# SLAM-Share: Visual Simultaneous Localization and Mapping for Real-time Multi-user Augmented Reality

Aditya Dhakal, Xukan Ran, Yunshu Wang, Jiasi Chen, K. K. Ramakrishnan
{adhak001,xran001,ywang1127}@ucr.edu,{jiasi,kk}@cs.ucr.edu
University of California, Riverside
Riverside, CA, USA

## ABSTRACT

Augmented reality (AR) devices perform visual simultaneous localization and mapping (SLAM) to map the real world and localize themselves in it, enabling them to render the virtual holograms appropriately. Current multi-user AR platforms fall short in that they only allow asymmetric sharing of this SLAM information, resulting in multiple "secondary" devices viewing holograms placed by a single "primary" device, instead of equal participation. The goal of this work is to enable *all* AR devices to participate equally, by constructing a common global map to which all AR devices can contribute. However, doing so with low latency and high accuracy is challenging on resource-constrained mobile devices. This work proposes an appropriate partitioning between clients and a server to achieve high-throughput, low latency, multi-user SLAM. In our system, SLAM-Share, the edge server performs the complex SLAM computations so that the client devices need only perform lightweight operations. The server utilizes shared memory and efficient map merging to build and update a global map from different clients. It also exploits the parallelism of GPU processing to achieve high-performance tracking. Evaluations show that SLAM-Share is able to achieve significant tracking speedups (up to 50% reduction compared to alternative approaches), maintain good localization accuracy, and merge and update maps within 200 ms.

## CCS CONCEPTS

• **Information systems** → **Multimedia information systems**; • **Networks** → **Location based services**;

## KEYWORDS

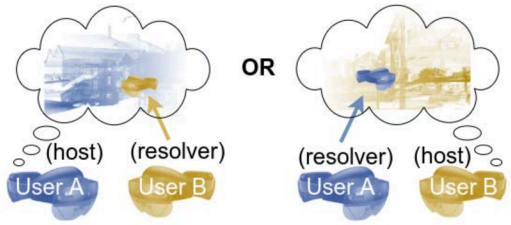
Augmented Reality, Simultaneous Localization and Mapping, Edge Offloading, GPU, Shared Memory

**ACM Reference Format:**
Aditya Dhakal, Xukan Ran, Yunshu Wang, Jiasi Chen, K. K. Ramakrishnan. 2022. SLAM-Share: Visual Simultaneous Localization and Mapping for Real-time Multi-user Augmented Reality. In *The 18th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '22), December 6–9, 2022, Roma, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3555050.3569142

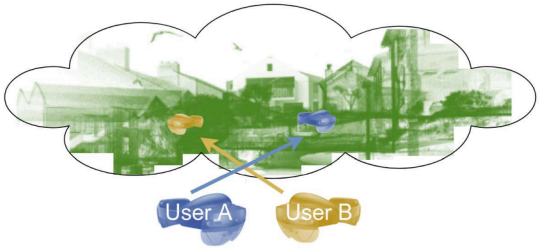


*CoNEXT '22, December 6–9, 2022, Roma, Italy*

ACM ISBN 978-1-4503-9508-3/22/12…$15.00
https://doi.org/10.1145/3555050.3569142




**(a) Asymmetric sharing. Either User B can localize in A's map, or User A can localize in B's map, but not both.**




**(b) In SLAM-Share, all users can localize in a shared map.**

**Figure 1: Current approaches [22, 36] only allow asymmetric map sharing, while SLAM-Share allows all devices to contribute and localize in a shared global map.**

## 1 INTRODUCTION

In augmented reality (AR), holograms are anchored to the real-world environment. For example, holographic graffiti should be anchored to a wall in the real world and remain there even as users move around the real world [12]. In order for an AR device to understand where the wall and the user are, and render the holographic graffiti at the right location on its display, an AR device creates a 3D map of the world (containing the wall) and localizes itself in that map [22, 39]. The process of creating the 3D map and tracking the device (localization) is commonly done using Simultaneous Localization and Mapping (SLAM), relying on camera and IMU sensors. Multiple works [14, 17, 27, 36] have shown that SLAM is a very expensive operation for mobile devices.

It is becoming increasingly common for multiple users to participate in a joint AR session. This may be for safety applications such as autonomous driving or driver-assistance [34], training applications, or a multi-player AR game [21, 32]. For example, in an AR game, if Player A goes to a new, unexplored room in the physical world to draw virtual graffiti, her 3D map should be updated with knowledge of the new room and then shared with the other AR player's devices; that way, if Player B enters the new room, his AR device will immediately recognize the new room and render the new graffiti on its display.

Edge computing can significantly facilitate AR in general and multi-user AR in particular. The simple and obvious reason is that edge computing can alleviate the burden of these computationally expensive tracking and 3D mapping operations of SLAM. For multi-user applications, an edge server can also serve as a central location to facilitate information sharing between AR devices on the current state of an AR task. Finally, edge computing helps compensate for the heterogeneous compute capabilities of different AR devices. Without edge computing, if devices have vastly varying compute power (*e.g.,* some devices have a higher power CPU, a GPU, or more memory), some devices may perform worse than their peers.

However, it is not straightforward for the edge server to provide the desired support for AR devices. The main reason is that the latency requirement of the multi-user AR application is tight. A naive implementation on an edge server may not meet the latency requirements due to all the moving parts that have to operate in sync. multi-user AR is inherently a distributed problem that is challenging to perform in real time. All of the AR SLAM computations – tracking and 3D mapping – need to be performed quickly so that users see the right holograms in the right positions. Tracking must be done in real-time because stale tracking results will cause outdated positions for the hologram(s) on the display as the user moves around (see Fig. 2b, which we discuss in §2). Stale mapping can have serious consequences in terms of misplaced holograms as seen by different users (see Fig. 2c). This is because each user's view is dependent on the merged maps from the other users. Hence, quickly merging all the users' 3D maps to create a consistent view is critical.

Recent approaches to using the edge cloud to support SLAM are insufficient because they focus either on single/few non-concurrent users or asymmetric information sharing. For example, Edge-SLAM [14] offloads some SLAM computations to an edge server for a single user. CarMap [8] performs map merging on the cloud for 1-2 vehicles that are not simultaneously operating. ARCore [22] and SPAR [36] only allow asymmetric sharing, as shown in Fig. 1a. On the left side of the figure, holograms can be placed by one host device (User A) and viewed by all resolver devices (such as User B). However, User B cannot place a hologram and have it viewed by User A without starting a new AR session with B as host (right side of figure). Several works [11, 29, 35, 45] focus on low-latency object detection in AR, which is orthogonal to this work. None enable SLAM for AR with multiple concurrent users, symmetric information sharing, low latency, and high accuracy.

In this paper, we take a first step towards answering the question: What techniques can help SLAM quickly and accurately perform tracking and 3D map merging for multiple AR devices? The fundamental complexity of multi-user AR is that the "state" of the system (*i.e.,* the shared merged map) needs to be updated and communicated to the clients through the tracking and mapping processes. Current approaches splinter this state across multiple devices (Fig. 1a), introducing complexity because it is updated and accessed by multiple user devices in a distributed, asynchronous manner. To overcome this, the main architectural approach we explore in this paper is the judicious use of edge computing for multi-user SLAM for AR. We carefully choose the essential functions to be performed on an end-user's device versus the edge server, leveraging both visual and IMU-based sensing. Our focus is on a server design that unifies all the work performed on behalf of the users, consolidating the state into a consolidated shared map (Fig. 1b).

In brief, the AR devices upload camera frames in real-time to an edge cloud server, which serves as a central vantage point to perform the AR SLAM computations and return only the key results (poses of devices and holograms) to the devices. Thus, the client processes access the shared information on the server for high-throughput, low latency merging of maps across users. This architecture leverages the increasing network bandwidth and low latency provided by current and future wireless networks.

**Contributions.** Our first contribution shifts the majority of the SLAM tracking and mapping computations to a server while running only very lightweight tracking computations on the device. The device performs inertial movement unit (IMU)-based computation only to provide the pose over the short-term while the device is waiting for the server to return a more accurate SLAM-computed pose. The server leverages hardware accelerators often found in edge servers (*e.g.,* GPU) for speed-up tracking.

In parallel with tracking, the 3D maps present on the edge server are merged together to provide a common view for all the devices. The 3D maps used by AR devices for SLAM can be quite large, nearly 40 MB for a minute-long time sequence (see Table 1 in §2), so transferring them over the network, or between processes on a server, to a common thread for merging can be time-consuming.

Our second contribution is a shared memory approach that enables the merging process to access each 3D map quickly. With all the 3D maps gathered in place, our third contribution is the map merging algorithm itself. Our method uses overlapping regions between multiple maps to identify how they are oriented and positioned with respect to each other, then merge them into a global map located in shared memory. The devices then use this global map as a basis for further tracking.

Our implementation, SLAM-Share, is built around a state-of-the-art SLAM framework, ORB-SLAM3 [16]. SLAM-Share reduces the tracking latency by up to 50% using the GPU compared to CPU-only processing and reduces the map merging latency by **at least 30×** compared to a baseline approach without shared memory.We utilize *absolute trajectory error* (ATE) to measure the accuracy of SLAM-Share. ATE is the average deviation of a device's estimated position from the ground truth, typically measured as the root mean squared error (RMSE). The map produced by SLAM-Share enables multiple devices to achieve positional accuracy as good as a single user observing the same environment (< 10cm. (ATE)). The open-source code is available[1].

## 2 BACKGROUND & MOTIVATION

SLAM is a foundational algorithm used in applications such as AR and robotics. The SLAM process typically involves two parallel steps, tracking and mapping. We next provide background on each of these steps to motivate SLAM-Share's design.

