Jaguar: Low Latency Mobile Augmented Reality with Flexible Tracking

ABSTRACT

In this paper, we present the design, implementation and evaluation of Jaguar, a mobile Augmented Reality (AR) system that features accurate, low-latency, and large-scale object recognition and flexible, robust, and context-aware tracking. Jaguar pushes the limit of mobile AR's end-to-end latency by leveraging hardware acceleration with GPUs on edge cloud. Another distinctive aspect of Jaguar is that it seamlessly integrates marker-less object tracking offered by the recently released AR development tools (*e.g.,* ARCore and ARKit) into its design. Indeed, some approaches used in Jaguar have been studied before in a standalone manner, *e.g.,* it is known that cloud offloading can significantly decrease the computational latency of AR. However, the question of whether the combination of marker-less tracking, cloud offloading and GPU acceleration would satisfy the desired end-to-end latency of mobile AR (*i.e.,* the interval of camera frames) has not been eloquently addressed yet. We demonstrate via a prototype implementation of our proposed holistic solution that Jaguar reduces the end-to-end latency to ~33 ms. It also achieves accurate six degrees of freedom tracking and 97% recognition accuracy for a dataset with 10,000 images.

1 INTRODUCTION

Augmented Reality (AR) recently draws tremendous attention from both the industry and the research community. An AR system augments the physical world by rendering virtual annotation contents in a device's camera view and aligning the pose of a virtual object with the pose of the camera. As a result, it overlays the rendered virtual objects on top of the camera view of the real world and creates an immersive experience for its users. We can divide existing software AR frameworks into two categories: the traditional systems that heavily utilize computer vision technologies for both object recognition and object tracking, and the recently released AR Software Development Kits (SDKs), such as Google ARCore [2] and Apple ARKit [3], which achieve marker-less object tracking by leveraging motion data from IMU (Inertial Measurement Unit).

Each AR framework has its own unique challenges and advantages. (1) Object recognition in computer vision involves computationally intensive tasks and a large-scale database of reference images, which requires the assistant from the cloud for both computation and storage. Thus, cloud offloading is a promising technology to fill the gap, where an AR system offloads the reference-image database and certain visual tasks (*e.g.,* feature extraction and object recognition) to the cloud [23]. However, on the client side computer vision based tracking still suffers from a low quality due to motion blur or view change. (2) The initial releases of ARCore and ARKit were able to detect horizontal planes and track them with a high quality. Although their tracking feature is flexible and scalable, as we will illustrate in § 2, these systems are not context aware because they do not have the object recognition capability. Thus, the key question we ask is *whether we can achieve the best of both worlds and combine their advantages in a holistic system?*

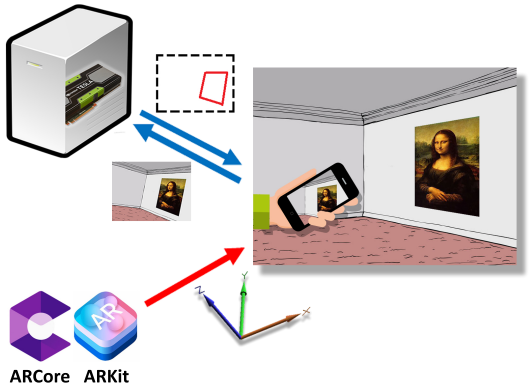

Figure 1: An illustrative example of Jaguar**.**

In this paper, we answer the above question through the design of Jaguar, a mobile AR system that features accurate, low-latency, and large-scale object recognition and flexible, robust, and context-aware tracking. We show an illustrative example of Jaguar in Figure 1. It has two parts, a low-latency image retrieval pipeline with GPU acceleration on edge cloud servers and a lightweight client application that enriches ARCore with object recognition by seamlessly integrating ARCore into its design. One of the primary goals of Jaguar server is to achieve an end-to-end latency close to the inter-frame interval for continuous recognition [17], which does not require a user to pause the camera at an object of interest for seconds. To achieve this goal, on the server side of Jaguar, we investigate the impact on the computational latency of various optimizations for the image retrieval pipeline, *e.g.,* offloading as many as possible computer vision tasks to GPUs.

The design of Jaguar is non-trivial and faces several key challenges. On the client side, since the tracking of ARCore works in the physical world scale, Jaguar needs to determine the actual size of a recognized object and transform the 2D boundary of the object into a 3D pose. Determining the object's physical size efficiently and precisely at runtime is challenging. Moreover, Jaguar runs a different instance of annotation rendering engine with ARCore, which requires an efficient and lightweight combination and alignment to render precisely. On the server side, there is a tradeoff between the accuracy of object recognition and computational latency. Jaguar needs to carefully design the image retrieval pipeline and strategically balance various system requirements.

By addressing the above challenges, we make the following contributions in this paper.

- We seamlessly integrate Google ARCore into Jaguar's client to provide precise runtime object size determination, as well as flexible and scalable object tracking capability (§ 3.1).
- We study how different optimizations of the image retrieval pipeline impact the computational latency on Jaguar server (§ 3.2)

and systematically investigate several practical deployment scenarios that leverage GPUs for acceleration.
• We implement a proof-of-concept for Jaguar (§ 4) and demonstrate that it can significantly reduce the end-to-end latency (§ 5). We also evaluate the scalability of GPU acceleration for supporting multiple Jaguar server instances.

2 BACKGROUND

In this section, we present the typical pipeline of mobile AR systems. We also review existing technologies on cloud offloading and object tracking, two major components for mobile AR.

2.1 Mobile AR Pipeline

A typical pipeline of existing mobile AR systems has 7 building blocks. It starts with *Frame Preprocessing* that shrinks the size of a camera frame, *e.g.,* by downscaling it from a higher resolution to a lower one. The next step is *Object Detection* that checks the existence of targets in the camera view of a mobile device and identifies the regions of interest (ROI) for the targets. It will then apply *Feature Extraction* to extract feature points from each ROI and *Object Recognition* to determine the original image stored in a database of to-be-recognized objects. *Template Matching* verifies the object-recognition result by comparing the target object with the recognized original image. It also calculates the pose of a target (*i.e.,* position and orientation). *Object Tracking* takes the above target pose as its initial input and tracks target object between frames in order to avoid object recognition for every frame. Finally, *Annotation Rendering* augments the recognized object by rendering its associated content. Note that in this paper we focus on AR systems that leverage image retrieval to actually *recognize* objects; whereas other types of AR applications *classify* objects for augmentation (which we will discuss in § 6).

