DeepVista: 16K Panoramic Cinema on Your Mobile Device

Wenxiao Zhang[*]
HKUST
wzhangal@cse.ust.hk

Feng Qian
University of Minnesota
fengqian@umn.edu

Bo Han
George Mason University
bohan@gmu.edu

Pan Hui
HKUST
panhui@cse.ust.hk

ABSTRACT

In this paper, we design, implement, and evaluate DeepVista, which is to our knowledge the first consumer-class system that streams panoramic videos far beyond the ultra high-definition resolution (up to 16K) to mobile devices, offering truly immersive experiences. Such an immense resolution makes streaming video-on-demand (VoD) content extremely resource-demanding. To tackle this challenge, DeepVista introduces a novel framework that leverages an edge server to perform *efficient, intelligent, and quality-guaranteed* content transcoding, by extracting from panoramic frames the viewport stream that will be delivered to the client. To support real-time transcoding of 16K content, DeepVista employs several key mechanisms such as dual-GPU acceleration, *lossless* viewport extraction, deep viewport prediction, and a two-layer streaming design. Our extensive evaluations using real users' viewport movement data indicate that DeepVista outperforms existing solutions, and can smoothly stream 16K panoramic videos to mobile devices over diverse wireless networks including WiFi, LTE, and mmWave 5G.

CCS CONCEPTS

• Information systems → Multimedia streaming; **• Computing methodologies → Virtual reality**; **• Human-centered computing → Mobile computing**.

ACM Reference Format:
Wenxiao Zhang, Feng Qian, Bo Han, and Pan Hui. 2020. DeepVista: 16K Panoramic Cinema on Your Mobile Device. In *Proceedings of the Web Conference 2021 (WWW '21), April 19–23, 2021, Ljubljana, Slovenia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3442381.3449829

1 INTRODUCTION

360° panoramic videos have recently registered high popularity on commercial platforms such as YouTube and Facebook [24, 33]. Despite its attractiveness, today's 360° video streaming faces a dilemma. On one hand, when using a headset (*e.g.,* a $10 Google Cardboard [8] with a smartphone plugged in), a viewer would expect a 360° video with a resolution much higher than that of regular videos due to the immersive requirement and the close distance from the eyes to the display. On the other hand, under the same perceived resolution, 360° videos require more bandwidth and hardware decoding resources than regular videos. The reason is that given the limited field of view (FoV), a viewer consumes only a small fraction (15% to 20% pixel-wise) of the panoramic frame that is usually delivered and decoded.

In this study, we set an ambitious goal of *streaming pristine-resolution 360° panoramic videos that are far beyond the ultra high-definition (UHD) quality on commodity mobile devices* such as smartphones and untethered virtual reality (VR) headsets. To this end, we propose DeepVista, a system offering video-on-demand (VoD) content that has up to 16K resolution for matching the perception limit of human eyes (§2). Having a panoramic resolution of 16K×8K pixels, 16K videos offer 16× and 64× pixel density compared to 4K and 1080p ones, respectively, which provide flawless, cinematic pictures and truly immersive experience. Although such a high resolution may not be necessary when watching regular videos, it can facilitate a truly immersive experience in a head-mounted VR headset setting *where the display is very close to the viewer's eyes.*

Achieving the above goal is challenging. A 16K video at 30 frames per second (FPS) has a raw bitrate of ~48 Gbps; even after encoding, its bitrate could be around 300 Mbps [26]. Moreover, we would like to stream 16K 360° videos to mobile devices with limited computation power and video decoding capability. As we will show in §3, none of today's mobile devices is capable of smoothly decoding 16K panoramic content.

Given the inadequacy of the client-only approach, we resort to edge computing by exploring the following idea. An edge server performs real-time transcoding by extracting the viewport and delivering only the viewport content to the client. As a typical viewport takes a small portion (around 1/4) of the panoramic view, we can drastically reduce bandwidth usage and *decoding workload* on the client. For 16K panoramic video streaming, after such transcoding, the client needs to decode only an 8K stream – feasible on today's COTS smartphones. Note that the industry has already developed 8K screens for mobile devices [13] and the physical dimension of smartphone/tablet screens has been increasing [4]. The edge proxy is a GPU-equipped machine. It can be placed at home or office, or be provided by an edge/cloud provider. For instance, Google offers high-end GPUs at as low as $0.2 per hour per GPU [9].

At a first glance, the idea of edge-assisted transcoding appears to be straightforward: the edge decodes a 16K stream, crops the predicted viewport (8K), encodes it, and then sends the viewport content to the client. In the literature, some existing system such as Freedom [50] does take this intuitive approach. However, we find that the visual quality of its resulting transcoded content is unsatisfactory, not to mention its lack of built-in support for 16K video content (§5.2.1). To this end, we make a key contribution by *designing and implementing an efficient, intelligent, and quality-guaranteed transcoding framework capable of handling extremely high-resolution VR content, and integrating this framework into* DeepVista*, a full-fledged edge-assisted 360° video streaming system.* We next describe the key challenges faced by DeepVista with our proposed solutions.

[*]Part of this work was done when Wenxiao was visiting University of Minnesota.

ACM ISBN 978-1-4503-8312-7/21/04.
https://doi.org/10.1145/3442381.3449829

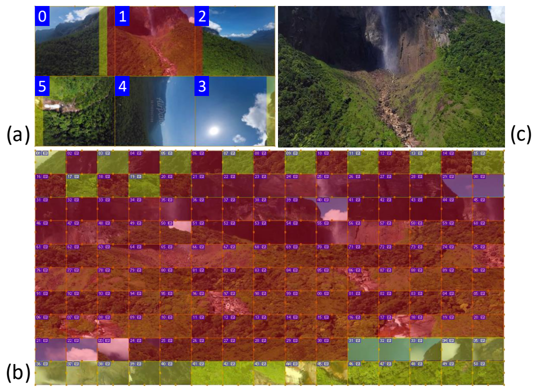

Figure 1: The basic idea of DeepVista**. (a) A 16K panoramic frame represented using CubeMap [7]. The red area corresponds to the viewport. (b) The transcoded 8K frame containing the viewport content. (c) The rendered view on client device.**

• **Efficient and Quality-guaranteed Viewport Transcoding.** 360° videos (and rendered VR content in general) are stored using various projection algorithms under which a viewport typically has an irregular shape (Figure 2). We develop a method that can precisely identify the viewport and fit it into a rectangular frame (§5.2). As illustrated in Figure 1, a projected panoramic frame is segmented into fixed-sized blocks; all blocks overlapping with the viewport are then losslessly rearranged into a rectangular frame in a concurrent manner, by leveraging GPU's massive parallelism. In contrast, existing transcoding solutions such as Freedom [50] cannot guarantee that the cropped viewport content fits the target rectangular frame. Hence, the resolution of viewport content usually needs to be lowered to various degrees, leading to quality degradation.
• **Efficient Processing of 16K Content.** We find that besides mobile devices, even today's state-of-the-art desktop and workstation GPUs cannot directly decode 16K videos using their dedicated decoding hardware [19]. To tackle this challenge, we design a first-of-its-kind pipeline that allows two GPUs to collaboratively perform transcoding. Each GPU first decodes a half-16K (8K×8K) stream separately; then according to the viewport information, both GPUs efficiently exchange the decoded data so that each possesses half of the viewport (4K×4K); the viewport is then encoded half-and-half by the two GPUs; the client will subsequently assemble the two encoded streams into the final 8K viewport (8K×4K). We carefully optimize the above pipeline by minimizing the dependency between two GPUs, guaranteeing that the transcoding meets the stringent timing requirement of 30+ FPS.
• **Intelligently Hiding the Transcoding Latency.** Due to the large size of 16K videos, their transcoding and wireless delivery may incur high latency that needs to be effectively hidden to ensure a smooth user experience. DeepVista takes two approaches to address this challenge. First, the edge employs deep learning (DL) to predict a viewer's future viewport and uses it (as opposed to the current viewport [50]) for transcoding. While some prior studies have applied DL to head movement prediction of VR users [30, 37, 55], we leverage DL-based viewport prediction to improve the transcoding performance. Specifically, we integrate the prediction module into the transcoding engine; we strategically reduce the resource footprint for online inference; and we carefully choose key parameters of the prediction model (§5.4). Second, to tolerate viewport prediction errors, the edge also delivers a low-resolution panoramic stream, which is properly synchronized with the high-resolution viewport stream for minimizing the stall probability (§5.3).

	Visual Quality of Transcoded Content	Supporting 16K Content	Viewport Prediction Guided Transcoding
Freedom	Low	No	No
DeepVista	High	Yes	Yes

Table 1: DeepVista **vs. Freedom [50] (details in §5.2.1).**

• **System-level Integration and Optimizations.** We integrate all the above components into a holistic edge-assisted 360° video streaming system. We introduce a series of system-level optimizations to boost the performance and enhance the quality of experience (QoE), such as deep pipelining, judicious buffer management, and fast client-side processing (§5.6). DeepVista employs a robust bitrate adaptation algorithm that adjusts the transcoded stream quality dynamically based on the network condition (§5.5).

We implement DeepVista on commodity GPU-equipped machines and smartphones in 9,500 LoC (§6). We extensively evaluate DeepVista using real 360° videos, real users' viewport movement data, and live WiFi, LTE, and mmWave 5G networks [25]. The key evaluation results consist of the following (§7).
• Over 802.11n WiFi, DeepVista typically experiences no 16K frame skip, and the 16K high-resolution content covers 98% of the viewport on average. The perceptual quality is close to visually lossless (SSIM, a widely used visual quality metric [52], is higher than 0.98).
• Under live LTE networks with fluctuating bandwidth and latency, DeepVista yields 2.6% of 16K frame skips on average (1.56 s/minute), and an average high-resolution ratio of 96%. Under mmWave 5G networks, DeepVista can achieve almost perfect quality with nearly no 16K frame skip. We spent more than 400GB of LTE/5G data on experiments over commercial cellular networks.
• DeepVista reduces the last-mile bandwidth consumption by 52% to 69%, compared to fetching the panoramic scene.
• When streaming 8K 360° videos, DeepVista outperforms the state-of-the-art solution (Flare [47]) by reducing the median stall from 22–34 seconds per minute to no stall, and reducing the median data transfer size by a factor of 3.0× to 3.6×. DeepVista offers significantly higher visual quality compared to Freedom, a recently proposed edge transcoding solution [50].

