Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications

Juheon Yi
johnyi0606@snu.ac.kr
Seoul National University
Seoul, Korea

Youngki Lee
youngkilee@snu.ac.kr
Seoul National University
Seoul, Korea

Abstract

We present Heimdall, a mobile GPU coordination platform for emerging Augmented Reality (AR) applications. Future AR apps impose an explored challenging workload: i) concurrent execution of multiple Deep Neural Networks (DNNs) for the physical world and user behavior analysis, and ii) seamless rendering in presence of DNN execution for immersive user experience. Existing mobile deep learning frameworks, however, fail to support such workload: multi-DNN GPU contention slows down inference latency (e.g., from 59.93 to 1181 ms) and rendering-DNN GPU contention degrades frame rate (e.g., from 30 to ≈12 fps). Multi-tasking for desktop GPUs (e.g., parallelization, preemption) cannot be applied to mobile GPUs as well due to limited architectural support and memory bandwidth. To tackle the challenge, we design a *Pseudo-Preemption* mechanism which breaks down the bulky DNN into smaller units, prioritizes and flexibly schedules concurrent GPU tasks. We prototyped Heimdall over various mobile GPUs (e.g., recent Adreno series) and multiple AR app scenarios that involve combinations of 8 state-of-the-art DNNs. Our extensive evaluation shows that 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 approach.

CCS Concepts

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computer systems organization → Real-time system architecture**.

Keywords

Mobile Deep Learning, Augmented Reality, Mobile GPUs, Multi-DNN and Rendering Concurrent Execution

ACM Reference Format:
Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications . In *The 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20), September 21–25, 2020, London, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372224.3419192

MobiCom '20, September 21–25, 2020, London, United Kingdom

ACM ISBN 978-1-4503-7085-1/20/09…$15.00
https://doi.org/10.1145/3372224.3419192

1 Introduction

Augmented Reality (AR) applications (apps) are getting increasing attention, with the expected market size of $198 billion in 2025 [1]. The life-immersive user experiences accelerate the penetration of AR apps into various domains including security, commerce, and education (Section 2.1). Also, new forms of AR devices (e.g., Microsoft HoloLens 2 [2], Magic Leap One [3]) are emerging. Despite the huge potential, truly immersive AR apps are yet to be developed.

The core challenge lies in the unique workload of AR apps to seamlessly combine virtual information over the physical world with resource-constrained AR devices (e.g., wearable and mobile devices). Specifically, AR apps have the following computational requirements. First, an AR app needs to accurately analyze the physical world and user behaviors (e.g., gestures and head movements) to decide which virtual contents to generate and where to display them. Such analysis often requires a continuous and simultaneous execution of multiple Deep Neural Networks (DNNs) on vision and sensor data streams (see Table 1). Second, the app should seamlessly synthesize and render virtual contents (e.g., 3D virtual objects, avatar's hand gestures) over analyzed scenes for immersive user experiences. Finally, background DNN computation and foreground UI rendering should be simultaneously performed in real-time under resource constraints. In particular, both DNN and rendering tasks should preferably run on the mobile GPU for low latency, causing serious contention. Without careful coordination, rendering and DNN performances degrade significantly even when the overall workload fits in the capacity of the mobile GPU.

In this paper, we present Heimdall, a mobile GPU coordination platform to meet the requirements of emerging AR apps. Heimdall newly designs and implements the *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 task 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 [5–8] that have mostly been designed for single DNN execution, emerging AR apps require a concurrent multi-DNN execution (Section 2.2). Not only are the individual state-of-the DNNs very complex to run in real-time (Section 3.1), 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) [5] and

(a) Criminal chasing. (b) Immersive online shopping. (c) Augmented Interactive Workspace (source: [4]).

Figure 1: Multi-DNN AR application scenarios.

Xiaomi MACE [6] over high-end Adreno 640 GPU incurs as high as 19.7× slowdown (Section 3.2). Although several recent studies aimed at running multiple DNNs concurrently on mobile [9–11], they have mostly focused on memory optimization [9, 10] or cloud offloading [11]; multi-DNN GPU contention remains unsolved.

• **Rendering-DNN GPU Contention.** More importantly, existing frameworks only consider a DNN running in an isolated environment where no other task is contending over 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 (Section 3.3)).

There have been studies to schedule concurrent tasks on a GPU designed for desktop/servers [12–19], either with *parallel execution* by dividing GPU cores (e.g., using NVIDIA Hyper-Q [20]) with hardware architectural support, or by *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 (Section 4.1.1). Even with architecture evolution, the need for an app-aware coordinator to dynamically adjust priorities between multiple DNNs and allocate resources persists (Section 10.1.1). We also can 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 GPU tasks. We take the *time-sharing* approach as baseline, and enable context switches only when a semantic unit of a DNN or rendering task is complete. This does not incur additional memory access cost, which is the core difficulty to apply conventional preemption (triggered by periodic hardware interrupt regardless of the application context) for mobile GPUs. On the other hand, our *Pseudo-Preemption* mechanism allows DNN and rendering tasks to time-share the GPU at a fine-grained scale with minimal scheduling overhead. With this new capability, we can flexibly prioritize and assign tasks to the GPU to meet the latency requirements. Our approach can also be useful for the emerging neural processors (i.e., NPUs/TPUs), as preempting hard-wired matrix multiplications are complicated and context switch overheads can be more costly due to larger state sizes (Section 10.1.2).

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

• **Preemption-Enabling DNN Analyzer.** The key to realize *Pseudo-Preemption* is breaking down the tasks into small schedulable units. Our *Preemption-Enabling DNN Analyzer* measures the execution times of DNN operators (ops) and rendering on target mobile device and partitions the DNN 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 op is sufficiently small (e.g., <5 ms for 89.8% of ops). Exploiting this, the analyzer groups several consecutive ops together as a scheduling unit, which can fit between two consecutive rendering tasks. Rendering task latencies are often very small (e.g., 2.7 ms for 1080p) and 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 [21], MaceEngine.Run() in MACE [6]), limiting multi-DNN and rendering to share the mobile GPU at a very coarse-grained scale.