2.2 Cloud Offloading for Mobile AR

To avoid performing computation intensive tasks (*e.g.,* feature extraction and object recognition) on mobile devices, AR systems can offload them to the cloud. There are two common offloading scenarios, depending on where feature extraction is running. (1) Systems, such as Overlay [29], offload tasks starting from feature extraction to the cloud by uploading camera frames; (2) Mobile devices can also perform feature extraction locally and send the compressed feature points to the cloud for object recognition, as is done by VisualPrint [30]. It is worth noting that cloud offloading for mobile AR is different from existing computation offloading solutions for mobile applications, such as MAUI [19], which offload the program code to the cloud and thus require to have the same cloud execution environment as that on mobile devices. Since we may need to use special hardware (*e.g.,* GPU) on the cloud to accelerate computer vision tasks and the programming of server GPU is usually different from mobile GPU, it is challenging to run the same code on both platforms. As a result, we cannot blindly apply existing schemes (*e.g.,* MAUI [19]), in our work.

With emerging technologies such as augmented reality, virtual reality, drones, and autonomous cars, data has been increasingly generated by end users, which demands real-time communication and efficient processing at the network edge [13]. Compared to traditional centralized cloud services offered by, *e.g.,* Amazon and Microsoft, edge computing can reduce network latency for cloud-based mobile AR applications [18, 23, 45], and provide the truly immersive and seamless user experience. An extreme of edge computing would be to offload computation tasks to nearby devices, as proposed in Serendipity [46]. The key challenge is to satisfy the stringent latency requirement of mobile AR through the intermittent connectivity among these devices.

2.3 Object Tracking in Mobile AR

One of the fundamental challenges of mobile AR is flexible and accurate object tracking, which is required especially in continuous object recognition where users can freely move their mobile devices during the recognition [17]. When users move their mobile devices, an AR system needs to track the updated pose of a target object within each frame to render the annotation content properly. For the non-continuous case, users need to pause the camera of a mobile device at the interested object before the recognition result is returned [29], either locally or from the cloud.

For marker-based AR which relies on the recognition of images or trackers (*e.g.,* QR codes) and the pre-knowledge of the environment (*e.g.,* a database of to-be-recognized objects), object tracking could be achieved though technologies such as optical flow tracking [27] of recognized object. With the emergence of precise sensors and high quality cameras on mobile devices, AR is transitioning to be markerless and allows users to lay annotations into the physical world without object recognition. Markerless AR (*a.k.a.* Dead Reckoning) usually solves the SLAM (Simultaneous Localization And Mapping) problem in run-time. Example systems include ARCore from Google [2], ARKit from Apple [3], Instant Tracking from Wikitude [6] and Smart Terrain from Vuforia [10].

Both ARCore [2] and ARKit [3] employ image frames from the monocular camera and motion data from IMU to track the position and orientation of a mobile device in the 3D space. Thus, their object tracking capability should be more flexible and scalable than other AR SDKs that track only planar objects through object recognition. Specially, ARCore utilizes concurrent odometry and mapping to understand the pose of a mobile device relative to the real world, by combining feature points from captured camera images and inertial measurements from IMU.

3 SYSTEM DESIGN OF JAGUAR

As shown in the illustrative example in Figure 1, Jaguar consists of two parts: Jaguar client and Jaguar server. Jaguar client utilizes the tracking method of ARCore and ARKit, and benefits from the large-scale object recognition capability of Jaguar server. What we add to ARCore/ARKit are the object recognition and runtime physical size estimation capabilities, which are crucial to make the AR system context-aware. We will present the design of Jaguar client, especially the flexible object tracking feature in § 3.1.

The goal of Jaguar server is to not only offload the computationally intensive tasks to the edge cloud, but also leverage hardware acceleration with GPUs to push the limit of end-to-end latency for mobile AR. We will present the design of Jaguar server in § 3.2. Another reason we employ the cloud-based architecture is that it is

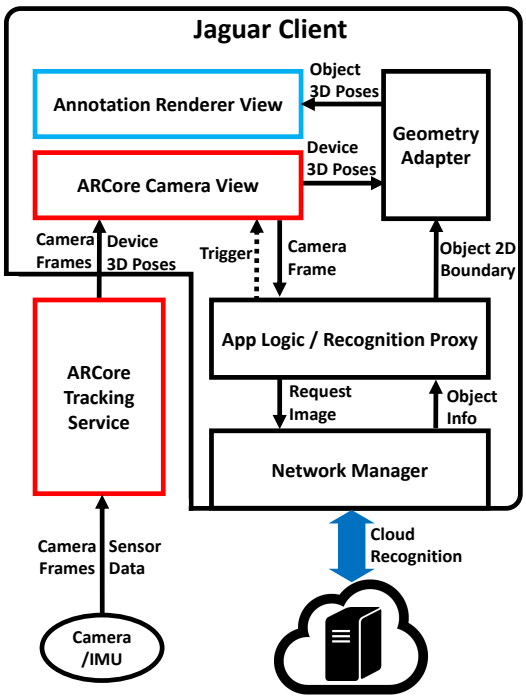

Figure 2: Architecture of Jaguar **client.**

necessary for large-scale image searching in a dataset with thousands of reference images, which is impossible for mobile devices due to their limited storage.

3.1 Flexible Object Tracking on Client

We show the architecture of Jaguar client in Figure 2. It enriches the functionalities of recently proposed ARCore [2] SDK with object recognition and physical size estimation. Although Jaguar is currently deployed on the Android platform with ARCore, we can easily extend its design for ARKit on the iOS Platform.

The traditional image-retrieval based AR solutions perform well on object recognition, but they suffer from low quality or even completely tracking loss due to occultation, motion blur or view change. The reason is that they rely on the recognized planar object for tracking, and the output of pose estimation is a relative pose to that planar object. AR SDKs such as ARCore and ARKit solve this problem by leveraging both inertial measurements from IMU and feature points from captured camera views. However, a key function needed by AR systems is context awareness, which is provided by object recognition. This important feature is missing from the initial releases of both ARCore and ARKit[1].

