LiFteR: Unleash Learned Codecs in Video Streaming with Loose Frame Referencing

Bo Chen, *University of Illinois at Urbana-Champaign;*
Zhisheng Yan, *George Mason University;* Yinjie Zhang, Zhe Yang,
and Klara Nahrstedt, *University of Illinois at Urbana-Champaign*

https://www.usenix.org/conference/nsdi24/presentation/chen-bo

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

LiFteR: Unleash Learned Codecs in Video Streaming with Loose Frame Referencing

Bo Chen[1], Zhisheng Yan[2], Yinjie Zhang[1], Zhe Yang[1], Klara Nahrstedt[1]
[1]*University of Illinois at Urbana-Champaign,* [2]*George Mason University*

Abstract

Video codecs are essential for video streaming. While traditional codecs like AVC and HEVC are successful, learned codecs built on deep neural networks (DNNs) are gaining popularity due to their superior coding efficiency and quality of experience (QoE) in video streaming. However, using learned codecs built with sophisticated DNNs in video streaming leads to slow decoding and low frame rate, thereby degrading the QoE. The fundamental problem is the tight frame referencing design adopted by most codecs, which delays the processing of the current frame until its immediate predecessor frame is reconstructed. To overcome this limitation, we propose LiFteR, a novel video streaming system that operates a learned video codec with loose frame referencing (LFR). LFR is a unique frame referencing paradigm that redefines the reference relation between frames and allows parallelism in the learned video codec to boost the frame rate. LiFteR has three key designs: (i) the LFR video dispatcher that routes video data to the codec based on LFR, (ii) LFR learned codec that enhances coding efficiency in LFR with minimal impact on decoding speed, and (iii) streaming supports that enables adaptive bitrate streaming with learned codecs in existing infrastructures. In our evaluation, LiFteR consistently outperforms existing video streaming systems. Compared to the existing best-performing learned and traditional systems, LiFteR demonstrates up to 23.8% and 19.7% QoE gain, respectively. Furthermore, LiFteR achieves up to a 3.2× frame rate improvement through frame rate configuration.

1 Introduction

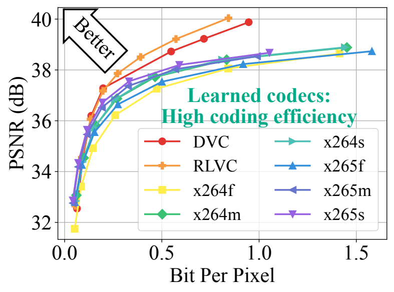
The video streaming industry has grown rapidly in recent years, with revenues of \$72.2 billion in 2021 and expected to reach \$115 billion by 2026 [20]. To achieve high-quality video streaming with minimal bandwidth usage, an essential component is the video codec, which compresses the video data while maintaining its visual fidelity.

Traditional codecs, such as AVC (x264) [57], HEVC (x265) [52], and MPEG-2 [33], are widely used and built on handcrafted modules such as block-based motion estimation, discrete cosine transform, and entropy coding. Recently, *learned codecs*, built entirely on deep neural network-based modules and end-to-end optimized, have been introduced and have demonstrated superior *coding efficiency*, i.e., the ability to encode videos with low bitrates while maintaining video quality [2,25,38,39,62,63]. As a result, learned codecs have the potential to offer a better quality of experience (QoE) in video streaming.

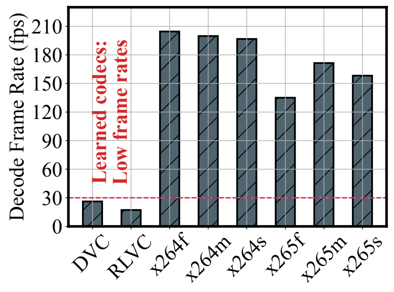
Despite the potential, the use of existing learned codecs in video streaming faces major limitations. The complicated neural processing incurs a slow decode speed at the video player, causing a low frame rate and QoE during video playback. Our results (§2) show that, on a *GPU*, the frame rate of systems using learned codecs is still one magnitude lower than that of systems using traditional codecs on a *CPU*. On many hardware configurations, learned codecs cannot reach a real-time frame rate for smooth streaming. While the QoE in systems using traditional codecs is mainly affected by slow video downloads, we discover that those using learned codecs can suffer additionally from slow decode speed.

The root cause of this limitation is the *tight frame referencing* (TFR) principle that has been in use for decades in traditional video codecs. This principle encodes and decodes each frame by referencing the immediate predecessor frame. For example, frame#1 is referenced by frame#2, and frame#2 is referenced by frame#3. It minimizes the difference between the current frame and the reference frame and makes coding efficient. However, the processing of the current frame must be delayed until the processing of its reference frame is completed. As traditional codecs are highly optimized in computation complexity [52,57], they can work with TFR with a reasonable frame rate. In contrast, TFR incurs an unacceptable frame rate in learned codecs with computation-intensive deep neural networks (DNNs). The frame dependency underlying TFR also impedes the possibility of parallel frame processing.

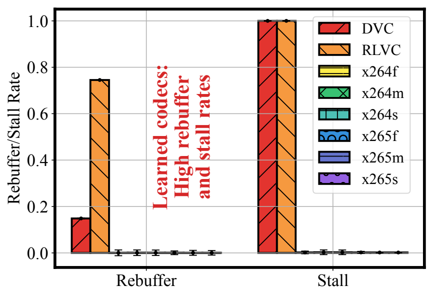
Aiming at improving the frame rate and achieving high QoE in video streaming, we propose to operate the learned



(a) Learned codecs has better coding efficiency.



(b) Learned codecs are slow in decoding.



(c) Learned codecs rebuffer and stall.

Figure 1: The potential and limitation of applying learned codecs in video streaming systems.

codec with *loose frame referencing* (LFR). In contrast to TFR, the reference of a frame in LFR only needs to be a temporally close frame, instead of the immediate predecessor. The rationale is that similarity exists between temporally close video frames, which are not necessarily adjacent. As such, frames with the same reference can be processed in parallel, which improves the frame rate of learned video codecs. More importantly, the similarity between a frame and its reference with LFR still preserves the coding efficiency.