**Background on tracking.** Tracking lets a client determine its pose (*i.e.,* position and orientation) in the real world. We call this process **Local Tracking** in Process A of Fig. 3. First, Local Tracking decodes images (potentially extracted from video) obtained from the

[1] https://github.com/network-lab2/slam-share

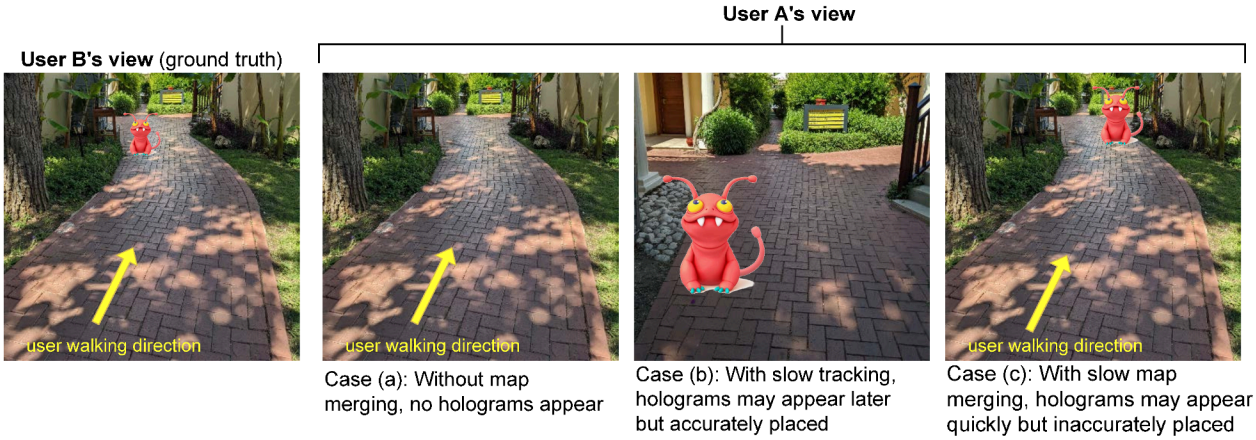



**Figure 2: Different AR views. User B placed a holographic character and user A wants to view it. (a) Current approaches [14, 22, 36] do not provide map merging sharing capabilities, so no holograms would appear. (b) Tracking latency (*e.g.,* 15 FPS [14]) would increase viewing latency, and (c) map merging latency (*e.g.,* > 2 s in our experiments) would reduce hologram placement accuracy.**

client's camera and extracts their image features (**Feature extraction** in Fig. 3). ORB-SLAM3 uses ORB (Oriented FAST and Rotated BRIEF); FAST (Features from Accelerated Segment Test) [38] finds corner features in an image, while Rotated BRIEF are the descriptors for those found features. FAST detects a feature when a pixel's intensity differs greatly from its surroundings. A number (~1000) of ORB features are extracted per image frame, which are compared with the features already seen by the device (**Search local point**) in its local map (described below) in order to localize the device.

Rapid tracking (30 FPS) is critical to a good AR user experience. Fig. 2 shows a mock example with two users. User B (leftmost figure) has placed a holographic character near the intersection (think of User B as the lead vehicle in a networked vehicular safety application). User A (*e.g.,* the vehicle behind that needs to be aware of an obstacle in front) then approaches the same area. If tracking is slow, as in case (b), it takes some time for User A to track its position. Thus, the hologram would only be displayed later (by which time User A may possibly be very close to the intersection). *Hence, rapid tracking is critical for a good AR user experience because it allows the user to view the holograms quickly.* The criticality of timely tracking for applications such as autonomous driving and driver assistance follows.

**Why offload tracking to an edge server?** While ORB-SLAM3's localization runs in real-time on a desktop [16], it is not able to run consistently in real-time (30 FPS) on a resource-constrained device such as an Android phone or an Nvidia Jetson TX2 in our experience. In our experience, when running ORB-SLAM2 on an Android device (ORB-SLAM3, released in 2021, does not yet have an Android version), the user experience is significantly impacted by considerable lag. There were particular slowdowns during certain complex parts of the trajectory, such as turns, where the visual information being processed is more complex. [14] similarly reports frame rate reductions to 15 FPS near turns. A key reason for this is the feature extraction step, which constitutes a majority (57%) of the tracking latency [16]. This motivates running tracking on an edge server in SLAM-Share to achieve real-time performance.

**Table 1: EuRoC MH04 [15] dataset map size.**

| No. of Keyframes | No. of Mappoints | Map Size (MBytes) |
|---|---|---|
| 10 | 825 | 2.74 |
| 20 | 1213 | 4.53 |
| 30 | 1435 | 6.33 |
| 40 | 2002 | 8.10 |
| 50 | 2551 | 10.03 |
| 210 (Full MH04) | 8415 | 38.81 |

**Background on mapping.** SLAM clients create a local map representing the real world concurrently with tracking. We call this process **Local Mapping** in Process A of Fig. 3. If the image read in by Local Tracking is from a previously unseen location and therefore not present on the local map, then SLAM marks the frame as a *Keyframe*. It next computes the poses of the Keyframe's features, known as *Mappoints*, and inserts them into the local map (**Mappoint creation**). Periodically, **Local Bundle Adjustment** runs to correct the pose error of each Mappoint.

When there are multiple users, their maps need to be merged into a global map (**Map Merging**) by synchronizing the coordinate system each client uses to describe its map. This allows devices to have a global map with a common coordinate system to describe the position and orientation of virtual holograms with respect to the real world. Current methods [22, 36] use shared visual landmarks to locate, for example, device B inside device A's map, but do not have true map merging and still rely on a single device's map (*e.g.,* user A, shown on the left side of Fig. 1a) only. In such cases where map merging is unavailable, as shown in Fig. 2a, User A will not be able to see the holograms placed by User B because User A has no idea of User B's map and the holograms contained therein. If map merging is slow or incomplete, then User A will see an inaccurately placed hologram, as shown in Fig. 2(c), because User A is tracking itself in an incomplete global map (without user B's information) and cannot accurately compute its location with respect to the holograms. *Hence rapid mapping is critical to the AR user experience because it allows the user to view the holograms accurately.*

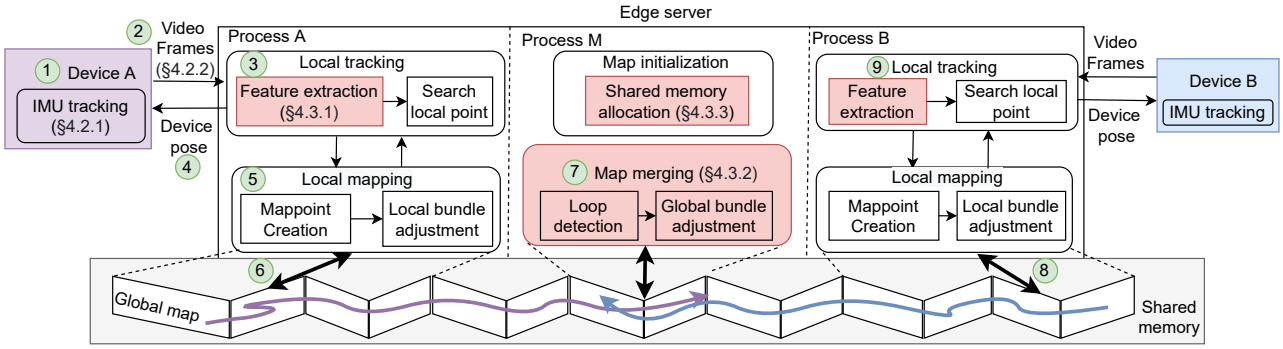



**Figure 3: Overview of SLAM-Share. (1) Device A performs IMU tracking and (2) Uploads frames to Process A on the edge server (3) Server performs local tracking (4) Server returns pose to device. Meanwhile, local mapping in Process A (5) produces map to be loaded (6) into global map using shared memory and merged with existing data where A and B's trajectories overlap (7). Device B also reads the updated global map (8) and tracks itself (9).**

**Why offload mapping to the edge server?** The mapping process has two significant limitations. First, the processing capacity and memory on a mobile user device are typically limited. Having real-time SLAM with a large map is a challenge for these devices. Merging maps on a server takes more than 2 seconds just for the computation (§5.2), and would be even slower on a mobile device. Second, the task of merging the maps of the different users requires them to be brought together in a merging thread. Exchanging new versions of the local maps generated by all clients requires additional processing and network bandwidth. This can dramatically increase latency, particularly when the map sizes are large, as shown in Table 1. Hence performing the mapping on an edge server is preferable to such a peer-to-peer approach.

**What about communication costs?** One consideration is the communication cost of shipping frames to the edge before tracking is performed on the edge server. In our experience, this is quite small, requiring approximately 1 Mbit/s when the frames are encoded as an H.264 video stream and uploaded to the edge server as in SLAM-Share (see §4.2.3). In contrast, even if tracking is performed on the client and is rapid enough (*e.g.*, mobile GPU) it would consume much higher bandwidth (approx. 4 Mbits/s) and energy to send the tracking results to the server for map merging, and get the results back, (based on sending Keyframes and Mappoints similar to [14]). Thus, tracking and mapping on an edge server in SLAM-Share, and transferring video from the client, requires far fewer network resources than alternative architectures.

## 3 RELATED WORK

**SLAM frameworks.** State-of-the-art SLAM frameworks [16, 33] do not provide a mechanism to share maps between clients because they are designed for single-user operation. Edge-SLAM [14] performs single-user SLAM by tracking wholly on the client while mapping is performed on the server. This could be extended to the multi-user scenario considered here, but merging existing maps on the server requires serialization and de-serialization of the maps to transfer them to a common merging thread (*e.g.*, Process M in Fig. 3), costing time and overhead. Minimizing this delay is very important, as it impacts how frequently map merging can occur. [14] also does not utilize accelerators (*e.g.*, GPU) to speed up tracking, resulting in lower frame rates during complex movements. A baseline derived from such an architecture [14] is evaluated for comparison in §5.

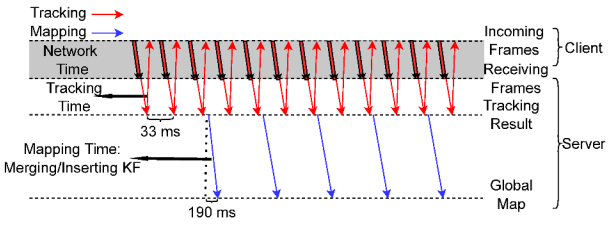
A few other SLAM frameworks [18, 23, 40, 43, 47] are designed for multiple collaborating robots, where clients create local maps, which are then remotely merged. However, this incurs heavy communication cost, having to send large Mappoints and/or Keyframes between the clients and the server, despite compression methods.

**Multi-user AR.** Google Play Services for AR [22] allows multiple AR users to share maps at the beginning of an AR session without further updates as the session progresses, unlike SLAM-Share which provides true map merging. AVR [17] and MARVEL [17] assume an offline 3D map is provided rather than computing it on-the-fly through SLAM. Like SLAM-Share, CarMap [8] focuses on vehicular scenarios and includes map stitching functionality but requires GPS to aid in feature matching due to its use of sparse 3D maps. However, GPS may not be available in indoor AR environments and is not accurate enough on its for the centimeter-level coordinate synchronization required by AR. SPAR [36] allows one user to track itself in another user's map by doing a one-time map alignment, but has no map merging capabilities, as stated in §1 and Fig. 1. SEAR [46] offloads object recognition to the edge to support multiple users. Here, we focus on tracking and mapping, key components of AR. Finally, many works [8, 34, 36, 40] rely on older frameworks such as ORB-SLAM2 [30] or VINS-Mono [33], which typically perform worse than the state-of-the-art ORB-SLAM3 [16] we use here.