Jaguar designs object recognition as an add-on of ARCore and enhances its motion tracking function to get an object's pose for each frame. ARCore takes control of the camera hardware and uses OpenGL to render the camera view. The view of annotation renderer in Jaguar is overlaid on top of ARCore's camera view. When the *App Logic* triggers an object-recognition request, Jaguar extracts the current camera frame from ARCore's camera view. It then sends the recognition request containing that frame to edge cloud servers via the *Network Manager*. The cloud recognition result of Jaguar includes the image object's boundary (vertices locations) within the camera frame, the image's size in pixels, and other information such as image ID, name, *etc.*

To render annotation contents, Jaguar transforms an object's 2D boundary into a 3D pose in six degrees of freedom (6DoF) with the *Geometry Adapter*. In OpenGL, the conversion of a 3D vertex coordinate v_{3D} (x, y, z) into a 2D screen coordinate v_{2D} (m, n) is accomplished through matrix transformations:

$$v_{2D} = P * V * M * v_{3D}$$

where M is the Model Matrix describing the object placement in 3D, V is the View Matrix describing the camera placement in 3D, and P is the Projection Matrix describing the 3D to 2D clip and projection transformation.

In Jaguar, P and V are provided by ARCore for each frame, v_{2D} (the locations of an object's vertices in the 2D camera view) and v_{3D} (derived from the object size in pixels) are returned from the server. However, with the four known values, we still cannot derive M from the above equation, since it's a multi-to-one mapping from its right hand side to the left. Imagine an object moving away from the camera but growing in size at the same time, it may look unchanged in the camera view. That is, for each v_{2D}, there are multiple possibilities of M corresponding to it, because the information on the object's physical size is still missing.

To determine the physical size of an object at runtime, Jaguar utilizes both the mapping information from the client and the recognition result from the server. The former models the environment in physical scale and the latter determines the boundary and location of the object within the environment. ARCore generates a sparse 3D point cloud of the environment as part of its SLAM functionality, which describes each point's location in the physical world scale. Some points in the point cloud could be fitted into a plane based on their 3D locations. In OpenGL, the camera imaging plane (camera view) is placed between the camera and the objects, and the 3D points within a truncated pyramid frustum in front of the camera are projected onto the imaging plane. When we perform a ray cast from the camera in the direction of an object's 2D vertex within the camera view, the first plane in the point cloud hit by the ray is the object plane, and the intersection point is exactly the 3D location of the corresponding physical object vertex. Based on these vertices, Jaguar derives the physical size and the Model Matrix M of the object, so that *Annotation Renderer* can place the annotation contents precisely aligned with the object. Currently Jaguar calculates the physical size of only planar objects, but not for non-planar 3D objects, which is more challenging. However, with the physical size and pose given as an input (as what ARKit does [9]), our system would be able to track any rigid object.

3.2 Low Latency Offloading to Edge Cloud

To accelerate computer vision tasks on edge cloud servers equipped with GPUs and provide low latency offloading for mobile AR, we study the impact on the computational latency by various optimizations for the image retrieval pipeline, *e.g.*, offloading as many

[1]We note that a simple image recognition feature (recognition of a few local images) has been recently introduced for ARKit in iOS 11.3 which was released in March 2018 [9], although it is still not available in ARCore (as of April 2018).

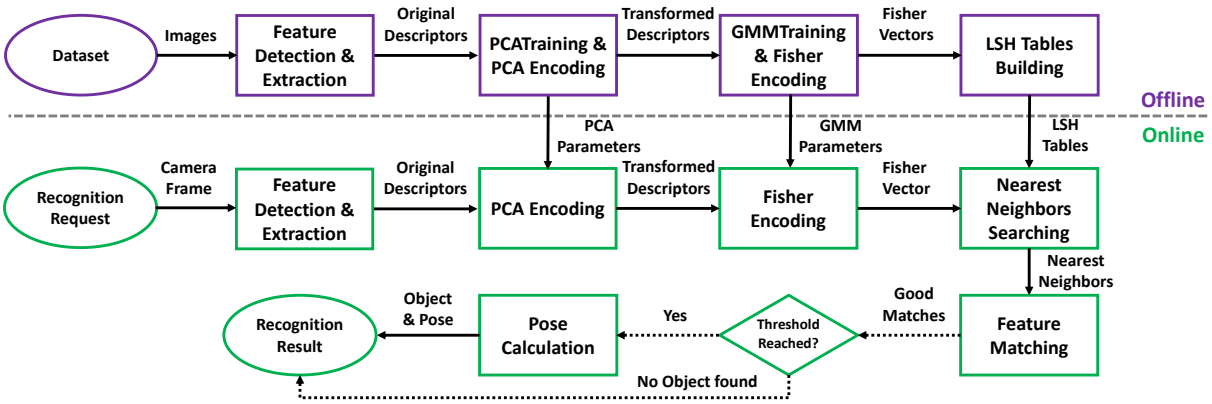

Figure 3: Image retrieval pipeline on Jaguar**'s edge cloud.**

as possible computer vision tasks to GPUs. Previous work demonstrated that cloud offloading can significantly reduce the processing latency of AR applications, from several seconds to a few hundreds of milliseconds [17, 29]. Another benefit of offloading to edge cloud is the reduced network transfer latency, which could be as low as a few milliseconds.

Ideally, the end-to-end latency should be lower than the camera-frame intervals, which is usually around 33 ms (*i.e.,* 30 frames per second), to achieve the best quality of user experience. Although GPU acceleration for most tasks in the image retrieval pipeline has been studied before in a standalone manner, *e.g.,* for feature extraction [16] and feature-point encoding [37], the question of whether the entire accelerated pipeline would satisfy the strict latency requirement of mobile AR has not been meaningfully addressed yet.

3.2.1 Offline Preparation. We can naturally divide server-side processing into two parts, offline preparation and online recognition. Jaguar's offline processing includes 4 steps, as shown in Figure 3. After extracting feature points from reference images in a database of to-be-recognized objects, it first reduces their dimension and trains a statistics model to compress them into a compact representation. Jaguar then generates hash tables of compressed features points for faster online object recognition.

Feature Extraction. For each image, Jaguar executes feature detection and extraction to get the feature descriptors. There is a huge body of feature extraction algorithms, such as SIFT [36], SURF [14], ORB [44], FAST+FREAK [12] and BRISK [34]. Among them SIFT outperforms others in terms of accuracy especially under cases of scaling, rotation and motion blur [15, 31]. Jaguar chooses SIFT in its design and implementation. As we will see later in § 5, when running on GPUs the processing time of SIFT is only ~2.4 ms.