2 BACKGROUND AND RELATED WORK

Playback and Representation. When displaying a 360° video, the player situates the viewer in the center of an imaginary sphere, and renders the content onto the inner surface of the spherical "screen". At a given time, the viewer can perceive only a small portion of the panoramic content, as determined by the viewing direction (latitude/longitude) and the FoV (*e.g.,* 100°×90°). Various *projection* methods such as Equirectangular projection [47], CubeMap [7], and Pyramid [15] have been leveraged to transform the *panoramic* content (used for storage and delivery), to the *displayed* content. For example, in CubeMap, the transformation is established by placing the sphere into a bounding cube, and performing 90° perspective projections of the sphere onto its six faces.

Content Streaming. Most of today's content providers such as YouTube and Facebook deliver 360° videos using a monolithic approach that always streams the entire panoramic scene [24]. This incurs a tremendous waste of traffic since the viewer consumes only a small fraction (around 15%–20% pixel-wise) of a panoramic frame. To overcome this limitation, *viewport-adaptive* streaming schemes have been developed for 360° videos, which fetch mainly the content overlapping with the viewport. From the *system* perspective, there exist only a few full-fledged viewport-adaptive 360° video streaming systems, such as PARSEC [29] (2020), Pano [33] (2019), Freedom [50] (2019), Flare [47] (2018), and Rubiks [35] (2018). Rubiks and Flare employ a tile-based approach where the panoramic content is spatially divided into *tiles*, which are selectively fetched. Pano optimizes the QoE for 360° video streaming by considering users' attention. PARSEC employs super-resolution to reduce bandwidth consumption. None of the above work leverages edge support except Freedom, which is the most relevant work to ours. We briefly compare DeepVista and Freedom in Table 1 and describe the details in §5.2.1. There exist numerous other proposals of viewport-adaptive 360° content delivery, albeit many backed up by analytical, simulation, or emulation results [27, 32, 46, 51, 53, 54, 58–60]. None of the above work handles 16K 360° videos.

VR and HD Video Streaming. The big context of DeepVista is VR and HD multimedia content streaming, on which a plethora of work has been done [39–41, 44]. Some recent efforts utilize 60 GHz millimeter wave links to deliver multimedia data [23, 42]. Despite its high bandwidth, mmWave is vulnerable to signal blockage and attenuation. More importantly, the mobile client may not be able to timely decode the high-resolution content [42]. The only proposal of 16K VR streaming that we are aware of was a preliminary study by Alface *et al.* [26]. However, in their design, the content displayed on mobile devices has only a 4K resolution.

3 MOTIVATION

Why 16K Resolution for 360° Videos? It is well known that VR requires a high resolution and low latency [40]. A human with 20/20 vision can perceive up to 60 pixels per degree [6, 12], or 3,600 pixels within an area of 1°×1°. Today's VR headsets (*e.g.,* Google cardboard [8] and Samsung Gear VR) typically cover an FoV of about 100° horizontally and 90° vertically [47]. To achieve the resolution of 3,600 pixels/deg^2, 32M (3600×100×90) pixels are expected to be in the FoV. However, an 8K video has only 8K×4K=32M pixels for the panoramic view, falling far short from the pristine resolution. Instead, 16K videos offer 4× pixel density compared to 8K, making the resolution within the FoV close to the pristine resolution.

Can COTS Mobile Devices Directly Decode 16K Videos? We test on mainstream smartphones, including Samsung Galaxy S8 (SGS8), SGS9, SGS10, Pixel 3, and popular laptops. We find that none is capable of decoding 16K H.264 or H.265 videos with the on-device hardware decoders. In fact, even state-of-the-art desktop and workstation GPUs (*e.g.,* NVIDIA GeForce/Quadro/Tesla series) cannot decode 16K videos [19]. Note that in this paper, the decoding capability of a GPU refers to decoding a video using the GPU's dedicated hardware decoder, as software-based video decoding is usually much slower than hardware-based solutions.

Is "Split-and-merge" Feasible? We consider an alternative "split-and-merge" approach: the server pre-segments a 16K video into tiles; the client fetches and decodes them in parallel, and then merges the decoded tiles into the original stream. One 16K×8K frame can be segmented into, for example, two 8K×8K tiles, four 8K×4K tiles, or sixteen 4K×2K tiles. We conduct experiments on an SGS8 with Exynos 8895 SoC. Table 2 benchmarks the performance of decoding an H.265 video with different resolutions. Each row corresponds to a target panoramic frame resolution, whereas each column represents parallel decoding using k threads, each invoking the Android MediaCodec API [2] to decode a tile whose size is $1/k$ of the panoramic frame. Each cell in Table 2 shows the decoding performance in FPS, with "N/A" denoting this option is not feasible. For example, the first row indicates that 16K video decoding is not possible; the second row shows that there are two ways to decode an 8K×8K video: using two threads each decoding an 8K×4K stream, or using four threads each decoding a 4K×4K stream. They yield an overall FPS of only 23.6 and 21.6, respectively (the merging overhead is not even considered). We make similar observations on other devices.

# Threads	k=1	k=2	k=4	k=8
16K×8K	N/A	N/A	N/A	N/A
8K×8K	N/A	23.6 ± 0.8	21.6 ± 0.6	N/A
8K×4K	45.9 ± 0.3	45.6 ± 1.2	24.9 ± 0.9	18.7 ± 0.4
4K×4K	58.1 ± 0.5	51.1 ± 1.9	38.8 ± 1.1	28.7 ± 0.6

Table 2: H.265 decoding performance on Samsung Galaxy S8. Reported FPS values are averaged over 10 runs of decoding the same video (Angel Falls [3]) with different resolutions.

How Much can Tile Based Viewport Adaptation Help? The "split-and-merge" approach can be improved by making it viewport-adaptive: the client fetches and decodes only a subset of the tiles based on the user's viewport [35, 47]. Unfortunately, this approach still cannot support 16K 360° video streaming on today's mobile devices. To stream the viewport-only portion of a 16K×8K stream, the player needs to fetch and decode up to 25% (*i.e.,* 8K×4K worth) of the panoramic content (§5.2). To realize this, the highlighted cells in Table 2 indicate two available options for achieving at least 30 FPS. However, in a tile-based scheme, the viewport boundary typically does not align with the tile boundary. Instead, as shown in Figure 2, the viewport may intersect with up to four tiles. Since a tile is the atomic decodable unit, the actual decoding workload is amplified by 2× to 4× compared to those in the highlighted cells in Table 2, making it impossible to achieve 30 FPS.

4 DEEPVISTA OVERVIEW

DeepVista strategically leverages an edge server to transcode in real-time a panoramic 16K video stream into an 8K stream that covers the user's viewport. The 8K stream can be efficiently decoded and rendered on mobile devices (8K screens for mobile devices are already available [13]). This leads to much less decoding overhead and bandwidth consumption than tile-based approaches, because the edge can precisely extract the viewport from the panoramic scene into a single video stream. In addition, the edge has more computation resources for running sophisticated algorithms such as deep viewport prediction (§5.4) to boost system performance.

DeepVista consists of a video content server, an edge proxy, and a client. Compatible with the DASH (Dynamic Adaptive Streaming over HTTP) standard, the server is simply a stateless HTTP(S) server. The key logic such as 16K to 8K transcoding, viewport prediction, and rate adaptation is performed on the edge. The thin client

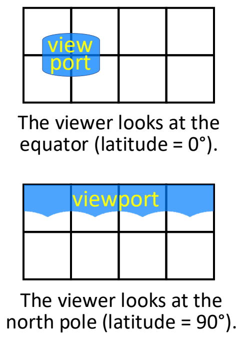
view port
The viewer looks at the equator (latitude = 0°).
viewport
The viewer looks at the north pole (latitude = 90°).

Figure 2: viewports overlap with 4 out of 8 tiles (2×4 segmentation, equirectangular projection).

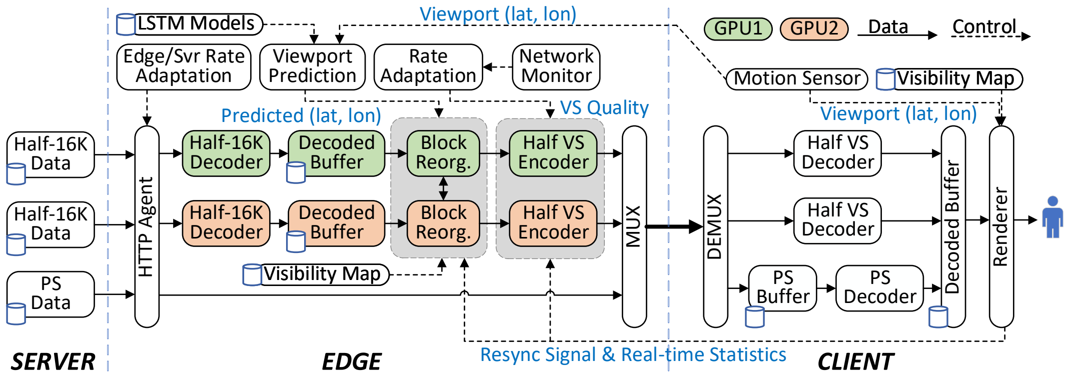

Figure 3: The DeepVista **system architecture (16K Resolution, Dual GPU mode).**

decodes/renders the 8K (viewport) stream and reports necessary data such as the viewport movement trajectory to the edge.