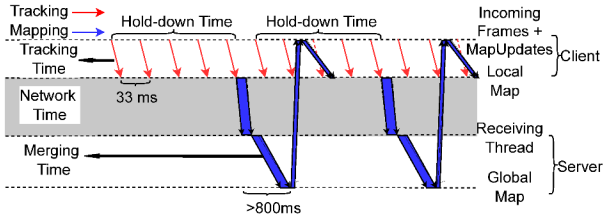
## 4 DESIGN OF SLAM-SHARE

### 4.1 Overview & Workflow

We illustrate the architecture and workflow of SLAM-Share through a running example, and the overall architecture of SLAM-Share is shown in Fig. 3. Consider the case where there are multiple users flying drones through an AR interface [28]. The users see what the drones see, and the AR interface highlights obstacles in the environment that the user must quickly navigate away from to avoid crashing the drone. The position and orientation of the AR highlights are stored in the shared map. Therefore, a drone needs accurate tracking in the shared map in order to understand where the obstacles are and draw the appropriate AR highlights.




**(a) SLAM-Share, where tracking and mapping run remotely on an edge server.**




**(b) Baseline approach (multi-user extension of [14]) where tracking is local and global mapping is remote.**

**Figure 4: Workflow of SLAM-Share vs. Baseline. Red (blue) arrows indicate tracking (mapping) data flow. SLAM-Share keeps the shared map consistent on the server, while the baseline distributes mapping across client and server, causing stale map updates and tracking results.**

**1) Tracking.** As drone A flies into a room, it uses its IMU to obtain an initial pose estimate (① in Fig. 3, §4.2.2). It also records camera frames, encodes them as a video, and streams them (②, §4.2.3) to process A running on the edge server. Process A performs local tracking ③, using the GPU to speed up feature extraction (§4.2.1), and determine the drone's pose in the room. Obstacles in the room are also detected and highlighted on the AR display (although not covered in this work, techniques such as [35] can be used). The computed pose is returned to drone/device A ④, enabling it to record precisely where the obstacles are in the room.

**2) Initial map merge.** In parallel to local tracking, process A generates a local map ⑤ and inputs its map into the global map maintained in shared memory (§4.3.2) across processes ⑥. Process M merges it with the existing global map (⑦, (§4.3.1). This allows drone B, which flies into the same room shortly thereafter, to quickly localize itself by reading the global map in shared memory ⑧ and performing local tracking ⑨. Drone/device B can then view the highlighted obstacles and quickly navigate away from them. If latency is too high without SLAM-Share, process A may not finish merging its map with the global map before drone B arrives in the room; in that case, drone B would view the highlighted obstacle late.

**3) Subsequent map updates.** The shared global map is continuously updated with new observations from drones A and B using SLAM-Share's methods (§4.3.1, §4.3.2) to improve the fidelity of its representation of the real world. If B explores the room further and finds the obstacle is actually closer than A originally marked, it updates this information in the global map ⑧. Because SLAM-Share enables fast map merging, both the initial map merge and subsequent map updates can be done continuously.

**Operation sequence timeline.** We created a timeline of the above workflow in Fig. 4a. The timeline runs from left to right, and the shaded grey areas indicate when data travels between a client and a server. The arrow width is proportional to the size of the network data transmissions. For tracking, frames sent by the client are indicated by thick red arrows at the top periodically, every 33 ms as frames are captured from the camera. They are sent to the server for SLAM processing. The frame is received and tracking is performed by the server. Then the pose (thin red arrow) is computed and returned to the client. Map merging occurs in parallel with tracking and takes around 190 ms. It occurs asynchronously, whenever a client observes something that matches the global map. The insertion of a map update in the global map is indicated by a blue arrow. Thus, in SLAM-Share, the state (shared map) is kept consistent because it's located and updated at a single server. Keeping tracking and mapping tasks together allows clients access to much more up-to-date and consistent state (*i.e.,* fresh map updates are automatically incorporated into tracking results).

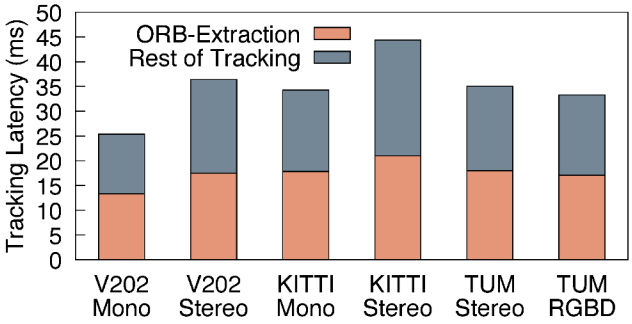



**Figure 5: ORB-SLAM3 tracking latency with CPU. ORB-Extraction is a large portion of tracking latency.**

We contrast SLAM-Share's approach to a representative of the state-of-the-art (by extending [14] to multiple users, evaluated in §5), where tracking is performed locally on the client and map merging is on the server. In Fig. 4b, the tracking results are batched and sent to the server for map merging. The shared map is only periodically updated on a client (*e.g.,* every 5 seconds), resulting in the client potentially having outdated maps to perform tracking in. In essence, the state is distributed (local map on client, global map on server), which leads to poor tracking results when the local and global states are inconsistent . This results in misplaced holograms.

## 4.2 Tracking Enhancements: GPU Tracking, IMU Assistance, & Video Transfers

SLAM-Share offloads the tracking to the edge server. We now describe the tracking enhancement in SLAM-Share, which uses GPU to achieve real-time tracking, client's IMU to increase tracking resiliency, and video to reduce bandwidth usage.

### 4.2.1 Real-time Tracking with GPU

**Problem.** Digging deeper into the root cause for the high tracking latency in SLAM, we find that feature extraction and search local points, the first crucial steps in SLAM tracking, are the bottleneck. We show a breakdown of the average tracking time in ORB-SLAM3 [30] in our testbed (details in §5.1) in Fig. 5. The ORB-SLAM3 feature extraction (red) takes more than 50% of the total tracking time, while search local points (grey) takes about 30%.

This holds across different datasets (KITTI [20], EuRoC V202 [15], TUM [41], and RGBD [41]), as well as different numbers of cameras (mono and stereo), and has also been recognized by others as being slow. [9] The slow feature extraction time in ORB-SLAM3's case leads to higher total tracking times (>34 ms) for a majority of datasets, making it impossible to track the device's pose in real-time (*i.e.,* 30 FPS) and render the holograms at the right locations. Our measurements also align with published benchmarks [30]. Thus, lowering the tracking time is a necessity for real-time AR.

**Our approach.** To address this, SLAM-Share performs tracking on the edge server, using a GPU to speed up its two slow steps: feature extraction and 3D point matching. For feature extraction, the key capability we introduce is the parallelization of "FAST" corner detection [38], needed in SLAM, with the GPU. FAST identifies images features used to find correspondences across images (see §2). This is in contrast to the default approach of searching for the FAST features sequentially in each frame. After GPU processing, the FAST features are transferred to the CPU for SLAM processing.

Our second speedup is to parallelize the code for the *search local point* module, which tries to match the Mappoints extracted from each frame with the Mappoints that exist in a small region of the entire map (known as a local map). The default implementation of *search local point* loops through all the Mappoints from a frame and sequentially matches them with the Mappoints in the local map. We parallelize this matching process in the GPU by creating a local tracking CUDA kernel, performing identical computation as in the original CPU version of the code, but parallelizing the loop iterations to benefit from parallel computations in the GPU.

While others have explored using GPUs to speedup ORB-SLAM2 [1–3, 9] or SLAM algorithm variants requiring 3D depth maps as input [26, 37], in this work we focus on the newer ORB-SLAM3 (released in 2021) without requiring 3D depth maps, as is common on smartphone-based AR devices. Not all SLAM operations can benefit from the parallelism offered by the GPU. SLAM's merging process includes many of random memory lookups (for Mappoints, Keyframes) and serialized operations (*e.g.,* bundle adjustment) that do not benefit from the GPU speedup. Tracking, however, can be parallelized and benefit from GPU acceleration. To the best of our knowledge, ours is the first to use a GPU for the acceleration of select ORB-SLAM3 functions beyond ORB extraction. Our GPU speedup methods improve both feature extraction, in line with previous work [9], as well as the "search local points" function, a key significant contributor to tracking latency. Furthermore, SLAM-Share utilizes spatio-temporal sharing of the GPU [19] to extract features simultaneously and search local points on the data received from multiple client updates. With this, SLAM-Share achieves higher processing throughput and reduces SLAM latency.

#### 4.2.2 Cooperative Client-IMU and Server Tracking

**Problem.** Tracking is a critical component of AR since it allows devices to update their poses in real-time and thus render the holograms at their correct locations on display. Since AR applications like drone navigation, autonomous driving, *etc.* require low-latency localization, it is imperative that the tracking results be available nearly every frame to prevent navigational errors. But tracking in SLAM using a device's camera and IMU is difficult to perform in real-time on mobile devices, as discussed in §2. While we sped up

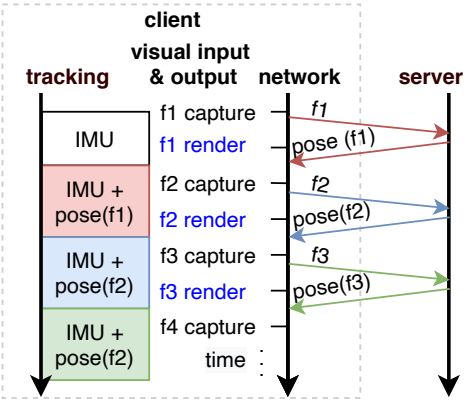


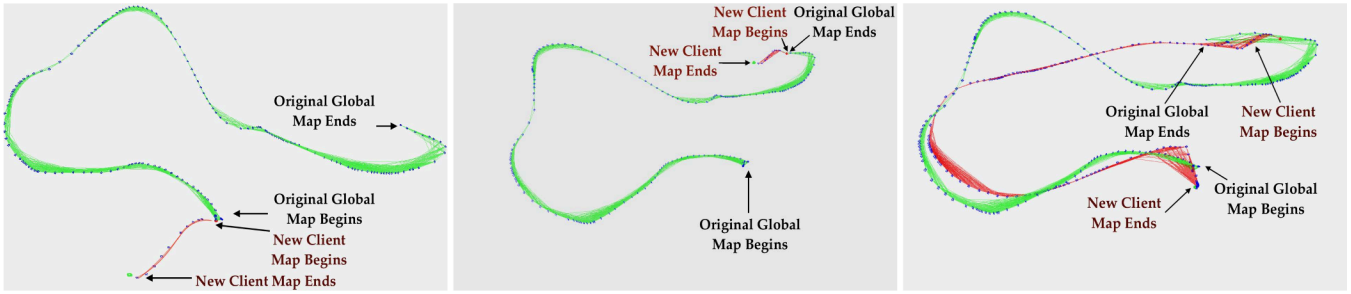

**Figure 6: IMU-assisted pose estimation. Frame 1 (*f1*) sent to the server; meanwhile, the device uses IMU for tracking. Resulting *pose(f1)* from server is incorporated into tracking.**

feature extraction and tracking by doing them on a server, SLAM-Share needs to be resilient to variable communication delays.

