Jigsaw: Robust Live 4K Video Streaming

Ghufran Baig[1*], Jian He[1*], Mubashir Adnan Qureshi[1], Lili Qiu[1], Guohai Chen[2], Peng Chen[2], Yinliang Hu[2]

[1]University of Texas at Austin, [2]Huawei

ABSTRACT

The popularity of 4K videos has grown significantly in the past few years. Yet coding and streaming *live* 4K videos incurs prohibitive cost to the network and end system. Motivated by this observation, we explore the feasibility of supporting live 4K video streaming over wireless networks using commodity devices. Given the high data rate requirement of 4K videos, 60 GHz is appealing, but its large and unpredictable throughput fluctuation makes it hard to provide desirable user experience. In particular, to support live 4K video streaming, we should (i) adapt to highly variable and unpredictable wireless throughput, (ii) support efficient 4K video coding on commodity devices. To this end, we propose a novel system, *Jigsaw*. It consists of (i) easy-to-compute layered video coding to seamlessly adapt to unpredictable wireless link fluctuations, (ii) efficient GPU implementation of video coding on commodity devices, and (iii) effectively leveraging both WiFi and WiGig through delayed video adaptation and smart scheduling. Using real experiments and emulation, we demonstrate the feasibility and effectiveness of our system. Our results show that it improves PSNR by 6-15dB and improves SSIM by 0.011-0.217 over state-of-the-art approaches. Moreover, even when throughput fluctuates widely between 0.2Gbps-2Gbps, it can achieve an average PSNR of 33dB.

CCS CONCEPTS

• **Human-centered computing** → **Ubiquitous and mobile computing systems and tools**; • **Information systems** → *Multimedia streaming*.

[*]Co-primary authors

MobiCom '19, October 21–25, 2019, Los Cabos, Mexico

ACM ISBN 978-1-4503-6169-9/19/10…$15.00
https://doi.org/10.1145/3300061.3300127

KEYWORDS

4k Video; Live streaming; 60 GHz; Layered coding

ACM Reference Format:
Ghufran Baig[1*], Jian He[1*], Mubashir Adnan Qureshi[1], Lili Qiu[1], Guohai Chen[2], Peng Chen[2], Yinliang Hu[2] . 2019. Jigsaw: Robust Live 4K Video Streaming. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19), October 21–25, 2019, Los Cabos, Mexico.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3300061.3300127

1 INTRODUCTION

Motivation: The popularity of 4K videos has grown rapidly, thanks to a wide range of display options, more affordable displays, and widely available 4K content. The first 4K TVs cost $20K in 2012, but now we can buy a good 4K TV under $200. Sony PS 4, Xbox One, Apple TV 4K, Amazon Fire TV, Roku and NVIDIA Shield all stream 4K videos. DirecTV, Dish Network, Netflix and YouTube all provide 4K programming. Gaming, Virtual Reality (VR) and Augmented Reality (AR) all call for 4K videos since the resolution has profound impact on the immersive user experience. Moreover, these interactive applications, in particular, VR and gaming, demand not only high resolution but also low latency (*e.g.*, within 60 ms [11, 14, 21]).

Many websites offer 4K video streaming. These videos are compressed in advance and require 35-68 Mbps [8] of throughput, which is affordable in many network conditions. In comparison, streaming live raw 4K videos (*e.g.*, in gaming, VR, AR) is much more challenging to support. Let us consider a raw 4K video is streamed at 30 frames per second (FPS) and uses 12-bit YUV color space. Without any compression, it requires ∼3 Gbps. Our commodity devices with the latest 60 GHz technology can only achieve 2.4 Gbps in the best case. With mobility, the throughput can quickly diminish and sometimes the link can be completely broken. Therefore, it is prohibitive to send raw 4K videos over current wireless networks.

Video coding has been widely used to significantly reduce the amount of data to transmit. Traditionally, a video is encoded using a few bitrates at the server side. The server then sends the video at an appropriate bitrate chosen based on the network bandwidth. The client receives the video and

decodes it to recover the raw video and then feeds it to the player.

However, coding 4K video is expensive. The delay requirement for streaming a live 4K video at 30 FPS means that we should finish encoding, transmitting, and decoding each video frame (having 4096 x 2160 pixels) within 60 ms [11, 14, 21]. Even excluding the transmission delay, simply encoding and decoding a 4K video can take long time. Existing commodity devices are not powerful enough to perform encoding and decoding in real-time. Liu et al [25] develop techniques to speed up 4K video coding using high-end GPUs. However, these GPUs cost over $1200, are bulky, consume high energy, and are not available for mobile devices. How to support live 4K videos on commodity devices still remains open.

Our approach: To effectively support live 4K video streaming, we identify the following requirements:

- **Adapting to highly variable and unpredictable wireless links:** A natural approach is to adapt the video encoding bitrate according to the available bandwidth. However, as many measurement studies have shown, the data rate of a 60 GHz link can fluctuate widely and is hard to predict. Even small obstruction or movement can significantly degrade the throughput. This makes it challenging to adapt video quality in advance.
- **Fast encoding and decoding on commodity devices:** It is too expensive to stream the raw pixels in 4K videos. Even the latest 60 GHz cannot meet its bandwidth requirement. On the other hand, the time to stream live videos includes not only the transmission delay but also encoding and decoding delay. While existing video coding (*e.g.*, H.264 and HEVC) achieves high compression rates, they are too slow for real-time 4K video encoding and decoding. Fouladi et al [15] show that YouTube H.264 encoder takes 36.5 minutes to encode a 15-minute 4K video at 24 FPS using H.264, which is too slow. Therefore, a fast video coding algorithm is needed to stream live 4K videos.
- **Exploiting different links:** WiGig alone is often insufficient to support 4K video streaming since its data rate may reduce by orders of magnitude even with small movement or obstruction. Sur et al [38] have developed approaches to detect when WiGig is broken based on the throughput in a recent window and then switch to WiFi reactively. However, it is challenging to select the right window size: a too small window results in unnecessary switch to WiFi and being constrained by the limited WiFi throughput and a too large window results in long link outage. In addition, even when WiGig link is not broken, WiFi can also complement WiGig by increasing the total throughput. The WiFi throughput can be significant compared with the WiGig throughput since the latter can be arbitrarily small depending on the distance, orientation, and movement. In addition, existing works mainly focus on bulk data transfer and do not consider delay sensitive applications like live video streaming.

We develop a novel system to address the above challenges. Our system has the following unique characteristics:

- **Easy-to-compute layered video coding:** To support links with unpredictable data rates, we develop a simple yet effective layered coding. It can effectively adapt to varying data rates on demand by first sending the base layer and then opportunistically sending more layers whenever the link allows. Different from traditional layered coding, our coding is simple and easy to parallelize on GPU.
- **Fast video coding using GPUs and pipelining:** To speed up video encoding and decoding, our system effectively exploits GPUs by parallelizing the encoding and decoding of multiple layers within a video frame. Our encoding and decoding algorithms can be split into multiple independent tasks that can be computed using commodity GPUs in real time. We further use pipelining to overlap transmission with encoding and overlap reception with decoding, which effectively reduces the end-to-end delay.
- **Effectively using both WiGig and WiFi:** While WiGig offers up to 2.4 Gbps of throughput, its data rate fluctuates a lot and is hard to predict. In comparison, WiFi is more stable even though its data rate is more limited (*e.g.*, 11ac offers up to 120 Mbps in our cards). Using both WiFi and WiGig not only improves the data rate but also minimizes the chances of video freezing under link outage. To accommodate the highly unpredictable variation in 60 GHz throughput, we defer video rate adaptation and scheduling, which determine how much data to send and which interface to use, as close to the transmission time as possible so that we do not require throughput prediction.

We implement our system on ACER laptops equipped with WiFi and WiGig cards without any extra hardware. Our evaluation shows that it can stream 4K live videos at 30 FPS using WiFi and WiGig under various mobility patterns.

2 MOTIVATION

Uncompressed 4K video streaming requires around 3Gbps of data rate. WiGig is the commodity wireless card that comes closest to such a high data rate. In this section, we examine the feasibility and challenges of streaming live 4K videos over WiGig from both system and network perspectives.

This study identifies major issues that should be addressed in supporting live 4K video streaming.

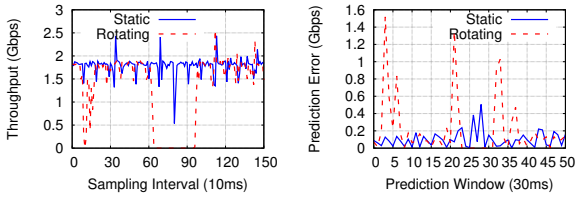
2.1 4K Videos Need Compression

WiGig throughput in our wireless card varies from 0 to 2.4 Gbps. Even in the best case, it is lower than the data rate of 4K raw videos at 30 FPS, which is 3 Gbps. Therefore, sending 4K raw videos is too expensive, and video coding is necessary to reduce the amount of data to transmit.