Dimension Reduction. Each feature point extracted by SIFT has 128 dimensions and some of them may be correlated. Jaguar leverages Principal Component Analysis (PCA) to decrease the dimension of feature descriptors to 82, which also increases the accuracy of object recognition. PCA is a statistical procedure that transforms possibly correlated variables into linearly uncorrelated ones. Before the transformation, we collect feature descriptors from all reference images to train the PCA parameters.

Feature Encoding. Since the number of feature points ranges from ~500 to 1,000 for different images, the goal of feature encoding is to create a more compact representation with the same size for each image. Jaguar builds a statistics model (*e.g.,* Gaussian Mixture Model, GMM [43]) using the feature points transformed by PCA and then utilizes Fisher Vector (FV) [40] to further compress them into vectors with a fixed length, one for each image.

LSH Hashing. To further accelerate the online search (*e.g.,* using the nearest neighbor), Jaguar utilizes Local Sensitive Hashing (LSH) and inserts all FVs of the reference images into LSH hash tables. Without hash tables, we need to calculate the distance (*e.g.,* L2 distance) between the FV of a target object and that of each reference image one-by-one, which is not scalable for a large dataset.

3.2.2 Online Recognition. Jaguar's online recognition follows a similar pipeline of its offline preparation. Upon receiving an object recognition request, it first extracts the SIFT feature points of the image, and reduces the feature dimension using PCA with the offline trained parameters. It then creates a single Fisher Vector of the image using the trained GMM model.

To find the original image in the dataset, Jaguar utilizes the LSH tables created in the offline preparation stage to search for the top K nearest neighbors, which are considered as recognition candidates. For each candidate, Jaguar executes SIFT feature matching with the target object in the request image (*i.e.,* a camera frame) to verify the object recognition result. Theoretically, only the true matching should have a number of matched feature descriptors larger than a certain threshold, so that we can find the correct image after feature matching. If no candidate reaches the threshold, there will be no result contained in the response of the object recognition request. After that, the client can issue another request. Finally, with the original image and the feature matching result, Jaguar calculates the pose of the target object within the camera frame.

4 IMPLEMENTATION

Cloud Side. We optimize the image retrieval pipeline by utilizing the CUDA technology from NVIDIA on edge cloud servers. Jaguar uses the GPU implementations of SIFT feature extraction and matching from Björkman *et al.* [16]. Based on the SIFT features, it leverages the OpenCV [7] implementation to train PCA parameters offline, and uses a GPU version of PCA implementation presented in Ma *et al.* [37] for online dimension reduction. With processed feature descriptors from PCA, Jaguar utilizes the VLFeat [11]

Task	Execution Time (ms)
Pre-Processing (client)	6.59 ± 1.21
Data Transfer	8.88 ± 3.29
SIFT Feature Extraction	2.40 ± 0.06
PCA Dimension Reduction	1.49 ± 0.04
FV Encoding with GMM	2.83 ± 0.07
LSH NN Searching	4.51 ± 0.38
Template Matching	3.66 ± 1.49
Post-Processing (client)	2.68 ± 1.32
Overall Latency	33.0 ± 3.79

Table 1: Breakdown of end-to-end latency under edge cloud scenario. The value after ± is standard deviation. Processing latency is measured on a Samsung Galaxy S8 smartphone.

implementation to calculate the Gaussian Mixture Model that represents a statistics distribution of the descriptors. For each reference image, it creates the Fisher Vector with the descriptors processed by PCA and GMM. The GPU Fisher Vector encoding implementation is also from Ma *et al.* [37]. It leverages a CPU based implementation from FALCONN [4] for LSH. In our implementation, we use 32 clusters in the GMM model, and 32 hash tables in LSH, with one hash function in each hash table.

To summarize, on the cloud side, Jaguar executes feature extraction, dimension reduction, feature encoding and matching on GPUs. It performs only the nearest neighbor searching on CPUs. Later we will show that the CPU processing time for this task is very short.

In order to evaluate the scalability of GPU acceleration for Jaguar, we build a Docker image for the server program so that multiple Docker containers can run simultaneously on an edge cloud server. We present the results of our performance evaluation in § 5.4. Ideally we should use Virtual Machines (VMs) to host Jaguar server instances for supporting data isolation. However, GPU virtualization is still not mature. For example, technologies based on device emulation create a virtual GPU inside a VM and multiple VMs share a single physical GPU via the host hypervisor, which suffers from extremely poor performance [49]. Another approach with API forwarding/remoting lacks the flexibility and could not support the full feature set of GPUs [47]. We plan to investigate how Jaguar can benefit from more recent proposals, such as gVirt [49] and gScale [50], in our future work.

Client Side. Jaguar client is an Android application running on a Samsung Galaxy S8 smartphone. It utilizes ARCore to track the device's pose in 6DoF and calculates the physical size of objects. It uses Rajawali [1] as the 3D Engine to render virtual annotation contents. During the cloud recognition procedure, Jaguar client has two tasks: *preprocessing* before sending a request to the cloud, and *post-processing* after receiving the recognition result. Preprocessing downscales the resolution of camera frames and encodes it into a JPEG file, the size of which is around only 10KB. Post-processing calculates the object's physical size and 3D pose based on the 3D point cloud generated by ARCore and the boundary information contained in the recognition result.

5 EVALUATION

In this section, we evaluate the performance of Jaguar using the following metrics, latency, object-recognition accuracy, scalability, object-tracking accuracy and energy consumption.

Execution Time (ms)	Cloud CPU	Cloud GPU	Edge GPU
Client Process	9.27 ± 1.88	9.27 ± 1.88	9.27 ± 1.88
Server Process	187 ± 83.9	17.8 ± 0.84	14.9 ± 2.41
Data Transfer	70.5 ± 22.9	70.5 ± 22.9	8.88 ± 3.29
Overall Latency	267 ± 84.6	97.6 ± 23.3	33.0 ± 3.79

Table 2: End-to-end latency under three scenarios, public cloud with CPU, public cloud with GPU and edge cloud with GPU. The value after ± is standard deviation. The CPU version of server implementation follows exactly the same pipeline with the GPU implementation. The edge cloud has a more powerful GPU than the public cloud (Nvidia GTX 1080 Ti vs. Tesla K80) which leads to reduced processing time (14.9 vs. 17.8 ms).