Designing and implementing DeepVista faces several major challenges. First, simultaneously decoding 16K frames and encoding 8K frames may not be an easy task even for the edge. For example, an NVIDIA GTX 1080Ti GPU can achieve 16K decoding performance at only around 16 FPS (a 16K frame is split into two 8K×8K frames for decoding). Second, we need an efficient transcoding scheme that can fill an 8K frame with the user's viewport content at 16K resolution without quality degradation. Third, we need robust methods to hide the transcoding and network delay, which may cause the transcoded/delivered viewport to deviate from the viewport that should be perceived. Last but not least, the system involves many components; it is far from trivial to implement various system-level optimizations and to integrate them into a holistic system.

Scaling to Multiple Users. Video transcoding is inherently resource-demanding. For large-scale deployment of DeepVista, we can provision resources at the Internet edge in a way similar to today's cloud gaming services such as Google Stadia [10], where each player has dedicated GPU resources in low-latency clouds. Such infrastructures already exist today (*e.g.,* NVIDIA EGX [17]).

Future Trend. Specialized hardware supporting 16K-video decoding may appear in the future. Complementing that, DeepVista enables *commodity* hardware to stream 16K panoramic videos. It can also drastically reduce the last-mile bandwidth consumption. We believe that in the foreseeable future, it is less likely that mobile GPUs will be capable of decoding 16K videos due to the inherent constraints of form factors, energy footprint, and heat dissipation. DeepVista's concept is therefore still valuable in the long term.

5 SYSTEM DESIGN

We now detail the design of DeepVista as shown in Figure 3.

5.1 Server-side Content Preparation

The DeepVista server is compatible with DASH and is thus a regular HTTP(S) server: the 16K content is split into chunks and encoded into different quality levels using standard encoders such as H.265. There are two non-trivial design decisions for the DeepVista server. First, for each quality level, instead of storing a single 16K×8K video, the server stores two 8K×8K streams of the original content. This facilitates distributing the workload to two GPU instances.

The other design decision is to let the server prepare a low-resolution version of the panoramic video, called the *Panoramic Stream (PS).* In our current design, we use 2K×1K resolution for the PS. Note that the edge transcodes only the 16K video. The transcoded stream is referred to as the *Viewport Stream (VS).* The PS is directly forwarded by the edge to the client without transcoding. The purpose of having the PS is twofold. First, it helps tolerate viewport prediction errors. Since the PS contains the full panoramic scene, the client can use it to cover any missing portion in the viewport that is not delivered by the VS due to inaccurate viewport prediction. We find that a missing portion, if any, typically occurs at the border of a viewport, so patching it using the PS typically incurs small QoE degradation. Second, since the delivery of the PS does not depend on the viewport, the client can maintain a large buffer for it, thus reducing the stall (*i.e.,* rebuffering) probability: if the VS is not delivered in time, at least the PS can be played. In contrast, for the VS, the client buffer has to be shallow because predicting the viewport in the long term is difficult.

5.2 16K-to-8K Viewport-aware Transcoding

We now focus on the edge design. Its core component is transcoding from the 16K stream (the two 8K×8K streams fetched from the server) to the 8K×4K viewport stream (VS). The high-level approach is intuitive: extracting from the 16K panoramic frame a region that (1) fully covers the predicted viewport (*i.e.,* the blue regions in Figure 2), and (2) can be fit into an 8K frame.

We face two challenges when realizing the above approach. First, we need to choose an appropriate projection method for the VS. For simple projection schemes such as Equirectangular, the viewport area of the panoramic frame, which we call *Projected Viewport Area (PVA),* may vary significantly depending on the viewer's orientation. This is illustrated in Figure 2: when the viewer looks at the equator (the top plot), the PVA is small; however, as he/she looks towards the north pole (the bottom plot), the PVA expands to more than 1/4 of the panoramic frame, making 16K-to-8K transcoding impossible. In contrast, the PVA of CubeMap (§2) has a much smaller variation (and thus lower distortion), and is always less than 1/4 of the panoramic frame under a typical FoV regardless of the viewer's orientation. DeepVista thus adopts CubeMap as the VS representation. It is possible to extend DeepVista to support other projection schemes (*e.g.,* Pyramid projection [15]).

The second challenge is to efficiently and losslessly "reorganize" the viewport content into an 8K×4K *rectangular* frame. This is necessary because the viewport typically has an irregular shape, and its bounding box oftentimes exceeds 8K×4K. To begin with, we divide each face of a CubeMap into small *blocks*. A block is somewhat similar to a tile depicted in Figure 2; however, the difference is that a tile is an independently decodable *video stream*, whereas blocks are merely "atomic" regions of pixels whose positions can be rearranged within a single *video frame*. Therefore, blocks can be made much more fine-grained than tiles, leading to less bandwidth waste and decoding overhead. Based on its position in the CubeMap, each block is assigned a unique ID.

There is a tradeoff between the reorganization overhead and the bandwidth saving. Having more blocks makes the partition more fine-grained, thus reducing the number of invisible pixels that belong to the blocks around the border of the viewport. However, this comes at the cost of a higher reorganization overhead. In DeepVista, we divide the whole CubeMap into 600 blocks (10×10 blocks on each face). We select 600 because its block reorganization time is short (<3ms); meanwhile, all blocks overlapping with the viewport can always be reorganized into an 8K frame. By enumerating all possible viewports, we find that under the $100^\circ \times 90^\circ$ FoV[1], at most 142 blocks are visible in the viewport, fewer than the capacity of 150 blocks (=600/4) that the 8K VS can carry. The above method ensures that the predicted viewport content (in 16K resolution) can losslessly fit an 8K frame, thus guaranteeing the transcoding quality. It is a general approach that can work with any projection scheme.

To identify the blocks to be included in the VS, we compute *offline* a visibility map, which contains mappings from a viewport (latitude, longitude) to the set of blocks overlapping with the corresponding viewport. In addition, since the number of visible blocks is less than 150, we complement the set by including extra blocks on the periphery of the visible blocks, to make the set contain exactly 150 blocks. This helps further tolerate viewport prediction errors. The visibility map has 181×361 entries that enumerate all possible viewports (lat $\in$ [-90°,90°], lon $\in$ [-180°,180°]) at the granularity of 1°, with each entry containing 150 block IDs.

We next detail the edge-side transcoding process. We use off-the-shelf GPUs that provide two key functionalities: (1) hardware-based video decoding and encoding, and (2) streaming multiprocessors (SMP) that can process the blocks in mass parallelism.

- The edge decodes the original 16K content using the NVIDIA Video Codec API. Recall from §5.1 that the server stores the 16K content in two 8K×8K streams. We find that using a single GPU (*e.g.,* GeForce GTX 1080 Ti) to decode the two streams yields a frame rate of only 16 FPS. Therefore, DeepVista supports using *two GPU instances* to decode both streams in parallel in order to keep up with the required frame rate. For performance consideration, DeepVista does not explicitly merge the two decoders' output. Instead, the decoded half-16K frames are kept in each GPU's memory.
- For each decoded 16K frame, the edge uses the predicted viewport as the key to look up the visibility map, and obtains the list of blocks. Figure 1(a) shows the six faces of a 16K CubeMap. It also exemplifies a viewport at (lat=0°, lon=0°). Pixels belonging to the 120 blocks within the viewport are marked in red, and those belonging to the 30 extra blocks around the viewport are marked in yellow. The edge then copies the 150 blocks to the 8K frame, placed from top-left to bottom-right according to their IDs. The blocks are arranged in 10 rows and 15 columns as shown in Figure 1(b). The copying of the blocks is parallelized over a large number of GPU (CUDA) threads, with each responsible for handling k pixels of a block (we empirically choose k=4 for the YUV420 format). The client also maintains the same visibility map so that the reorganized blocks can be restored (§5.6). If dual GPUs are used, each GPU is responsible for producing half of the viewport (75 blocks in a 4K × 4K frame). Cross-GPU block exchange, *i.e.,* copying blocks from one GPU to the other, is only performed here, in an on-demand manner. This helps minimize the coupling between the GPUs. The cross-GPU transfers go through the PCI-E interface without the CPU involvement.
- The transcoded frames are encoded into the VS using the NVIDIA Video Codec API. Under a single GPU (dual GPUs), the VS consists of one 8K×4K (two 4K×4K) encoded stream(s), which are transmitted to the client.

5.2.1 Compare to State-of-the-art Transcoding Approach. To further illustrate the advantages of DeepVista's transcoding approach, we compare it with the mechanism used by Freedom [50], a recent system that also leverages the edge to perform viewport transcoding (a preliminary design was published by the same authors in [49]). Freedom is the most relevant work compared to DeepVista. It employs a representative transcoding approach: based on the current FoV F, the edge crops a region R, encodes it, and sends it to the client. To tolerate the client's viewport movement, R needs to be expanded from F by Δ. We denote this as $R = F + \Delta$. Despite its simplicity, Freedom has a major limitation. Since it does not rely on real-time viewport prediction, Δ has to be fairly large (*e.g.,* ±30° for yaw and ±15° for pitch). In addition, Δ is directly encoded with F at the *same* quality. This leads to quality reduction as the resolution of $F + \Delta$ needs to be shrunk to fit the target (transcoded) frame.

We conduct the following experiment to quantify the above limitation. We transcode an 8K video into a 4K stream based on 12 users' viewing trajectory (details in §7.1). We implement Freedom's approach for FoV $F = 100^\circ \times 90^\circ$ using three configurations of Δ: $17^\circ \times 15^\circ$, $28^\circ \times 25^\circ$, and $39^\circ \times 35^\circ$ [50]. When the FoV becomes $F + \Delta$, Freedom has to sacrifice the quality by shrinking the cropped frame, which is larger than the normal FoV, to 4K. Using the viewport content from the original video as the ground truth, the measured SSIM is 0.953±0.002, 0.941±0.002, and 0.929±0.003, respectively, for the above three configurations of Δ. Note that the original Freedom system performs 8K to 1080p (instead of 4K) transcoding, leading to an even higher quality degradation. Also note that Freedom does not have built-in support of 16K video content.