**Our approach.** To increase tracking resiliency, we utilize the IMU-based tracking used in ORB-SLAM3 to compute pose in the client itself over a short timescale. Most AR devices (*e.g.,* smartphones, Hololens) have an IMU that provides sensor readings at a high sampling rate (*e.g.,* 1000 Hz [25]), from which position and orientation can be computed. Unlike ORB-SLAM3, which performs both SLAM and IMU-based tracking on the client device, SLAM-Share only performs IMU-based pose prediction on the client. We still depend on the SLAM-based tracking performed at the server since relying on the IMU alone for long periods of time is known to introduce drift errors [25]. Prior approaches [14, 16, 36] have not considered this split processing between the client and server.

We show the client-and-server based tracking timeline in Fig. 6. First video frame 1 (f1) is captured by the camera and transferred to the server for SLAM (red arrow). Meanwhile, the client computes the pose based solely on the IMU since it does not yet have pose information from the server (white box). Once f1's pose information is obtained from the server, the client's tracking is updated (red box) by solving an optimization problem that minimizes the residual error from the IMU-based pose and the server-based pose, trying to ensure that the two sets of observations are consistent with each other [16]; *i.e.,* the optimization problem solves for the pose that best agrees with both sets of observations (using a weighted sum). This should happen when Frame 2 (f2) is captured in order for tracking to be real-time. Incorporating both sources of information (IMU from the client, vision-based pose from the server) achieves higher accuracy than relying on the IMU alone (details in Appendix A).

One potential concern is if it takes too long to get an update from the server. Since tracking based on the IMU alone becomes inaccurate over time because of accumulated drift-related errors, we rely on the IMU-based tracking only during the brief interim period while the client waits for results to be returned from the edge server. In SLAM-Share, with this combination, we limit the accumulated drift error over this short time interval to be very small. This is validated by our experimental results in (§5.3). In the worst case, if the RTT is very high, then a client can fall back to single-user operation with tracking in a local map for the short-term.




(a) Before merging (b) After merging (c) Update after merging

**Figure 7: EuRoc MH4 dataset. Green: Original global map, Red: Map made by new client. Blue dots: Keyframes. SLAM-Share merges the red and green maps into a shared map.**

#### 4.2.3 Using Video Transfers in SLAM-Share

**Problem.** Many open-source visual SLAM systems rely on images (*e.g.,* images in PNG format), and many popular SLAM datasets [15, 41] also provide separate images captured at a certain frame rate. However, each image file can be quite large, and many images are needed to achieve a high enough frame rate. For example, the EuRoC machine hall and Vicon room datasets [15] have PNG images of 331 Kilobytes on average. The KITTI driving dataset's average image size is 225 Kilobytes. Transferring the image files at 30 frames/sec. requires almost 80 Mbps. Stereo SLAM, with two images of that size, doubles the required bandwidth per user. This may be difficult to sustain, especially with wireless uplinks.

**Our Approach:** As explored in the past [13], SLAM-Share encodes images as a H.264 video stream to reduce the bandwidth consumption, rather than transferring PNG compressed images. We quantify the savings compared to network image transfers in §5.4.

### 4.3 Mapping Innovations: Multi-Map Merging Using Shared Memory

To minimize the overhead of communicating individual maps for multiple users, SLAM-Share merges the maps (§4.3.1) using a shared memory framework (§4.3.2) to consolidate all the users' maps.

#### 4.3.1 Merging Maps Between Multiple Clients

**Problem.** In multi-user AR, each user needs to view the same holograms at the same locations and orientations as other users. However, by default, each SLAM client constructs its own real-world map, with each device having a different origin point (*i.e.,* local coordinate $(0, 0, 0)$) for its map. In multi-user scenarios, this poses difficulties because each device has its own understanding of where a hologram, (*e.g.,* with coordinates $(1, 0, 0)$) is located in the real world. A mechanism to merge the maps of the individual clients, and determine a common coordinate system in which the coordinates of the virtual holograms can be represented, is necessary. This problem differs from that in prior approaches [22, 36] ,which perform alignment but not true map merging (Fig. 1).

**Our Approach.** We create a common, shared *global map* representing the real world that all the clients can access. This global map has a common coordinate system in which the positions of the virtual holograms can be accurately represented. The advantage of co-locating mapping and tracking together on the server, compared to distributing them across client and server [14, 40], is two-fold. First, the tracking can immediately take advantage of any map updates without waiting for the usual map refinement processes (*e.g.,* global bundle adjustment) to complete. In contrast, if the shared map needs to be sent to the client for tracking, the map needs to be fully refined before being sent, incurring delays (*e.g.,* global bundle adjustment takes multiple seconds). Second, the reduction of downlink data volume saves time since only the tracking pose updates need to be sent in SLAM-Share (which is a small 4×4 matrix), instead of the maps themselves (Table 1).

To correctly merge the individual maps and generate a global map, it is necessary to find where in the global map a client map "fits" or overlaps. Based on those overlapping regions, the two maps can be aligned, and duplicate regions of the map can be merged. Instead of designing our own ad hoc alignment algorithm, we leverage existing alignment techniques in ORB-SLAM3 [16] that can merge two maps of a single user – however, we modify these methods to merge two maps from two *different users.* By leveraging these mature methods, we expect to achieve good accuracy, thus robustly extending the techniques to the multi-user case (see Appendix B).

Although conceptually simple, it is not easy to make this work in practice. First, when multiple clients merge their maps, there are conflicts between their Keyframe and Mappoint indices, because each client normally starts its indexing with 0. Therefore, we set different starting indices for each client. This involved careful pointer updates and keeping track of relationship Mappoints/Keyframes belong and clients. Second, the multi-user scenario requires merging two existing client maps; this occurs, for example, if one client decides to join the multi-user AR session later and wishes to contribute its own existing map to the global map. The original single-user map merging in ORB-SLAM3 can only trigger a merge based on newly arrived camera frames but does not use previously seen Keyframes in an existing map to trigger a merge, as it assumes past Keyframes have already been checked for a possible merge opportunity. In other words, if we naively extended ORB-SLAM3 to multiple clients, a client that joins later would have to wait until it saw a view with enough overlap with the global map to trigger a merge, instead of merging its map immediately upon joining. To overcome this, when the client joins the session, SLAM-Share checks all of the Keyframes in the client's map for merging opportunities.

**Example.** We illustrate the steps of map merging in an example scenario of a drone that enters a previously explored area, shown in Fig. 7. The map merging is driven by traces from the EuRoc

MH4 dataset [15]. The blue dots in the figure indicate Keyframes in the global map, while the green lines represent the relationship between the Keyframes in the global map. The red line is the map created by a new client, to be merged with the global map.

The leftmost map (Fig. 7a) shows a small map initially created by the new client ("New Client Map") when it enters the room. However, the client's map is not aligned correctly with the global map. SLAM-Share finds where the client's map fits on the global map, merges the two, and runs bundle adjustment to correct the pose of the client's Keyframes and Mappoints. These actions result in the client's map having the correct pose relative to the global map. Fig. 7b shows the maps immediately after merging and bundle adjustment. The pose of the client's Keyframes and Mappoints are corrected and thus, the client's small red trajectory snaps to the correct place in the global map (see the position of the "New Client Map"). As the client continues to explore the room, new images from the environment are processed to update the global map. Fig. 7c shows the entire scenario, where the client continues extending the global map (longer red line). The result is a shared map with the client's information incorporated into the original map, analogous to Fig. 1b.

#### 4.3.2 Shared Memory for Merging Maps

**Problem.** Transferring a large amount of data (1-2 MB or more in Table 1) for the maps of 1-2 secs either across the network (for client-client and client-server) or even between processes in the same node involves considerable overhead in terms of serialization and deserialization of the map, protocol stack processing, and network delays. These overheads impede sharing map updates in real-time.

**Our approach.** In SLAM-Share, we place the global map in a shared memory buffer accessible by all the mapping processes corresponding to each client. Clients localize themselves based on this shared global map and also update the map. The shared memory approach avoids the problem of large file transfers and consequent delays. All the mapping processes are co-located on the edge server (thus no communication over the network). They can access the data directly in the shared memory buffer (no inter-process communication time). The shared memory approach avoids data serialization / de-serialization because the data is placed directly in a memory buffer with all the data structures preserved. Having a single global map in the shared memory implies that any change in the map is instantly available to all the clients. Thus, there are no synchronization delays between the client processes. Finally, shared memory allows zero-copy operations, reducing processing overhead. Once a data structure is initialized in shared memory, it can be accessed by all cooperating client processes, eliminating the need to copy the data or place it in each process' memory.

Utilizing shared memory brings new challenges. Existing libraries such as OpenCV and ORB-SLAM3 do not natively utilize a shared memory buffer. We implemented the necessary constructors, and other primitives and refactored existing ORB-SLAM3 code to exploit the shared memory. ORB-SLAM3 utilizes mutexes to synchronize updates of crucial data structures, such as keyframes and mappoints. In SLAM-Share, we use these mutexes to mediate access to shared memory. To avoid clients idling on a lock, we use Boost's [4] `named-utilities`, which helps us implement a shareable mutex that allows concurrent reads of shared data by threads of multiple processes, while restricting writes to be serialized. This allows SLAM-Share to reduce the amount of locking used for accessing data structures in shared memory. Thus, we do not expect shared memory to be a bottleneck even with more (tens) of users.