5.1 Experimental Setup

To evaluate the latency, accuracy and scalability of Jaguar's image retrieval pipeline, we crawl 10,000 book-cover images of 10,000 different books from Amazon. They have a resolution of 375×500. We use all 10,000 images as the training dataset in the offline processing to train PCA and GMM parameters, generate their Fisher Vectors and insert them into LSH tables. We randomly choose 100 images from the training dataset and take photos of them using a smartphone. For each book cover, we create three versions of photo copies with different resolutions, small (200×267), medium (300×400) and large (400×533), to understand the impact of image quality on recognition accuracy.

We measure the end-to-end latency for both public cloud and edge cloud scenarios. Our edge cloud server is a PC equipped with Intel I7-5820K CPU, 64GB memory and a NVIDIA GTX 1080Ti GPU. We create a WiFi hotspot on it and connect the smartphone directly to the hotspot over 802.11g at 2.4GHz, with 16 Mbps (measured) uplink throughput and ~4 ms RTT. Regarding the public cloud, we create a VM on the Google Cloud Platform with 4 vCPUs, 15GB memory and a NVIDIA Tesla K80 GPU. We connect the smartphone to the Internet over 802.11g 2.4GHz WiFi as well, with 4.4 Mbps (measured) uplink throughput and ~45 ms RTT to the Google Cloud.

5.2 End-to-End Latency

We measure the time consumed in each step of the end-to-end edge cloud based Jaguar system and summarize the experimental results in Table 1. During the experiments, we trigger 100 object recognition requests on the client. The preprocessing (frame resizing, JPEG encoding) takes ~6.59 ms, and the post-processing (physical size estimation, 2D to 3D transformation, renderer update) takes ~2.68 ms. The overall client-side processing latency is thus ~9.27 ms.

On the server side, the total processing latency is ~14.9 ms, with template matching for the nearest neighbor. When Jaguar executes nearest neighbors searching on CPU, it takes ~4.51 ms for the dataset with 10,000 images. For $K > 1$ nearest neighbors, the time taken by LSH searching will not increase, but the feature matching in template matching will be repeated up to K times. However, as we will show next, our image retrieval pipeline achieves close to 100% accuracy with only one nearest neighbor (for images with 400×533 resolution). We emphasize that although the size of request images will affect the recognition accuracy, it has almost no impact on the computational latency, because we create at most 1,000 SIFT feature points for each image regardless of its size (which can already ensure a high recognition accuracy).

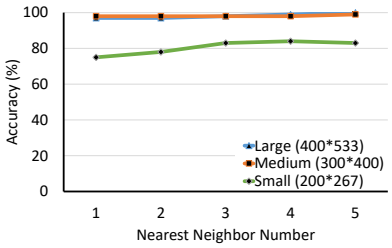

Figure 4: Recognition accuracy for a small dataset with 1,000 images.

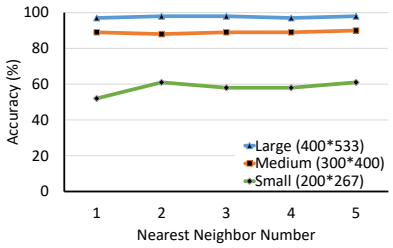

Figure 5: Recognition accuracy for a large dataset with 10,000 images.

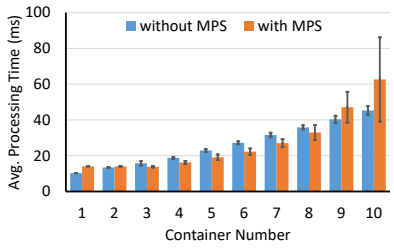

Figure 6: Processing time for different number of server instances.

We compare the end-to-end latency of three offloading scenarios in Table 2, public cloud with and without GPU, and edge cloud with GPU. The CPU version of server implementation follows exactly the same pipeline with the GPU implementation. As we can see from this table, GPU acceleration reduces the server processing time from ~187 ms to less than 20 ms. Our edge cloud has a more powerful GPU which results in a shorter processing time than the public cloud. Benefiting from the closeness of edge cloud to mobile devices, the data transfer latency is reduced to less than 10 ms. As a result, Jaguar achieves an end-to-end latency of ~33.0 ms. With the help of ultra-low latency and high throughput 5G networks, we expect the end-to-end latency of mobile AR could be further optimized.

5.3 Object Recognition Accuracy

We define the object-recognition accuracy as the ratio between the number of successful recognition requests over the total number of requests. We show the experimental results for a dataset with 1,000 images and another one with 10,000 images in Figure 4 and Figure 5, respectively. The x-axis of these figures is the number of nearest neighbors used in template matching. The small dataset is a subset of the large one and we create it to understand the impact of the size of search space on the recognition accuracy.

There are two key observations from Figure 4. First, for a small search space with only 1,000 images, the accuracy is 100% for images with a 400×533 resolution and 5 nearest neighbors for LSH searching. Second, we can achieve more accurate recognition for high-resolution images than that for low-resolution ones. It demonstrates the tradeoff between object-recognition accuracy and bandwidth usage (*i.e.,* uploading high-resolution images can improve the recognition accuracy at the cost of consuming more data usage).

The recognition accuracy drops for the dataset with 10,000 images, as shown in Figure 5. When using only one nearest neighbor, the recognition accuracy is 52%, 88%, and 97% for images with 200×267, 300×400, and 400×533 resolutions, respectively. It is worth noting that the number of nearest neighbors for LSH searching has a limited impact on the recognition accuracy. The accuracy is close to 100% even with one nearest neighbor for high-resolution images.

5.4 Scalability

To understand the scalability of Jaguar with GPU acceleration on edge cloud servers, we measure the processing time for different numbers of Jaguar server instances, from 1 to 10. We launch various numbers of Docker containers simultaneously on a single edge cloud server and run a Jaguar server instance inside each container. We send 100 object recognition requests sequentially to each instance and measure the processing time for every request. We plot the average processing time for two scenarios in Figure 6.