Compared to Freedom, DeepVista takes a different transcoding approach (Table 1), which guarantees the cropped viewport content can losslessly fit the target frame. Under the same setup, the SSIM of DeepVista (considering both VS and PS, with viewport prediction) is measured to be 0.995, significantly better than Freedom.

5.3 VS/PS Coordination & Buffer Management

VS Resync. In DeepVista, the pace of video playback is determined by the PS due to its reliability: the PS can enjoy a deep on-client

[1]We focus on a typical FoV of 100°×90°. Larger FoVs may need slightly more blocks, and can be supported by slightly increasing the VS resolution.

buffer since it depends on neither the viewport nor transcoding. Normally, the VS playback is synchronized with the PS as ensured by the edge. However, due to network anomalies, the VS delivery may fall behind the PS, causing missing VS frames.

Once the client detects a missing VS frame F_m, it sends a *resync* message to the edge, which will immediately stop working on the current VS frame and start processing a new frame with an ID of $F_m + \lceil r \cdot T_{pipe} \rceil$, where r is the FPS of the VS, and T_{pipe} is the estimated latency of the edge-to-client pipeline for processing the new frame, including the delay of transcoding, network transfer, client decoding, and client-side buffering. DeepVista continuously profiles the above components in order to accurately estimate T_{pipe}. Intuitively, $\lceil r \cdot T_{pipe} \rceil$ represents the smallest number of VS frames that have to be skipped due to the client/edge processing time and network delay, so that the VS can catch up with the PS again. During the brief resync phase, the user sees only the PS, but the video still plays smoothly without any stall (unless the PS is not delivered in time).

Edge-side Buffer Management. For each GPU, the edge maintains a frame buffer storing several decoded half-16K frames, in order to prevent the encoder from starvation. The decoded frame buffer also facilitates the above resync process by having the new frame ready (*i.e.,* already decoded) after skipping several frames. One decoded half-16K frame in YUV420 format requires ~96 MB of memory. We buffer 60 frames, which occupy about 6.7GB memory on each GPU. An important design decision we make is that the edge does not explicitly maintain any buffer for transcoded VS frames. This is because such frames contain predicted viewports (§5.4), and need to be delivered to the client as soon as possible. Buffering them will inflate the viewport prediction window ($(F_p - F_c)/r$, see §5.4), making the viewport prediction less accurate.

5.4 Viewport Prediction

In DeepVista, viewport prediction is executed on the edge server. We develop two concrete prediction methods. The first one is a simple linear regression [47] used as the baseline for comparison (§7.8). The second method is based on deep learning (DL). We build an offline DL model of multiple prior viewers' viewport trajectories of the same video, and apply this model to predict a new user's viewport at runtime. We adopt Long Short-Term Memory (LSTM) [36] due to its good accuracy for time series data prediction. We train two LSTM models for each video, one for latitude predication and the other for longitude prediction, using many users' viewport movement trajectories collected when they watch the same video (we use up to 30 users in our evaluation in §7.8). We train the network to minimize the mean absolute error (MSE), which is used as the loss function.

While some prior studies have applied DL to head movement traces of VR users [30, 37, 55], our contribution here is to *leverage DL-based viewport prediction to improve the transcoding performance.* We address several system-related design issues as detailed below.

• First, we need to properly integrate the viewport prediction module with the transcoding engine. Specifically, a prediction is performed when a VS frame is about to be generated (after the original 16K content is decoded, as this step is viewport-independent). The client continuously uploads the user's real-time viewport trajectory to the edge, with the sampling rate of 30 Hz, consuming only 3 Kbps of uplink bandwidth. At each prediction instance, the edge uses this trajectory data to estimate the viewport at time t_p, the time when the to-be-transcoded VS frame (with an ID of) F_p will be displayed to the viewer. Let t_c be the current time at the client, F_c be the ID of the current frame being played at the client, r be the (fixed) FPS of the VS. t_c and F_c are piggybacked with the trajectory data uploaded to the edge. Since the frames are sequentially played, t_p can be estimated as $t_c + (F_p - F_c)/r$.

• Second, our LSTM models should have a low resource footprint at runtime. We thus employ a lightweight single-layer LSTM model with 64 neurons. The LSTM layer is followed by a dropout layer and a dense layer. The resulting size of each model is less than 0.5 MB, making it portable. It takes ~20ms for a single inference over CPU. If using GPU, the inference time can be reduced to less than 3ms (see §7.1 for detailed experimental setup).

• Third, we need to carefully choose several key parameters, in particular the size of the prediction window. We test using the 2, 5, 18, and 30 most recent samples for viewport prediction. Among them, using the 2 most recent samples yields the best prediction results. A possible explanation is that compared to a long window, a short one such as 2 most recent samples can better adapt to users' sudden movement.

5.5 Rate Adaptation for Real Time Transcoding

Rate adaptation dynamically adjusts the video quality based on the network condition. DeepVista consists of two rate adaptation modules: one controls the quality of the content fetched from the server, and the other determines the quality of the VS content transcoded by the edge. For the former, any traditional DASH-friendly rate adaptation scheme [38, 43, 57] can be used. We thus focus on the latter. By again following the DASH standard, the edge encodes the VS into one of M qualities and we empirically choose M to be 5 [32, 35, 47]. The bitrates of the quality levels are 48Mbps, 32Mbps, 20Mbps, 12Mbps, and 8Mbps. The highest bitrate for the VS (48 Mbps) is selected based on the highest bitrate of the three YouTube 8K videos we used for experiments. The four other bitrates are selected by following Netflix's recommendation [20] where the encoded bitrate ratio between two consecutive quality levels is roughly 1:1.5. As the VS is delivered continuously without an explicit chunk boundary, we define a *virtual chunk* (V-chunk) as a group of 10 consecutive frames. A V-chunk is the smallest granularity of VS rate adaptation. Right before encoding each V-chunk, the edge executes the rate adaptation logic and configures the encoders' output bitrate accordingly.

DeepVista's rate adaptation is based on discrete optimization. The concept was used in prior streaming systems [47, 56, 57], but we tailor it to real-time transcoding. It maximizes the QoE over a finite horizon of the next N V-chunks. We empirically choose N=5 (setting it to 3 or 7 does not qualitatively change the results in §7). Let q_i be the quality level for the i-th V-chunk. The algorithm determines $q_1, ..., q_N$ by considering the following QoE metrics.

• *The Average Quality Level* over the finite horizon is defined as $Q = h \sum_{i=1}^{N} q_i / N$. The coefficient h represents the average high-resolution ratio (HRR) over the next N V-chunks. HRR is defined as the fraction of the area occupied by the VS (as opposed to the PS) in a viewport. DeepVista uses HRRs of previously played frames to estimate the future HRR. A non-perfect viewport prediction may lead to an HRR less than 1.

• *The Bandwidth Requirement* B is a 0/1 binary prediction indicating whether the network bandwidth is too low to sustain the bitrate of VS and PS, *i.e.,* $B = 1$ iff $\sum_{i=1}^{N} \beta_i/N + \beta_{ps} > \eta \cdot \beta_{pred}$ where β_i is the encoded bitrate of q_i, β_{ps} is the bitrate of the PS, β_{pred} is the predicted network bandwidth, and η is a parameter that tolerates the bandwidth prediction error and encoded bitrate variation. We empirically set η to 0.9.

• *The Temporal Quality Switch* quantifies the total number of quality level changes. It is defined as $TS = \frac{h \sum_{i=1}^{N-1} |q_{i+1} - q_i|}{N}$ where h is HRR that is defined above. A small TS is preferred as it offers smooth and gradual quality changes.

• *The Spatial Quality Switch* quantifies the QoE impact incurred by a mixture of the VS and PS content in a viewport. It is defined as $SS = (1/2 - |1/2 - h|) \sum_{i=1}^{N} q_i$. Given $\{q_i\}$, SS reaches its maximum when HRR is 1/2, *i.e.,* VS and PS each occupies half of the viewport.

Having the above QoE components defined, the overall QoE is calculated as their weighted sum, *i.e.,* $QoE = Q - w_s B - w_i(TS + SS)$ where w_s and w_i are the weights empirically chosen as $w_s = 5$ (giving the bandwidth requirement high importance) and $w_i = 1/2$ based on our tests using different combinations of (w_s, w_i). Since the search space is relatively small, DeepVista enumerates all combinations of $\{q_i\}$ and selects the one yielding the highest QoE.

5.6 Client-side Design

DeepVista adopts a thin-client paradigm. The client de-multiplexes the VS and PS from the received data. For the PS, the client maintains a large encoded frame buffer (30 seconds) to cushion the bandwidth fluctuation and minimize the stall. The VS is much more delay-sensitive than the PS, so the client immediately decodes VS frames without any *prior-to-decoding buffering*. The edge coordinates with the client by pacing its VS transcoding with the client-side playback and performing a resync when the VS falls behind.

Video decoding on mobile devices is time-consuming [35, 40]. Compared to tile-based 360° video streaming that requires large decoded buffers to instantly stitch the tiles [47], DeepVista involves only the VS and PS streams, and the randomness of the viewport movement is already taken into account by the edge. Henceforth, the client-side decoding has a much lower overhead and becomes independent of the viewport movement. In DeepVista, the client only maintains a shallow *decoded buffer* of 5 frames (empirically chosen). The rationale for a shallow buffer is to reduce the transcoding-to-rendering latency to make viewport prediction more accurate, while absorbing the jitter of the network and client-side decoding.