• **Pseudo-Preemptive GPU Coordinator.** We design a GPU coordinator schedules rendering and DNN ops (as grouped by the analyzer) on GPU and CPU. It can enable various scheduling policies based on multiple factors: profiled latencies, scene variations, and app/user-specified requirements. As the base scheduling policy, the coordinator puts the top priority to the rendering tasks and executes the rendering tasks 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 to run on GPU. It also decides whether to offload some DNNs to CPU to in case there is a high level of contention on 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 *Pseudo-Preemption* mechanism to overcome the limitations of mobile GPUs in 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 [6], an OpenCL-based mobile deep learning framework, and conduct an extensive evaluation with 8 state-of-the-art DNNs (see Table2) and mobile GPUs (i.e., Adreno series) to verify the effectiveness.

2 Applications and Requirements

2.1 Application Scenarios

Criminal Chasing (Figure 1(a)). A police officer chasing a criminal in a crowded space (e.g., shopping mall) sweeps the mobile

Table 1: DNN and rendering requirements for the example AR app scenarios.

	Criminal Chasing	Immersive Online Shopping	Augmented Interactive Workspace	AR Emoji	Surroundings Monitoring
Continuously executed DNNs (fps)	- Face detection [22] - Face recognition [23] (< 1s per scene)	- Image segmentation [24] (1-5 fps) - Object detection [25] (1-5 fps) - Hand tracking [26] (1-10 fps)	- Text detection [27] (1-5 fps) - Hand tracking [26] (1-10 fps)	- Face detection [22] (1-10 fps) - Image segmentation [24] (1-10 fps)	- Face detection [22] (1-10 fps) - Object detection [25] (1-10 fps)
Event-driven DNNs (response time)		- Image style transfer [28] (< 0.1s)		- Image style transfer [28] (< 0.1s)	- Pose estimation [29] (< 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	- Camera frames (1080p, 30 fps)[1] - Bounding boxes - Human body joints

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

Table 2: DNNs for the above AR 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 [25]	416×416×3	0/33	95 ms
Face detection	RetinaFace [22]	1,920×1,080×3	6/129	230 ms
Face recognition	ArcFace [23]	112×112×3	0/106	149 ms
Image segmentation	DeepLab-v3 [24]	513×513×3	0/101	207 ms
Image style transfer	StyleTransfer [28]	640×480×3	14/106	60 ms
Pose estimation	CPM [29]	192×192×3	0/187	14 ms
Hand tracking	PoseNet [26]	192×192×3	0/74	256 ms
Text detection	EAST [27]	384×384×3	8/117	214 ms

device to take the video of the area from distance. The mobile device processes the video stream to detect faces and find the matching face with the criminal. Specifically, it continuously runs face detector per scene and face recognizer per each detected face. Detection results are seamlessly overlayed on top of the camera frames and rendered on screen, so as to narrow down a specific area to search.

Immersive Online Shopping (Figure 1(b)). An online shopper wearing AR glasses positions a virtual couch in his room to see if the couch matches well before purchase. The smart 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 to run object detection and image segmentation simultaneously to analyze the room, run hand tracking, and image style transfer to recognize user's hand movements and adjust the style of the couch, while seamlessly rendering the virtual couch on the right spot.

Augmented Interactive Workspace (Figure 1(c)). A student wearing AR glasses creates an interactive workspace by combining virtual documents and physical textbooks. When he encounters a concept he does not understand, he commands the AR glass 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 AR Apps include AR emoji [30] (face detection + segmentation + style transfer) or surroundings monitoring for visual support [9] (object and face detection + pose estimation).

2.2 Workload Characterization

Real-time, Concurrent Multi-DNN Execution. The core of AR apps is accurately analyzing the physical world and user behavior to combine the virtual contents, which requires to run multiple DNNs concurrently (see Table 1 and 2 for examples). Also, such analysis needs to be continuously performed over a stream of images to seamlessly generate and overlay virtual information, especially with fast-changing scenes (e.g., criminal chasing). Moreover, DNNs need to run over high-resolution inputs for accurate analysis. For instance, recognizing small hand-writings or distant faces requires over 720p or 1080p frames [11, 27]). These characteristics are clearly distinguished from prior work [7–9, 31] that has mostly considered running a single DNN over simple scenes (e.g., with a few main objects) that can be analyzed with smaller resolution (e.g., 300×300).

Seamless Rendering on Top of Concurrent DNN Execution. AR apps need to seamlessly augment virtual information (e.g., virtual objects, hand gestures) over the analyzed scenes for immersive user experiences. Such foreground rendering should be continuously performed in real-time in presence of DNN computation, causing serious contention on resource-constrained mobile GPUs.

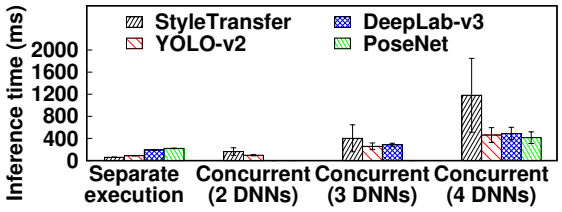
Summary. Concurrent execution of multi-DNN and rendering necessitate the need to prioritize and coordinate their execution on the mobile GPU. Careful coordination will become more important if an app requires audio tasks (e.g., voice command recognition, spatial audio generation) along with the vision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

3 Preliminary Studies

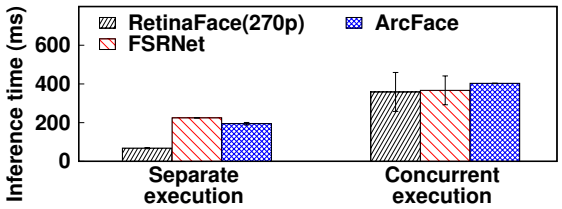
We conduct a few motivational studies to analyze the limitations of existing frameworks in handling of emerging AR app workload.