As TFR has been the de facto design for both traditional and learned codecs that maximizes coding efficiency, implementing LFR in video streaming presents three challenges. First, optimally balancing the coding efficiency and decoding speed with LFR is non-trivial. Additionally, the reference frames in LFR could lead to significant memory consumption. Second, the existing architecture of the learned codec cannot handle the degradation in coding efficiency caused by LFR. Third, the existing streaming infrastructure is compatible with the learned codec regarding bitrate adaptation, frame rate configuration, and buffer level.

We design LiFteR, a novel on-demand video streaming system that leverages a learned codec with LFR, to address the above challenges. First, we introduce the *LFR video dispatcher* that routes frames to the learned codec iteratively. It strikes a balance between coding efficiency and decoding speed by defining frame dependency based on a binary tree and pre-order traversal. The memory consumption of it is constrained by processing frames in a GOP on a per-tree basis. Second, we design a unique LFR learned codec. It improves coding efficiency by exploiting the inter-frame correlation presented by LFR with self-attention. More importantly, it imposes minimal impact on the frame rate due to highly parallelized motion estimation. Finally, we adopt bitrate-adaptive training, frame rate configuration, and an enhanced adaptive bitrate (ABR) algorithm to bridge the gap between learned codecs and modern streaming infrastructure.

We compare LiFteR to video streaming systems built on different traditional codecs (x264 and x265) and learned codecs (DVC [39] and RLVC [63]). Results show that LiFteR consistently outperforms other streaming systems on different GPU capabilities and network conditions. Compared to the best-performing baselines using learned and traditional codecs, LiFteR achieves up to 23.8% and 19.7% QoE gain, respectively. We also demonstrate the capability of LiFteR to boost system frame rate by up to 3.2× via LFR on everyday GPUs. In summary, the contributions of this work are as follows.

- We identify the limitations of applying learned codecs in video streaming and propose the LFR paradigm.
- We build LiFteR to showcase a practical system through the design of the LFR video dispatcher, the LFR learned codec, and streaming supports for the learned codec.
- We evaluate LiFteR to demonstrate its multi-fold benefits in frame rate, video quality, and rebuffer rate.

2 Background and Motivation

Adaptive video streaming has been predominantly used. At the server, a video is segmented and encoded into different bitrates in advance. A client-side ABR algorithm [51, 61] adaptively downloads video segments of the appropriate bitrate level based on network conditions. The received segments are then decoded for playback. Studies have shown that the QoE of video streaming [51] is primarily determined by video quality Q, measured by PSNR, and playback smoothness, measured by the rebuffer rate R (the rebuffer duration divided by the video session duration). Therefore, the QoE metric can be formally defined as in Equation 1.

$$QoE = Q - \gamma R, \tag{1}$$

where γ is a parameter balancing Q and R. The value of γ in this paper is chosen according to BOLA [51].

2.1 Advantages of Learned Codecs

Video codecs play a crucial role in ensuring QoE in video streaming by improving the trade-off between video quality and bitrate. Though, traditional codecs like x264, x265, and MPEG [33, 52, 57] have been widely utilized in video streaming, the advancement in deep learning allows learned codecs [2, 38, 39, 63] to challenge the dominance of them. Unlike traditional codecs, learned codecs implement the codec pipeline with DNN modules instead of handcrafted modules. These DNN modules are trained end-to-end to optimize video quality and bitrate.

In Figure 1(a), the coding efficiency of two learned codecs (DVC [39] and RLVC [63]) is compared to three *presets*, i.e., veryslow (s), medium (m), and veryfast (f), of traditional codecs, i.e., x264 and x265. For instance, x264f represents the codec x264 using the veryfast preset. The comparison is performed on the UVG dataset [42] with the same group of pictures (GOP) size as specified in §4. Each point on the figure represents a *rate-distortion* trade-off for a codec, where bits per pixel (*bpp*) is the rate metric and PSNR is the distortion metric. A rate-distortion curve closer to the top-left side of the figure indicates higher coding efficiency, thus demonstrating the superiority of learned codecs. For example, DVC achieves a similar PSNR (38.73 dB) as x265s (38.68 dB), the best-performing traditional codec, while reducing the *bpp* by almost half (0.55 vs. 1.05 *bpp*). Similarly, RLVC improves the PSNR by 1 dB compared to x265s (39.22 dB vs. 38.20 dB) with a similar *bpp* (0.57 vs. 0.59 *bpp*).

2.2 Learned Codecs Are Slow In Decoding

Despite the benefits of learned codecs, they exhibit a relatively low decode frame rate due to the computation overhead of DNNs, which may lead to rebuffering or stall and affect QoE in video streaming. For simplicity, we will use the term "frame rate" whenever referring to the decode frame rate throughout the remainder of this paper. To assess such impacts, we developed a video streaming prototype with various traditional and learned codecs (see §5 for a detailed setup). Our rate adaptation algorithm was BOLA [51], a widely used industrial-level ABR algorithm. We used the UVG [42] and MCL-JCV [56] video datasets, along with 1,000 traces from the FCC broadband network data [23] to emulate the streaming environment. The video client utilizing traditional codecs and learned codecs was run on an Intel Core i9-8950HK CPU and an NVIDIA GTX 1080 Ti GPU, respectively.

Figure 1(b) shows that, despite the advantage of the hardware, the frame rates of systems using learned codecs remain one order of magnitude lower than traditional codecs. It is worth noting that the "veryfast" preset in x265 typically yields a faster frame rate compared to the "medium" and "veryslow" presets. However, the impact of presets may vary based on distinct video contents and configurations, such as quantization parameters and resolution. Consequently, scenarios may arise where x265f does not achieve a faster frame rate than x265m and x265s, as depicted in Figure 1(b). Figure 1(c) compares the QoE of systems via rebuffer rate and stall rate (the number of segments that cannot be played immediately after its previous segment, divided by the total number of segments). The rebuffer rates of DVC (0.15) and RLVC (0.75) are drastically higher than those of traditional codecs, indicating the videos are freezing 15% and 75% of the time.

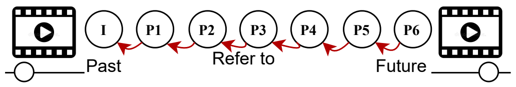


Figure 2: The frame processing dependency in today's codecs.