2.2 Rate Adaptation Requirement

WiGig links are sensitive to mobility. Even minor movement at the transmitter (TX) or receiver (RX) side can induce a drastic change in throughput. In the extreme cases, where an obstacle is blocking the line-of-sight path, throughput can reduce to 0. Evidently, such a throughput drop results in severe degradation in the video quality.

Large fluctuations: Fig. 1(a) shows the WiGig link throughput in our system when both the transmitter and receiver are static (Static) and when the TX rotates around a human body slowly while the RX remains static (Rotating). The WiGig link can be blocked when the human stands between the RX and TX. As you can see, the amount of throughput fluctuation can be up to 1 Gbps in the *Static* case. This happens due to multipath, beam searching and adaptation [31]. Since WiGig links are highly directional and have large attenuation factor due to high frequency, the impacts of multipath and beam direction on throughput are severe. In the *Rotating* case, the throughput reduces from 2 Gbps to 0 Gbps. We observe 0 Gbps when the user completely blocks the transmission between the TX and RX. Therefore, drastic throughput fluctuation is common in 60 GHz, and a desirable video streaming scheme should adapt to such fluctuation.



(a) Throughput Variation (b) Prediction Error

Figure 1: Example 60GHz Throughput Traces

Large prediction error: In order to predict the throughput fluctuation, existing streaming approaches use historical throughput measurements to predict the future throughput. The predicted throughput is then used to determine the amount of data to be sent. Due to large throughput fluctuation in WiGig links, the throughput prediction error can be large. In Fig. 1(b), we evaluate the accuracy of using the average throughput of the previous 40 ms window to predict the average throughput of the next 30 ms window. We observe large prediction error when there is a drastic change in the throughput. Even if throughput remains relatively stable in the *Static* case, the prediction error can reach up to 0.5 Gbps. When link blockage happens, the prediction error can be even higher than 1 Gbps in *Rotating* case. Such large fluctuation will cause selecting too low video rate or too high rate. The former degrades the video quality while the latter results in partially blank video frame because only part of the video frame can arrive in time.

Insight: *The large and unpredictable fluctuation of 60 GHz link suggests that we should adapt promptly to the unanticipated throughput variation. Layered coding is promising since the sender can opportunistically send less or more data depending on the network condition instead of selecting a fixed video rate in advance.*

2.3 Limitations of Existing Video Codecs

We explore the feasibility of using traditional codecs like H.264 and HEVC for 4K video encoding and decoding. These codecs are attractive due to their high compression ratios. We also study the feasibility of the state-of-the-art layered encoding scheme since it is designed to adapt to varying network conditions.

Cost of conventional software/hardware video codecs: Recent work [15] shows that YouTube's H.264 software encoder takes 36.5 minutes to encode a 4K video of 15 minutes and VP8 [10] software encoder takes 149 minutes to encode 15 minute video. It is clear these coding schemes are too slow to support live 4K video streaming.

To test the performance of hardware encoding, we use NVIDIA NVENC/NVDEC [4] hardware codec for H.264 encoding. We use a desktop equipped with 3.6GHz quad-core Intel Xeon Processor, 8GB RAM, 256GB SSD and NVIDIA Geforce GTX 1050 GPU to collect encoding and decoding time. We measure the 4K video encoding time where one frame is fed to the encoder every 33 ms. Table 1 shows the average encoding and decoding time of a single 4K video frame played at 30 FPS. We use ffmpeg [1] for encoding and decoding using NVENC and NVDEC codecs. The maximum tolerable encoding and decoding time is 60ms as mentioned earlier. We observe the encoding and decoding time is well beyond this threshold even with GPU support. This means that even using the latest hardware codecs and commodity hardware, video content cannot be encoded in time and result in unacceptable user experience. Such large coding time is not surprising. In order to achieve high compression rate, these codecs use sophisticated motion compensation [37, 40], which are computationally expensive.

A recent work [25] uses a very powerful desktop equipped with an expensive NVIDIA GPU ($1200) to stream 4K videos in real time. This work is interesting, but such GPUs are not available on common devices due to their high cost, bulky size (*e.g.*, $4.376'' \times 10.5''$ [3]) and large system power consumption (600W [3]). This makes it hard to deploy on laptops and smartphones. Therefore, we aim to bring the capability of live 4K video streaming to devices with limited GPU capabilities like laptops and other mobile devices.

Codec	Video	Enc(ms)	Dec(ms)	Total(ms)
H.264	Ritual	160	50	210
H.264	Barscene	170	60	230
HEVC	Ritual	140	40	180
HEVC	Barscene	150	50	200

Table 1: Codecs encoding and decoding time per frame

Cost of layered video codecs: Scalable video coding (SVC) is an extension of the H.264 standard. It divides the video data into base layer and enhancement layers. The video quality improves as more enhancement layers are received. The SVC uses layered code to achieve robustness under varying network conditions. But this comes at a cost of high computational cost since SVC needs more prediction mechanisms like inter-layer prediction [32]. Due to its higher complexity, SVC has rarely been used commercially even though it has been standardized as an H.264 extension [26]. None of the mobile devices have hardware SVC encoders/decoders.

To compare the performance of SVC with standard H.264, we use OpenH264 [5], which has software implementation of both SVC and H.264. Our results show that SVC with 3 layers takes 1.3x encoding time as that of H.264. Since H.264 alone is not feasible for live 4K encoding, we conclude that the SVC is not suitable for live 4K video streaming.

Insight: *Traditional video codecs are too expensive for 4K video encoding and decoding. We need a cheap 4K video coding that is fast to run on commodity devices.*

2.4 WiGig and WiFi Interaction

Since WiGig links may not be stable and can be broken, we should seek an alternative way of communication. As WiFi is widely available and low-cost, it is an attractive candidate. There have been several interesting works that use WiFi and WiGig together, however they are not sufficient for our purpose as they do not consider delay sensitive applications like video streaming.

Reactive use of WiFi: Existing works [38] use WiFi in reactive manner (only when WiGig fails). Reactive use of WiFi falls short for two reasons. First, WiFi is not used at all as long as WiGig is not completely disconnected. However, even when WiGig is not disconnected, its throughput can be quite low and WiFi throughput becomes significant. Second, it is hard to accurately detect the WiGig outage. A too early detection may result in unnecessary switch to WiFi, which tends to have lower throughput than WiGig, while a too late detection may lead to a long outage.

WiFi throughput: Second, it is non-trivial to use WiFi and WiGig links simultaneously since both throughput may fluctuate widely. This is shown in Fig. 2. As we can see, the WiFi throughput also fluctuates in both static and mobile scenarios. In the static case, the fluctuation is mainly due to contention. In the mobile cases, the fluctuation mainly comes from both wireless interference and signal variation due to multipath effect caused by mobility.

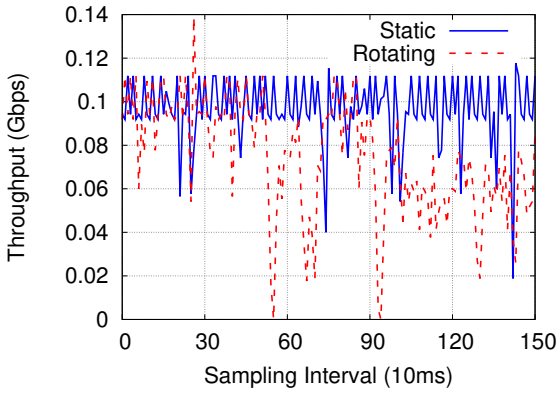


Figure 2: Example WiFi Throughput Traces

Insight: *It is desirable to use WiFi in a proactive manner along with the WiGig link. Moreover, it is important to carefully schedule the data across the WiFi and WiGig links to maximize the benefit of WiFi.*

3 JIGSAW

In this section, we describe the important components of our system for live 4K video streaming: (i) light-weight layered coding, (ii) efficient implementation using GPU and pipelining, and (iii) effective use of WiGig and WiFi.

3.1 Light-weight Layered Coding

Motivation: Existing video codecs, such as H.264 and HEVC, are too expensive for live 4K video streaming. A natural approach is to develop a faster codec albeit with a lower compression rate. If a sender sends more data than the network can support, the data arriving at the receiver before the deadline may be incomplete and insufficient to construct a valid frame. In this case, the user will see a blank screen. This can be quite common for WiGig links. On the other hand, if a sender sends at a rate lower than the supported

data rate (*e.g.*, also due to prediction error), the video quality degrades unnecessarily.