An alternative approach to SLAM-Share using a separate SLAM process per client could be to spawn new CPU threads within a single SLAM process for each new client. However, this "new thread per client" approach brings additional complexity and the potential for interference between threads. Even a single SLAM client already requires multiple threads to perform all of SLAM's operations [24]. Thus, it requires additional effort to provide the needed performance isolation and management of the threadpool, all of which is avoided with SLAM-Share. Furthermore, SLAM-Share's one process per client model enables portability to containers to take advantage of the horizontal scaling provided by cloud-based services.

**Implementation details.** We utilize the Boost interprocess library [4] in C++ to create a shared memory framework (gray block at the bottom of Fig. 3). This memory is allocated by an orchestrator process (not shown), separate from the clients using SLAM. We allocate 2 GB of memory buffer for the shared memory. We selected this value based on the size of maps created by different datasets, *e.g.*, EuRoC MH-04 [15] requires approximately 40 MBytes of memory for a map made from the entire trajectory 1. When Process A on the server starts (see Fig. 3), it searches and attaches the shared memory buffer to its own virtual address space. Then, when Process A receives video frames from Device A and generates new Keyframes and Mappoints, it writes those updates to the global map in shared memory (as well as reads others' updates). We created special allocators to allocate complex variable types used in the map directly in the shared memory buffer. We modified ~2000 lines of code from the original ORB-SLAM3 to implement SLAM-Share.

## 5 EXPERIMENTAL EVALUATION

We first describe the setup (§5.1), then evaluate individual components of the framework (§5.2-5.4) , followed by overall accuracy and latency (§5.5), hologram positioning (§5.6), impact of network conditions (§5.7), and resource utilization (§5.8)

### 5.1 Setup

**Testbed.** Our testbed used a Dell PowerEdge R740xd with Intel(R) Xeon(R) Gold 6148 CPU with 40 cores, 256 GB of system memory, one NVIDIA Tesla V100 GPU, and an Intel X710 10GbE quadport NIC as our edge server. The evaluation datasets were the EuRoC [15] and KITTI [20] datasets. The former contains trajectories of drones flying around a large room, while the latter contains vehicular traces. We specifically used the MH04 and MH05 traces from EuRoC, comprising 68 seconds (2032 frames) and 75 seconds (2273 frames) respectively. We also used KITTI-00 and KITTI-05 comprising 151 sec (4541 frames) and 92 sec (2762 frames), respectively. Some experiments use the full 10 Gbit/s client-server link with negligible delay. We also used `tc` [5] to modify the link in both directions, adding either delay (300 ms) or bandwidth constraints (18.7 Mbit/s or 9.4 Mbit/s). 18.7 Mb/s is the minimum bandwidth for the server to send the largest map to the client within 5 seconds, in the baseline method described below, without the map's packets incurring significant queuing delay waiting for transmission on the link. Then, we further restrict the bandwidth to half of that (9.4 Mbit/s).

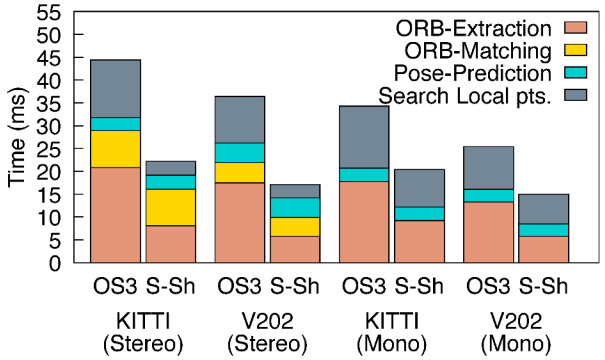



**Figure 8: ORB-SLAM3 (OS3) vs. SLAM-Share with GPU (S-Sh) tracking latency with mono and stereo version of KITTI and EuRoC (V202). SLAM-Share greatly reduces tracking latency.**

**Metrics.** We evaluate two main metrics: latency and absolute trajectory error (ATE). Rendering times are typically a tiny proportion of the overall AR pipeline [11, 44], so we neglect this latency component. We measure the cumulative ATE (along the whole trajectory of the client), and the short-term ATE (along the last 5 seconds of the client's trajectory). The former is the typical ATE measurement for SLAM, while the latter represents the user's most recent experience (see Appendix C). If the ATE is high, then the hologram will be misplaced on the user's display [39]. We also measured the FPS for all experiments.

**Baseline.** Our baseline is a multi-user extension of [14], with each client performing tracking and mapping locally (no GPU). The map merging takes place on a server, as in [18] (see Fig. 4b). This local map at the client is serialized, *i.e.,* local map's data (Keyframes, Mappoints, and their relationship) is stored in a single transmittable buffer to send across the network to the server. At the server it is deserialized, *i.e.,* converted to map data usable by the SLAM program and merged with any other maps present. A portion of the global map (containing approximately 6 keyframes) is sent back to the client and merged with its existing local map. Tracking then continues on this local map. This occurs every 150 frames. We implemented this baseline from scratch.

## 5.2 Real-Time Server GPU Tracking

In this set of experiments, we examined the tracking latency of the default ORB-SLAM3 running on the server without GPU and SLAM-Share on the server using our GPU speedup techniques (§4.2.1). We show the breakdown of tracking time in Fig. 8, including feature extraction (ORB-Extraction), feature matching (ORB-Matching), pose prediction, and finally search local point. SLAM-Share's ORB feature extraction (peach bars) reduces the tracking latency by more than 50% compared to the default ORB-SLAM3. SLAM-Share also achieves a 25-50% reduction in the local tracking time (gray blocks) compared to the default ORB-SLAM3. Overall, SLAM-Share reduces the total tracking latency by ~40% with monocular datasets and by more than 50% with stereo datasets. Thus, SLAM-Share can achieve real-time performance (< 33 ms per frame).

## 5.3 Client-IMU Assist for Tracking

We present experimental results validating §4.2.2 to show the usability of IMU for accurate tracking over short periods of time. A

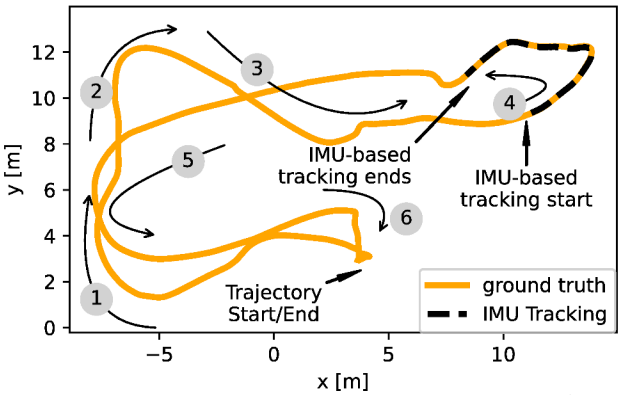



**Figure 9: Map of MH04 dataset with a small region (black dashed line) where server contact is lost.**

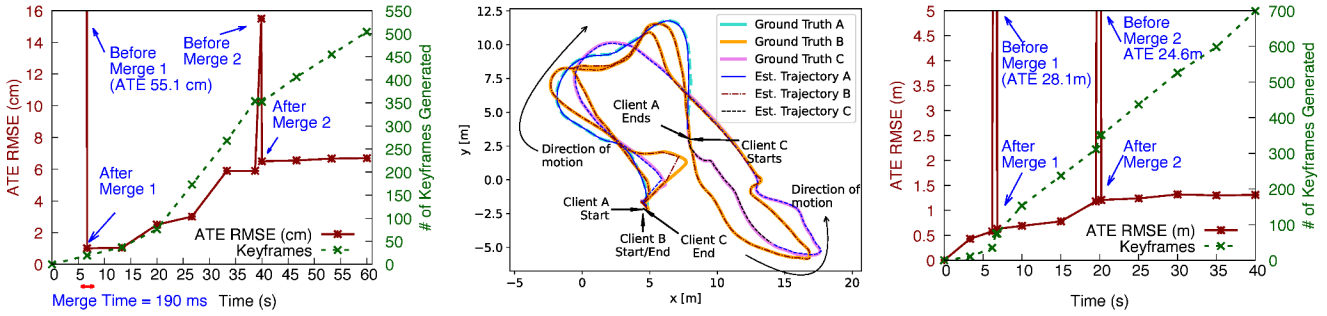
**Table 2: Accuracy of IMU-compensated pose computation with increasing RTT.**

| RTT (ms) | Whole Map ATE RMSE (cm) | Small Map Region ATE RMSE (cm) |
|---|---|---|
| 0 (baseline) | 5.91 | 2.41 |
| 30 | 5.91 | 2.41 |
| 60 | 5.91 | 2.41 |
| 90 | 5.92 | 2.45 |
| 167 | 5.97 | 2.61 |
| 200 | 6.08 | 2.67 |
| 300 | 6.12 | 2.71 |
| 1000 | 6.58 | 3.13 |

**Table 3: Video vs. image transfer, monocular & stereo. SLAM-Share consumes less bandwidth.**

| | Image Transfer | | SLAM-Share | |
|---|---|---|---|---|
| | KITTI-00 Stereo | MH-05 Mono | KITTI-00 Stereo | MH-05 Mono |
| 30-fps Bitrate | 131 (Mbits/s) | 81 | 1.93 | 1.1 |
| Encoding (ms) | - | - | 2.7 | 2.6 |
| Decoding (ms) | 1.2 | 1.1 | 1.2 | 1.1 |
| ATE RMSE (m) | 1.31 | 0.07 | 1.31 | 0.07 |

ground truth trajectory (gold color) from the EuRoC dataset (trajectory ID MH04) is plotted in Fig. 9. The client starts at the bottom middle and moves around the room in the direction of arrows 1-6. We also show how the loss of server pose information in a small region of the map (dashed black line) for a period of time can affect the device's tracking accuracy. SLAM-Share's client uses the last SLAM-based pose returned from the server, *i.e.,* at ④, combined with the client's IMU motion model to estimate the pose in that region. We chose this region of the map with a sharp turn as a stress test. We varied the time span when the pose returned from the server is delayed (*i.e.,* the RTT) and measured the ATE in Table 2. When using IMU-based tracking for a short period of time, both the overall accuracy of the map and the accuracy of the small section of the map do not decrease significantly. In situations where RTT is high (100ms-300ms), SLAM-Share still creates a very accurate map with ATE, similar to the 0 RTT ATE. Even when the pose for an entire second (30 frames) is missing from the server, the accuracy is only reduced slightly. This is in contrast to IMU-alone based tracking, which suffers from 300 cm error after 10 seconds [42].