Table 3: Complexity comparison between state-of-the-art DNNs and their backbone networks.

	State-of-the-art DNN		Backbone (input size scaled)	
Input size	Model	FLOPs	Model	FLOPs
1,920×1,080	RetinaFace [22]	9.54 G	MobileNet-v1-0.25 [32]	1.65 G
112×112	ArcFace [23]	10.13 G	ResNet [33]	0.95 G
513×513	DeepLab-v3 [24]	16.48 G	MobileNet-v2 [34]	1.54 G



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



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

Figure 2: Multi-DNN GPU contention.

3.1 Complexity of the State-of-the-art DNNs

One might think multi-DNN execution on mobile devices is becoming less challenging due to the emergence of lightweight network architectures (e.g., MobileNet [32, 34]) and the increasing computing power of mobile GPUs. However, the challenge still exists. One main reason is that state-of-the-art DNNs do not employ the lightweight networks directly, but elaborate them with complex task-specific architectures to achieve higher accuracy.

Table 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 network. For instance, RetinaFace [22] detector employs feature pyramid [35] on top of MobileNet-v1 [32] to accurately detect tiny faces, whereas ArcFace [23] recognizer adds batch normalization layers on ResNet [33] and replaces 1×1 kernel to 3×3 for higher accuracy. Similar holds for DeepLab-v3 [24] (segmentation model), which adds multiple branches to the backbone MobileNet-v2 [34] to analyze the input image in various scales.

3.2 Multi-DNN GPU Contention

Existing mobile deep learning frameworks [5–8] are mostly designed to run only a single DNN. The only way to run multiple DNNs concurrently is to launch multiple instances (e.g., TF-Lite's Interpreter, MACE's MaceEngine) on separate threads. However, multiple DNNs contending over limited mobile GPU resources incur contention, unexpectedly degrading overall latency. More importantly, the uncoordinated execution of multiple DNNs makes it difficult to guarantee performance for mission-critical tasks with latency constraints.

Algorithm 1 OpenCL-based DNN inference in MACE

```
1: for Operator in Graph do
2:     TargetDevice ← Operator.GetTargetDevice()
3:     if TargetDevice == GPU then
4:         Kernel ← Operator.GetKernel()
5:         clCommandQueue.enqueueNDRangeKernel(Kernel)
6:     if TargetDevice == CPU then
7:         clCommandQueue.finish()
8:         Operator.RunOnCPU()
```

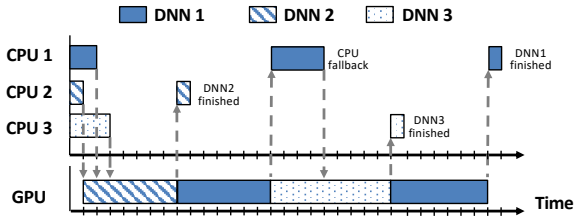


Figure 3: Multi-DNN GPU contention example.

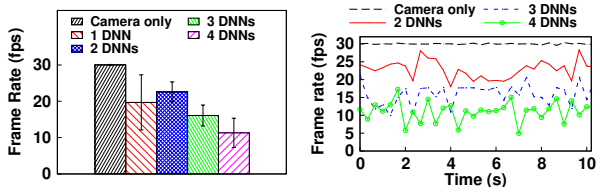
3.2.1 *Measurement on Existing Frameworks*

To evaluate the impact of multi-DNN GPU contention on latency, we run 4 DNNs in the immersive shopping scenario in Table 2 on MACE over LG V50. Figure 2(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 individual DNN inference times are sufficient to satisfy app requirements (e.g., the sum of the inference times of 4 the DNNs are 560.02 ms, indicating that they can run at ≈2 fps when coordinated perfectly). However, uncoordinated execution of multiple DNNs 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 guarantee the latency requirement of AR apps. We observe a similar trend in TF-Lite: Figure 2(b) shows that running 3 DNNs in the criminal chasing scenario incurs significant and latency overhead.

3.2.2 *Analysis*

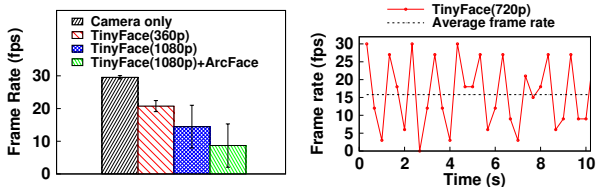
Algorithm 1 presents the OpenCL-based DNN inference flow in MACE.[1] Upon the inference start, the framework executes a series of operators (ops) constituting the DNN. Per each op, the framework first identifies if the it is executed on GPU or CPU (lines 1–2). GPU op is executed by enqueueing its kernel to the command queue to be executed by the GPU driver (lines 3–5). 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 previous GPU ops

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



(a) Average frame rate. (b) Frame rate over time.

Figure 4: Rendering-DNN GPU contention on MACE over LG V50 (immersive online shopping scenario).



(a) Average frame rate. (b) Frame rate over time.

Figure 5: Rendering-DNN GPU contention on TF-Lite over Google Pixel 3 XL (criminal chasing scenario).

are finished and the result is available to the CPU via CPU/GPU synchronization (lines 6–8).

Figure 3 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 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 shows how frequently this occurs for different DNNs; more details are in Section 7.2). In such a case, DNN#1 encounters another contention: even when the CPU execution is finished, it cannot access the GPU until already running DNN#3 finishes. As a result, the inference latency of DNN#1 is delayed significantly.

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(a), StyleTransfer [28] containing 14 CPU ops suffers the most latency overhead compared to other DNNs that contain no CPU ops.

3.3 Rendering-DNN GPU Contention

More importantly, existing frameworks consider only a single DNN running in an isolated environment (i.e., no other task contending over the mobile GPU), and are ill-suited for AR apps that require concurrent execution of rendering on top of multiple DNNs. Figure 4 and 5 shows the 1080p camera frame rendering rate in presence of multiple DNNs, with the same DNN setting as in Figure 2. Figure 4(a) shows that with DNNs are running, rendering frame rate drops due to the similar contention scenario as in Figure 3, dropping to as low as 11.31 fps when all 4 DNNs are running. To make matters worse, GPU contention incurs frame rate fluctuating over time as shown in Figure 4(b), significantly degrading perceived rendering quality to users. We observe a similar trend on TF-Lite when running TinyFace [36] detector and ArcFace [23] recognizer concurrently with rendering task (Figure 5).