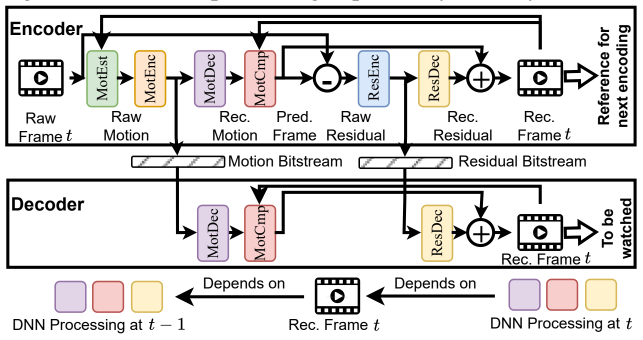


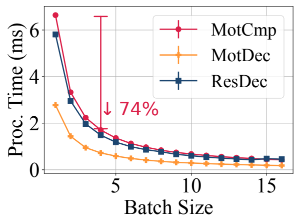
Figure 3: The canonical pipeline of learned codecs. This iterative pipeline means DNN processing at timestamp t is dependent on that at timestamp $t-1$.

The stall rate of 1.0 for both learned codecs implies that video playback freezes for every segment, as the decoding frame rate is lower than the target frame rate of the source of 30 fps.

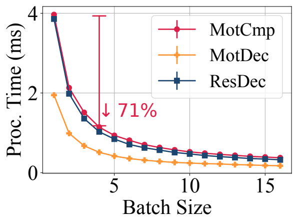
2.3 Preliminary Analysis of Learned Codecs

Background. Similar to traditional codecs, a video is divided into GOPs in learned codecs for encoding/decoding. Typically, the processing of a GOP starts from an I frame (intra-coded) that can be processed by an image codec. As depicted in Figure 2, all other frames in the GOP are P frames (predicted) and undergo predictive coding with the immediate predecessor frame as the reference frame, i.e., TFR. An alternate implementation inserts additional B frames (bi-directional) between I frame and P frames, referencing past and future frames [62]. Still, the fundamental frame referencing of I and P frames of such an implementation follows TFR.

The canonical pipeline for encoding a P frame at timestamp $t, t = 1,2,...,$ with learned codecs, is illustrated in Figure 3 (top), involving four DNN-based modules: motion estimation (MotEst), motion encoder/decoder (MotEnc/MotDec), motion compensation (MotCmp), and residual encoder/decoder (ResEnc/ResDec). Initially, MotEst estimates the motion, a vector representing the displacement of each pixel, based on the current frame (Raw Frame t) and the reference frame (Reconstructed Frame $(t-1)$), the reconstructed immediate predecessor frame. MotEnc compresses the raw motion data into bitstreams and MotDec decompresses bitstreams into the reconstructed motion, representing the motion received by the video decoder. MotCmp then generates the predicted frame based on the reconstructed motion and the reference frame. The raw residual data between the predicted and raw current frame is then calculated via subtraction and processed by ResEnc and ResDec into the reconstructed residual. Finally, the reconstructed residual and the predicted frame are added to produce the reconstructed current frame (Rec. Frame t). The decoding process partially follows the encoder pipeline in Figure 3 (top) that produces the current reconstructed frame (Rec. Frame t) from the bitstreams of the motion and residual.



(a) NVIDIA GTX 1080 Ti



(b) NVIDIA RTX 2080 Ti

Figure 4: DNNs for decoding save time via parallelism.

The need and potential of parallelism. Unlike heavily-engineered modules in traditional codecs that can run fast enough to support real-time encoding/decoding, the sophisticated DNN modules in learned codecs are inherently slow. As a result, even a single iteration in Figure 3 pipeline can be time-consuming, resulting in a low frame rate.

Parallelism, which accelerates DNN by processing multiple instances of data simultaneously in a batch, is a potential solution for improving the frame rate of learned codecs, by processing frames in parallel. To demonstrate the potential of parallelism, we isolate DNN modules for decoding, i.e., MotCmp, MotDec, and ResDec, from learned codecs. Then, we separately measure the mean processing time per frame of these DNN modules for different batch sizes, ranging from 1 to 16 frames, on NVIDIA GTX 1080 Ti and RTX 2080 Ti. Figure 4 demonstrates the average processing time per frame when the batch size is one, the case with TFR, can be significantly improved by adopting a larger batch size. When the batch size increases from one to four, the speed of the motion compensation modules can be boosted by 74% and 71% on NVIDIA GTX 1080 Ti and RTX 2080Ti, respectively.
Parallelism is infeasible with TFR. Parallelism requires that the processing of different frames $t = 1,2,\ldots$ is independent. For existing learned codecs with TFR, DNN processing of modules like MotDec, MotCmp, and ResDec at timestamp t happens after the reconstructed frame $t-1$. The reconstructed frame $t-1$, produced by DNN modules MotDec, MotCmp, and ResDec, happens after DNN processing of MotDec, MotCmp, and ResDec at timestamp $t-1$. As a result, the processing of MotDec, MotCmp, and ResDec at timestamp t depends on the processing at timestamp $t-1$, as illustrated in Figure 3 (bottom). Such a dependency contradicts parallelism.

2.4 Intuition and Challenges

Intuition. Our intuition is that frame similarity exists not only between adjacent frames but also between non-adjacent, temporally close frames. By leveraging the temporally close frames for reference in video coding, i.e., loose frame referencing (LFR), multiple frames can share the same reference frame. Therefore, these frames with the same reference can be processed in parallel, improving the frame rate. Meanwhile, the similarity between a frame and its reference, a temporally close frame preserves coding efficiency.
Challenges. Designing a learned video streaming system with LFR presents three main challenges:

1. **Processing pipeline:** It remains a question of how to configure LFR that 1) optimally balances the coding efficiency and decoding speed and 2) minimizes the memory usage due to buffering multiple reference frames.
2. **Learned codec:** LFR, by its design, introduces a larger difference between the raw and reference frames than TFR. The existing learned codec design cannot handle such discrepancies.
3. **Streaming infrastructure:** The existing video streaming infrastructure is incompatible with learned codecs, which exhibit different behaviors from traditional ones.