**(a) Cumulative ATE vs. time of SLAM-Share's global map (EuRoC dataset) at 30 FPS.**

**(b) Final trajectory with 3 clients, corresponding to Fig. 10a.**

**(c) Cumulative ATE vs. time of SLAM-Share's global map (KITTI-05 dataset) at 30 FPS.**

**Figure 10: SLAM-Share maintains high accuracy (low cumulative ATE) as multiple clients maps merge with global map.**

## 5.4 Video from Client in SLAM-Share

We show the benefits of uploading video from the client camera rather than individual images and the subsequent bandwidth savings. The maximum bandwidth between the client and server is 1 Gbps, so the link was not the bottleneck. As seen in the top half of Table 3, sending images at 30 FPS requires wireless bandwidth of 80 Mbits/sec for monocular SLAM. It only takes an average 1.1 of Mbits/sec to transfer the images as video frames (over TCP). Note that there is a small delay to encode the video on the client (< 3 millisec), unlike images. At the receiver, both video and images have to be decoded to raw pixel values before processing by SLAM, taking approximately the same time for decoding. The ATE when using images versus video as input to vanilla ORB-SLAM3 is about the same. Video transfers in SLAM-Share provide accurate inputs for SLAM and are much more efficient than image transfers.

## 5.5 Mapping Accuracy and Latency

**Accuracy.** We consider a timeline of three clients creating their local maps, merging them on the server into a global map, and continuously updating the global map (§4.3.1). We show the accuracy of the global map compared to the ground truth over this period.

First, a global map using 200 frames is created by client A (based on MH04). Then, client B joins with its local map containing 200 frames (based on MH05). We show the progression of the resulting accuracy in terms of ATE in Fig. 10a. At first, before merging client B's local map into the global map, the ATE of the global map is very high (55.1 cm). This is because the two maps exist separately as two different fragments with different origins, and are not aligned correctly (similar to Fig. 7a). SLAM-Share performs map merging at the 6.67 second mark ("Before Merge 1" in Fig. 10a). The ATE immediately drops to 1.01 cm upon merging ("After Merge 1" in Fig. 10a) because the pose from the client B's map is adjusted to the pose of the global map (similar to Fig. 7b). This reduces the error.

After the initial map merge, both clients A and B move and add to the global map. The Keyframe count increases with time, as the larger global map is built. The ATE grows slightly, with the increase being higher (*e.g.,* 25-30 s) in a difficult part of the trajectory (clients A and B are at the top corner in Fig. 10b). After 40 seconds, user A stops and a 3$^{rd}$ client (client C) joins the AR session and adds its local map containing 200 frames. This results in the ATE initially going up to about 15 cm ("Before Merge 2" in Fig. 10a), but immediately drops back down after merging completes ("After Merge 2"). Subsequently, only clients B and C continue. The ATE is flat (~6.5cm vs. 7.1cm for baseline), which is highly desirable.

With regards to scalability, a possible concern is that as more clients join and as each contributes information to the global map, the ATE might increase drastically. This may be because the map covers a larger area and it is difficult to have good tracking performance (low error) in all parts of the map. However, Fig. 10a suggests this is not an issue. While the ATE does spike initially when user B or C joins, it reduces immediately after the merge.

We illustrate the corresponding shared map estimated by SLAM-Share in the above experiment in Fig. 10b. The estimated trajectory is shown by the blue line (Client A), red dash-dot line (Client B), and black dashed line (Client C). These trajectories are overlaid over the ground truth of their trajectories shown in the same but lighter colors. Clients A and B start from the same location and move towards the northeast using different paths. Meanwhile, client C starts in the middle and moves southeast. We can see that the estimated trajectories are close to the ground truth. Thus, SLAM-Share's merged global map is accurate for tracking.

**Vehicular dataset.** We perform a similar experiment using a vehicular KITTI-05 dataset in Fig. 10c. We divide the KITTI-05 dataset into 3 parts, with each segment considered as a different client driving on the street in an area of 500 x 600 square meters. Client A starts initially, and at the 6.5 second mark, client B joins. The map error increases to 28.1 meters, but once SLAM-Share merges client B's map with the global map (in about 150 ms), the map's accuracy comes back to a relatively low error of 0.6 m. Another merge occurs at about 20 seconds. This third client's map is merged into the global map, and the ATE increases prior to the merge but again decreases to a low value once the merge is completed in about 180 ms. Finally, the ATE remains about 1.68 meters until the end of the trajectory. For comparison, the single user ORB-SLAM3 baseline ATE was 1.72 m. Overall, these results show SLAM-Share successfully merges multiple maps into a global map, and multiple clients can continue simultaneously updating it with high accuracy.

**Latency.** We break down the components of the map merging time for SLAM-Share and the baseline. We take the average across 10 runs of the EuRoC dataset for this breakdown, shown in Table 4. The data transfer time is measured from when the data transmission starts at the sender to when the final ACK is received back.

**Table 4: Avg. Latency breakdown of SLAM-Share & baseline.**

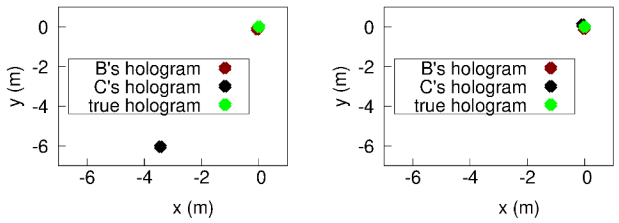
| Component | Baseline (ms) | SLAM-Share (ms) |
|---|---|---|
| 1. Hold-down Time | 5000 | N/A |
| 2. Serialization | 78.1 | N/A |
| 3. Encoding | N/A | 3 |
| 4. Data Transfer 1 | 66 | 0.11 |
| 5. Deserialization | 390.8 | N/A |
| 6. Map Merging | 2339 | 190 |
| 7. Data Processing | 132 | N/A |
| 8. Data Transfer 2 | 6.4 | 0.1 |
| 9. Load Map | 19.8 | N/A |
| **Total** | **8006** | **193** |

The total latency of the baseline approach ranges from 5.8 to 10.8 seconds, depending on the map size and complexity - 8 seconds on average. The major components contributing to the baseline's total latency are the user-specified hold-down time and the map merging time. The hold-down time is needed to batch enough data to send to the server for map merging; a shorter time would result in more frequent network communications and increased overhead. The map merging time is long for the baseline because it first needs to wait for the client to send the map update information to the server (serialization, data transfer 1, and de-serialization steps) and then perform full map merging before sending the updated map back to the client (data transfer 2). In contrast, SLAM-Share operates directly with the map in memory without communicating large map updates back and forth. This results in SLAM-Share taking only 193 ms on average. Overall, SLAM-Share reduces latency at least 30× for merging two maps compared to the baseline. Subsequently, SLAM-Share takes less than 200 ms for map updates thereafter, whereas the baseline requires > 5.8 seconds every merging round.

In terms of frame rate, our method always achieves at least 30 FPS for all traces throughout the entire device trajectory, even with turns (see trajectory examples in Figs. 7, 9, 10b). This is in contrast to prior work [14], which reported lower frame rates, especially at difficult turns. The key enablers of this are SLAM-Share's IMU-based tracking when network delay is high and fast computations at the server, allowed results to be returned in < 33 ms.

## 5.6 Impact of Sharing on Positioning

To illustrate the impact of ATE on hologram positioning, we plot the position of a common hologram as perceived by users B and C in Fig. 11. The scenario considered is as follows. User B first places a hologram and then moves around as in Fig. 10b. Subsequently, when user C joins and locates the hologram on her viewport, we capture a snapshot of the hologram's position as perceived by each user. Fig. 11a shows the perceived positions of the hologram in the real world if no map merging is available, and Fig. 11b shows the hologram's perceived positions with SLAM-Share. The only information shared between users is the coordinates of the hologram. Ideally, all users should perceive the hologram's position identically (*i.e.,* all the dots should be on top of each other in the plot). With SLAM-Share (Fig. 11b), this is indeed the case (the slight dissimilarities are a result of small amounts of ATE). However without sharing (Fig. 11a), device C's estimated hologram position (black dot) is 6.94 m away from the ground truth (green dot). This is because device C starts at a different place compared to user B and does not know




**(a) Without sharing.** **(b) With SLAM-Share.**

**Figure 11: (a) Without sharing, clients B and C have inconsistent views of the hologram's position. (b) With sharing, the clients' estimated hologram positions are close to the truth.**

where it is located on the shared map. Device C assumes that its starting location is the origin and erroneously places the hologram relative to that origin. This result quantifies the benefit of sharing with SLAM-Share compared to the no map merging scenario.

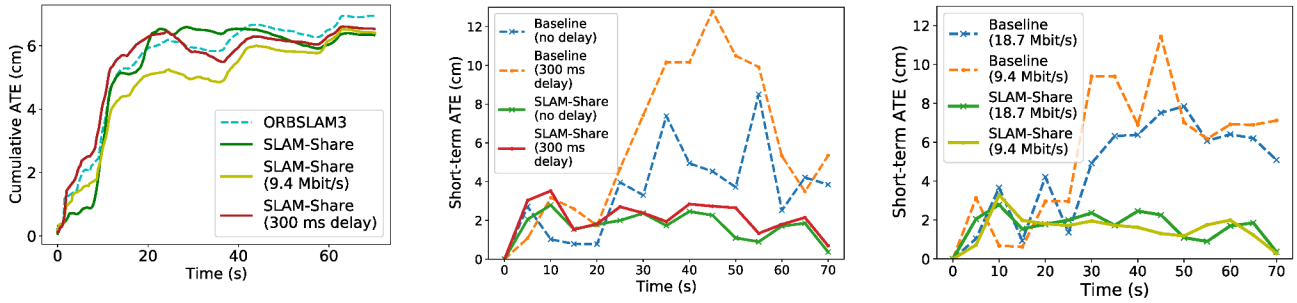
## 5.7 Impact of Network Conditions

We now evaluate SLAM-Share and the baseline for the same scenario as Fig. 10b, to demonstrate the resilience of SLAM-Share as network conditions (bandwidth, delay) change.