In comparison, layered coding is robust to throughput fluctuation. The base layer is small and usually delivered even under unfavorable network conditions so that the user can see some version of a video frame albeit at a lower resolution. Enhancement layers can be opportunistically sent based on network conditions. While layered coding is promising, the existing layered coding schemes are too computationally expensive as shown in Section 2. We seek a layered coding that can (i) be easily computed, (ii) support parallelism to leverage GPUs, (iii) compress the data, and (iv) take advantage of partial layers, which are common since a layer can be large.

3.1.1 Our Design. A video frame can be seen as a 2D array of pixels. There are two raw video formats: RGB and YUV where YUV is becoming more popular due to better compression than RGB. Each pixel in RGB format can be represented using three 8-bit unsigned integers in RGB (one integer for red, one for green, and one for blue). In the YUV420 planar format, four pixels are represented using four luminance (Y) values and two chrominance (UV) values. Our implementation uses YUV but the general idea applies to RGB.

We divide a video frame into non-overlapping 8x8 blocks of pixels. We further divide each 8x8 block into 4x4, 2x2 and 1x1 blocks. We then compute the average of all pixels in each 8x8 block which makes up the base layer or Layer 0. We round each average into an 8-bit unsigned integer, denoted as $A^0(i,j)$, where (i,j) denotes the block index. Layer 0 has only 1/64 of the original data size, which translates to around ~50 Mbps. Since we use both WiGig and WiFi, it is very rare to get the total throughput below ~50 Mbps. Therefore, layer 0 is almost always delivered. [1] While only receiving layer 0 gives a 512x270 video, which is a very low resolution, it is still much better than partial blank screen, which may happen if we try to send a higher resolution video than the link can support.

Next, we go down to the 4x4 block level and compute averages of these smaller blocks. Let $A^1(i,j,k)$ denote the average of a 4x4 block where (i,j,k) is the index of the k-th 4x4 block within the (i,j)-th 8x8 block and $k = 1,2,3,4$. $D^1(i,j,k) = A^1(i,j,k) - A^0(i,j)$ forms layer 1. Using three of these differences, we can reconstruct the fourth one since the 8x8 block contains the average of the four 4x4 blocks. This reconstruction is not perfect due to rounding error. The rounding error is small: MSE is 1.1 in our videos where the maximum pixel value is 255. This has minimum impact on the final video quality. Therefore, the layer 1 consists of $D^1(i,j,k)$ for $k = 1$ to 3 from all 8x8 blocks. We call $D^1(i,j,1)$ the first sublayer in layer 1, $D^1(i,j,2)$ the second sublayer, and $D^1(i,j,3)$ the third.

[1] If the typical throughput is even lower, one can construct 16x16 blocks and use average of these blocks to form layer 0.

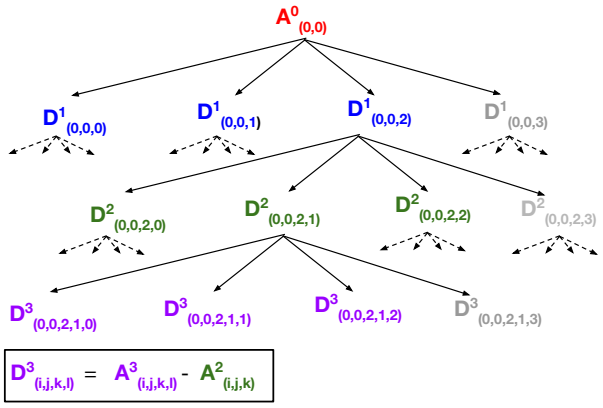


Figure 3: Our layered coding

Following the same principle, we form layer 2 by dividing the frame into 2x2 blocks and computing $A^2 - A^1$, and form layer 3 by dividing into 1x1 blocks and computing $A^3 - A^2$. As before, we omit the fourth value in layer 2 and layer 3 blocks. The complete layering structure of an 8x8 block is shown in Fig. 3. The values marked in grey are not sent and can be reconstructed at the receiver. This layering strategy gives us 1 average value for layer 0, 3 difference values for layer 1, 12 difference values for layer 2 and 48 difference values for layer 3, which gives us a total of 64 values per 8x8 block to be transmitted. Note that due to spatial locality, the difference values are likely to be small and can be represented using less than 8 bits. In YUV format, an 8x8 block's representation requires 96 bytes. Our strategy allows us to use fewer than 96 bytes.

Encoding: Difference values calculated in the layers 1 - 3 vary in magnitude. If we use the minimum number of bits to represent each difference value, the receiver needs to know that number. However, sending that number for every difference value defeats the purpose of compression and we will end up sending more data. To minimize the overhead, we use the same number of bits to represent the difference values that belong to the same layer of an 8x8 block. Furthermore, we group eight spatially co-located 8x8 blocks, referred to as *block-group* and use the same number of bits to represent their difference values for every layer. This serves two purposes:(i) It further reduces the overhead and (ii) the compressed data per *block-group* is always byte aligned. To understand (ii), consider if b bits are used to represent difference values for a layer in a *block-group*, $b * 8$ will always be byte-aligned. This enables more parallelism in GPU threads

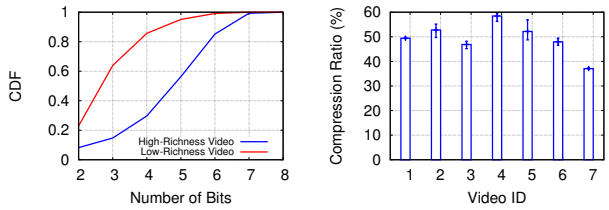
as explained in Sec. 3.2. Our meta data is less than 0.3% of the original data.

Decoding: Each pixel at the receiver side is constructed by combining the data from all the received layers corresponding to each block. So the pixel value at location (i, j, k, l, m), where $(i, j), k, l, m$ correspond to 8x8, 4x4, 2x2 and 1x1 block indices respectively, can be reconstructed as

$$A^0(i, j) + D^1(i, j, k) + D^2(i, j, k, l) + D^3(i, j, k, l, m)$$

If some differences are not received, they are assumed to be zero. When partial layers are received, we first construct the pixels for which we received more layers and then use them to interpolate the pixels with fewer layers based on the average of the larger block and the current blocks received so far.

Encoding Efficiency: We evaluate our layered coding efficiency using 7 video traces. We classify the videos into two categories based on the spatial correlation of the pixels in the videos. A video is considered *rich* if it has low spatial locality. Fig. 4(a) shows the distribution across *block-groups* for the minimum number of bits requires to encode difference values for all layers. For less rich videos, 4 bits are sufficient to encode more than 80% of difference values. For more detailed or rich videos, 6 bits are needed to encode 80% of difference values. Fig. 4(b) shows the average compression ratio for each video, where compression ratio is defined by the ratio between the encoded frame size and the original frame size. Error bars represent maximum and minimum compression ratio across all frames of a video. Our layering strategy can reduce the size of data to be transmitted by 40-62%. Less rich videos achieve higher compression ratio due to relatively smaller difference values.



(a) Difference bits (b) Compression ratio

Figure 4: Compression Efficiency

3.2 Layered Coding Implementation

Our layered coding encodes a frame as a combination of multiple independent blocks. This allows encoding of a full frame to be divided into multiple independent computations and makes it possible to leverage GPU. GPU consists of many cores with very simple control logic that can efficiently support thousands of threads to perform the same task on different inputs to improve throughput. In comparison, CPU consists of a few cores with complex control logic that can handle only a small number of threads running different tasks to minimize the latency of a single task. Thus, to minimize our encoding and decoding latency, we implement our schemes using GPU. However, achieving maximum efficiency over GPU is not straightforward. To maximize the efficiency of GPU implementation, we address several significant challenges.

GPU threads synchronization: An efficient GPU implementation should have minimal synchronization between different threads for independent tasks. Synchronization requires data sharing, which greatly impacts the performance. We divide computation into multiple independently computable blocks. However, blocks still have to synchronize since blocks vary in size and have variable writing offsets in memory. We design our scheme such that we minimize the synchronization need among the threads by putting the compression of a *block-group* in a single thread.

Memory copying overhead: We transmit encoded data as sublayers. The receiver has to copy the received data packets of sublayers from CPU to GPU memory. A single copy operation has non-negligible overhead. Copying packets individually incurs too much overhead, while delaying copying packets until all packets are received incurs significant startup delay for the decoding process. We design a mechanism to determine when to copy packets for each sublayer. We manage a buffer to cache sublayers residing in GPU memory. Because copying a sublayer takes time, GPU is likely to be idle if the buffer is small. Therefore, we try to copy packets of a new sublayer while we are decoding a sublayer in the buffer. Our mechanism can reduce GPU idle time by parallelizing copying new sublayers and decoding buffered sublayers.