Since the edge delivers 2 or 3 streams (1 PS + 1 or 2 VS), the client utilizes 2 or 3 decoding threads, respectively, which output the decoded VS/PS frames into their corresponding buffers. When rendering the viewport, the client first checks if the PS buffer has the required decoded frame. If not, a stall occurs; otherwise, the client applies the CubeMap projection to project the PS frame based on the user's current viewport. Next, the client examines the VS buffer. An empty VS buffer indicates that the VS is falling behind the PS, and thus a *resync* is issued (§5.3). Otherwise, the client employs a modified projection method to render the VS in the viewport, with the rendered portion overwriting that of the PS. The modification comes from the fact that the VS is received with reorganized blocks (Figure 1(b)) so the client needs to project each block back onto the original location in the viewport. This location is obtained through the same block visibility map used by the edge (§5.2). Figure 1(c) shows the rendered viewport.

6 SYSTEM IMPLEMENTATION

We implement DeepVista on commodity machines and smartphones. The edge is developed in C++ and Python (6.9K LoC). We use the NVIDIA Video Codec SDK [18] for hardware-assisted decoding and encoding, and NVIDIA CUDA API [16] for block reorganization. To enable accurate network performance measurement, we employ the libpcap API [21] to get access to the TCP packet-level data. The downlink (edge-to-client) throughput is estimated by monitoring the uplink TCP ACK stream. We use Keras [11] for LSTM offline training and real-time inference. We integrate the components from §5.2 to §5.5 into a holistic edge system. To optimize the performance, we adopt a deep pipelining design where 7–9 threads (depending on one or two GPUs) process the data and control planes at different stages. We also employ a simple throughput-based rate adaptation algorithm [38] between the server and edge.

The client is developed using the Android SDK (2.6K LoC). It employs the low-level MediaCodec API [2] to perform video decoding, and OpenGL ES for CubeMap rendering.

Our prototype supports three modes: 2GPU/16K, 1GPU/16K, and 1GPU/8K. In these modes, the VS outputs of the edge are: two 4K×4K streams, one 8K×4K stream, and one 4K×2K stream, respectively. The PS has a resolution of 2K×1K. Both the VS and PS are encoded in H.265 format.

7 EVALUATION

7.1 Experimental Setup

Edge and Server. Unless otherwise stated, we conduct experiments using a machine with an Intel i7-5820K CPU and two NVIDIA GeForce GTX 1080 Ti GPUs (launched in Q1 2017) as the edge. It runs Ubuntu 18.04 with Linux kernel 4.18. We benchmark the performance of a better GPU, RTX 2080 Ti, in §7.7. We use another machine as the video server, which is a stateless HTTP server.

Mobile Devices. We use two off-the-shelf smartphones. One is an SGS8 (launched in Q2 2017) with an octa-core Cortex-A53 CPU, a Mali-G71 MP20 GPU, and 4GB memory, running Android 9.0. The other is a Samsung Galaxy S10 5G (Q2 2019) with an octa-core Kryo 485 CPU, an Adreno 640 GPU, and 8 GB memory, running Android 10.0. We use the SGS8 by default, and use the SGS10 only for experiments over commercial mmWave 5G networks as the SGS8 does not have a 5G modem. Neither phone is rooted[2].

Networking. Since our evaluation focuses on the edge and the client, we do not consider the scenario where the server-edge path becomes the performance bottleneck. We therefore inter-connect the server and the edge over 1Gbps Ethernet. We experiment with three types of networks between the edge and the client.

[2]Both phones' screen resolutions are lower than 8K. However, this does not affect our evaluation results because the phones indeed download, decode, buffer, and render 8K content for the VS.

Synthetic	H.264 (Mbps)		H.265 (Mbps)	
16K Video	CRF18	CRF23	CRF18	CRF23
Interpolated (Angel Falls)	246	219	159	80
Stitched from four 8K videos*	328	277	232	122

*Angel Falls [3], NYC [14], Roller Coaster [22], and New Year [1].

Table 3: Comparison of 16K videos generated in two ways.

• **An 802.11n WiFi network** with a peak bandwidth of 100 Mbps and a latency of less than 2ms. This represents a typical usage scenario of DeepVista where the edge is a PC at the user's home.
• **Commercial LTE networks** provided by a large U.S. cellular carrier. We traveled to several locations (a university campus, a residential area, a shopping plaza, and a business district) in a large U.S. city to perform live experiments. The bandwidth ranges from 30 to 80 Mbps, and the client-edge RTT is between 40 and 90ms.
• **Commercial mmWave 5G networks** offered by another major cellular carrier in the U.S. We also performed on-site tests at several places with mmWave 5G coverage (28 GHz): a residential area and two busy downtown areas, where the bandwidth ranges from 300 Mbps to 1 Gbps and the client-edge RTT is between 25 and 40 ms.

Video Content. We select three high-quality 360° videos from YouTube: scenery (Angel Falls [3]), urban (NYC Drive [14]), and entertainment (roller coaster [22]). They had a total number of 14.4M+ views as of October 2020. One problem is that YouTube provides only up to 8K resolution for these videos, and we are unable to find any 16K 360° videos online. We address this issue in two ways. First, we create synthetic 16K videos by enlarging the above three videos' resolution from 8K to 16K using pixel-wise interpolation. These interpolated 16K videos have meaningful content and the desired resolution, but they are more compressible and therefore having slightly lower bitrates than "real" 16K videos. Second, we create another type of 16K videos by stitching together four content-wise different 8K videos shown in Table 3. Complementing the interpolated videos, the stitched 16K videos do not have meaningful content, but their encoded bitrate is equivalent or even higher than real 16K videos due to their complex content, so streaming them is expected to be even more challenging than "real" 16K videos.

Table 3 compares the encoded bitrates (in Mbps) of the two types of synthetic videos, using two encoders (FFmpeg H.264 and H.265) and two quality levels (CRF 18 and 23). The results validate the extrapolation from [26] that high-quality 16K videos require up to 300Mbps bandwidth. We test both the interpolated videos and the stitched videos on DeepVista, and observe very similar performance in terms of decoding/reorganization/encoding latency on the proxy side and the decoding/rendering latency on the client side. Unless otherwise noted, we use the interpolated 16K videos as 16K sources because our collected viewport traces are based on their content. On the server side, they are encoded using H.265 + CRF 18, an almost lossless encoding configuration [5]. We create the PS for each video at ~2Mbps. All videos have an FPS of 30.

Therefore, *despite the limitation of using synthetic 16K videos, we believe that our evaluations are sound* due to two reasons.
• The stitched 16K videos, whose content is more complex compared to "real" 16K videos, show similar transcoding performance compared to the interpolated ones.
• The bitrates of our transcoded VS streams at 8K match those of real 8K videos (§5.5). Hence, using 16K synthetic videos does not change the workload over wireless links.

Mode	Edge Dec.	Edge Enc.	Phone Dec.
(1) 2GPU, 16K	32.3±0.1	56.7±0.2	37.9±0.1
(2) 1GPU, 16K	16.1±0.1	28.3±0.2	38.9±0.2
(3) 1GPU, 8K	60.6±0.0	109±2.6	88.8±0.1

Table 4: Hardware performance benchmark averaged over all frames of Angel Falls video. The unit is FPS (frames-per-second). Edge-side encoding and decoding are performed concurrently.

Mode	Edge-only FPS	% Skipped VS Frames (End-to-end at 30 FPS)
2GPU/16K	30.2±0.1	0.69%
1GPU/16K	16.0±0.1	46.7%
1GPU/8K	60.4±0.2	0

Table 5: Overall performance of DeepVista**. The high skipped frame ratio of 1GPU/16K+PS is due to edge-side slowing down.**

Viewport Movement Data. We conduct an IRB-approved user study involving 42 voluntary participants (undergraduates, graduate students, and faculty) from a large university, with 40% of them being female. We ask each participant to watch the three videos when wearing a Samsung Gear VR headset and capture their viewport movement trajectories at 30 Hz. We notice that the head movements and their predictability indeed exhibit high heterogeneity. Among the users, the linear regression prediction accuracy[3] ranges from 51.0% to 96.5% (average: 76.8%, stdev: 7.9%). From the 42 users, we uniformly sample 12 users who are "representative" in terms of their viewport prediction accuracy, which ranges from 59.6% to 94.9% (average: 76.7%, stdev: 9.5%). We replay their viewport trajectories in our evaluation. Unless otherwise noted, in our experiments, we use LSTM models trained from the remaining 30 users (not overlapping with the 12 users) for viewport prediction. As described in §5.4, we train separate models for each video.

7.2 Benchmarking Hardware Performance

We begin with profiling our edge and client's hardware capability of video encoding and decoding. Note that this experiment does *not* use DeepVista. Instead, we develop two simple benchmark programs: one runs on the edge and performs simultaneous decoding and encoding (without transcoding); the other performs decoding on the client. Table 4 shows the benchmarking results for DeepVista's three working modes (§6). On the edge, the bottleneck is decoding: using two GPUs can barely reach 32 FPS due to the immense resolution of 16K content. The phone alone can achieve a satisfactory decoding FPS for viewport content. The above results provide a performance *upper bound* for DeepVista, limited by the hardware capability. They also imply that the edge must utilize the deep pipelining approach (§6), otherwise the end-to-end FPS can easily drop below 30 FPS. Using the videos stitched from four 8K videos (§7.1) yields similar results.

7.3 DeepVista Overall Performance

Table 5 summarizes the overall performance of DeepVista. The workload is the Angel Falls video replayed using the head movement trace of an average user in terms of the viewport prediction accuracy. The other videos (including the stitched videos) and users

[3]We use a prediction window of 500ms and a history window of 250ms. A prediction is performed for every frame, and is considered to be accurate if the errors of both the predicted latitude and longitude are less than 10°. The prediction accuracy of one playback is defined as the fraction of accurate predictions over all predictions.