3.4 Summary

The workload of upcoming AR apps is unique in that it runs multiple computation-intensive DNNs and while seamlessly rendering 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.

4 Heimdall System Overview

4.1 Approach

The core challenge in supporting concurrency with mobile GPU lies in the lack of support for parallelization or preemption. As analyzed in Section 3, mobile GPU can run only a single task at a given time, making it hard to provide stable DNN execution latency and rendering performance. Existing mobile deep learning frameworks, however, fail to account for 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., 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), and ii) they provide no means to prioritize a certain task over others, making it challenging to guarantee performance under contention.

4.1.1 *Why Not Apply Desktop GPU Scheduling?*

One possible approach is to implement parallelization or preemption in mobile GPUs. There have been prior studies to support multitask scheduling for desktop/server-grade GPUs [12–17]. However, they are either designed for CUDA-enabled NVIDIA GPUs (which are unsupported in mobile devices) or require hardware modifications (e.g., memory hierarchy [37]). This makes it difficult to apply them for 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 [15, 17] (e.g., NVIDIA GPUs can be parallelized in units of Streaming Multiprocessors (SMs) using Hyper-Q [20] starting from Kepler architecture [38] released in 2012) or fusing parallelizable kernels with compiler techniques [16, 39]. 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 [12–14], as well as enabling high-priority tasks to preempt the GPU even when others are running [17] (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× smaller than that of NVIDIA RTX 2080Ti (616 GB/s). Each context switch requires 120 MB memory transfer (=20 cores×24 execution lanes/core×64 registers/lane×32 bits), which incurs at least 4.36 ms latency even when assuming the GPU fully utilizes the shared memory bandwidth. Recent Qualcomm GPUs (Adreno 630 and above) support

preemption [40], which can be utilized by setting different context priorities in OpenCL. However, we observed that each context switch (both between rendering–DNN and DNN–DNN) incurs 2–3 ms overhead on LG V50 (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 multi-DNN and rendering workload, where context switch should occur at a 30 fps (or higher) scale.

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 baseline. To mimic the effect of preemption while avoiding burdensome context switch memory overhead, we divide the DNN and rendering tasks into smaller chunks (i.e., scheduling units) and switch between tasks only when each task chunk is finished. This enables multi-DNN and rendering tasks to time-share the GPU at a fine-grained scale. Our approach can be useful for NPUs/TPUs as well, as context switch overhead can be more painful for such processors due to even larger state sizes than mobile GPUs (Section 10.1.2). 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 (Section 5.1).

4.2 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 as well as fluctuation due to GPU contention, which harms the user experience.

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 [10]) 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.

4.3 System Architecture

Figure 6 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

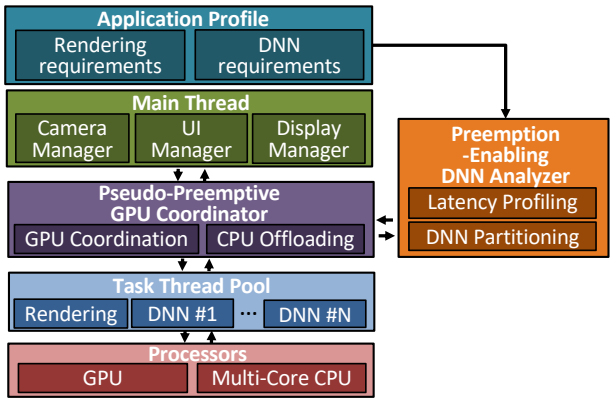


Figure 6: System Architecture of Heimdall.

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 (Section 5.2). Second, it partitions the DNNs into chunks (scheduling unit) so as to fit in between rendering events with minimal inference latency overhead (Section 5.3).

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 (Section 6.2), and coordinates the execution on GPU, as well as dynamically offload some DNNs to the CPU to reduce the GPU contention (Section 6.3).

5 Preemption-Enabling DNN Analyzer

5.1 Overview

What Should We Analyze? The goals of the analyzer are: i) profile rendering and DNN inference latencies on the target device (varies depending on the mobile SoC and GPU) to let the coordinator get a grasp on how it can dynamically schedule multi-DNN execution, and ii) partition the bulky DNN 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), 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 DNN partitioning approach is feasible. However, the execution times of DNNs on CPUs may fluctuate due to resource contention; Figure 9 shows that inference times on CPU increase and fluctuate when 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. At the operator (op)-level, the execution times remain small enough, making fine-grained partitioning feasible to fit in between rendering events. For example, for the 7 DNNs in Table 2 whose

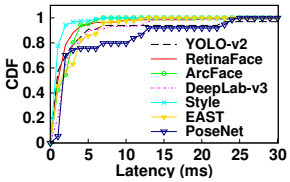


Figure 7: Operator-level latency distribution.

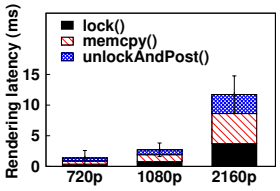


Figure 8: Camera frame rendering latency.

total inference latency is over 33 ms, Figure 7 shows that on average 89.8% of the ops run within 5 ms on Google Pixel 3 XL. Therefore, it suffices to partition the DNN at the op-level and not below (e.g., convolution filter). However, note that dividing the model too finely also has its downside: it incurs higher latency overhead, as it loses the chance for the GPU driver to batch more consecutive ops to enhance GPU utilization.

5.2 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 DNNs can occupy the GPU (i.e., $\frac{1}{f} - T_{render}$ between rendering events). For example, rendering 1080p frames on Adreno 640 GPU in LG V50 takes 2.7 ms (Figure 8), leaving 30.3 ms for DNNs when the frame rate is 30 fps.