Our implementation works for a wide range of GPUs, including low to mid-range GPUs [2], which are common on commodity devices.

3.2.1 GPU Implementation. **GPU background:** To maximize the efficiency of GPU implementation, we follow the two guidelines below: (i) A good GPU implementation should divide the job into many small independently computable tasks. (ii) It should minimize memory synchronization across threads as memory operations can be expensive.

Memory optimization is critical to the performance of GPU implementation. GPU memory can be classified into three types: *Global*, *Shared* and *Register*. Global memory is standard off-chip memory accessible to GPU threads through bus interface. Shared memory and register are located on-chip, so

their access time is ~100× faster than global memory. However, they are very small. But they give us an opportunity to optimize performance based on the type of computation.

In GPU terms, a function that is parallelized among multiple GPU threads is called *kernel*. If different threads in a kernel access contiguous memory, global memory access for different threads can be coalesced such that memory for all threads can be accessed in a single read instead of individual reads. This can greatly speed up memory access and enhance the performance. Multiple threads can work together using the shared memory, which is much faster than the global memory. So the threads with dependency can first read and write to the shared memory, thereby speeding up the computation. Ultimately, the results are pushed to the global memory.

Implementation Overview: To maximize the efficiency of GPU implementation, we divide the encoder and decoder into many small independently computable tasks. In order to achieve independence across threads, we should satisfy the following two properties:

No write byte overlap: If no two threads write to the same byte, there is no need for memory synchronization among threads. This is a desired property, as thread synchronization in GPU is expensive. Since layers 1-3 may use fewer than 8 bits to represent the differences, a single thread processing a 8x8 block may generate output that is not a multiple of 8-bits. In this case, different threads may write to the same byte and require memory synchronization. Instead, we use one thread to encode difference values per *block-group*, where a *block-group* consists of 8 spatially co-located blocks. Since all blocks in the *block-group* use the same number of bits to represent difference values, the output size from *block-group* is an multiple of 8 and always byte aligned.

Read/write independence among blocks: Each GPU thread should know in advance where to read the data from or where to write the data to. As the number of bits used to represent the difference values vary across *block-groups*, its writing offset depends on how many bits were used by the previous *block-groups*. Similarly, the decoder should know the read offsets. Before spawning GPU threads for encoding or decoding, we derive the read/write memory offsets for that thread using cumulative sum of the number of bits used per *block-group*.

3.2.2 Jigsaw GPU Encoder. **Overview:** Encoder consists of three major tasks: (i) calculating averages and differences, (ii) constructing meta-data, and (iii) compressing differences. Specifically, consider a YUV video frame of dimensions $a \times b$. The encoder first spawns $\frac{ab}{64}$ threads for Y values and $\frac{0.5ab}{64}$ for UV values. Each thread operates on an 8x8 block to (i) compute the average of an 8x8 block, (ii) compute the averages of all 4x4 blocks within the 8x8 block and take the difference between the averages of 8x8 block and 4x4 block, and similarly compute the hierarchical differences for 2x2 blocks and 1x1 blocks, and (iii) coordinate with the other threads in its *block-group* using the shared memory to get the minimum number of bits required to represent difference values for its *block-group*. Next the encoder derives the meta-data and memory offsets. Then it spawns $\frac{x*y*1.5}{64*8}$ threads, each of which compresses the difference values for every 8 blocks.

Calculating averages and differences: The encoder spawns multiple threads to compute averages and differences. Each thread processes an 8x8 block. It calculates all the averages and difference values required to generate all layers for that block. It reads eight 8-byte chunks for a block one by one to compute the average. In GPU architecture, multiple threads of the same kernel execute the same instruction for a given cycle. In our implementation, successive threads work on contiguous 8x8 blocks. This makes successive threads access and operate on contiguous memory, so the global memory reads are *coalesced* and memory access is optimized.

After computing the average at each block level, each thread computes the differences for each layer and writes it to an output buffer. The thread also keeps track of the minimum number of bits required to represent the differences for each layer. Let b^i denote the number of bits required for layer i. All 8 threads for a *block-group* use atomic operations in a *shared memory* location to get the number of bits required to represent differences for that *block-group*. We use B^i_j to denote the number of bits used by *block-group* j for layer i.

Meta data processing: We compute the memory offset where the compressed value for the i^{th} layer of *block-group* j should be written to using a cumulative sum $C^i_j = \sum_{k=0}^{k=j} B^i_k$. Based on the cumulative sum, the encoding kernel generates multiple threads to process the difference values concurrently without write byte overlap. B^i_j values are transmitted, based on which the receiver can compute the memory offsets of compressed difference values. 3 values per *block-group* per layer are transmitted except the base layer and we need 4 bits to represent one meta-data value. Therefore a total of $\frac{3*x*y*1.5}{64*8*2}$ bytes are sent as meta-data, which is within 0.3% of the original data.

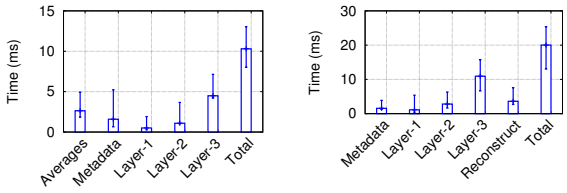
Encoding: A thread is responsible for encoding all the difference values for a *block-group* in a layer. A thread for *block-group* j uses B^i_j to compress and combine the difference values for layer i from all its blocks. It then writes the compressed value in an output buffer at the given memory offset. Our design is to ensure consecutive threads read and write to contiguous memory locations in the global GPU memory to take advantage of *memory coalescing*. Fig. 5(a) shows average running time of each step in our compression algorithm. On

average, all steps finish around 10ms. Error bars represent the maximum and minimum running times across different runs.

3.2.3 Jigsaw GPU Decoder. **Overview:** The receiver first receives the layer 0 and meta-data. It then processes the meta-data to compute the read offsets. Each layer except the base layer has 3 sublayers as described in Section 3.1. Once all data corresponding to a sublayer are received, a thread is spawned to decode the sublayer and add the difference value to the previously reconstructed lower level average. This process is repeated for every sublayer that is received completely.

The decoder consists of the following steps: (i) processing meta-data, (ii) decoding, and (iii) reconstructing a frame.

Meta-data processing: Meta-data contains the number of bits used for difference values per layer per *block group*. The kernel calculates the cumulative sum C_j^i from the meta-data values similarly to the encoder described previously. This cumulative sum indicates the read offset at which *block-group* j can read the compressed difference values for any sublayer in layer i. Since all sublayers in a layer are generated using the same number of bits, their relative offset is the same.



(a) Encoder Modules (b) Decoder Modules

Figure 5: Codec GPU Modules Running Time

Decoding: A decoding thread decodes all the difference values for a *block-group* corresponding to a sublayer. This thread is responsible for decompressing and adding the difference value to the previously reconstructed lower level average. This process is repeated for every sublayer that is received completely. Each block is composed of 4 smaller sub-blocks. Difference values for three of these sub-blocks are transmitted. Once sublayers corresponding to the three sub-blocks of the same block are received, the fourth sublayer is constructed based on the average principle.

Received sublayers reside in the main memory and should be copied to GPU memory before they can be decoded. Each memory copy incurs overhead so copying every packet individually to GPU has significant overhead. On the other hand, memory copy for a sublayer cannot be deferred to the point at which its decoding kernel is scheduled. Otherwise, it would stall the kernel till the sublayer is completely copied to GPU memory.

We implement a **smart delayed copy** mechanism. It parallelizes memory copy for one sub-layer with the decoding of another sublayer. We always keep a maximum of 4 sublayers in GPU memory. They are next in line to decode. As soon as one of these 4 sublayers is scheduled to be decoded in GPU, we choose a new complete sublayer to copy from CPU to GPU memory. If no new complete sublayer is available, we copy a partial sublayer to GPU. In the latter case, all future incoming packets for that sublayer are directly copied to GPU without any delay. Our smart delayed copy mechanism allows us to reduce the memory copy time between GPU and main memory by 4x at the cost of only 1% of the kernels experiencing 0.15ms stall on average due to delayed memory copying. The total running time of the decoding process of a sublayer consists of the memory copy time and the kernel running time. Because of the large memory copy time, our mechanism significantly reduces the total running time.

Frame reconstruction: Once the deadline for a frame is imminent, we stop decoding its sublayers and prepare for reconstruction. We interpolate for the partial sublayers as explained in Section 3.1. The receiver reconstructs a pixel based on all the received sublayers. Finally, reconstructed pixels are organized as a frame and rendered on the screen.

Fig. 5(b) shows the average running time of each step in our decompression algorithm, where the error bars represent the maximum and minimum values. On average, the total running time is around 19ms.