When using CUDA on a single GPU, multiple applications cannot hold the CUDA context at the same time. As a result, the CUDA kernels are executed sequentially for these applications, in a round-robin fashion. In this case, the processing time increases almost linearly with the number of instances, 10.23 ms for 1 instance and 45.28 ms when the number of instances increases to 10. We also utilize the NVIDIA Multi-Process Service (MPS) [39], in order to better utilize GPU resources and improve the scalability of Jaguar. MPS enables multiple CUDA kernels to be processed concurrently on the same GPU. It improves the resource utilization when a GPU's compute capacity is underutilized by a single application, by reducing both on-GPU context storage and GPU context switching.

There are three main observations from Figure 6. First, when there is only one Jaguar server instance, the processing time with MPS is higher (14.37 vs. 10.23 ms) due to the "detour" overhead caused by MPS. Second, when we increase the number of Jaguar server instances, MPS reduces the processing time by up to ~20% by better utilizing GPU resources. Third, when we have more than 9 instances running simultaneously, the performance of using MPS degrades. One possible reason is the imbalance of its scheduling algorithm, evidenced by the higher standard deviation shown in Figure 6. For example, when there are 10 instances, the minimal processing time is 16.35 ms, whereas the maximum is 95.88 ms. We plan to address this scalability limitation of MPS in our future work (*e.g.,* by designing our own scheduler).

5.5 Object Tracking Accuracy

The quality of object tracking is a crucial factor in providing a seamless AR experience. As Jaguar offers 6DoF tracking, we measure the translation error in centimeters (cm) and rotation error in degrees for a sequence of operations to the phone. At the beginning of the experiment, Jaguar utilizes our edge cloud server to recognize a movie poster, determines its physical size on the client, and then tracks the movie poster afterwards. It renders the boundary of the movie poster in red and a green cuboid on top of it to demonstrate the orientation in 3D. We record the tracking results in 6DoF from the phone's screen during a sample sequence of 400 frames, as shown in Figure 7, during which obvious translation and rotation happen. In order to get the ground truth and calculate the tracking errors, we manually label the boundary of the movie poster for each frame and calculate the 6DoF pose accordingly.

Figure 7: Screen records of object tracking in a sequence of smartphone translations and rotations. The image boundary in red is rendered according to the tracking results. The green cuboid is rendered on top of the image to demonstrate the orientation.

Error	Phase 1	Phase 2	Phase 3
X (cm)	0.08 ± 0.04	0.14 ± 0.07	0.23 ± 0.14
Y (cm)	0.06 ± 0.03	0.16 ± 0.10	0.24 ± 0.15
Z (cm)	0.16 ± 0.08	0.10 ± 0.06	0.33 ± 0.20
Roll (degree)	0.40 ± 0.22	0.55 ± 0.28	0.91 ± 0.73
Pitch (degree)	0.45 ± 0.28	1.77 ± 0.86	0.42 ± 0.32
Yaw (degree)	0.49 ± 0.26	0.63 ± 0.29	2.07 ± 1.34

Table 3: Tracking error in three different phases.

We show the translation and rotation errors in Figures 8 and Figure 9, respectively. It is worth mentioning that the physical size of the image shown in Figure 7 is 12.5cm×18.5cm, and the calculated size on the client is 12.8cm×18.7cm (with an error of only around 2.4%). We divide the frames in Figures 8 and Figure 9 into three phases. During the first phase, we rotate the phone as shown in the five sampled frames of the first row in Figure 7. We move the phone left and right in the second phase (the second-row frames of Figure 7) and forward and backward in the third phase (the third-row frames of Figure 7). We present the translation and rotation errors in each phase in Table 3, which demonstrate an increasing trend (with two exceptions). The reason is that when moving the phone left and right, the movement is more dramatic than rotation in phase 1. similarly, forward and backward movement in phase 3 is more intensive than that in phase 2.

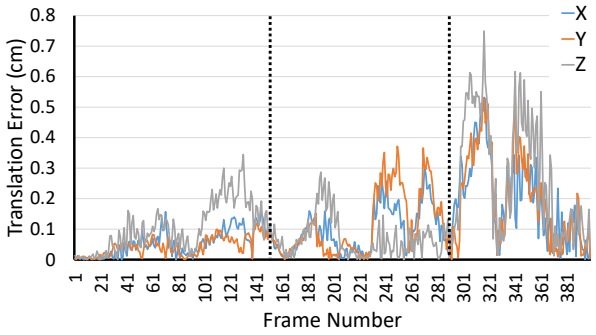

Figure 8: Translation error in centimeters (cm). The physical size of the movie poster is 12.5cm×18.5cm.

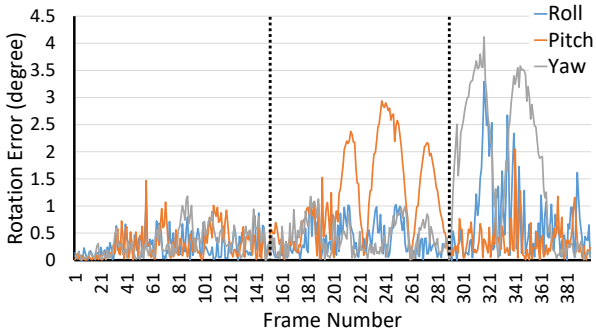

Figure 9: Rotation error in degrees.

5.6 Energy Consumption

We use Batterystats [8], an official Android battery usage profiling tool provided by Google, to measure the energy consumption of Jaguar's client application. We run the application on a fully-charged Samsung Galaxy S8 phone for 10 minutes and export the battery usage profile. During the 10 minutes, the phone consumes 4.71% of the overall battery capacity, indicating that our Jaguar client can run more than 3.5 hours on a fully-charged phone.

6 DISCUSSION

In this section, we discuss the opportunities to further improve the performance of Jaguar and the limitations of our current implementation and evaluation.

6.1 Deep Learning

Deep learning is a promising solution to improve the performance of object-classification based AR systems. For example, Ran *et al.* [42] proposes a distributed infrastructure that combines both front-end mobile devices and cloud servers to enable deep learning for object detection and classification in AR applications. We note that deep learning based image detection methods, such as AlexNet [32], ResNet [26] and GoogLeNet [48], are capable of conducting *object detection and classification*, but not *object recognition*, as their goal is to classify millions of images in various categories (*e.g.,* 1000 different classes for the ImageNet [5] database). However, we cannot blindly apply the deep neural network architecture of AlexNet, ResNet and GooLeNet into Jaguar for image-retrieval based object recognition, because they solve different problems. For example,

we need to know not only the object in an image is a car, but also the model or owner of the car by comparing this image with the reference images in a large-scale database. For object classification, we usually have a large number of reference images in a specific category/class to train the model. For object recognition, each reference image should compose a dedicated class, which makes it difficult to train the model.