DNN Latency. Secondly, the analyzer measures the inference latencies of DNNs on GPU and CPU. Figure 10 shows an example of profile results for different processors (GPU or CPUs in 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.

5.3 DNN Partitioning

Basic Operation. 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 that each partition execution time lie within the rendering interval,

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

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 it loses the chance for the GPU driver to batch more consecutive ops to enhance GPU utilization: Figure 12 shows that executing only a single op at a time incurs 13 to 70% latency overhead compared to running the entire model at once. The 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 (this happens commonly at the end of the model where there are not enough ops left). To alleviate the issue, we relax the constraint in Equation (1) and allow the partition execution time to exceed rendering interval by a small margin (e.g., 5 ms), so that more ops can be packed to maximize GPU utilization.

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

6 Pseudo-Preemptive GPU Coordinator

6.1 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 and device driver layer where the workloads are highly abstracted.

Operational Flow. The coordinator puts the top priority to the rendering tasks and executes it at the target frame rate (e.g., 30 fps). We take this design decision as the degradation or fluctuation in rendering frame rate immediately affects the usability of AR apps. It is possible to change the scheduling policy to make rendering 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 Preemption-Enabling DNN Analyzer) to run on 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 app requirements. It also decides whether to offload some DNNs to CPU in case GPU contention level is too high.

6.2 Utility Function

To schedule multiple DNNs, we need a formal way to compare which DNN operators are 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,

$$U_{D_i}(t) = L_{D_i}(t, t^i_{start,k}) + \alpha \cdot C_{D_i}(t^i_{start,k}, t^i_{start,k-1}), \tag{2}$$

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).

6.2.1 Latency Utility

The latency utility of the DNN D_i is calculated as,

$$L_{D_i}(t, t^i_{start,k}) = L^0_{D_i} - \left(\beta_i \cdot (t - t^i_{start,k})^{\gamma_i}\right)^2. \tag{3}$$

The latency utility is modeled as a concave function so that it decreases more rapidly over time to prevent the coordinator from

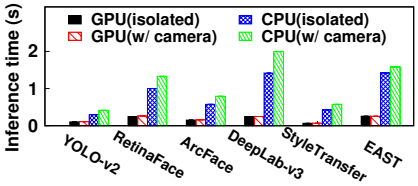


Figure 9: DNN inference latency with and without camera.

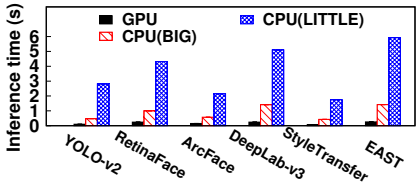


Figure 10: Example DNN latency profiling result on Google Pixel 3 XL.

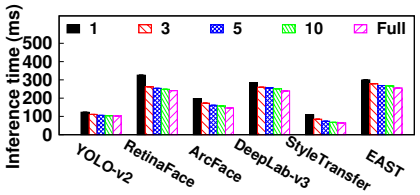


Figure 11: DNN inference latencies for varying partition sizes.

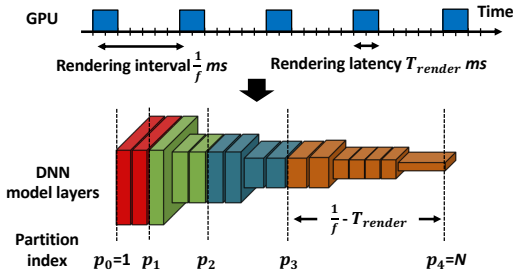


Figure 12: Operation of DNN partitioning.

delaying the execution too long. Three parameters can be configured to set priority 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 value but decrease more rapidly, possibly allowing the coordinator to enable the DNN to preempt the GPU before its utility drops.

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) [41] between the two frames. However, this is infeasible in mobile devices due to computational complexity. Alternatively, we take the approach in [42] and compute the difference between the Y value (luminance) Y^k of the two frames (which has a high correlation with the SSIM and requires only $O(N)$ computations),

$$C_{D_i}(t^i_{start,k}, t^i_{start,k-1}) = \sum_{h=1}^{H}\sum_{w=1}^{W} |Y^k_{h,w} - Y^{k-1}_{h,w}|, \qquad (4)$$

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

6.3 Scheduling Problem and Policy

Given the DNNs and their utilities, the coordinator schedules their execution to maximize 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 CPU to resolve contention.

6.3.1 *GPU Coordination Policy*

Among many possible policies, we define two common GPU coordination policies, following the similar approach in [10]. 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,

$$\begin{aligned} &\min_i U_{D_i}(t). \\ &\text{s.t.} \quad t^i_{end,k} - t^i_{start,k} \le t_{i,max} \end{aligned} \qquad (5)$$

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 1).

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

$$\begin{aligned} &\max_i \quad \textstyle\sum_{t=1}^{N} U_{D_i}(t). \\ &\text{s.t.} \quad t^i_{end,k} - t^i_{start,k} \le t_{i,max} \end{aligned} \qquad (6)$$

Under the MaxTotalUtility policy, the coordinator favors a DNN with higher utility (i.e., allow it to preempt the GPU) and runs the remaining DNNs are executed 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 1).

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 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 CPU to reduce 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,

$$\max_{P_1,P_2,...,P_N} \quad \textstyle\sum_{t=1}^{N} U_{D_i,P_i}(t), \qquad (7)$$

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 measured by the analyzer). As changing the target processor (allocating memory for the model weights and feature maps) incurs around 50 ms latency, the processor mapping is determined at every 1 second interval.

6.4 Greedy Scheduling Algorithm

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

GPU Coordination. For each scheduling event, the coordinator first checks for each DNN how many partitions are left to execute. Based on the profiled latencies of the remaining partitions, the scheduler 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 scheduler runs it immediately. If otherwise, the scheduler determines which DNN to execute based on current utility values. Specifically, MaxMinUtility policy selects a DNN with the current lowest utility. MaxTotalUtility policy iteratively computes the expected sum of utilities at the current scheduling event (without consideration of the future) assuming that a specific DNN chunk is executed, and selects the DNN which maximizes the sum. 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 inference 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 term to avoid ping pong effect between CPU and GPU); per each scheduling event, only one DNN is offloaded to the CPU. If no DNNs are 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.