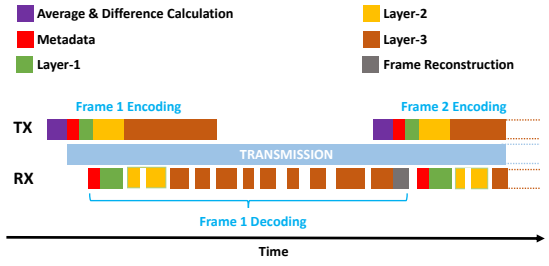


Figure 6: Jigsaw Pipeline

3.2.4 Pipelining. A 4K video frame takes significant amount of time to transmit. Starting transmission after the whole frame finishes encoding increases delay; similarly, decoding after receiving a whole frame is also inefficient. Pipelining can be potentially used to reduce the delay. However, not all encoders and decoders can be pipelined. Many encoders have no intermediate point where transmission can be pipelined. Due to data dependency, it is not possible to start decoding before all dependent data has been received.

Our layering strategy has nice properties that (i) the lower layers are independent of the higher layers and can be encoded or decoded independently and (ii) sublayers within a layer can be independently computed. This means our encoding can be pipelined with data transmission as shown in Fig. 6. Transmission can start as soon as the average and difference calculation is done, which can happen as soon as 2ms after the frame is generated. This is possible since the base layer consists of only average values and does not require any additional computation. Subsequent sublayers are encoded in order and scheduled for transmission as soon as any sublayer is encoded. Average encoding time for a single sublayer is 200-300us.

A sublayer can be decoded once all its lower layers have been received. As the data are sent in order, by the time a sublayer arrives, its dependent data are already available. So a sublayer is ready to be decoded as soon as it is received. As shown in Fig. 6, we pipeline sublayer decoding with data reception. Final frame reconstruction cannot happen until all the data of the frame have been received and decoded. As the frame reconstruction takes around 3-4ms, we stop receiving data for that frame before the frame playout deadline minus the reconstruction time. Pipelining reduces the delay from 10ms to 2ms at the sender, and reduces the delay from 20ms to 3ms at the receiver.

3.3 Video Transmission

We implement a driver module on top of the standard network interface bonding driver in Linux that bonds WiFi and WiGig interfaces. It is responsible for the following important tasks: (i) adapting video quality based on the wireless throughput, (ii) selecting the appropriate interface to transmit the packets (*i.e.*, intra video frame scheduling), and (iii) determining how much data to send for each video frame before switching to the next one (*i.e.*, inter video frame scheduling). Our high-level approach is to defer the decision till transmission to minimize the impact of prediction errors. Below we describe these components in detail.

Delayed video rate adaptation: A natural approach to transmit data is to let the application decide how much data to generate and pass it to the lower network layers based on its throughput estimate. This is widely used in the existing video rate adaptation. However, due to unpredictable throughput fluctuation, it is easy to generate too much or too little data. This issue is exacerbated by the fact that the 60 GHz link throughput can fluctuate widely and rapidly.

To fully utilize network resources with minimized effect on user quality, we delay the transmission decision as late as possible to remove the need for throughput prediction at the application layer. Specifically, for each video frame we let the application generate all layers and pass them to the driver in the order of layers 0, 1, 2, 3. The driver will transmit as much data as the network interface card allows before the sender switches to the next video frame.

While the concept is intuitive, realizing it involves significant effort as detailed below. First, our module determines whether a packet belongs to a video stream based on the destination port. Instead of forwarding all video packets directly to the interfaces, they are added to a large circular buffer. This allows us to make decisions based on any optimization objective. Packets are dequeued from this buffer and added into the transmission queue of an interface whenever a transmission decision is made. Moreover, to minimize throughput wastage, this module drops the packets that are past their deadline. The module uses the header information attached to each video packet to determine which video frame the packet belongs to and its deadline. Before forwarding the packets to the interface, the module estimates whether that packet can be delivered within the deadline based on current throughput estimate and queue length of the interfaces. The interface queue depletion rate is used to estimate the achievable throughput. If a packet cannot make the deadline, all the remaining packets belonging to the same video frame are marked for deletion since they have the same deadline. The buffer head is moved to the start of next frame. Packets marked for deletion are removed from the socket buffer in a separate low priority thread since the highest priority is to forward packets to interfaces to maintain throughput.

Intra-frame scheduling: Interface firmware beneath the wireless driver is responsible for forwarding packets. Transmission is out of control of the driver once the packet is put to interface queue of an interface. This means that the packet cannot be dropped even if it misses its deadline. To minimize such waste, we only enqueue a minimum number of packets to both interfaces to sustain the maximum throughput. The interface card notifies the driver whenever a packet has been successfully received, and the driver removes it from that interface queue automatically.

Our module keeps track of the queue lengths for both WiFi and WiGig interfaces. As soon as the queue size of any interface goes below the threshold, it forwards packets from the packet buffer to that interface. As shown in Section 2, both interfaces have frequent inactivity intervals during which they are unable to transmit any packets. This can result in significant delay in the packets sent to the inactive interface. Such delay can sometimes cause the lower layer packets to get stuck at one interface queue and the higher layer packets to be transmitted at the other interfaces. Since our layered encoding requires the lower layer packets to be received in order to decode the higher layer packets, it is important to ensure the lower layer packets to be transmitted before the higher layer.

To achieve this goal, the sender monitors the queue on both interfaces. If no packet is removed from that queue for T duration, it declares that interface is inactive. When the queue of the active interface reduces, which means that it is now sending packets and is ready to accept more packets, we compare the layer of the new packet with that of the packet queued at the inactive interface. If the latter is a lower layer (which means it is required in order to decode the higher layer packets), we move the packet from the inactive interface to the active interface. No rescheduling is done for the packets in the same layer since these packets have the same priority. In this way, we ensure that all the lower layer packets are received prior to the reception of higher layer packets. T is adapted dynamically based on the current frame's deadline. If the deadline is more than 10 ms away, we set $T = 4ms$, otherwise it is set to $2ms$. This is because we need to react more quickly if the deadline is closer so that all the enqueued packets can be transmitted on the active interface and received before the deadline.

Inter-frame scheduling: If the playout deadline for a frame is larger than the frame generation interval, the module can have packets from different video frames at a given instant. For example, in a 30 FPS video, a new frame is generated every 33 msec. If the deadline to play a frame is set to 60 ms from its generation as suggested in recent studies [11], two consecutive frames will overlap for 27 ms.

In this case, we have an opportunity to decide when to start transmitting the next video frame. Certainly, no packets should be transmitted after their deadline. However, we sometimes need to move on to transmit the next video frame before the deadline of the previous frame arrives, to ensure similar numbers of packets are transmitted for two consecutive frames and there is no significant variation in the quality of consecutive frames.

To achieve this, whenever the interface queue has room to accept a new packet, our module predicts the number of packets that can be transmitted for the next frame. If the estimate is smaller than the number of packets already sent for the current frame minus a threshold (80 packets in our system), we switch to transmitting the next video frame. This reduces variation in the quality of the two consecutive video frames. Note that a frame is never preempted before its base layer finishes transmission (unless it passes the deadline) to ensure at least the lowest resolution frame is fully transmitted.

4 EVALUATION

In this section, we first describe our evaluation methodology. Then we perform micro-benchmarks to understand the impact of our design choices. Finally, we compare *Jigsaw* with the existing approaches.

4.1 Evaluation Methodology

Evaluation methods: We classify our experiments into two categories: *Real-Experiment* and *Emulation*. Experiments are done on two laptops equipped with QCA9008 AC+AD WiFi module that supports 802.11ad on 60 GHz and 802.11ac on 2.4/5 GHz band. Each laptop has 2.4GHz dual-core processor, 8 GB RAM and NVIDIA Geforce 940M GPU. One laptop serves as the sender and the other serves as the receiver. The sender encodes the data while the receiver decodes the data and displays the video.

While real experiments are valuable, the network may vary significantly when we run different schemes even for the same movement pattern. This makes it hard to compare different schemes. Emulation allows us to run different schemes using exactly the same throughput trace. For emulation, we connect two desktops, each with 3.6GHz quad-core processor, 8 GB RAM, 256GB SSD and NVIDIA Geforce GTX 1050 GPU, a 10 Gbps fiber optic link. We collect packet traces over WiFi and WiGig using our laptops and use these traces to run trace-driven emulation. The sender and receiver code run on two desktops in real time. We emulate two interfaces, and delay the packets according to the packet trace from WiFi and WiGig before sending them over the 10 Gbps fiber optic link. We verify the correctness of our emulation by comparing the instantaneous throughput between the real traces and the emulated experiment, and find that the emulated throughput is within 0.5% of the real trace's throughput.