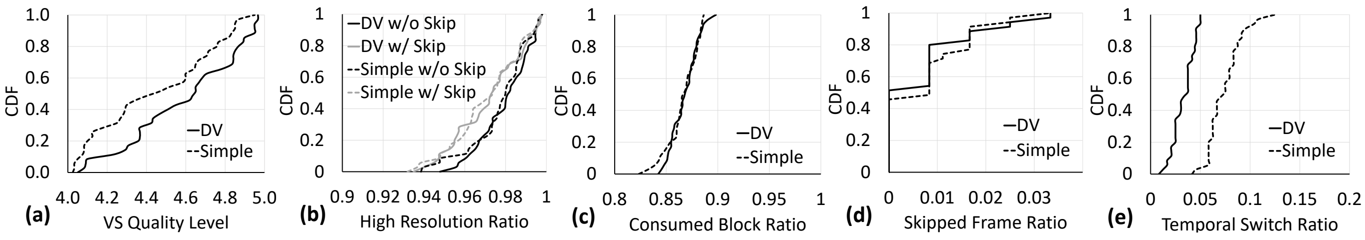

Figure 4: Comparison of DeepVista **(DV) and its simple version over unthrottled WiFi: (a) VS Quality Level, (b) High Resolution Ratio (HRR), (c) Consumed Block Ratio (CBR), (d) Skipped Frame Ratio (SFR), and (e) Temporal Switch Ratio (TSR).**

yield similar results. First, to measure the edge-side FPS, we connect the edge to a *dummy client* and let the edge execute as fast as possible. For 2GPU/16K, the edge manages to achieve 30.2 FPS, only a 6.5% drop from the upper bound shown in Table 4. The drop is mostly caused by block reorganization that competes for GPU resources with encoding and decoding. The results confirm the effectiveness of DeepVista's pipelined approach. For 8K streaming, the edge can achieve 60 FPS on a single GPU.

We next measure the percentage of skipped VS frames under an *end-to-end* setting where the client is connected to the edge over unthrottled 802.11n WiFi. The rightmost column of Table 4 indicates that for 2GPU/16K and 1GPU/8K (with the PS), DeepVista can achieve an end-to-end FPS of 30 with negligible or no frames being skipped. For 1GPU/16K, DeepVista cannot reach 30 FPS due to the edge-side slowing down. Therefore, unless otherwise noted, we evaluate 16K streaming using 2 GPUs in the rest of this section.

Last-mile Bandwidth Savings. Compared to the monolithic approach of fetching the panoramic scene, DeepVista can significantly reduce the last-mile bandwidth usage (more precisely, between the edge and the client). The actual savings depend on the content and viewport trajectory. It can be quantified as $1 - \frac{B_{VS}+B_{PS}}{B_{PAN}}$ where B_{VS}, B_{PS}, and B_{PAN} are the consumed bytes of the VS, the PS, and the panoramic stream (at the same quality as the VS), respectively. Using the viewport trace of an average user (in terms of the viewport prediction accuracy), we calculate the savings to be 69%, 71%, and 52% for the three videos when the bitrate of the VS is 48Mbps. Lowering the VS bitrate leads to similar savings.

7.4 16K Streaming Quality of Experience (QoE)

We now assess the QoE of 16K video streaming. To make the experiments reproducible, we lively replay the 12 representative users' viewport traces of the three videos (§7.1).

We calculate five QoE metrics for each playback. (1) The **Average VS Quality Level** ranging from 1 (lowest) to 5 (highest). They correspond to the five bitrate levels described in §5.5. (2) The **Average High Resolution Ratio (HRR)** as defined in §5.5, which quantifies the fraction of the viewport occupied by the VS as opposed to the PS, across all frames. (3) The **Consumed Block Ratio (CBR)**, defined as the ratio between the total number of user-consumed (*i.e.,* perceived) blocks to the total number of transferred blocks across all VS frames. A high CBR indicates high bandwidth efficiency. (4) The **Skipped Frame Ratio (SFR)**, defined as the fraction of skipped VS frames. (5) The **Temporal Switch Ratio (TSR)**, defined as the number of total VS quality level changes divided by the maximum possible number of VS level changes (4 levels per pair of consecutive V-chunks). A lower TSR is preferred as rapid changes in the quality level are known to be detrimental to the QoE [57]. (6) The **Stall (Rebuffering) Duration**, which is found to be 0 in all the playbacks over WiFi, LTE, and 5G. This is because DeepVista uses the PS whose exact purpose is to eliminate stalls and blank areas not covered by the VS. We therefore focus on the first five metrics in the remainder of this section.

Since we are not aware of any well-documented solution for 16K 360° video streaming, we compare DeepVista with its simplified version. They differ in the rate adaptation module. The simple version takes a greedy approach: the quality level of each V-chunk is independently determined to be the highest level that the estimated bandwidth can sustain.

WiFi Networks. We first evaluate DeepVista under 802.11n WiFi. This represents the scenario where the edge and the client are in the same wireless LAN at the user's home or office. We show the results in Figure 4, where the subplots correspond to the aforementioned five metrics. Each curve consists of 12 users × 3 videos = 36 data points (playbacks). We highlight the key results below. For DeepVista, since the bandwidth is quite high, the rate adaptation properly determines the VS quality level to be between 4 and 5, as shown in Figure 4(a). Figure 4(b) indicates the median HRR is around 98%, confirming that most area in the viewport is covered by the VS. The "w/o Skip" and "w/ Skip" curves calculate the HRR in different ways: the former ignores skipped frames, while the latter assumes a skipped frame has an HRR of 0 (so the HRR becomes statistically lower). Figure 4(c) shows the median CBR is around 87%, implying that most downloaded blocks are indeed consumed. Figure 4(d) demonstrates the low SFR: about 51% of the playbacks do not experience any skipped frames, and the 90% percentile of SFR is only 2.5% (1.5 sec per minute). Figure 4(e) shows the low frequency of VS quality switches. Regarding the simple rate adaptation scheme, due to its "shortsighted" greedy nature, it underperforms DeepVista in the VS quality level and TSR.

Commercial LTE and 5G Networks. We now present the results of live experiments over commercial LTE/5G networks. At each location (§7.1), we repeatedly play the Angel Falls video for 50 times using the 12 users' head movement traces in a random order at different times of the day. Each curve in Figure 5 thus consists of 200 runs for LTE and 150 runs for 5G. We spent more than 400 GB of LTE/5G data on the experiments. We first examine the LTE results. Compared to Figure 4, several metrics in Figure 5 degrade: the limited bandwidth reduces the median VS quality level to 3.8; the bandwidth/RTT fluctuation makes rate adaptation more challenging and thus increases the SFR. Despite these, DeepVista maintains decent performance: the median HRR is 95.6% (98.9%) when the skipped frames are accounted (ignored). The median SFR

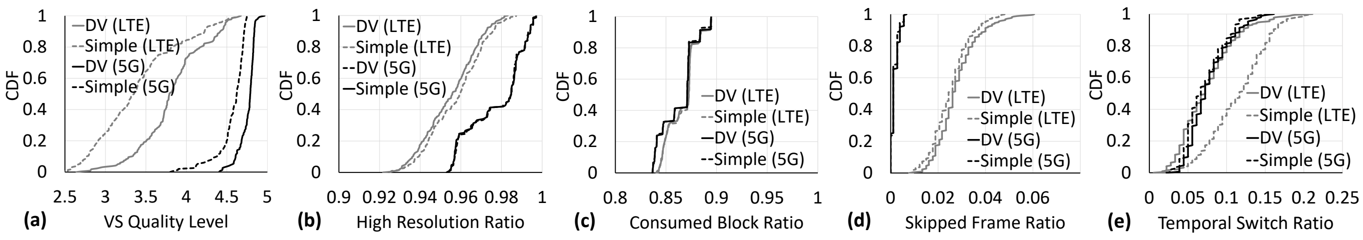

Figure 5: Comparison of DeepVista **(DV) and its simple version over live LTE/mmWave 5G Networks. Subplot (b) shows HRR with skips.**

Mode	Decoding (ms)	Reorg. (ms)	Encoding (ms)
2GPU/16K	31.0 ± 0.1	2.6 ± 0.0	17.6 ± 0.1
1GPU/16K	61.9 ± 0.1	0.3 ± 0.0	35.5 ± 0.0
1GPU/8K	16.5 ± 0.0	0.1 ± 0.0	9.2 ± 0.3

Table 6: DeepVista **edge performance breakdown.**

remains as low as 2.6% – equivalent to skipping 1.56-second worth of VS content every minute. Note that even if a VS frame is skipped, the PS frame is still played so the viewer will not perceive any stall.

More excitingly, we observe excellent results for mmWave 5G attributed to its high bandwidth and low RTT variation (§7.1). The median quality level and SFR across all runs are 4.8 and 0.1%, respectively, close to the perfect QoE. The other metrics including HRR, CBR, and TSR are also satisfactory. Note that the quality level is slightly lower than 5.0 because it needs to gradually ramp up at the beginning of a playback. Overall, the results in Figure 5 indicate that DeepVista can provide good QoE even when the edge is not in the immediate vicinity of the client.

PS's Impact on Image Quality. In DeepVista, a viewport may consist of both the VS and PS content. To understand the impact of the PS on the image quality (*i.e.,* the impact of a non-perfect HRR that is less than 1), we calculate the SSIM [52] for three viewport streams: *Perfect VS* (generated offline using the real viewport trajectory, so HRR=1), *Predicted VS + PS* (DeepVista's approach, HRR≤1), and *Predicted VS Only* (removing the PS). The ground truth is the viewport stream extracted from the original 16K video. We use the viewport trajectory trace of one average user (in terms of the HRR), the Angel Falls video, and the 48Mbps VS bitrate to conduct the calculation. The above three streams yield an average SSIM of 0.985, 0.980, and 0.969, respectively, across their frames. These numbers already take the quality loss due to video codec into account (so that the first number is less than 1). Note that an SSIM index higher than 0.98 implies that the image is visually lossless compared to the ground truth [28]. The results indicate that (1) compared to only showing the predicted VS, adding the PS can enhance the perceived quality; and (2) compared to the prefect VS, the small area patched by the PS has a very small impact on the image quality.