3 LiFteR Overview

The overview of LiFteR is illustrated in Figure 5, which comprises offline and online stages. In the offline stage, we construct the *LFR learned codec* (§3.2), a unique codec design that mitigates the impact of LFR by using an *elastic compression component* (§3.2.2) and maintains the decoding speed through *highly-parallelized motion estimation* (§3.2.1). Then, the codec is trained and configured before being deployed on the media server and the client.

After deployment, the server utilizes the *LFR video dispatcher* (§3.1) to route the raw video to the video encoder iteratively based on LFR. The LFR video dispatcher balances coding efficiency and decoding speed with a *dependency graph* (§3.1.1) and constrains memory usage via a *frame iterator* (§3.1.2). The video encoder compresses the raw video dispatched by the LFR video dispatcher into video segments of different bitrates, ready for Dynamic Adaptive Streaming over HTTP (DASH).

During the online stage, the client employs an ABR algorithm to download video segments with appropriate bitrates from the media server. These downloaded segments are routed to the decoder by the LFR video dispatcher and reconstructed into video frames. To integrate LiFteR into existing ABR streaming infrastructures, we build streaming supports that enable *adaptive bitrate* in training (§3.3.1), *frame rate configuration* (§3.3.2), and the *enhanced ABR algorithm* (§3.3.3).

3.1 LFR Video Dispatcher

LFR video dispatcher models frame processing dependency with the *dependency graph* (§3.1.1). Driven by the depen-

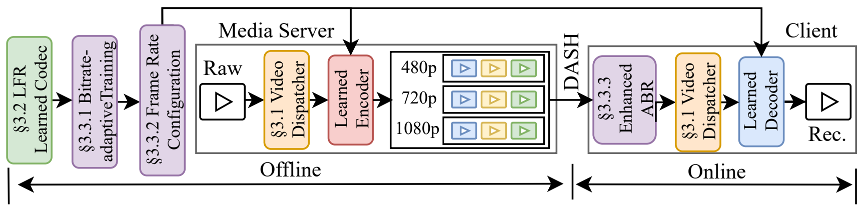


Figure 5: Overview of the video streaming pipeline in LiFteR.

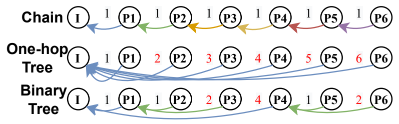


Figure 6: Frame reference structures. A binary tree balances reference cost and depth.

dency graph, it iteratively feeds frames to codec (§3.1.2).

3.1.1 Dependency Graph

In the dependency graph, each vertex with a unique positive integer label corresponds to a video frame processed at a particular timestamp, and the directed edge represents the dependency, pointing from a reference frame to the to-be-processed frame. Traversing the dependency graph is equivalent to processing video frames, which starts from a visited vertex representing the I frame. Then, graph traversal proceeds in iterations until all vertices are visited. In each iteration, we can visit vertices pointed by directed edges starting from visited vertices in previous iterations. The visited vertices in the same iteration represent frames that can be processed in parallel. As this paper focuses on the dependency of I and P frames (Figure 2) where a frame with a smaller timestamp is typically processed earlier, we constrain this dependency graph such that the end of a directed edge has a larger index than the start. We aim to minimize two metrics when deciding the shape and labeling of indices in the dependency graph.

1) Reference cost: The sum of the difference in indices between the end and start of directed edges, which indicates the difference between the frames and their reference in video coding and relates to coding efficiency.

2) Reference depth: The maximum depth of a vertex, which indicates the number of iterations needed to process all frames and relates to parallelism.

Shape: binary tree. It is challenging to balance the reference cost and depth. The dependency graph representing TFR has the shape of a chain. In Figure 6 (top), we provide a visual representation of the chain while also labeling index differences between frames and their references. Iterations are colored differently for clarity. The chain displays a low reference cost of 6 and a high reference depth of 6. Alternatively, we construct the dependency graph as a one-hop tree in Figure 6 (middle), which represents a fully parallelized way where all P frames reference the same I frame. This tree boasts a small reference depth (1) but has a high reference cost (21).

Our insight is that the binary tree as the dependency graph effectively balances the reference cost and depth. First, unlike the chain, the reference depth in a binary tree increases slowly with the number of vertices in the tree in a logarithmic manner. Second, in contrast to the one-hop tree, the reference cost is more constrained as one frame is used at most twice for frame referencing. While N-ary trees ($N = 2,3,\ldots$), e.g., the 3-ary tree, may have similar properties as the binary tree, as the first effort to explore frame dependency with a tree, we focus on the binary tree for simplicity.

Labeling: pre-order traversal. The derivation of the dependency graph is essentially finding the Minimum Spanning Tree (MST) in the particular shape of the binary tree, termed Minimum Spanning Binary Tree (MSBT), within a dense graph. Assuming there are N frames, the dense graph consists of N vertices indexed by $0,1,\ldots,N-1$. In this dense graph, there exists an edge between every pair of vertices, whose absolute difference in indices is the edge weight. It is important to minimize the reference cost in labeling.

Naively, we can construct the MST by leveraging the well-known Prim's algorithm to add vertices and edges in the dense graph into a tree starting with the vertex 0. However, the derived MST would have the shape of the chain like Figure 6 (top) instead of the binary tree. To tackle this problem, we modify the Prim's algorithm by skipping the addition of vertices and edges that 1) cause the MST to have a depth that is higher than a complete binary tree of N vertices, i.e., $\lceil \log_2(N+1) \rceil$ and 2) cause one vertex to have more than two children. The resulting algorithm is equivalent to assigning frame indices (from small to large) to vertices of a binary tree via pre-order traversal. Intuitively, this algorithm greedily minimizes the reference costs of both children of any vertex.

Theoretical analysis. Figure 6 (bottom) illustrates the dependency graph with a binary tree and pre-order traversal. Such a tree results in a reference depth (2) that is considerably smaller than that of the chain while using only half the reference cost of the one-hop tree, i.e., 6. In Figure 7, we delve further into how the reference cost and depth change as the number of frames in a GOP increases. The number of frames is expressed as a function of the full binary tree depth D. Our analysis reveals that the binary tree strikes a better balance between the frame rate (reference depth) and the coding efficiency (reference cost) and displays a more pronounced advantage when processing more frames.