**Cumulative ATE.** In Fig. 12a, we present the cumulative ATE of SLAM-Share for the multi-user trajectory as in Fig. 10b, just from user B's perspective (MH05). We examine it with added delay (300 ms) or bandwidth constraints (18.7 Mbit/s or 9.4 Mbits/sec), as described in §5.1. We also plot the cumulative ATE of by a single client following the same trajectory (MH05) in the vanilla ORB-SLAM3 (the "ORBSLAM3" line in Fig. 12a). From Fig. 12a, we see that all the cumulative ATEs show a rise in ATE upto ~30 seconds, after which the ATE levels off, achieving close to the final ATE values reported in [16]. We find that even with increased network delay or decreased bandwidth, SLAM-Share provides about the same or better (lower) ATE compared to the single-user ORB-SLAM3 at the same point of time in the trajectory, despite supporting multiple clients. SLAM-Share's low bandwidth requirement for client-server communication, as well as the use of IMU to produce accurate pose even when with the network delay, enables it to create maps with accuracy as good or better than single-user ORB-SLAM3.

**Short-term ATE.** The short-term ATE (the last 5 seconds of the device's trajectory) reflects the device's most recent tracking error (see Appendix C). Therefore, we evaluate the impact of increased delay and reduced bandwidth on the short-term ATE, for both the baseline and SLAM-Share. Fig. 12b shows the impact of adding a 300 ms delay. SLAM-Share has a better (lower) short-term ATE compared to the baseline. For example, SLAM-Share's short-term ATE consistently stays below 4 cm, whereas the baseline's ATE fluctuates and can reach 12 cm due to the additional delay, as the baseline does not get updates from the server in real-time. On the other hand, SLAM-Share has the IMU to help compensate for the delayed responses from the server.

Fig. 12c shows the impact of bandwidth on performance. Again, we see that SLAM-Share is able to consistently maintain a low ATE, regardless of the bandwidth restriction. However, the baseline's short-term ATE with 18.7 Mbit/s is already higher than without any bandwidth restriction (compare with "Baseline (no delay)" in Fig. 12b. This is because, with even more bandwidth, the baseline's




**(a) Cumulative ATE: Effect of reducing network bandwidth or increasing network delay for SLAM-Share. Compared to ORB-SLAM3.**

**(b) Short-term ATE: Effect of increased network delay - baseline vs. SLAM-Share, no bandwidth limitation.**

**(c) Short-term ATE: Effect of reduced network bandwidth - baseline vs. SLAM-Share, no added network delay.**

**Figure 12: Impact of network conditions on SLAM-Share and the baseline. Bandwidth restrictions or increased delay hurt the baseline's performance (higher ATE), while SLAM-Share is robust to different network conditions and maintains lower ATE.**

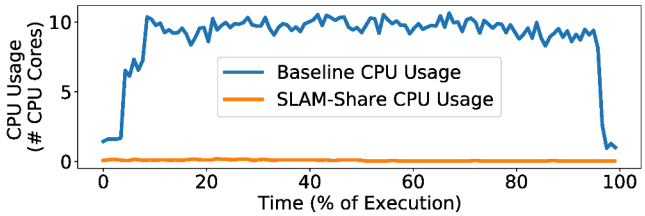



**Figure 13: Client B's CPU usage comparison between baseline and SLAM-Share. Client is moving along the MH05 trajectory.**

client receives the server map updates more quickly and positions itself earlier, reducing the short-term ATE. The baseline's short-term ATE increases even more when the bandwidth is halved, because its high bandwidth requirement is unmet and updates from the server arrive late. In fact, by the end of the trajectory, a baseline client misses 38% of the updates from the server, and thus suffers from higher ATE. Finally, note that the baseline's ATE rises in the middle part of the trajectory because it is a challenging region where client B makes a quick turn. However, SLAM-Share is able to maintain low ATE because the server provides client B with global information about that region, seen by user A.

With regard to resilience to communication constraints, SLAM-Share requires approximately 1-2 Mbit/s on the uplink (Table 3) per client and is not significantly affected by moderate bandwidth restrictions (Fig. 12c), unlike the baseline that requires nearly 20 Mbit/s per client. With typical 5G uplink bandwidths exceeding 200 Mbps [31] and WiFi exceeding 300 Mbps [6], the network seems unlikely to be a bottleneck impacting SLAM-Share's performance even as the number of clients increases. Also note that the typical scale we envisage for a multi-user AR scenario of clients sharing the same physical space and global map, as studied in this work, would be on the order of tens of users.

### 5.8 Resource Utilization

Since client devices may be battery-powered mobile devices, we believe SLAM-Share can provide considerable energy savings, as the computation tasks performed on the client are considerably simplified, We focus on computation tasks because communication tasks in multi-user AR generally consume less energy [10]). SLAM-Share requires the client to perform much less processing on the client in terms of device tracking, local bundle adjustment, map merging, etc. Even though a SLAM-Share client still has to perform video compression, we expect it to use less CPU (and hence energy) than a baseline client, which has to run full SLAM.

To quantify this, we measured the CPU utilization on clients for both SLAM-Share and the baseline. We utilize the `psutil` tool [7] to capture CPU utilization. We run it in the background while running the client of baseline or SLAM-Share on a server in our testbed (§5.1). `psutil` measures the CPU cycles used by deserializing the application while executing user-space or kernel-level code, reporting the overall CPU utilization as *CPU% utilized* (for our experiment, 100% CPU utilization means all the 40 CPU cores are fully utilized). Fig. 13 shows the relative CPU consumption of both methods over the entire MH05 trajectory of user B. The baseline uses almost 10 CPU cores throughout the experiment (25% of 40 cores). On the other hand, SLAM-Share hardly consumes any CPU (0.7% of one CPU core). We believe this 35× decrease will directly translate to critical energy savings on battery-powered client devices.

## 6 CONCLUSIONS

In SLAM-based multi-user AR, the participating devices need to merge their maps of the real world together into a common shared map. They use this shared map to localize themselves and render the virtual holograms. Instead of distributing the shared map (representing the "state" of the AR session) across different devices, leading to inconsistent state accesses by different devices and hence inconsistent hologram visualizations, we propose an edge-centric architecture that consolidates the work performed on behalf of the users into a unified state at the server. Our framework, SLAM-Share, does this quickly and accurately through cooperative tracking between the server and client and fast map merging using shared memory. With SLAM-Share, AR devices can achieve tracking rates of 30 FPS and merge their maps in $< 200$ ms, enabling devices to localize themselves accurately and view the virtual holograms in real-time. *This work does not raise ethical issues.*

## ACKNOWLEDGMENTS

We thank the anonymous reviewers and shepherd for their valuable comments, from which this paper greatly benefited. This work was partially supported by the NSF grants CNS 1817216, CAREER 1942700, and CRI 1823270.