7.5 DeepVista Performance Breakdown

We study the performance breakdown of the edge (Table 6) and client (Table 7), using the same workload as that used in Table 5. For the edge, we again let it execute as fast as possible by connecting it with a dummy client. Table 6 shows the per-frame execution time for three major tasks: decoding, block reorganization, and VS encoding. For 2GPU/16K, the ratio among them is about 12:1:7, consistent with our findings in Table 4 that decoding remains the performance bottleneck. The block reorganization phase for 2GPU/16K takes a much longer time than that for 1GPU/16K (2.6ms vs. 0.3ms) due to the cross-GPU block exchange (§5.2), which is the only performance penalty incurred when dual GPUs are used.

Table 7 profiles the client performance when the client is connected to the edge over unthrottled 802.11n WiFi. It shows per-frame execution time of three major client-side components: decoding a frame, copying a decoded frame to the buffer, and CubeMap rendering. Decoding still remains the most time-consuming task.

7.6 Comparison with Flare

We compare DeepVista with Flare [47], a representative viewport-adaptive 360° video streaming system. Flare employs a client-only, tile-based approach (§2). Since Flare does not support 16K video streaming, we compare DeepVista with it at 8K resolution. To ensure apple-to-apple comparisons, we take the following measures. (1) Both DeepVista and Flare use the same three videos and 12 viewport traces on the same testbed as described in §7.1. We prepare three segmentation schemes for Flare: 2×4 (Figure 2), 4×4, and 4×6. (2) We disable both systems' rate adaptation by using only one quality level. (3) Both use linear regression for viewport prediction. (4) DeepVista uses the "1GPU, 8K" mode in Table 4 with 2 decoding threads; Flare uses 3 decoding threads, which is the best decoder configuration on SGS8.

We show the results in Figure 6 where each bar corresponds to 3 videos × 12 users = 36 playbacks. Figure 6(a) compares the HRR: DeepVista achieves a median HRR of around 0.98 whereas Flare always has an HRR of 1, because Flare needs to fetch all the tiles within the viewport (*i.e.,* the player will stall when any tile is missing). Figure 6(b) compares the CBR. The CBR for Flare has a similar definition except that we consider tiles instead of blocks. This figure shows that compared to Flare, DeepVista improves the CBR by a factor of 1.4× to 1.9×. The reason is that Flare inherently needs to fetch much more tiles than actually consumed in order to combat inaccurate viewport prediction. While DeepVista also delivers additional blocks, it is less aggressive; instead, DeepVista uses the PS as a "protection" for missing VS blocks within a viewport.

Figure 6(c) examines the network traffic size. DeepVista drastically reduces the median downloaded bytes by a factor of 3.0× to 3.6× compared to Flare. Note that for each video, we make sure the encoded content inside the viewport has roughly the *same bitrate* between DeepVista and Flare. The PS is also counted when calculating the downloaded bytes for DeepVista. There are three main reasons for such great disparities. First, Figure 6(b) already shows that Flare is much more aggressive than DeepVista in terms of fetching content. Second, spatially, a block is much smaller than a tile so that DeepVista can more precisely follow the contour of a predicted viewport than Flare. Third, in Flare, video frames are

Mode	Frame Decoding	Texture Copying	CubeMap Rendering
2GPU/16K+PS	26.4 ± 0.1	18.6 ± 2.2	4.7 ± 1.1
1GPU/16K+PS	25.7 ± 0.2	14.7 ± 0.6	4.7 ± 1.1
1GPU/8K+PS	11.3 ± 0.0	4.7 ± 0.6	3.4 ± 0.6

Table 7: Client performance breakdown (in ms).

Mode	Edge Decoding FPS		Edge Encoding FPS		Overall Edge FPS	
	1080 Ti	2080 Ti	1080 Ti	2080 Ti	1080 Ti	2080 Ti
2GPU,16K	32.3±0.1	60.6±0.8	56.7±0.2	52.4±0.0	30.2±0.1	45.9±0.6
1GPU,16K	16.1±0.1	31.2±0.2	28.3±0.2	25.7±0.3	16.0±0.1	24.6±0.2
1GPU,8K	60.6±0.0	118.1±0.3	109±2.6	101±2.3	60.4±0.2	88.5±1.7

Table 8: Hardware performance benchmark averaged over all frames of the Angel Falls video, using 1080Ti vs. 2080Ti GPU(s).

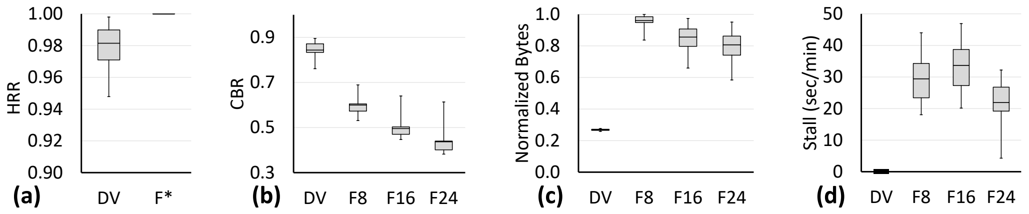

Figure 6: Comparison of DeepVista **with Flare on 8K streaming. Schemes: DV =** DeepVista**, F8 = Flare with 2×4 tiles, F16 = Flare with 4×4 tiles, F24 = Flare with 4×6 tiles. Subplots: (a) High Resolution Ratio (HRR), (b) Consumed Block/Tile Ratio (CBR), (c) Downloaded bytes (normalized), (d) Stall (seconds per minute).**

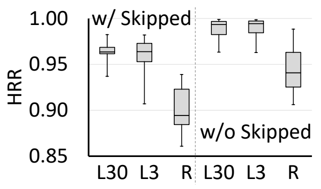

Figure 7: Impact of three viewport prediction schemes on HRR.

aggregated into 1-sec chunks. Even if only one frame is required, the entire chunk needs to be downloaded. This restriction does not exist in DeepVista, which makes the decision frame-by-frame due to its real-time transcoding nature.

The most striking difference is shown in Figure 6(d). Flare incurs long duration of stalls because of the high tile decoding overhead. Based on our experience, given the same panoramic video, segmenting it into multiple tiles will increase the decoding overhead, in particular when the number of to-be-decoded-tiles is larger than the number of hardware decoders. This explains why Flare, which uses a tile-based approach, incurs high decoding overhead when streaming 8K 360 videos – for every chunk, up to 24 tiles in Flare (36 in Rubiks [35]) need to be decoded. This issue does not exist in DeepVista, where viewport extraction is offloaded to the edge.

7.7 Impact of Edge-side GPU

We wonder how a more powerful GPU can help improve the overall performance of DeepVista. Table 8 compares the performance of two models: NVIDIA GTX 1080 Ti (launched in Q1 2017) and NVIDIA RTX 2080 Ti (Q4 2018). We consider three working modes: 2GPU/16K, 1GPU/16K, and 1GPU/8K. Recall that all our results reported in other subsections are obtained using 1080 Ti, whose corresponding numbers in Table 8 are copied from Tables 4 and 5.

We make some interesting observations from Table 8. First, 2080 Ti's decoding capability is superior – its decoding FPS almost doubles compared to 1080 Ti. This shifts the edge-side bottleneck to the encoding stage, which, to our surprise, is not improved. Due to its significantly improved decoding performance, 2080 Ti yields an FPS increase of 52%, 54%, and 47% for 2GPU/16K, 1GPU/16K, and 1GPU/8K, respectively, compared to 1080 Ti. However, due to 16K videos' extremely high resolution, a single 2080 Ti GPU is still incapable of transcoding them at 30 FPS, not to mention an even higher FPS such as 60 or 90. This makes DeepVista's dual-GPU transcoding scheme essential.

7.8 Viewport Prediction

Figure 7 compares the HRR under three viewport prediction methods: LSTM trained using 30 users (L30), LSTM trained using 3 randomly selected users (L3), and linear regression (R). For linear regression, the history window is set to be half of the prediction window according to Flare [47]. The workload is the Angel Falls video replayed using the 12 users' head movement traces over unthrottled WiFi. The results confirm the effectiveness of LSTM: compared to linear regression, it improves the median HRR by 7% (96.4% vs. 89.4%, assuming skipped frames have an HRR of 0) or 5.3% (99.4% vs. 94.1%, ignoring skipped frames).

We make an interesting observation that the models trained from 3 and 30 users yield similar HRR. This makes applying LSTM easier, but appears to be counter-intuitive to our initial expectation that LSTM requires a large number of users' head movement data for training. One possible reason might be that there exist some intrinsic similarities among viewers when watching the same 360° video. Another possible explanation is that, although LSTM outperforms linear regression by capturing non-linear patterns, the LSTM models may still underfit the data due to the inherent complexity of human head movement. Exploring whether more sophisticated models can capture such complexities is our future work.

7.9 CPU, GPU, Memory, and Energy Usage

Edge-side. We report the resource consumption of DeepVista using the same workload as that used in §7.3. We consider the 2GPU/16K mode. The edge uses up to three logical CPU cores, and its maximum main memory usage is 1.86 GB. We use the *nvidia-smi* tool [45] to monitor the GPU usage. For each individual GPU, its maximum utilization is 62%. The GPU memory usage is configurable by adjusting the decoded cache size (§5.3). When we set the cache size to 60 half-16K frames, the maximum memory usage for each GPU is 6.7 GB (out of 11 GB offered by GeForce GTX 1080 Ti).

Client-side Hardware Resource Utilization. We monitor the client-side resource utilization using Android Profiler [31]. During a 16K video playback, the maximum CPU usage is only 23%, and the total memory usage is 0.6GB due to its shallow cache design. Note that SGS8's video memory is shared with its main memory.

Client-side Energy and Thermal Overhead. We fully charge the SGS8 and then play the 16K Angel Fall video for 30 minutes using DeepVista. After that, the battery level drops to 91%. Our SGS8 does not expose an interface to measure the device's internal temperature. Nevertheless, after a 30-minute playback, the phone feels only moderately warm. We believe the thermal overhead should not be a concern given DeepVista's low energy consumption.