3.1.2 Frame Iterator

With the dependency graph, a straightforward way is to include all frames of a GOP in it. However, a GOP can have a large number of frames, which requires processing more frames in parallel than a system can afford. To scale to arbi-

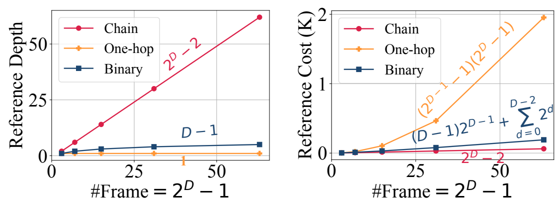


Figure 7: The binary tree achieves a better trade-off between reference cost and depth.

trary GOP sizes, we process frames in a GOP on a per-tree basis, as illustrated in Figure 8, where each tree is processed with three steps: slice, map, and iterate.

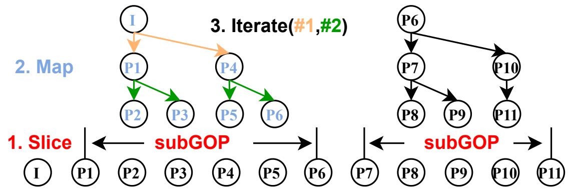


Figure 8: Video processing on a per-tree basis.

Slice: Frames within a GOP (excluding the I frame) are sliced into non-overlapping, consecutive *subGOPs*, each containing N sequential frames. Here, N is the subGOP size, which equals the number of non-root vertices in a full binary tree. For instance, full binary trees of depth $D = 2, 3, 4$ correspond to subGOP sizes $N = 2^D - 2 = 2, 6, 14$.

Map: Frames in each subGOP are mapped to the binary tree via pre-order traversal as shown in Figure 8. The root of a tree is the latest I or P frame preceding a subGOP. The reference of each frame is determined as the frame at its parent vertex. It is worth noting that the last subGOP might not have enough frames to fill all vertices in the full binary tree, which marginally affects the functionality of our approach.

Iterate: To achieve parallelism in the video encoding/decoding, the frames are processed level by level iteratively in the tree. The processing of frames at the same level of the tree is parallelized. For example, the codec first processes frames that reference the tree root (I frame), such as P1 and P4 in Figure 8. Then, it processes frames (P2, P3, P5, and P6) that reference the previously processed frames (P1 and P4).

3.2 LFR Learned Codec

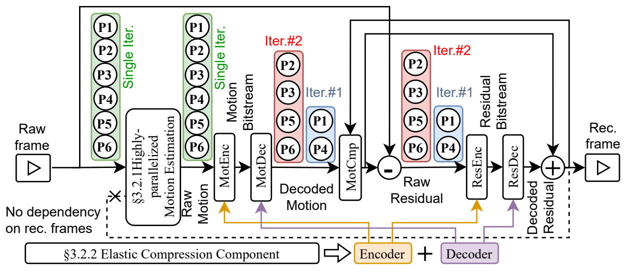


Figure 9: Design of LFR Learned Codec.

LFR learned codec (Figure 9) replaces motion estimation in canonical learned codec (Figure 3) with *highly-parallelized motion estimation* (§3.2.1) and encodes/decodes motion/residual of one or multiple frames via *elastic compression component* (§3.2.2).

3.2.1 Highly-parallelized Motion Estimation

In Figure 8, the processing of frames at the lower level (e.g., two frames at level 1 of the tree) are less parallelized than those at the higher level (e.g., four frames at level 2), which hinders the decoding speed. The fundamental problem is that DNN modules in the learned codec, i.e., motion estimation and compensation, rely on the reconstructed version of the reference frame(s).

Frame approximation. The key intuitions are that 1) the raw and reconstructed frames are similar and 2) motion estimation is performed only on the encoder. Therefore, it is possible to approximate (replace) the reconstructed reference frames with the raw reference frames with the same index in motion estimation. As such, the processing of all frames (e.g., frames P1, P2, ..., and P6 in Figure 8), relying on only raw frames, are parallelized. Such approximation allows us to speed up motion estimation, motion encoder, and motion decoder. More importantly, the coding efficiency is negligibly affected due to the frame similarity.

3.2.2 Elastic Compression Component

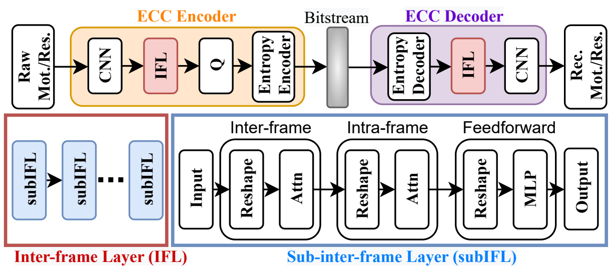


Figure 10: Elastic compression component design

As shown in Figure 10, the elastic compression component (ECC) comprises an encoder and a decoder. Following existing efforts on learned codecs [39, 63], the encoder utilizes a convolutional neural network (CNN) to transform the raw motions or residuals, into high-level features. Then, quantization and entropy encoding [58] are performed to compress quantized features into more compact bitstreams. In reverse, the decoder adopts entropy decoding to convert the bitstreams to quantized features, and a CNN to transform the features to the same shape as the raw motions or residuals. However, as the difference between a frame and its reference is increased by LFR, the coding efficiency could be compromised.

Inter-frame layer. We make two key observations. First, LFR expands the frame dimension of the input to DNN modules from one to multiple. The expanded dimension presents inter-frame dependency in video coding that does not exist in existing learned codecs [38, 39, 63]. Such dependency presents an opportunity to enhance coding efficiency. Second, the self-attention module [55] is a generic learner of dependency adopted in numerous video-related tasks [9, 26, 44, 46] for spatial and temporal dependency.

Based on these observations, we design the *inter-frame layer* (IFL) that leverages inter-frame dependency in LFR with self-attention. The IFL involves sequentially connected sub-inter-frame layers (subIFL). Each subIFL is constructed with inter-frame, intra-frame, and feedforward blocks. Given common practice, the IFL is placed after CNN in the encoder and before CNN in the decoder.