6.2 Latency & Bandwidth Tradeoff

When offloading computation intensive tasks of mobile AR to cloud servers, there is a tradeoff between end-to-end latency, object-recognition accuracy, uplink bandwidth utilization, and mobile energy consumption. Here we discuss the tradeoff between latency and bandwidth. There are two choices if we choose to offload tasks starting from object detection by sending camera frames to the cloud [29]. We can upload either individual images or video streams. Although video streams usually have a smaller size and consume less data than individual frames for the same user perceived video quality (*e.g.,* due to inter-frame compression), they buffer frames for encoding on mobile devices and thus introduce extra latency. Also, during the offloading we can either send feature points or upload their compressed fingerprints [30]. The size of fingerprints should be smaller than that of raw feature points, but their generation may consume more energy on mobile devices and introduce extra latency, as it is also a computation intensive task. We plan to improve the performance of Jaguar by further exploring its design space and investigating the above tradeoffs.

6.3 Limitations

There are a few limitations of the current design of Jaguar that we intend to address in the future. (1). Jaguar uses SIFT for feature extraction mainly due to its high accuracy for object recognition. However, there is usually a tradeoff between the accuracy and the feature-extraction time. There may exist other lightweight feature extraction algorithms with reasonable accuracy for object recognition. (2). Although it is more flexible and scalable than others, ARCore currently tracks only the pose change of *static* objects and cannot track *moving* objects. Jaguar inherits this limitation from ARCore. (3). We have not considered offloading motion tracking and rendering of annotation content to the cloud.

7 RELATED WORK

In this section, we briefly review existing work related to augmented reality and continuous vision on mobile devices, and hardware acceleration and image search for various applications.

Augmented Reality. There is a plethora of work on mobile AR. For example, Nestor [24] is a mobile AR system to recognize planar shapes and estimate their 6DoF poses, which applies recursive tracking for achieving an interactive frame rate. Overlay [29] leverages sensor data to narrow down the search space of object recognition by taking into consideration the geometric relation of objects. VisualPrint [30] reduces the bandwidth requirement for uplink by sending compressed feature points (*i.e.,* their fingerprints) extracted from a camera frame to the cloud for object recognition.

Augmented Vehicular Reality (AVR) [41] supports the sharing of visual information among nearby vehicles by aligning automatically coordinate frames of reference. CARS [52] is a collaborative framework to enable cooperations among users of mobile AR applications. Zhang *et al.* [51] evaluated the cloud-offloading feature of commercial AR systems and pinpointed the dominating components of the end-to-end latency for cloud-based mobile AR. Different from the above work, Jaguar explores GPU-based hardware acceleration on cloud servers to significantly reduce the end-to-end latency of mobile AR and complements the recently released ARCore with object recognition capability.

Continuous Vision has recently become an active research topic of mobile computing. In order to improve the energy efficiency of continuous mobile vision, LiKamWa *et al.* [35] identified two energy-proportional mechanisms, optimal clock scaling and aggressive standby mode. Glimpse (software) [17] is a continuous face and road-sign recognition system, which hides the cloud-offloading latency by tracking objects using an active cache of camera frames. Glimpse (hardware) [38] investigates how to accurately and flexibly discard uninteresting video frames to improve the efficiency of continuous vision, by re-designing the conventional mobile video processing pipeline. There are also several recent proposals leveraging deep learning for continuous mobile vision. For example, DeepEye [33] is a small wearable camera that can support rich analysis of periodically captured images by performing locally multiple cloud-scale deep learning models. DeepMon [28] is a deep learning inference system for continuous vision which accelerates the processing by offloading convolutional layers to mobile GPUs. The above work focuses on enhancing the performance and efficiency of continuous mobile vision. Jaguar can potentially benefit from them to optimize the client side execution and further accelerate the cloud side object recognition.

Hardware Acceleration (*e.g.,* with GPU) has been increasingly popular in multiple application domains [20–22, 25]. OpenVIDIA [21] is a library for accelerating image processing and computer vision using GPUs. To facilitate high-quality mobile gaming, Kahawai [20] utilizes collaborative rendering by combining the output of mobile GPUs and server GPUs for display. Georgiev *et al.* [22] optimizes the performance of mobile GPUs for deep-inference audio sensing. PacketShader [25] is a high-performance software router that accelerates general packet processing using GPUs. Similarly, Jaguar leverages server GPUs to speed up cloud-side processing of mobile AR and boost the quality of user experience.

8 CONCLUSION

In this paper, we design, implement and evaluate Jaguar, a low-latency and edge-cloud based mobile AR system with flexible object tracking. Jaguar's client is built on top of ARCore from Google for benefiting from its marker-less tracking feature, and it enhances ARCore with object-recognition and physical size estimation capabilities for context awareness. We systematically investigate how to optimize the image retrieval pipeline in mobile AR by leveraging GPU acceleration on edge cloud and compare its performance with offloading to public cloud with CPUs and with GPUs. Our proof-of-concept implementation for Jaguar demonstrates that it can significantly reduce the end-to-end latency of mobile AR and achieve accurate 6DoF object tracking.

REFERENCES

[1] Android OpenGL ES 2.0/3.0 Engine. https://github.com/Rajawali. [Online; accessed 05-April-2018].

[2] ARCore from Google. https://developers.google.com/ar/. [Online; accessed 05-April-2018].

[3] ARKit from Apple. https://developer.apple.com/arkit/. [Online; accessed 05-April-2018].

[4] FAst Lookups of Cosine and Other Nearest Neighbors (FALCONN). https://falconn-lib.org/. [Online; accessed 05-April-2018].

[5] ImageNet. http://image-net.org/. [Online; accessed 05-April-2018].

[6] Instant Tracking from Wikitude. https://www.wikitude.com/blog-wikitude-markerless-augmented-reality/. [Online; accessed 05-April-2018].

[7] Open Source Computer Vision Library. https://opencv.org/. [Online; accessed 05-April-2018].

[8] Profile Battery Usage with Batterystats and Battery Historian. https://developer.android.com/studio/profile/battery-historian.html. [Online; accessed 05-April-2018].