7 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 adequate form for the task, and iii) possible ops in the model that are unsupported by the GPU backend of the mobile deep learning framework and are executed on CPU. Granting GPU access to a DNN that currently needs to run such steps incurs unwanted GPU idle time, slowing down overall inference latency. This becomes especially significant when processing high-resolution complex scene images. For example, RetinaFace [22] detector with inference pipeline shown in Figure 13 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.

7.1 Preprocessing and postprocessing

Before enqueueing the DNN inference to the task queue for the Pseudo-Preemptive GPU Coordinator to schedule, we run the following steps in parallel in separate thread (in parallel with other DNNs running on the GPU), so that the DNN fully occupies 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 format. For example, face detection requires converting the output feature map to bounding boxes and performing non-maximum suppression to filter out redundant ones.

7.2 CPU Fallback Operators

GPU backend of a mobile deep learning framework typically supports a limited number of ops (a subset of the ops supported in the cloud framework). In case an operator is unsupported by the GPU, it falls back to CPU execution. We identify the indexes of the CPU fallback operators at the profiling stage and run them in parallel with other GPU operators at runtime.

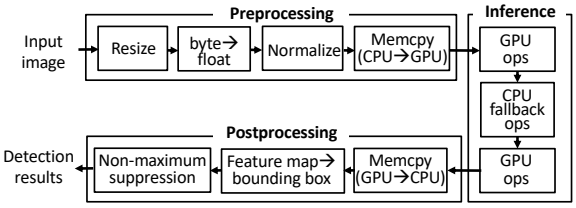


Figure 13: End-to-end DNN inference pipeline example for RetinaFace detector.

Note that CPU fallback occurs frequently especially for state-of-the-art DNNs. For example, TF-Lite does not support tf.image.resize() required in feature pyramid [35], 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().

8 Implementation

We implement Heimdall by extending MACE [6], an OpenCL-based mobile deep learning framework, to partially run a subset of the operators in the DNN 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 evaluated 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). We implement RetinaFace [22], ArcFace [23], EAST [27], PoseNet [26] using TensorFlow 1.12.0. For MobileNet-v1 [32], CPM [29], and StyleTransfer [28], we use the models provided in the MACE model zoo [43]. For DeepLab-v3 [24] and YOLO-v2 [25], we use the pre-trained models from the original authors.

9 Evaluation

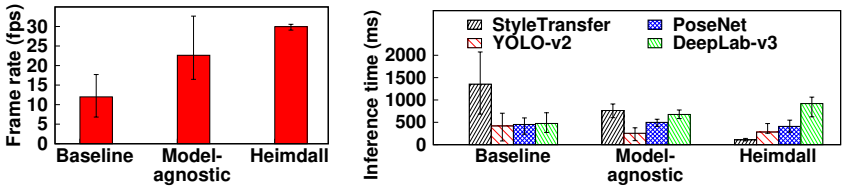
9.1 Experiment Setup

Scenarios. We evaluate Heimdall for 3 scenarios in Table 1 with DNNs in Table 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 (Section 7).

Comparison Schemes.



(a) Rendering frame rate. (b) DNN inference latency.

Figure 14: Performance overview of Heimdall on LG V50.

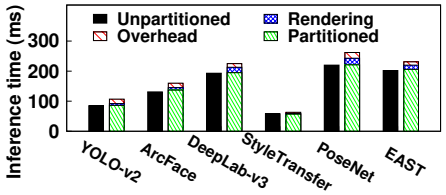
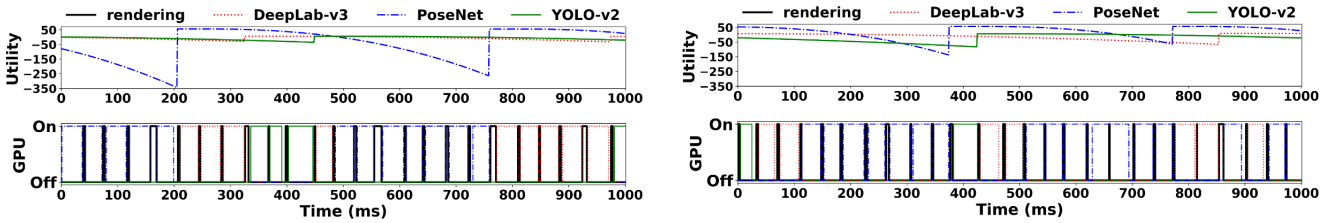


Figure 15: DNN partitioning overhead.



(a) MaxMinUtility. (b) MaxTotalUtility.

Figure 16: Performance comparison of GPU coordination policies.

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

9.2 Performance Overview

We first evaluate Heimdall with 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 14(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 (6.82-17.70 fps, 11.99 on average) frame rate. 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 14(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 runs at average 109, 409, 919 ms, respectively (maximum 139, 548, 1064 ms). Note that the worst-case inference latency of StyleTransfer is also reduced by 14.92× (from 2074 to 139 ms). This is done by i) giving preemptive access to StyleTransfer, ii) running DeepLab-v3 at the minimum and PoseNet more frequently to satisfy the latency constraints of both tasks, and iii) offloading YOLO-v2 to CPU to reduce GPU contention level (which also benefits YOLO-v2 as well). Baseline and model-agnostic partitioning that cannot support such coordination fails to satisfy the app requirement, especially for StyleTransfer which is more vulnerable to GPU contention due to a number of CPU fallback operators, as analyzed in Section 3.2.2.