Video Traces: We use 7 uncompressed videos with YUV420 format from Derf's collection under Xiph [7] with resolution 4096x2160. We choose videos with different motion and spatial locality to evaluate their impact. Videos are streamed at 30 FPS and each video is concatenated to itself to generate desired duration. We classify the videos into two categories based on spatial correlation of pixels. We use variance in the Y values in 8x8 blocks for all video frames of a video to quantify spatial correlation a video. Videos that have high variance are classified as high richness (HR) videos and video with low variance are classified as low richness (LR) videos. We use 2 HR videos and 5 LR videos.

Mobility Patterns: We collect traces by fixing the sender and moving the receiver. We use the following movement patterns in our experiment. (i) Static: No movement in sender or receiver; (ii) Watching a 360 video: A user watches a 360^o video, and rotating the receiver laptop up and down, right and left according to the video displayed on the laptop and the static sender is around 1.5m away; (iii) Walking: The user walks around with the receiver laptop in hand within a 3m radius from the sender, (iv) Blockage: Sender and receiver are static, but another user moves between them and may block the link from time to time thus inducing environment mobility.

For our real experiments, we run each experiment 5 times for each mobility pattern and then average the results. For emulation, we collect 5 different packet level traces for each mobility pattern.

Performance metrics: We use Peak Signal to Noise Ratio (PSNR) and Structural Similarity Index (SSIM) to quantify the video quality. PSNR is a widely used video quality metric. Videos with PSNR greater than 45 dB are considered to have excellent quality, between 33-45 are considered good and between 27-33 are considered fair. 1 dB difference in PSNR is already visible and 3 dB difference indicates that the video quality is doubled. Videos with SSIM greater than 0.99 are considered to have excellent quality, 0.95-0.99 are considered good, and 0.88-0.95 are considered fair [29].

In all our results, the error bars represent maximum and minimum values unless otherwise specified.

4.2 Micro-benchmarks

We use emulation to quantify the impact of different system components since we can keep the network condition identical.

Impact of layers on video quality: Our layered coding allows us to exploit spatial correlation. It can not only adapt the video quality on the fly but also reduce the amount of data to transmit. We quantify the compression rate for different types of videos. Fig. 7 shows the frame PSNR as we vary the number of layers. For the LR videos, receiving only 2 layers can achieve an average PSNR close to 40dB and receiving only 3 layers gives PSNR of 42 dB. For the HR videos, the average PSNR is around 7dB lower than the LR video when receiving 3 layers. Our scheme can achieve similar PSNR values for both kind of videos when all the layers are received. Moreover, as we would expect, less rich videos have higher compression ratios due to smaller difference values, which can be represented using fewer bits.

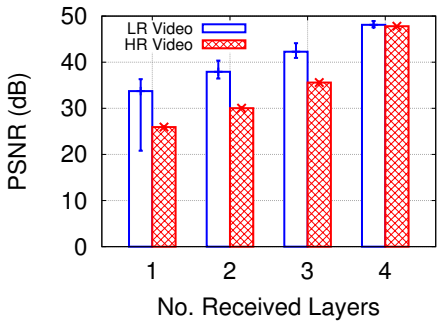


Figure 7: Video quality vs no. of received layers.

Impact of using WiGig and WiFi: Fig. 8 shows the PSNR per frame when using WiGig only. We use the throughput trace collected when a user is watching 360^o videos. Without WiFi, PSNR drops below 10dB when disconnection happens. When WiGig has drastic changes, PSNR can decrease by more than 10dB even if WiGig is not completely disconnected. WiFi improves PSNR by over 25dB when WiGig is disconnected, and by 2dB even when WiGig is not disconnected. The disconnection of WiGig can result in partial blank frames because the available throughput may not be able to transmit the whole layer 0 in time. The complementary throughput from WiFi can remove partial blank frames effectively, so we observe large improvement in PSNR when WiGig disconnection happens. We can transmit more layers when using both WiGig and WiFi because of higher total available throughput. Thus, WiFi still improves PSNR even if WiGig is not disconnected.

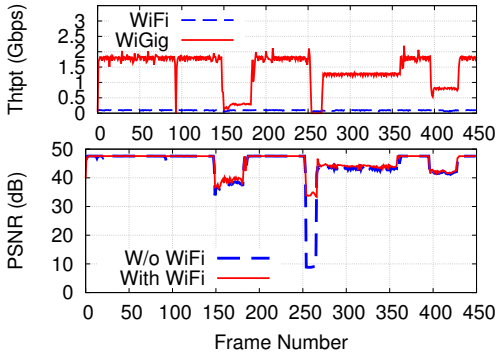
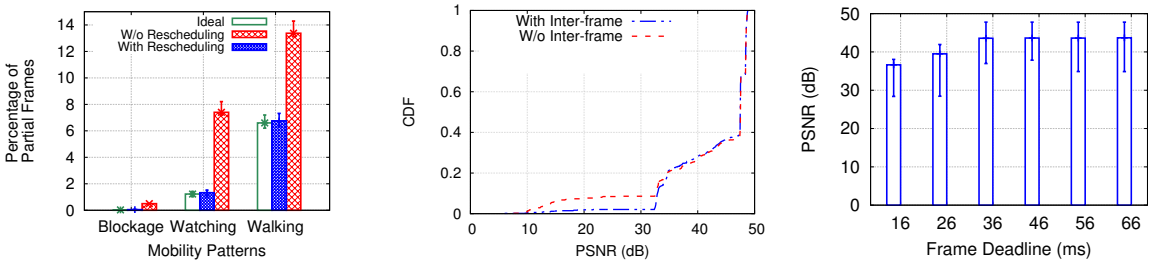


Figure 8: Impacts of using WiGig and WiFi.

Impact of interface scheduler: When the WiGig throughput reduces, data at the WiGig interface gets stuck. Such situation is especially bad when the data packets queued at WiGig are from the lower layer than those queued at WiFi. In this case, even if WiFi successfully delivers the data, they are not decodable without the complete lower layer data. To avoid this situation, our scheduler can effectively move the data from the inactive interface to the active interface whenever the inactive interface has lower layer data. When Layer-0 data get stuck, the user will see part of the received frame in blank. Fig. 9(a) shows the percentage of partial frames that do not have complete Layer-0 under various mobility patterns. Our scheduler reduces the percentage of partial frames by 90%, 82% and 49% for watching, walking, and blockage traces, respectively. In the static case, we do not observe any partial frame. Our scheme gives a partially blank frame only when the throughput is below the minimum required – 50 Mbps. As shown in Fig. 9(a) the number of partially blank frames for our scheme are within 0.1% of what we receive in ideal case for these traces.

Impact of Inter-frame Scheduler: When throughput fluctuates, the video quality can vary significantly across frames.



(a) Impacts of interface scheduler (b) Impacts of inter-frame scheduler (c) Impacts of frame deadline

Figure 9: Microbenchmarks

Our inter-frame scheduler tries to balance throughput allocated to consecutive frames. Without this scheduling, we just send frame data until its deadline. As shown in Fig. 9(b), this improves the quality of frames around 10dB when the throughput is low. The impact is minimal when the throughput is already high and stable.

Impact of GPU: Next we study the impact of GPU on the encoding and decoding time. Table 2 shows the performance of *Jigsaw* for three types of GPUs with different number of cores. More powerful GPU has more cores and reduces the encoding and decoding time significantly, leaving more time to transmit data. However, this comes at the cost of high power consumption. Even the low to mid-end GPUs that are generally available on mobile devices can successfully encode and decode the data in real time.

GPU Cores	Power Rating (W)	Encoding Time (ms)	Decoding Time (ms)
384	30	10.1	19.3
768	75	1.7	5.7
2816	250	1.1	5.3

Table 2: Performance over different GPUs

Impact of Frame Deadline: Frame deadline is determined by the desired user experience of the application. Fig. 9(c) shows the video PSNR when varying frame deadline from 16 ms to 66 ms. The performance of *Jigsaw* using 36 ms deadline is similar to that using 66 ms deadline. 36 ms is much lower than the 60 ms delay threshold for interactive mobile applications [11, 14, 21]. We use 60ms as our frame deadline threshold because this is the threshold that is deemed tolerable by the earlier works [11, 14, 21]. However, as we show that we can tolerate frame delay as low as 36ms, so even if future applications require lower delay, our system can still support.