The input dimension of the subIFL is $F \times C \times H \times W$, where F, C, H, and W represent the number of input frames, the number of channels, and the height and width of the feature, respectively. The output of the subIFL has the same dimension as its input. The inter-frame and intra-frame blocks exploit inter-frame and intra-frame dependency with a reshape function and the multi-head Attention (Attn) [55]. The input of Attn has three dimensions ($batch \times sequence \times feature$), which correlates data in the *sequence* dimension, with no correlation along the *batch* dimension. We reshape the input to shapes $HW \times F \times C$ and $F \times HW \times C$ for inter-frame (F) and intra-frame (HW) dependency, respectively. The last block converts the input shape to $BHW \times C$ and applies a multi-layer perceptron (MLP) consisting of two linear neural network layers to refine features.

3.3 Streaming Supports

We integrate our system into existing streaming infrastructure through *bitrate-adaptive training* (§3.3.1), *frame rate configuration* (§3.3.2), and *enhanced ABR algorithm* (§3.3.3).

3.3.1 Bitrate-adaptative Training

To leverage ABR algorithms that require multiple bitrates of the same segment, a learned codec must be configurable to encode and decode videos at different compression levels, which correspond to different trade-offs of bitrates and distortion. However, a typical learned codec cannot adjust the trade-offs of bitrates and distortion as easily as traditional codecs with the quantization parameter (QP) [52, 57]. A naive approach that prepares different versions of the codec of different compression levels is memory-consuming.

Compression-level embedding. To attain similar rate adaptability with learned codecs, we adopt a *compression-level embedding* approach [48]. In this approach, the compression level is treated as an input, in addition to video frames, to the learned codec. Specifically, multiple compression levels $l = 1$, 2, ..., are converted into one-hot vectors, spatially tiled, and concatenated to the input of DNN modules. This approach allows the trade-off of bitrates and distortion to be conveniently adjusted by changing the input, without storing multiple versions of the codec. Meanwhile, this approach minimally degrades the coding efficiency compared to a canonically trained learned codec.

3.3.2 Frame Rate Configuration

When working with a specific hardware platform, it is critical to attain a targeted frame rate. For canonical learned codecs, adjusting the frame rate may require compressing and retraining DNNs in the learned codec, which is time-consuming.

SubGOP probing. In LiFteR, this can be easily achieved by configuring the subGOP size. Specifically, we probe the target hardware with the LiFteR decoder using different subGOP sizes. Based on the frame rates achieved at various subGOP sizes, we select the smallest one that surpasses the targeted frame rate, e.g., 30 fps. Choosing a subGOP size than what is needed is not recommended, as it would consume more GPU resources while offering only marginal enhancements to coding efficiency (Figure 18).

3.3.3 Enhanced ABR Algorithm

The buffer level is a critical factor in the decision-making process of adaptive bitrate (ABR) algorithms, such as BOLA [51], which adjusts video quality according to network conditions. In video streaming systems, the buffer level represents the duration of video frames in the playback buffer. A low buffer level indicates low network bandwidth, causing the ABR algorithm to download segments of lower bitrates to avoid rebuffering. Conversely, a higher buffer level triggers the ABR algorithm to download segments of higher bitrates. However, for learned codecs, the buffer level does not accurately reflect network conditions and may mislead ABR algorithms for learned codecs. For instance, when the decoding rate of frames equals the consumption rate, the playback buffer remains almost empty, as each decoded frame is immediately consumed. This is likely to happen with learned codecs whose decoding rate is more comparable to the consumption rate than traditional codecs. As the buffer level is low, the ABR algorithm may download segments of low bitrates, even though the network bandwidth is abundant, leading to wastage.

Virtual buffer. To tackle this issue, we introduce the *virtual buffer*. It captures the length of received but unwatched segments, irrespective of whether they are stored in the replay buffer. By utilizing the level of the virtual buffer instead of the actual buffer level in the ABR algorithm, we avoid potential conservative decision-making by the ABR algorithm.

4 Implementation

Model choices and streaming configuration. We implement motion estimation using convolutional neural networks [47] and motion compensation using a warping function and an interpolation network [39]. As per [9], we configure the attention mechanism's number of heads and each head's channel number to 8 and 64, respectively, and the number of sub-inter-frame dependencies (subIFL) to 12. During training, we set the subGOP size to six, which we empirically find to be sufficient for the high coding efficiency and frame rate of LiFteR. We utilize BOLA [51], an industrial-level ABR algorithm [19], as the ABR algorithm in LiFteR, which decides the bitrate every time the system tries to download a segment. The segment and GOP size are set to 5s following [21].

Training. We jointly optimize our model using the Vimeo-90k dataset [60]. In this dataset, every training sample consists of seven images, with the first image being the I frame ($k = 0$) and the remaining images being P frames ($k = 1, \dots K$). K corresponds to the number of P frames in a GOP during training. The I frame is encoded and decoded using an image codec, Better Portable Graphics (BPG) [7], while the learned codec compresses the P frames on a per-tree basis. We scale the training images' resolution to 256×256. We use the Adam optimizer [31] with a learning rate of 10^{-4}, which is reduced by a factor of 10 after convergence until 10^{-6}. Our loss function $\mathcal{L}$ is defined as follows.

$$\mathcal{L} = \sum_{k=1}^{K} \mathbb{E}_{l}[\lambda^{l} \mathcal{D}(x_k, \hat{x}_k{}^{l}) + \hat{v}_k{}^{l} + \hat{r}_k{}^{l}]. \quad (2)$$

We set $\lambda^l = 256, 512, 1024, 2048, 4096, 8192, 16384$ at different compression levels $l = 0, 1, ..., 6$, which cover bitrates from 1 Mbps to 16 Mbps. The estimated bits per pixel (bpp) of the motion or residual for the compression level l is denoted by $\hat{v}_k{}^{l}$ or $\hat{r}_k{}^{l}$. The distortion, measured by mean square error (MSE), between the raw frame x_k and the reconstructed frame at compression level l, $\hat{x}_k{}^{l}$, is denoted by $\mathcal{D}(x_k, \hat{x}_k{}^{l})$. In training, the compression level l in each sample is randomly and iteratively generated, following ELFVC [48].