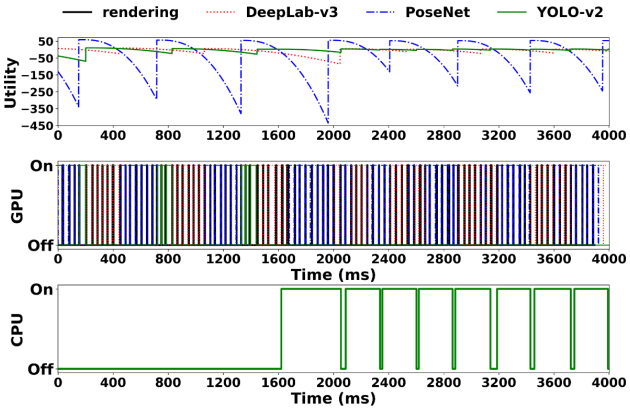


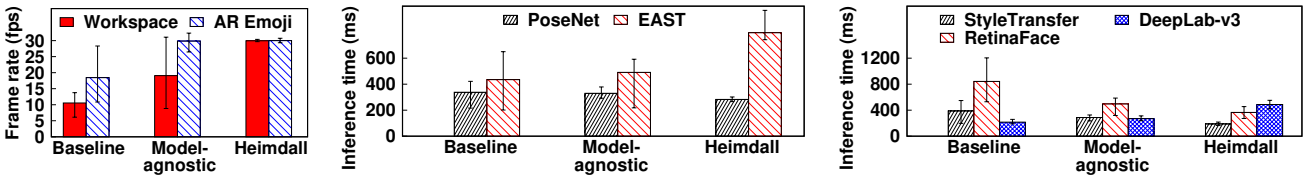
Figure 17: Opportunistic CPU offloading performance.

9.3 DNN Partitioning/Coordination Overhead

We evaluate the DNN partitioning and coordination overhead on inference latency when executed with camera rendering 1080p frames at 30 fps. Figure 15 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 operators as possible. The remaining overhead other than the rendering latency includes multiple factors, including 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 future work to support more number of concurrent DNNs.

9.4 Pseudo-Preemptive GPU Coordinator

GPU Coordination Policy. Figure 16 show 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



(a) Rendering frame rate. (b) DNN inference latency (workspace scenario). (c) DNN inference latency (AR emoji scenario).

Figure 18: Performance of Heimdall for other AR app scenarios.

in Section 6.3.1. Figure 16(a) shows that MaxMinUtility policy executes a DNN with the currently lowest utility and enables a fair resource allocation between the 3 DNNs. Figure 16(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 (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. Now, we incorporate opportunistic CPU offloading in the same setting as in Figure 16(a). Figure 17 shows the GPU/CPU occupancy and utility over time for 3 DNNs. When CPU offloading is triggered at around t=1600 ms, YOLO-v2 (which had the least priority and thus had been executed sporadically) is offloaded to CPU. This benefits the two DNNs on GPU by reducing the contention level (notice that the utility of PoseNet becomes higher after CPU offloading), as well as benefit YOLO-v2 as it experiences faster inference latency as compared to when it was contending with the other two DNNs on GPU.

9.5 Performance for Various App Scenarios

Figure 18 shows the performance of Heimdall on two different scenarios: Augmented Interactive Workspace and AR Emoji. Overall, we observe consistent results. Figure 18(a) shows that Heimdall enables higher and stable rendering. Figure 18(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 18(c) shows that for the AR Emoji scenario, Heimdall prioritizes StyleTransfer to guarantee low latency, while balancing the latencies between RetinaFace and DeepLab-v3.

9.6 DNN Accuracy

We evaluate the impact of Heimdall on DNN accuracy for the AR Emoji scenario. For repeatable evaluation, we sample 5 videos from 300-VW dataset [44] consisted of a single talking person. As the dataset does not provide face bounding box and person segmentation mask labels, we run our models on every frame and use the results as ground truth to be compared with runtime detection results. Table 4 shows the detection accuracy in terms of mean Intersection over Union (IoU). For baseline multi-threading, face detection accuracy remains low, as RetinaFace (with a number of CPU fallback ops) runs at only ≈1 fps due to contention with DeepLab-v3 (Figure 18(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 face detection accuracy at the cost of relatively smaller loss in 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 overall accuracy.

Table 4: 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

9.7 Energy Consumption

Finally, we report the impact of Heimdall on energy consumption. We use Qualcomm Snapdragon Profiler [45] 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 [46] (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, where 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.

10 Discussion

10.1 Will the Challenge Persist?

10.1.1 *How Will Mobile GPU Evolution Affect Heimdall?*

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 [47]) 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 [48] support instruction-level

preemption at 0.1 ms scale [49]), we can consider employing prior multi-DNN scheduling for desktop GPUs [17, 18]. 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.

10.1.2 Will Heimdall be Useful for NPUs/TPUs?

Recently, neural processors are being embedded in mobile devices (e.g., Google Pixel 4 edge TPU [50], Huawei Kirin NPU [51]). Such processors maximize computing power by packing a large number of cores specialized for DNN inference. For example, Google TPUs employ a 128×128 systolic array-based matrix unit (MXU), which accelerates matrix multiplication by hard-wired calculation without memory access. We envision that our Pseudo-Preemption mechanism can also benefit in coordinating multiple concurrent tasks; for such neural processors, it is challenging to preempt the hard-wired MXU. Also, context switch overhead on bandwidth-limited mobile SoCs can be more costly due to larger state sizes than GPU.

10.2 Other Discussions

Generality. We believe Heimdall can be extended to other deep learning frameworks such as TF-Lite as it does not require OS or underlying system supports. The key requirement of Heimdall is to partially run a subset of the operators in the DNN graph. On TF-Lite, it can be implemented by modifying the *Interpreter.Invoke()* function and *Subgraph.Invoke()* function to take the start and end index of the *TFLiteNode* to compute as input parameters.

Scalability. To scale Heimdall to more number of concurrent DNNs with diverse app requirements, further optimizations can help improve performance; such include advanced profiling techniques to enable more efficient scheduling (especially for the runtime CPU task profiling as analyzed in Section 5.1), and incorporating more intelligent utility-based scheduling techniques (possibly motivated from QoS-based scheduling in wireless networking or cloud computing systems). Furthermore, in cases where multiple concurrent AR apps are running, an OS-level extension would be required.