[9] Recognizing Images in an AR Experience. https://developer.apple.com/documentation/arkit/recognizing_images_in_an_ar_experience. [Online; accessed 05-April-2018].

[10] Smart Terrain from Vuforia. https://library.vuforia.com/articles/Training/Getting-Started-with-Smart-Terrain. [Online; accessed 05-April-2018].

[11] VLFeat. http://www.vlfeat.org/. [Online; accessed 05-April-2018].

[12] A. Alahi, R. Ortiz, and P. Vandergheynst. Freak: Fast Retina Keypoint. In *Proceedings of CVPR*, 2012.

[13] AT&T Labs & AT&T Foundry. AT&T Edge Cloud (AEC) - White Paper. https://about.att.com/content/dam/innovationdocs/Edge_Compute_White_Paper%20FINAL2.pdf, 2017. [Online; accessed 05-April-2018].

[14] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded Up Robust Features. *Proceedings of ECCV*, 2006.

[15] D. Bekele, M. Teutsch, and T. Schuchert. Evaluation of binary keypoint descriptors. In *Proceedings of the 20th IEEE International Conference on Image Processing (ICIP)*, 2013.

[16] M. Björkman, N. Bergström, and D. Kragic. Detecting, segmenting and tracking unknown objects using multi-label MRF inference. *Computer Vision and Image Understanding*, 118:111–127, 2014.

[17] 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 SenSys*, 2015.

[18] J. Cho, K. Sundaresan, R. Mahindra, J. Van der Merwe, and S. Rangarajan. ACACIA: Context-aware Edge Computing for Continuous Interactive Applications over Mobile Networks. In *Proceedings of CoNEXT*, 2016.

[19] E. Cuervo, A. Balasubramanian, D. ki Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of MobiSys*, 2010.

[20] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings of MobiSys*, 2015.

[21] J. Fung, S. Mann, and C. Aimone. OpenVIDIA: Parallel GPU Computer Vision. In *Proceedings of Multimedia*, 2005.

[22] P. Georgiev, N. D. Lane, C. Mascolo, and D. Chu. Accelerating Mobile Audio Sensing Algorithms through On-Chip GPU Offloading. In *Proceedings of MobiSys*, 2017.

[23] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of MobiSys*, 2014.

[24] N. Hagbi, O. Bergig, J. El-Sana, and M. Billinghurst. Shape recognition and pose estimation for mobile augmented reality. In *Proceedings of IEEE International Symposium Mixed and Augmented Reality (ISMAR)*, 2009.

[25] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proceedings of SIGCOMM*, 2010.

[26] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of CVPR*, 2016.

[27] B. K. Horn and B. G. Schunck. Determining Optical Flow. *Artificial intelligence*, 17(1-3):185–203, 1981.

[28] L. N. Huynh, Y. Lee, and R. K. Balan. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of MobiSys*, 2017.

[29] P. Jain, J. Manweiler, and R. Roy Choudhury. Overlay: Practical Mobile Augmented Reality. In *Proceedings of MobiSys*, 2015.

[30] P. Jain, J. Manweiler, and R. Roy Choudhury. Low Bandwidth Offload for Mobile AR. In *Proceedings of CoNEXT*, 2016.

[31] L. Juan and O. Gwun. A comparison of SIFT, PCA-SIFT and SURF. *International Journal of Image Processing (IJIP)*, 3(4):143–152, 2009.

[32] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of NIPS*, 2012.

[33] A. M. 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 MobiSys*, 2017.

[34] S. Leutenegger, M. Chli, and R. Y. Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Proceedings of ICCV*, 2011.

[35] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl. Energy Characterization and Optimization of Image Sensing Toward Continuous Mobile Vision. In *Proceedings of MobiSys*, 2013.

[36] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Journal of Computer Vision*, 60(2):91–110, 2004.

[37] W. Ma, L. Cao, L. Yu, G. Long, and Y. Li. GPU-FV: Realtime Fisher Vector and Its Applications in Video Monitoring. In *Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval*, 2016.

[38] S. Naderiparizi, P. Zhang, M. Philipose, B. Priyantha, J. Liu, and D. Ganesan. Glimpse: A Programmable Early-Discard Camera Architecture for Continuous Mobile Vision. In *Proceedings of MobiSys*, 2017.

[39] NVIDIA. Multi-Process Service - NVIDIA Developer Documentation. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2017. [Online; accessed 05-April-2018].

[40] F. Perronnin, Y. Liu, J. Sánchez, and H. Poirier. Large-Scale Image Retrieval with Compressed Fisher Vectors. In *Proceedings of CVPR*, 2010.

[41] H. Qiu, F. Ahmad, R. Govindan, M. Gruteser, F. Bai, and G. Kar. Augmented Vehicular Reality: Enabling Extended Vision for Future Vehicles. In *Proceedings of HotMobile*, 2017.

[42] X. Ran, H. Chen, Z. Liu, and J. Chen. Delivering Deep Learning to Mobile Devices via Offloading. In *Proceedings of VR/AR Network*, 2017.

[43] D. A. Reynolds, T. F. Quatieri, and R. B. Dunn. Speaker verification using adapted Gaussian mixture models. *Digital Signal Processing*, 10(1-3):19–41, 2000.

[44] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. ORB: An efficient alternative to SIFT or SURF. In *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2011.

[45] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[46] C. Shi, V. Lakafosis, M. H. Ammar, and E. W. Zegura. Serendipity: Enabling Remote Computing among Intermittently Connected Mobile Devices. In *Proceedings of MobiHoc*, 2012.

[47] Y. Suzuki, S. Kato, H. Yamada, and K. Kono. GPUvm: Why Not Virtualizing GPUs at the Hypervisor? In *Proceedings of USENIX ATC*, 2014.

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going Deeper with Convolutions. In *Proceedings of CVPR*, 2015.

[49] K. Tian, Y. Dong, and D. Cowperthwaite. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proceedings of USENIX ATC*, 2014.

[50] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In *Proceedings of USENIX ATC*, 2016.

[51] W. Zhang, B. Han, and P. Hui. On the Networking Challenges of Mobile Augmented Reality. In *Proceedings of VR/AR Network*, 2017.

[52] W. Zhang, B. Han, P. Hui, V. Gopalakrishnan, E. Zavesky, and F. Qian. CARS: Collaborative Augmented Reality for Socialization. In *Proceedings of HotMobile*, 2018.