8 CONCLUDING REMARKS

By judiciously offloading sophisticated tasks such as viewport-adaptive transcoding and deep viewport prediction to the edge, DeepVista enables streaming 16K 360° videos to commodity mobile devices at the line rate. It makes an important initial step towards efficient delivery and processing of extremely high-resolution panoramic content. We believe that many concepts of DeepVista are applicable to other multimedia applications such as volumetric video streaming [34] and real-time VR/AR streaming [40, 44, 48].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. This work was supported in part by NSF Award #1903880 and #1915122, Project 16214817 from the Research Grants Council of Hong Kong, and the 5GEAR and FIT projects from Academy of Finland.

REFERENCES

[1] 2019 New Year Illumination in Moscow, Russia (360 Video). https://www.youtube.com/watch?v=LdpzR_pZ9-w.
[2] Android MediaCodec API. https://developer.android.com/reference/android/media/MediaCodec.
[3] Angel Falls (360 Video). https://www.youtube.com/watch?v=L_tqK4eqelA.
[4] Big Screens, Big Data: The Future For Smartphones. https://www.npd.com/wps/portal/npd/us/blog/2018/big-screens-big-data-the-future-for-smartphones/.
[5] CRF Guide (Constant Rate Factor in x264, x265 and libvpx). https://slhck.info/video/2017/02/24/crf-guide.html.
[6] Explaining 360 video resolution: how to measure it, quality comparisons, and other metrics to look at. https://www.immersiveshooter.com/2017/08/31/explaining-360-video-resolution-how-measure-quality-comparisons/.
[7] Google AR and VR: Bringing pixels front and center in VR video. https://blog.google/products/google-ar-vr/bringing-pixels-front-and-center-vr-video/.
[8] Google Cardboard. https://vr.google.com/cardboard/index.html.
[9] Google Compute Engine Pricing. https://cloud.google.com/compute/pricing.
[10] Google Stadia. https://stadia.com/.
[11] Keras: The Python Deep Learning library. https://keras.io/.
[12] Making 12K 360° VR Streaming a Reality. https://medium.com/visbit/making-12k-360%C2%BA-vr-streaming-a-reality-why-and-how-we-did-it-ce65e9aa0bc3.
[13] New 8K OLED Displays for Tablets and Laptops: 8.3 and 13.3 Inches. https://www.anandtech.com/show/13742/new-8k-oled-displays.
[14] New York City VR 360 Drive (360 Video). https://www.youtube.com/watch?v=2Lq86MKesG4.
[15] Next-generation video encoding techniques for 360 video and VR. https://code.fb.com/virtual-reality/next-generation-video-encoding-techniques-for-360-video-and-vr/.
[16] Nvidia CUDA Runtime API. https://docs.nvidia.com/cuda/cuda-runtime-api/.
[17] NVIDIA Launches Edge Computing Platform to Bring Real-Time AI to Global Industries. https://nvidianews.nvidia.com/news/nvidia-launches-edge-computing-platform-to-bring-real-time-ai-to-global-industries.
[18] Nvidia Video Codec SDK. https://developer.nvidia.com/nvidia-video-codec-sdk.
[19] NVIDIA Video Encode and Decode GPU Support Matrix. https://developer.nvidia.com/video-encode-decode-gpu-support-matrix.
[20] Per-Title Encode Optimization. https://medium.com/netflix-techblog/per-title-encode-optimization-7e99442b62a2.
[21] Programming with pcap. https://www.tcpdump.org/pcap.html.
[22] Roller Coaster (360 Video). https://www.youtube.com/watch?v=s14wKgPPQ-c.
[23] O. Abari, D. Bharadia, A. Duffield, and D. Katabi. Enabling High-Quality Untethered Virtual Reality. In *Proceedings of NSDI*, 2017.
[24] S. Afzal, J. Chen, and K. Ramakrishnan. Characterization of 360-Degree Videos. In *Proceedings of the Workshop on VR/AR Network*, pages 1–6. ACM, 2017.
[25] S. Aggarwal, S. Paul, P. Dash, N. S. Illa, Y. C. Hu, D. Koutsonikolas, and Z. Yan. How to evaluate mobile 360° video streaming systems? In *ACM HotMobile*, 2020.
[26] P. R. Alface, M. Aerts, D. Tytgat, S. Lievens, C. Stevens, N. Verzijp, and J.-F. Macq. 16K Cinematic VR Streaming. In *ACM Multimedia*, 2017.
[27] B. Chen, Z. Yan, H. Jin, and K. Nahrstedt. Event-driven stitching for tile-based live 360 video streaming. In *ACM MMSys*, 2019.
[28] 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 ACM MobiSys*, 2015.
[29] M. Dasari, A. Bhattacharya, S. Vargas, P. Sahu, A. Balasubramanian, and S. R. Das. Streaming 360 videos using super-resolution. In *IEEE INFOCOM*, 2020.
[30] C.-L. Fan, J. Lee, W.-C. Lo, C.-Y. Huang, K.-T. Chen, and C.-H. Hsu. Fixation Prediction for 360 Video Streaming in Head-Mounted Virtual Reality. In *ACM NOSSDAV*, 2017.
[31] Google. Android Profiler. https://developer.android.com/studio/profile/android-profiler.
[32] M. Graf, C. Timmerer, and C. Mueller. Towards bandwidth efficient adaptive streaming of omnidirectional video over HTTP: Design, implementation, and evaluation. In *Proceedings of MMSys 2017*, pages 261–271. ACM, 2017.
[33] Y. Guan, C. Zheng, X. Zhang, Z. Guo, and J. Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *SIGCOMM*, 2019.
[34] B. Han, Y. Liu, and F. Qian. ViVo: visibility-aware mobile volumetric video streaming. In *ACM MobiCom*, 2020.
[35] J. He, M. A. Qureshi, L. Qiu, J. Li, F. Li, and L. Han. Rubiks: Practical 360-Degree Streaming for Smartphones. In *Proceedings of MobiSys 2018*. ACM, 2018.
[36] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.
[37] X. Hou, S. Dey, J. Zhang, and M. Budagavi. Predictive View Generation to Enable Mobile 360-degree and VR Experiences. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*, pages 20–26. ACM, 2018.
[38] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming With Festive. In *ACM CoNEXT*, 2012.
[39] T. Kämäräinen, M. Siekkinen, J. Eerikäinen, and A. Ylä-Jääski. Cloudvr: Cloud accelerated interactive mobile virtual reality. In *ACM Multimedia*, 2018.
[40] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *ACM MobiCom 2017*.
[41] Y. Li and W. Gao. Muvr: Supporting multi-user mobile virtual reality with resource constrained edge cloud. In *IEEE/ACM Symp. on Edge Computing*, 2018.
[42] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser. Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering. In *ACM MobiSys*, 2018.
[43] H. Mao, R. Netravali, and M. Alizadeh. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of SIGCOMM 2017*, pages 197–210. ACM, 2017.
[44] J. Meng, S. Paul, and Y. C. Hu. Coterie: Exploiting Frame Similarity to Enable High-Quality Multiplayer VR on Commodity Mobile Devices. In *ASPLOS*, 2020.
[45] NVIDIA. NVIDIA System Management Interface. https://developer.nvidia.com/nvidia-system-management-interface.
[46] S. Petrangeli, V. Swaminathan, M. Hosseini, and F. De Turck. An HTTP/2-based adaptive streaming framework for 360° virtual reality videos. In *ACM MM 2017*.
[47] F. Qian, B. Han, Q. Xiao, and V. Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *ACM MobiCom*, 2018.
[48] X. Ran, C. Slocum, M. Gorlatova, and J. Chen. ShareAR: Communication-efficient multi-user mobile augmented reality. In *ACM HotNets*, 2019.
[49] S. Shi, V. Gupta, M. Hwang, and R. Jana. Mobile vr on edge cloud: a latency-driven design. In *ACM MMSys*, 2019.
[50] S. Shi, V. Gupta, and R. Jana. Freedom: Fast recovery enhanced vr delivery over mobile networks. In *ACM MobiSys*, 2019.
[51] L. Sun, F. Duanmu, Y. Liu, Y. Wang, Y. Ye, H. Shi, and D. Dai. Multi-path multi-tier 360-degree video streaming in 5g networks. In *ACM MMSys*, 2018.
[52] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.
[53] L. Xie, X. Zhang, and Z. Guo. Cls: A cross-user learning based system for improving qoe in 360 video adaptive streaming. In *ACM Multimedia*, 2018.
[54] X. Xie and X. Zhang. POI360: Panoramic Mobile Video Telephony over LTE Cellular Networks. In *Proceedings of CoNEXT 2017*, pages 336–349. ACM, 2017.
[55] T. Xu, B. Han, and F. Qian. Analyzing viewport prediction under different VR interactions. In *ACM CoNEXT*, 2019.
[56] Z. Yan and C. W. Chen. RnB: Rate and Brightness Adaptation for Rate-distortion-energy Tradeoff in HTTP Adaptive Streaming over Mobile Devices. In *ACM MobiCom*, 2016.
[57] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In *ACM SIGCOMM*, 2015.
[58] A. Zare, A. Aminlou, M. M. Hannuksela, and M. Gabbouj. HEVC-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proceedings of MM 2016*, pages 601–605. ACM, 2016.
[59] Y. Zhang, Y. Guan, K. Bian, Y. Liu, H. Tuo, L. Song, and X. Li. EPASS360: QoE-aware 360-degree Video Streaming over Mobile Devices. *IEEE Transactions on Mobile Computing*, 2020.
[60] Y. Zhang, P. Zhao, K. Bian, Y. Liu, L. Song, and X. Li. DRL360: 360-degree Video Streaming with Deep Reinforcement Learning. In *IEEE INFOCOM*, 2019.