5 Evaluation

We evaluate LiFteR regarding streaming performance, coding efficiency, and adaptability. Our highlights are

1. LiFteR achieves superior QoE compared with systems using learned and traditional codecs across different GPUs and network conditions (Figures 11-14).

2. LiFteR improves the rate-distortion trade-off of learned and traditional codecs (Figure 16).

3. LiFteR maintains a real-time frame rate across different GPUs (Figure 15) and improves the frame rate by adapting the subGOP size (Figure 17).

4. The component designs of LiFteR show effectiveness in contributing to the overall gain (Figures 18-22).

5.1 Methodology

Hardware setup. We conducted our experiments on Linux desktops featuring various NVIDIA GeForce GPUs and CPUs, which are listed in Table 1. The learned codecs are executed on all three Linux desktops, "1080", "2080", and "3090". Although the traditional codecs always run on "2080", the choice of the three hardware platforms has a minimal impact on the QoE for them.

Table 1: Hardware setup.

Name	GPU	CPU
1080	GTX 1080 Ti	Intel Core i9 @ 2.90GHz
2080	RTX 2080 Ti	Intel Core i7 @ 3.60GHz
3090	RTX 3090 Ti	AMD Ryzen 9 @ 4.95GHz

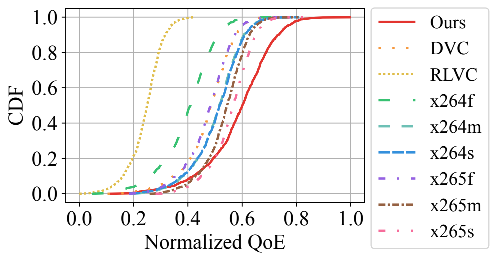
Network traces. We use network traces from FCC [23], randomly selecting 1,000 traces from two tests conducted for "video streaming" and "http get". These traces represent a range of diverse network scenarios. The "video streaming" and "http get" traces have an average bandwidth of 3.9 Mbps and 15.8 Mbps, respectively, indicating limited and adequate bandwidth.

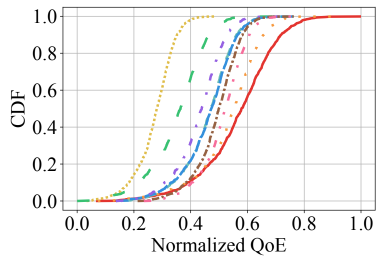
Video datasets. We merged two video datasets, UVG [42] and MCL-JCV [56]. This unified dataset consists of 37 videos with a resolution of 2K, operating at 30 frames per second with a total runtime of around 5 minutes.

Baselines. In our evaluation, we compare LiFteR with video streaming systems that use state-of-the-art learned codecs (DVC [39] and RLVC [63]) and traditional codecs (x264 [57] and x265 [52]), all employing BOLA for rate adaptation with the same GOP and segment size as LiFteR. To test the traditional codecs, we use the FFmpeg [54] implementation and configure each codec into three modes: veryfast, medium, and veryslow. These modes are denoted by x264-veryfast (x264f), x264-medium (x264m), x264-veryslow (x264s), x265-veryfast (x265f), x265-medium (x265m), and x265-veryslow (x265s). Commands for the different modes can be found in Appendix A. To optimize PSNR, DVC and RLVC are configured in the same way as LiFteR. To ensure a fair evaluation, we encode videos for all baseline systems into seven bitrates, covering a range that is comparable to LiFteR's videos. Note that there are learned codecs [2, 48] that adopt alternate designs of motion estimation and compensation instead of that of DVC, RLVC, and ours, which are not included in the evaluation for fairness. However, as they still rely on TFR, we claim the performance gain achieved with LFR is applicable to them.

5.2 End-to-end performance

End-to-end QoE. Figures 11 and 12 illustrate that the overall QoE of LiFteR outperforms other baselines on network traces with limited ("video stream") and adequate ("http get") bandwidth, respectively. The QoE metric is calculated


(a) NVIDIA GTX 1080 Ti


(b) NVIDIA RTX 2080 Ti

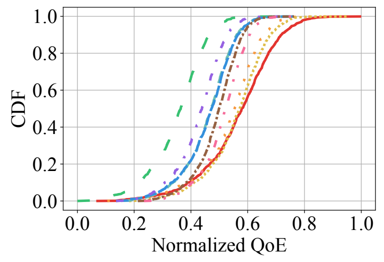
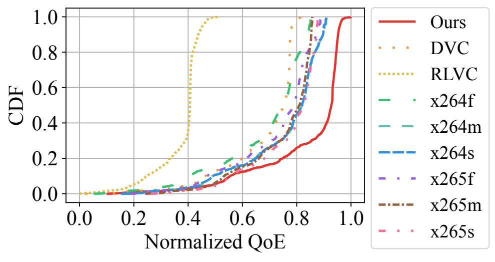
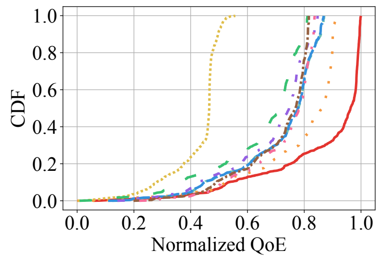

(c) NVIDIA RTX 3090 Ti

Figure 11: LiFteR shows consistent advantages in QoE across different hardware on the “video stream” traces.


(a) NVIDIA GTX 1080 Ti


(b) NVIDIA RTX 2080 Ti

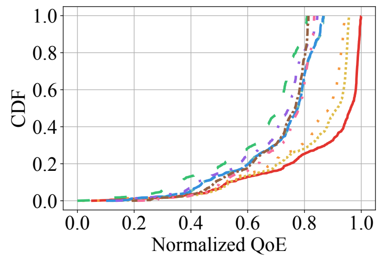
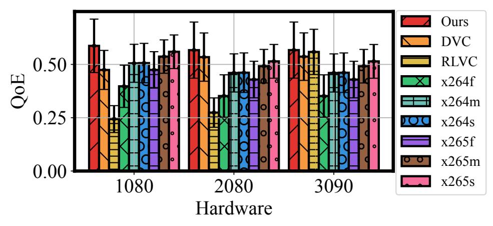
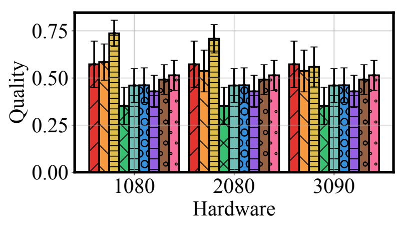

(c) NVIDIA RTX 3090 Ti

Figure 12: LiFteR demonstrates consistent advantages in QoE across different hardware on the “http get” traces.