4.3 System Results

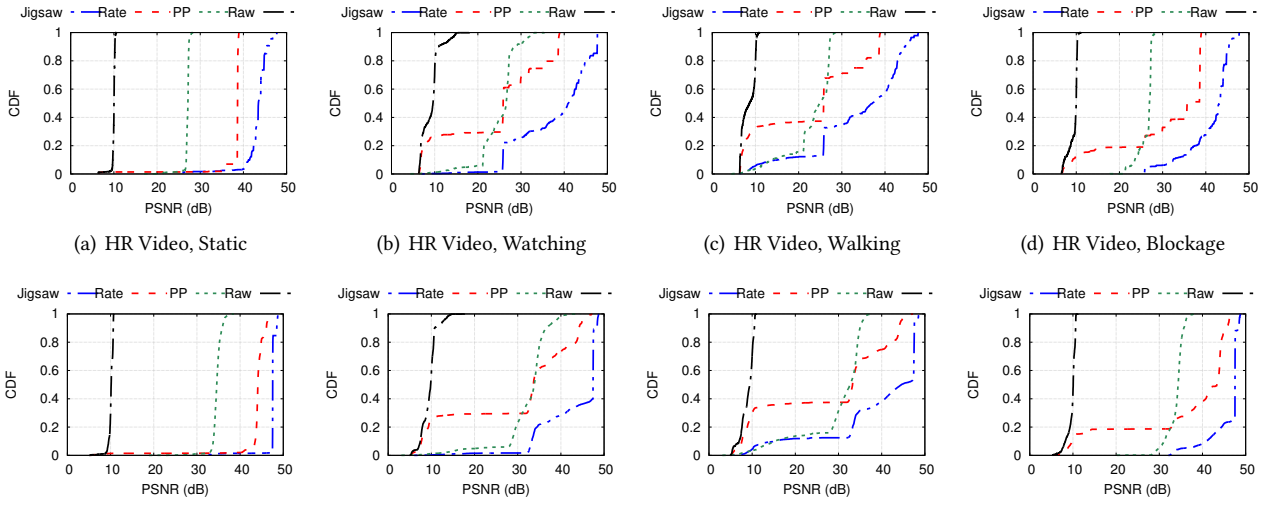
We compare our system with the following three baseline schemes using real experiments as well as emulation.

- **Pixel Partitioning(PP):** Video is divided into 8x8 blocks. The first pixels from all blocks are transmitted, followed by the second pixels in all blocks, and so on. If not all pixels are received for a block, the remaining ones are replaced by average of the remaining pixels, which can be computed based on the average of the larger block and the current blocks received so far.
- **Raw:** Raw video frame is transmitted. This is uncompressed video streaming. Packets after their deadline are dropped to avoid wastage.
- **Rate adaptation (Rate):** Throughput is estimated using historical information. Based on this estimate, the uncompressed video is down-sampled accordingly before transmission. The receiver up-samples it to 4K and displays it. This is similar in concept to DASH streaming, which is the current standard of video-on-demand streaming.

Benefits of *Jigsaw*: Fig. 10 shows the video quality for all schemes under various mobility patterns and videos. We perform real experiments by running each video 5 times over each mobility traces and report the average. We make the following observations.

First, *Jigsaw* achieves much higher PSNR than the other schemes across all mobility scenarios and videos. *Raw* always has partial-blank frames and has a very low PSNR of around 10dB in all settings as the throughput can not support full 4K frame transmission even in the best case. *PP* also transmits complete 4K frame, so it is never able to transmit a complete frame in any setting, however the impact of not receiving all data is not as severe as *Raw*. This is because some data from different parts of the frame is received, and frames are no longer partially blank. Moreover, using interpolation to estimate the unreceived pixels also improves quality. *Rate* achieves high PSNR in static scenarios since the throughput prediction is more accurate and it can adapt the resolution. However, it cannot transmit at full 4K resolution.

Second, the benefit of *Jigsaw* is higher under mobility scenarios. *Jigsaw* improves the median PSNR by up to 6 dB over *Rate*, 12 dB over *PP* and 35 dB over *Raw* in static settings.



(a) HR Video, Static (b) HR Video, Watching (c) HR Video, Walking (d) HR Video, Blockage

(e) LR Video, Static (f) LR Video, Watching (g) LR Video, Walking (h) LR Video, Blockage

Figure 10: Performance of *Jigsaw* under various mobility patterns. (HR: High-Richness, LR: Low-Richness)

The corresponding improvement is 15 dB over *Rate*, 16 dB over *PP* and 38 dB over *Raw* because *Jigsaw* can quickly adapt to the throughput changes in the mobile case due to its layered coding design, delayed video rate adaptation, and smart scheduling.

Third, for all schemes HR videos suffer more when less data can be transmitted. All schemes achieve higher PSNR for the LR videos. The median PSNR for the LR (HR) videos is 46dB (41dB), 38dB (31dB), 33dB (26dB) and 10dB (9dB) for *Jigsaw*, *Rate*, *PP* and *Raw*, respectively. These results show that *Jigsaw* significantly out-performs the existing approaches for a variety of network conditions and videos.

Table. 3 shows the median video SSIM for all schemes. We can observe similar trend as PSNR. *Jigsaw* achieves the highest SSIM. *Jigsaw* can achieve at least good video quality for all videos and mobility traces.

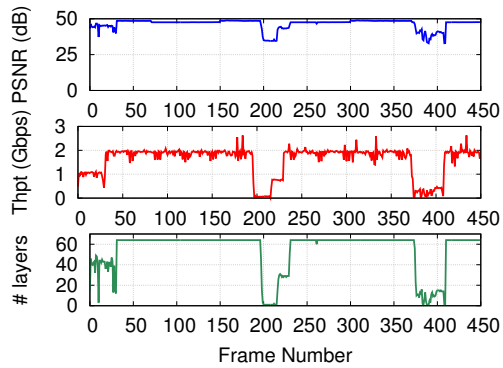


Figure 11: Frame quality correlation with throughput.

Effectiveness of layer adaptation: Fig. 11 shows the correlation between throughput and video frame quality for *Jigsaw*. We can see that the changes of video quality has very similar pattern as the throughput changes. *Jigsaw* only receives Layer-0 and partial Layer-1 when throughput is close to 0. In those cases, the frame quality drops to around 30dB. When the throughput stays close to 1.8Gbps, the frame quality reaches around 49dB. Because we keep our interface queues small, our packet transmission rate closely follows the packet depletion rate from the interface queues. Hence, our layer adaptation can quickly respond to any throughput fluctuation.

	Jigsaw	Rate	PP	Raw
HR,Static	0.993	0.978	0.838	0.575
HR,Watching	0.965	0.749	0.818	0.489
HR,Walking	0.957	0.719	0.805	0.421
HR,Blockage	0.971	0.853	0.811	0.511
LR,Static	0.996	0.985	0.949	0.584
LR,Watching	0.971	0.785	0.897	0.559
LR,Walking	0.965	0.748	0.903	0.481
LR,Blockage	0.984	0.862	0.907	0.560

Table 3: Video SSIM under various mobility patterns.

4.4 Emulation Results

In this section, we compare time series of *Jigsaw* with the other schemes when using the same throughput trace. We use emulation to compare different schemes under the same condition. Fig. 12 shows the quality of each frame using an example throughput trace collected from the *Watching* mobility pattern. As we can see, *Jigsaw* achieves the highest quality for frames and least variation.

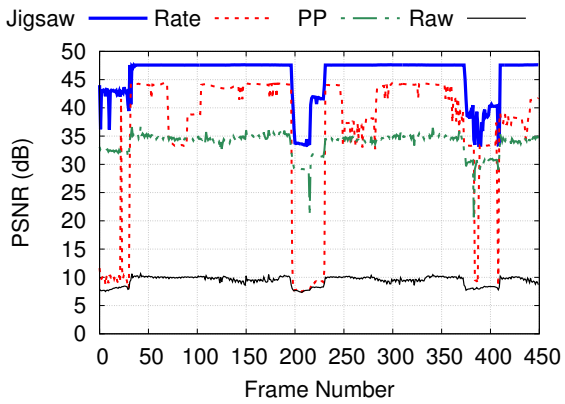


Figure 12: Frame quality comparison

5 RELATED WORK

Video Streaming over Internet: DASH [36] is the current standard protocol for Internet video streaming services. A video is divided into chunks, each of which contains a few seconds video content. The client adapts the bitrate of video chunks according to the network condition. Significant work has been done on understanding and improving the performance of video rate adaptation algorithms [18, 20, 22, 28, 43]. However, most of them only investigate video-on-demand (VoD) services in which entire videos are encoded and stored at the server side before users start downloading. Compared with VoD, live video streaming [13, 30, 42] is more delay sensitive. Existing live video streaming approaches stream video content in chunks, and users have to wait for a few seconds before a video chunk is ready to play. In this work, we focus on supporting live video streaming in which the delay of a video frame is as small as only tens of milliseconds. Such live streaming is crucial for interactive applications, like gaming, virtual reality and augmented reality [11, 21].

