Instance and Semantic Segmentation," http://presentations.cocodataset.org/COCO17-Stuff-FAIR.pdf. Accessed: 15 Feb. 2023.

[156] T.-W. Chin, R. Ding, and D. Marculescu, "Adascale: Towards real-time video object detection using adaptive scaling," *Proceedings of machine learning and systems*, vol. 1, pp. 431–441, 2019.

[157] K. Yang, J. Yi, K. Lee, and Y. Lee, "Flexpatch: Fast and accurate object detection for on-device high-resolution live video analytics," in *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 2022, pp. 1898–1907.

[158] F. Yu, W. Xian, Y. Chen, F. Liu, M. Liao, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving video database with scalable annotation tooling," *arXiv preprint arXiv:1805.04687*, vol. 2, no. 5, p. 6, 2018.

[159] Z. Tu, J. Hu, H. Chen, and Y. Wang, "Toward accurate post-training quantization for image super resolution," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2023, pp. 5856–5865.

[160] Q. Zhou, S. Guo, Z. Qu, J. Guo, Z. Xu, J. Zhang, T. Guo, B. Luo, and J. Zhou, "Octo:{INT8} training with loss-aware compensation and backward quantization for tiny on-device learning," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 177–191.

[161] A. Chaurasia and E. Culurciello, "Linknet: Exploiting encoder representations for efficient semantic segmentation," in *2017 IEEE Visual Communications and Image Processing (VCIP)*. IEEE, 2017, pp. 1–4.

[162] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, pp. 91–110, 2004.

[163] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I 9*. Springer, 2006, pp. 404–417.

[164] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving DNNs like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.

[165] "Google Cloud Serverless Computing," https://cloud.google.com/serverless?hl=en. Accessed: 15 Feb. 2023.

[166] "AWS Lambda Serverless Computing," https://aws.amazon.com/ko/lambda/pricing/. Accessed: 15 Feb. 2023.

[167] "CppFlow," https://github.com/serizba/cppflow. Accessed: 15 Feb. 2023.

[168] "Secure Reliable Transport (SRT) Protocol," https://github.com/Haivision/srt. Accessed: 15 Feb. 2023.

[169] A. Kirillov, K. He, R. Girshick, and P. Dollár, "A unified architecture for instance and semantic segmentation," 2017.

[170] K. Zhang, B. He, J. Hu, Z. Wang, B. Hua, J. Meng, and L. Yang, "G-NET: Effective GPU sharing in NFV systems," in *15th USENIX Symposium on Networked Systems Design and Implementation NSDI 18)*, 2018, pp. 187–200.

[171] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded gpu," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, 2010, pp. 344–350.

[172] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile gpus," *arXiv preprint arXiv:1907.01989*, 2019.

[173] "NVIDIA's next generation CUDA compute architecture: Kepler GK110, 2012." https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf. Accessed: 25 Mar. 2020.

[174] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron, "Enabling task parallelism in the CUDA scheduler," in *Workshop on Programming Models for Emerging Architectures*, vol. 9. Citeseer, 2009.

[175] "Snapdragon 845: Immersing you in a brave new world of XR." https://www.qualcomm.com/news/onq/2018/01/18/snapdragon-845-immersing-you-brave-new-world-xr. Accessed: 25 Mar. 2020.

[176] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[177] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song, "RAVEN: Perception-aware optimization of power consumption for mobile games," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. ACM, 2017, pp. 422–434.

[178] "XiaoMi Mobile AI Compute Engine (MACE) model zoo," https://github.com/XiaoMi/mace-models. Accessed: 25 Mar. 2020.

[179] S. Zafeiriou, G. Tzimiropoulos, and M. Pantic, "The 300 videos in the wild (300-VW) facial landmark tracking in-the-wild challenge," in *ICCV Workshop*, vol. 32, 2015, p. 73.

[180] "Qualcomm Snapdragon Profiler," https://developer.qualcomm.com/software/snapdragon-profiler. Accessed: 25 Mar. 2020.

[181] T. Jin, S. He, and Y. Liu, "Towards accurate GPU power modeling for smartphones," in *Proceedings of the 2nd Workshop on Mobile Gaming*, 2015, pp. 7–11.

[182] J. J. Kang and S. Adibi, "Bushfire disaster monitoring system using low power wide area networks (lpwan)," *Technologies*, vol. 5, no. 4, p. 65, 2017.

[183] "NVIDIA CUDA on Arm," https://developer.nvidia.com/cuda-toolkit/arm. Accessed: 25 Mar. 2020.

[184] "NVIDIA Tesla P100, 2016." https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf. Accessed: 25 Mar. 2020.

[185] "R. Smith and Anandtech. Preemption improved: Fine-grained preemption for time-critical tasks, 2016." http://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/10. Accessed: 25 Mar. 2020.

[186] "Google Edge TPU," https://cloud.google.com/edge-tpu?hl=en. Accessed: 25 Mar. 2020.

[187] "Huawei Kirin SoC with NPU," http://www.hisilicon.com/en/Products/ProductList/Kirin. Accessed: 25 Mar. 2020.

[188] Y. Xie, F. Yi, and K. Jamieson, "Pbe-cc: Congestion control via endpoint-centric, physical-layer bandwidth measurements," 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. 451–464.

[189] Y. Lin, Z. Zhang, H. Tang, H. Wang, and S. Han, "Pointacc: Efficient point cloud accelerator," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 449–461.

초 록

주요어: 실시간 비디오 분석, 엣지-클라우드 협력, 모바일/엣지 AI
학번: 2020-39481

감사의 글

감사합니다

2024 년 2월

이주헌 올림