## REFERENCES


[1] 2019. ORB-SLAM2 4 MatlabSimulink. https://github.com/falfab/orb_slam_cuda. (2019).
[2] 2019. ORB-SLAM2-GPU. https://github.com/yunchih/ORB-SLAM2-GPU2016-final. (2019).
[3] 2019. ORB_SLAM2_CUDA. https://github.com/thien94/ORB_SLAM2_CUDA. (2019).
[4] 2021. Boost C++ Libraries. https://www.boost.org/. (2021).
[5] 2021. tc(8) - Linux Manual Page. https://man7.org/linux/man-pages/man8/tc.8.html. (2021).
[6] 2022. AT&T Speed Test. https://www.highspeedinternet.com/tools/speed-test/att. (2022).
[7] 2022. psutil library. (2022). Retrieved October 21, 2022 https://psutil.readthedocs.io/en/latest/.
[8] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. 2020. {CarMap}: Fast 3D Feature Map Updates for Automobiles. In *USENIX NSDI*.
[9] Stefano Aldegheri, Nicola Bombieri, Domenico D Bloisi, and Alessandro Farinelli. 2019. Data flow ORB-SLAM for real-time performance on embedded GPU boards. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE.
[10] Kittipat Apicharttrison, Jiasi Chen, Vyas Sekar, Anthony Rowe, and Srikanth Krishnamurthy. 2022. Breaking edge shackles: Infrastructure-free collaborative mobile AR. *ACM SenSys* (2022).
[11] Kittipat Apicharttrisorn, Xukan Ran, Jiasi Chen, Srikanth V Krishnamurthy, and Amit K Roy-Chowdhury. 2019. Frugal following: Power thrifty object detection and tracking for mobile augmented reality. In *ACM SenSys*.
[12] AT&T Labs. [n. d.]. Air Graffiti Mobile Application. https://www.att.com/gen/press-room?pid=22691. ([n. d.]).
[13] Christoph Bachhuber, Alvaro Sanchez Martinez, Rastin Pries, Sebastian Eger, and Eckehard Steinbach. 2019. Edge cloud-based augmented reality. In *IEEE International Workshop on Multimedia Signal Processing (MMSP)*.
[14] Ali J Ben Ali, Zakieh Sadat Hashemifar, and Karthik Dantu. 2020. Edge-SLAM: edge-assisted visual simultaneous localization and mapping. In *ACM MobiSys*.
[15] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. 2016. The EuRoC micro aerial vehicle datasets. *The International Journal of Robotics Research* (2016).
[16] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. 2021. Orb-slam3: An accurate open-source library for visual, visual–inertial, and multimap slam. *IEEE Transactions on Robotics* 37, 6 (2021), 1874–1890.
[17] Kaifei Chen, Tong Li, Hyung-Sin Kim, David E Culler, and Randy H Katz. 2018. Marvel: Enabling mobile augmented reality with low energy and low latency. In *ACM SenSys*.
[18] HA Daoud, Sabri AQ Md, CK Loo, and AM Mansoor. 2018. SLAMM: Visual monocular SLAM with continuous mapping using multiple maps. *PloS one* 13, 4 (2018).
[19] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *ACM Symposium on Cloud Computing (SoCC)*.
[20] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *IEEE CVPR*.
[21] Google. [n. d.]. Just a Line - Draw anywhere, with AR. https://justaline.withgoogle.com/. ([n. d.]).
[22] Google. 2022. Cloud Anchors allow different users to share the experience. https://developers.google.com/ar/develop/cloud-anchors. (2022).
[23] Marco Karrer, Patrik Schmuck, and Margarita Chli. 2018. CVI-SLAM—collaborative visual-inertial SLAM. *IEEE Robotics and Automation Letters* 3, 4 (2018), 2762–2769.
[24] Georg Klein and David Murray. [n. d.]. Parallel tracking and mapping for small AR workspaces. In *IEEE and ACM International Symposium on Mixed and Augmented Reality*.
[25] Steven LaValle. 2016. *Virtual reality*. Cambridge University Press.
[26] Donghwa Lee, Hyongjin Kim, and Hyun Myung. 2012. Gpu-based real-time rgb-d 3d slam. In *IEEE International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*.
[27] Peiliang Li, Tong Qin, Botao Hu, Fengyuan Zhu, and Shaojie Shen. 2017. Monocular visual-inertial state estimation for mobile augmented reality. In *IEEE ISMAR*. IEEE, 11–21.
[28] Chuhao Liu and Shaojie Shen. 2020. An Augmented Reality Interaction Interface for Autonomous Drone. In *IEEE/RSJ IROS*.
[29] Luyang Liu, Hongyu Li, and Marco Gruteser. 2019. Edge assisted real-time object detection for mobile augmented reality. In *ACM MobiCom*.
[30] Raul Mur-Artal and Juan D Tardós. 2017. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE transactions on robotics* 33, 5 (2017), 1255–1262.
[31] Arvind Narayanan, Xumiao Zhang, Ruiyang Zhu, Ahmad Hassan, Shuowei Jin, Xiao Zhu, Xiaoxuan Zhang, Denis Rybkin, Zhengxuan Yang, Zhuoqing Morley Mao, et al. 2021. A variegated look at 5G in the wild: performance, power, and QoE implications. In *ACM SIGCOMM*.
[32] Niantic. 2022. Shared AR Experience with your Buddy. https://niantic.helpshift.com/hc/en/6-pokemon-go/faq/2146-shared-ar-experience-with-your-buddy/. (2022).
[33] Tong Qin, Peiliang Li, and Shaojie Shen. 2018. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics* 34, 4 (2018), 1004–1020.
[34] Hang Qiu, Fawad Ahmad, Fan Bai, Marco Gruteser, and Ramesh Govindan. 2018. Avr: Augmented vehicular reality. In *ACM MobiSys*.
[35] Xukan Ran, Haoliang Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. 2018. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM*.
[36] Xukan Ran, Carter Slocum, Yi-Zhen Tsai, Kittipat Apicharttrisorn, Maria Gorlatova, and Jiasi Chen. 2020. Multi-user augmented reality with communication efficient and spatially consistent virtual objects. In *ACM CoNEXT*.
[37] Adrian Ratter, Claude Sammut, and Matthew McGill. 2013. GPU accelerated graph SLAM and occupancy voxel based ICP for encoder-free mobile robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*.
[38] Edward Rosten and Tom Drummond. 2006. Machine learning for high-speed corner detection. In *European conference on computer vision*. Springer, 430–443.
[39] Dieter Schmalstieg and Tobias Hollerer. 2016. *Augmented reality: principles and practice*. Addison-Wesley Professional.
[40] Patrik Schmuck and Margarita Chli. 2017. Multi-uav collaborative monocular slam. In *IEEE ICRA*.
[41] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. 2012. A Benchmark for the Evaluation of RGB-D SLAM Systems. In *IEEE/RSJ IROS*.
[42] Yunshu Wang, Lee Easson, and Feng Wang. 2021. Testbed development for a novel approach towards high accuracy indoor localization with smartphones. In *ACM Southeast Conference*.
[43] Jingao Xu, Hao Cao, Zheng Yang, Longfei Shangguan, Jialin Zhang, Xiaowu He, and Yunhao Liu. 2022. {SwarmMap}: Scaling Up Real-time Collaborative Visual {SLAM} at the Edge. In *USENIX NSDI*.
[44] Wenxiao Zhang, Bo Han, and Pan Hui. 2017. On the networking challenges of mobile augmented reality. In *ACM SIGCOMM Workshop on Virtual Reality and Augmented Reality Network*.
[45] Wenxiao Zhang, Bo Han, and Pan Hui. 2018. Jaguar: Low latency mobile augmented reality with flexible tracking. In *ACM Multimedia*.
[46] Wenxiao Zhang, Bo Han, and Pan Hui. 2022. SEAR: Scaling Experiences in Multi-user Augmented Reality. *IEEE Transactions on Visualization & Computer Graphics* (2022).
[47] Danping Zou and Ping Tan. 2012. Coslam: Collaborative visual slam in dynamic environments. *IEEE transactions on pattern analysis and machine intelligence* 35, 2 (2012), 354–366.


# APPENDIX

## A CLIENT TRACKING ALGORITHM

**Algorithm 1:** Pose Computation with IMU Model

```
Function ApproxPose_UpdateMM(C_IMU,i):
    PF_MM := Poses[i-1]    // prev. frame motion model
    CRot:= PF_MM.Rot×C_IMU.RotΔ
    CPos:= IMUPosition(PF_MM.Pos,C_IMU.PosΔ)
    CVel:= IMUVelocity(PF_MM.Vel,C_IMU.VelΔ)
    Velocity:= PoseVelocity(CRot, CPos, CVel)
    CurrentPose:= LastFramePose×Velocity
    Poses[i]:= CurrentPose
return CurrentPose
Function Recv_SLAMPose(SLAMPose, SLAMIndex):
    PastPoses[SLAMIndex]:= SLAMPose
    /* Update Motion Model */
    for j ←SLAMIndex to len(Poses) do
        ApproxPose_UpdateMM(Poses[j+1],j+1)
    end
```

The IMU-based pose estimation module used in SLAM-Share (§4.2.2) is in Algorithm 1. The client uses the `ApproxPose _UpdateMM` function to compute each frame's pose. It uses the previous frame's IMU rotation, position and velocity values (lines 3-5) and updates their values based on the difference between the previous and the current frames. The updated IMU-based values can then be used to derive a "velocity" (line 6) that then approximates the current pose relative to the previous frame's pose (line 7), and is stored. This client computation proceeds in parallel with the server's pose computation. When a SLAM-based pose is obtained from the server, `Recv_SLAMPose` uses the more accurate server-computed pose to update the IMU-based motion model for subsequent frames, based on the received pose value (lines 12-13).

## B MAP MERGING ALGORITHM

**Algorithm 2:** Map Merging

```
1  Function MapMerge(CMap):
2      foreach Mappoint ∈ CMap, Keyframe ∈ CMap do
3          GMap.AddMapPoint(Mappoint)
4          GMap.AddKeyFrame(Keyframe)
5      end
       /* Loop through every client's Keyframe */
6      foreach KF ∈ CMap do
7          LW:= DetectCommonRegion(KF, GMap)
8          if LW> 0 then
9              T:= 3DAlign(KF, LW)
10             foreach Mappoint visible by KF do
11                 Mappoint.pose × T
12             end
13             if mbLoopDetected then
14                 BundleAdjustment(KFs ∈ CMap &
                    LocalKFs)
15                 EssentialGraphOptimization()
16     end
17 return GMap            // Merged Global Map is returned
```

We present SLAM-Share's map merging method (§4.3.1) in greater detail in Alg. 2. The steps follow the single-user map merge procedure present in ORB-SLAM3, with the differences in SLAM-Share highlighted below. SLAM-Share initiates map merging as soon as a client map (`CMap`) is created and placed in the server's shared memory (§4.3.2). As described in §4.3.1, this initiation process differs from single-user map merging, which only initiates map merging based on incoming keyframes, not based on an existing map. As the first step, SLAM-Share adds the Mappoints and Keyframes from a client's map into the global map (`GMap`) data structure (lines 2-5). SLAM-Share re-numbers the Mappoint and Keyframe indices from `CMap` and `GMap` so that there are no collisions between clients. As both maps exist in the same server shared memory (unique to SLAM-Share), this only adds pointers to the global map database, without any data copying. SLAM-Share then iterates through each Keyframe (denoted "KF" in Alg. 2) from the client's map and runs the `DetectCommonRegion` function to detect common regions

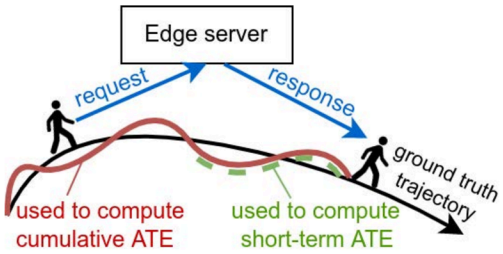



**Figure 14: Illustration of cumulative ATE vs. short-term ATE.**

seen in that KF, and in the global map (lines 6-7). This iterative process is unique to SLAM-Share, as the default ORB-SLAM3 only checks the current active Keyframe, not all past keyframes in the `CMap`. `DetectCommonRegion` uses a Bag of Words (BoW) to search through all the global map's KFs to find the closest ones, which it returns as a list called LW (line 7). If LW is not empty, *i.e.,* if the algorithm can find a common region between client map and global map, the program proceeds ahead on aligning the maps, by finding the 3D alignment $T$ between KF and LW and applying this alignment to each Mappoint visible in the client's KF (lines 8-12 in Alg. 2). This provides an initial merge. Once the initial merge of a single Keyframe from the client map to the global map is completed, we check if there is a loop been detected. (line 13) If a loop has been detected, then run bundle adjustment and essential graph optimization to reduce the error and correct the map (line 14-15).

## C CUMULATIVE VS. SHORT-TERM ATE

In Fig. 14, we illustrate the differences between the cumulative ATE (a common SLAM evaluation metric [16]), and the short-term ATE (which reflects the user's experience in the last 5 seconds). The cumulative ATE is based on the distance between the estimated trajectory (red line) and ground truth (black line), for the entire length of the client's trajectory until the present time. The short-term ATE is based on the distance between the last 5 seconds of the estimated trajectory (green dashed line) and the corresponding 5 seconds of ground truth (portion of black line). We use the short-term ATE metric for evaluation because it (a) captures the client's most recent experience, and (b) since SLAM continuously updates all parts of the estimated trajectory, we need to take a snapshot of the ATE for each portion of trajectory as it is walked, to reflect the client's real-time position estimate and hologram positioning.

Fig. 14 also helps illustrate why SLAM-Share's lower data transfers and hence faster server response time improve (reduce) the short-term ATE. The time in between when a client requests help from the server and receives a response from the edge server (*i.e.,* the portion of the trajectory between the two human figures in Fig. 14) is generally the most inaccurate part of the trajectory, since help from the server is pending. If the request/response is slow (as in the baseline), then the inaccurate portion will be longer, and the last 5 seconds of the trajectory (green dashed line) will lie fully within the inaccurate portion, resulting in a higher short-term ATE. If the request/response is fast (as in SLAM-Share), then the inaccurate portion will be shorter, and the last 5 seconds of the trajectory will only overlap with the inaccurate portion for a short period, resulting in a lower short-term ATE.