Video Streaming over non-60GHz Wireless Networks: Wireless links are not reliable and have large fluctuation in available throughput. Flexcast [9] incorporates rateless code into the existing video codec such that video quality does not drop drastically when network condition varies. Softcast [19] exploits 3D DCT to achieve similar target. Parcast [27] investigates how to transmit more important video data to more reliable wireless channels. PiStream [41] uses physical layer information from LTE links to enhance video rate adaptation. However, these works do not study 4K video streaming. Due to large size, existing video coding cannot code 4K video content in real time. Furion [23] tries to use multiple codec instances to encode and decode VR video content in parallel, but it supports a much lower resolution than the 4K resolution. Rubiks [16] designs a layered coding scheme to minimize bandwidth for 360-degree video streaming. However, it does not consider 4K video encoding cost, and hence cannot support 4K live video. Some dedicated hardware can support 4K videos in real time [25], but it is generally not available on laptops and mobile devices. Our work focuses on supporting live 4K video encoding, transmission, and decoding in real time on commodity devices.

Video Streaming over WiGig: Owing to the large bandwidth in WiGig, streaming uncompressed video has become a key application for WiGig [35, 39]. Choi et al [12] proposes a link adaptation policy to allocate different amount of resource to different data bits in a pixel such that the total amount of allocated resource is minimized while maintaining good video quality. He et al [17] try to encode an uncompressed video into multiple descriptions using RS coding. The video quality improves as more descriptions are received. [12, 17] use unequal error protection to protect different bits in a pixel based on importance. Shao et al [33] compresses pixel difference values using run length coding, which is difficult to parallize since run length codes since different pixels can not be known beforehand. Singh et al [34] propose to partition adjacent pixels into different packets and adapt the number of pixels to send based on throughput estimation. But its video quality degrades significantly when throughput drops since it has no compression. Li et al [24] develops an interface that notifies the application layer of the parallelize between WiGig and WiFi so that the video server can determine an appropriate resolution of the video to send. In general, these works do not consider encoding time, which can be significant for 4K videos. They also do not explore how to leverage multiple links for video streaming, which our system addresses.

Other wireless approaches: Developing other high-bandwidth reliable wireless networks is also an interesting solution to support 4K video streaming. A recent router [6] claims to support Gbps-level throughput by exploiting one 2.4GHz and two 5GHz bands. However, it still cannot support raw 4K video transmission. Our system can efficiently fit 4K video into this throughput range.

6 CONCLUSION

In this paper, we study the feasibility of streaming live 4K videos and identify the major challenge in realizing this goal is to tolerate large and unpredictable throughput fluctuation. To address this challenge, we develop a novel system to stream live 4K videos using commodity devices. Our design consists of a new layered video coding, an efficient implementation using GPU and pipelining, and implicit video rate adaptation and smart scheduling data across WiFi and WiGig without requiring throughput prediction. Moving forward, we are interested in exploring applications of live 4K video streaming, such as VR, AR, and gaming.

REFERENCES

[1] ffmpeg. https://www.ffmpeg.org/.

[2] Nvidia geforce 940m. https://www.notebookcheck.net/NVIDIA-GeForce-940M.138027.0.html.

[3] Nvidia titan x specs. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/.

[4] Nvidia video codec. https://developer.nvidia.com/nvidia-video-codec-sdk.

[5] Openh264. https://www.openh264.org/.

[6] Tp-link archer c5400x mu-mimo tri-band gaming router. https://venturebeat.com/2018/09/06/tp-link-launches-gaming-router-for-4k-video-stream-era/.

[7] Video dataset. url="https://media.xiph.org/video/derf/.

[8] Youtube 4k bitrates. https://support.google.com/youtube/answer/1722171?hl=en.

[9] S. Aditya and S. Katti. Flexcast: Graceful wireless video streaming. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 277–288. ACM, 2011.

[10] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. Vp8 data format and decoding guide. RFC 6386, Google Inc., November 2011.

[11] C.-M. Chang, C.-H. Hsu, C.-F. Hsu, and K.-T. Chen. Performance measurements of virtual reality systems: Quantifying the timing and positioning accuracy. In *Proceedings of the 2016 ACM on Multimedia Conference*, pages 655–659. ACM, 2016.

[12] M. Choi, G. Lee, S. Jin, J. Koo, B. Kim, and S. Choi. Link adaptation for high-quality uncompressed video streaming in 60-ghz wireless networks. *IEEE Transactions on Multimedia*, 18(4):627–642, 2016.

[13] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback control for adaptive live video streaming. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 145–156. ACM, 2011.

[14] J. Deber, R. Jota, C. Forlines, and D. Wigdor. How much faster is fast enough?: User perception of latency & latency improvements in direct and indirect touch. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1827–1836. ACM, 2015.

[15] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, 2017. USENIX Association.

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

[17] Z. He and S. Mao. Multiple description coding for uncompressed video streaming over 60ghz networks. In *Proceedings of the 1st ACM workshop on Cognitive radio architectures for broadband*, pages 61–68. ACM, 2013.

[18] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *ACM SIGCOMM Computer Communication Review*, 44(4):187–198, 2015.

[19] S. Jakubczak and D. Katabi. A cross-layer design for scalable mobile video. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 289–300. ACM, 2011.

[20] J. Jiang, V. Sekar, H. Milner, D. Shepherd, I. Stoica, and H. Zhang. Cfa: A practical prediction system for video qoe optimization. In *NSDI*, pages 137–150, 2016.

[21] T. Kämäräinen, M. Siekkinen, A. Ylä-Jääski, W. Zhang, and P. Hui. Dissecting the end-to-end latency of interactive mobile video applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, pages 61–66. ACM, 2017.

[22] R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of tcp-based rate-control algorithms for adaptive internet streaming of h. 264/svc. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 157–168. ACM, 2010.

[23] 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 *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 409–421. ACM, 2017.

[24] Y.-Y. Li, C.-Y. Li, W.-H. Chen, C.-J. Yeh, and K. Wang. Enabling seamless wigig/wifi handovers in tri-band wireless systems. In *Network Protocols (ICNP), 2017 IEEE 25th International Conference on*, pages 1–2. IEEE, 2017.

[25] 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 *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 68–80. ACM, 2018.

[26] X. Liu, Q. Xiao, V. Gopalakrishnan, B. Han, F. Qian, and M. Varvello. 360° innovations for panoramic video streaming. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets-XVI, pages 50–56, New York, NY, USA, 2017. ACM.

[27] X. L. Liu, W. Hu, Q. Pu, F. Wu, and Y. Zhang. Parcast: Soft video delivery in mimo-ofdm wlans. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, pages 233–244. ACM, 2012.

[28] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[29] A. Moldovan and C. H. Muntean. Qoe-aware video resolution thresholds computation for adaptive multimedia. In *2017 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–6, June 2017.

[30] K. Pires and G. Simon. Youtube live and twitch: a tour of user-generated live streaming systems. In *Proceedings of the 6th ACM Multimedia Systems Conference*, pages 225–230. ACM, 2015.

[31] T. Rappaport. *Wireless Communications: Principles and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[32] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the h.264/avc standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, Sept 2007.

[33] H.-R. Shao, J. Hsu, C. Ngo, and C. Kweon. Progressive transmission of uncompressed video over mmw wireless. In *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, pages 1–5. IEEE, 2010.

[34] H. Singh, J. Oh, C. Kweon, X. Qin, H.-R. Shao, and C. Ngo. A 60 ghz wireless network for enabling uncompressed video communication. *IEEE Communications Magazine*, 46(12), 2008.

[35] H. Singh, X. Qin, H.-r. Shao, C. Ngo, C. Y. Kwon, and S. S. Kim. Support of uncompressed video streaming over 60ghz wireless networks. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 243–248. IEEE, 2008.

[36] T. Stockhammer. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[37] G. J. Sullivan, J.-R. Ohm, W.-J. Han, T. Wiegand, et al. Overview of the high efficiency video coding(hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[38] S. Sur, I. Pefkianakis, X. Zhang, and K.-H. Kim. Wifi-assisted 60 ghz wireless networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, pages 28–41, New York, NY, USA, 2017. ACM.

[39] T. Wei and X. Zhang. Pose information assisted 60 ghz networks: Towards seamless coverage and mobility support. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 42–55. ACM, 2017.

[40] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[41] X. Xie, X. Zhang, S. Kumar, and L. E. Li. pistream: Physical layer informed adaptive video streaming over lte. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 413–425. ACM, 2015.

[42] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li. Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky. In *Proceedings of the 17th ACM international conference on Multimedia*, pages 25–34. ACM, 2009.

[43] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *ACM SIGCOMM Computer Communication Review*, volume 45(4), pages 325–338. ACM, 2015.