Extension to Complex Rendering Tasks. Heimdall is currently evaluated on 30 fps 1080p frame rendering task. Though we expect Heimdall can be applied similarly to more complex rendering tasks (e.g., 60 fps, 2160p), we conjecture that joint coordination and optimization of multi-DNN and rendering will help resolve GPU contention. Heimdall can be incorporated with recent works that dynamically adjust rendering quality to optimize resource consumption and meet dynamically changing app demands (e.g., AR/VR [52–58], games [42, 59], or web browsing [60]).

Integration with Cloud Offloading. Heimdall can be integrated with cloud offloading systems [11, 61, 62] to collaboratively execute the multi-DNN workload. For example, we can extend the opportunistic CPU offloading to determine which DNN to offload to the cloud, depending on GPU contention level and network conditions.

AR Glass Support. Heimdall can be implemented similarly on AR glasses as they are typically equipped with comparable computing units with the smartphones used in our evaluation (e.g., Snapdragon 850 and 845 for HoloLens 2 [2] and Nreal Light [63], respectively). However, more careful optimizations will be required for power consumption and heat dissipation. Furthermore, in case the DNN inferences for the AR glass inputs are offloaded to the smartphone to save energy, network latency will also need to be considered.

11 Related Work

Continuous Mobile Vision and AR. LiKamWa *et al.* [64] optimize energy consumption of continuous mobile vision systems, while Starfish [65] supports concurrency between multiple apps. Gabriel [66], OverLay [67] and MARVEL [68] utilize cloud for cognitive assistance and mobile AR. Heimdall designs a mobile GPU coordination platform for future multi-DNN enabled AR apps.

Mobile Deep Learning Framework. Several frameworks have been developed from industry [5, 6, 69, 70] and academia [7, 8, 31, 71–76]. However, 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 to Heimdall. DeepEye [9] and NestDNN [10] mainly focuses on memory optimization. DeepEye [9] parallelizes fully connected layer parameter loading and convolutional layer computation but runs only a single DNN on GPU at each time. NestDNN [10] dynamically adapts model size considering available resources but does not consider multi-DNN inference coordination. Lee et al. [77] and Mainstream [78] focus on sharing weights and computations between multiple DNNs. EagleEye [11] runs a multi-DNN face identification pipeline, but offloads most of the computation to the cloud. Most importantly, none of the existing studies considered rendering-DNN GPU contention.

Multi-Task Scheduling on Desktop GPUs. Several studies aimed at enabling efficient GPU sharing on desktop/server GPUs, either by multiplexing multiple kernels temporally [12–14] or spatially [20, 37, 79–82]. Such techniques have been also applied for multi-DNN workloads [17–19, 83]. However, they are ill-suited for mobile GPUs due to limited architecture support and memory bandwidth (see Section 4.1.1 for analysis).

12 Conclusion

We presented Heimdall, a mobile GPU coordination platform for emerging AR apps. To coordinate multi-DNN and rendering tasks, Preemption-Enabling DNN Analyzer partitions the DNN into smaller units to enable fine-grained GPU time-sharing between DNNs and rendering with minimal DNN inference latency overhead. Furthermore, Pseudo-Preemptive GPU Coordinator flexibly schedules multi-DNN and rendering tasks on GPU and CPU to satisfy app requirements. Heimdall efficiently supports multiple AR 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.

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant (No. 2019R1C1C1006088). Youngki Lee is the corresponding author of this work.

References

[1] "Augmented Reality (AR) market size worldwide in 2017, 2018 and 2025," https://www.statista.com/statistics/897587/world-augmented-reality-market-value/. Accessed: 25 Mar. 2020.

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

[3] "Magic Leap One AR glass," https://www.magicleap.com/magic-leap-1/. Accessed: 25 Mar. 2020.

[4] Z. Li, M. Annett, K. Hinckley, K. Singh, and D. Wigdor, "HoloDoc: Enabling mixed reality workspaces that harness physical and digital content," in *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019, pp. 1–14.

[5] "TensorFlow-Lite GPU Delegate," https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/delegates/gpu. Accessed: 25 Mar. 2020.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J. Yi, S. Choi, and Y. Lee, "EagleEye: VR-based empathetic app design for accessibility," in *Proceedings of the 16th Annual International Conference on Mobile Computing and Networking*. ACM, 2020.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] H. Zhou, S. Bateni, and C. Liu, "S^{3}DNN: 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] 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.

[30] "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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] "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.

[39] 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.

[40] "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.

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

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

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

[49] "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.

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

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

[52] J. Choi, H. Park, J. Paek, R. K. Balan, and J. Ko, "LpGL: Low-power graphics library for mobile AR headsets," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2019, pp. 155–167.

[53] 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.

[54] W. Kim, K. T. W. Choo, Y. Lee, A. Misra, and R. K. Balan, "Empath-D: VR-based empathetic app design for accessibility," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 123–135.

[55] T. K. Wee, E. Cuervo, and R. Balan, "FocusVR: Effective 8 usable VR display power management," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, p. 142, 2018.

[56] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han, "Rubiks: Practical 360-degree streaming for smartphones," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 482–494.

[57] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan, "Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices," in *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*. ACM, 2018, pp. 99–114.

[58] S. Shi, V. Gupta, and R. Jana, "Freedom: Fast recovery enhanced vr delivery over mobile networks," in *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2019, pp. 130–141.

[59] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan, "Adaptive display power management for mobile games," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 57–70.

[60] M. Dong and L. Zhong, "Chameleon: a color-adaptive web browser for mobile OLED displays," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 85–98.

[61] 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.

[62] 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.

[63] "NReal Light Mixed Reality Glasses," https://www.nreal.ai/specs/. Accessed: 25 Mar. 2020.

[64] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*. ACM, 2013, pp. 69–82.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

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

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

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

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

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

[80] 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.

[81] 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.

[82] 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.

[83] 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.