(a) Norm. QoE


(b) Norm. Video Quality

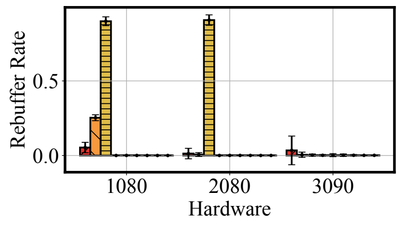
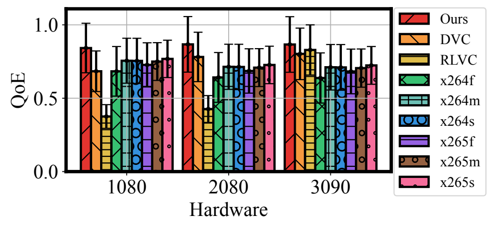
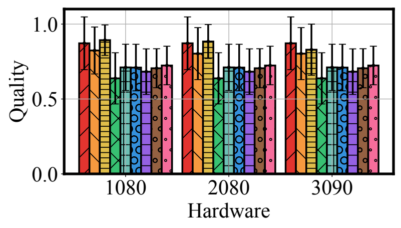

(c) Norm. Rebuffer Rate

Figure 13: LiFteR’s normalized QoE is 1.5%-23.8% and 5%-10.3% higher than the best-performing learned and traditional approaches, respectively, with limited bandwidth (“video stream” traces).


(a) Norm. QoE


(b) Norm. Video Quality

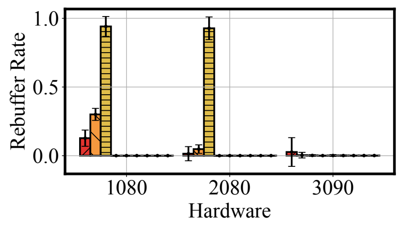

(c) Norm. Rebuffer Rate

Figure 14: LiFteR demonstrates 4.5%-23.2% and 9.6%-19.7% improvements in normalized QoE than the best-performing learned and traditional approaches, respectively, with adequate bandwidth (on “http get” traces).

based on Equation 1, normalized per hardware and trace, with the lowest and highest values mapped to 0 and 1. On the “video stream” traces with limited bandwidth, LiFteR’s normalized QoE is 1.5%-23.8% and 5%-10.3% higher than the best-performing learned and traditional streaming systems, respectively (Figure 13(a)). On the “http get” traces with adequate bandwidth, the advantages in normalized QoE become 4.5%-23.2% and 9.6%-19.7%, respectively (Figure 14(a)), indicating that LiFteR better utilizes adequate bandwidth than other approaches. The performance of DVC and RLVC is significantly impacted by the hardware platform. DVC has the second-best performance on “2080” but one of the worst performances on “1080”, whereas RLVC has the second-best performance on “3090” but the worst performance on “1080” and “2080”. In contrast, LiFteR performs well on different hardware consistently.

QoE breakdown. To improve visualization, the metrics of video quality and rebuffer rate were normalized per hardware and trace, with the lowest and highest values being mapped to 0 and 1, respectively. In Figure 13(b) and Figure 14(b), the

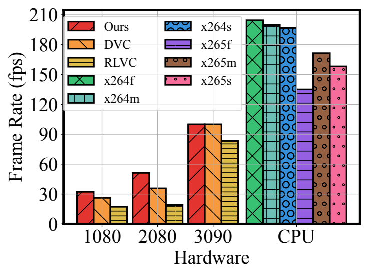


Figure 15: LiFteR decodes faster than other learned codecs.

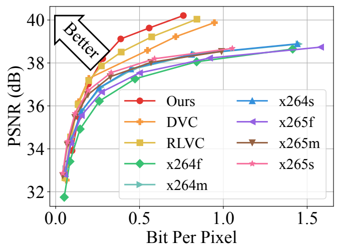


Figure 16: LiFteR exhibits the best coding efficiency.

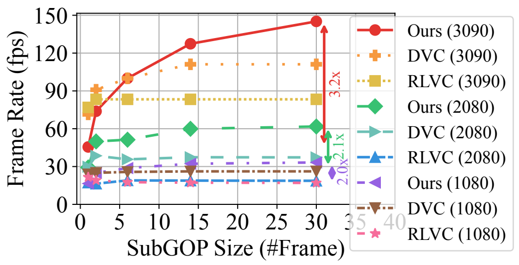


Figure 17: LiFteR can boost the frame rate by 3.2× via adaptation.

video quality of DVC and RLVC is mostly better than others because their virtual buffers are consumed slower or increasing, resulting in the ABR algorithm downloading segments with higher bitrates. Regarding the rebuffer rate (Figure 13(c) and Figure 14(c)), LiFteR significantly outperforms DVC and RLVC due to its real-time frame rate (Figure 15). For high-end hardware like NVIDIA RTX 3090 Ti, the rebuffer rate of LiFteR is lightly above others because it tends to wait for bitstreams of several frames before parallel processing. It causes rebuffering more easily than learned codecs with TFR (DVC and RLVC), which plays frames immediately after receiving them. Although this downside is outweighed by the benefits of LiFteR's superior frame rates on low-end hardware, it becomes noticeable for high-end hardware. Nevertheless, LiFteR's overall QoE is better than others due to its superior quality. We also notice a gap in rebuffer rate between our system and traditional systems, particularly on less powerful hardware. The reason is primarily the longer decoding latency of LiFteR than traditional codecs. It causes LiFteR to take more time on average to display the first frame after rebuffering. Nevertheless, the subsequent frames in LiFteR do not experience any additional delays. As a result, the QoE of LiFteR is not significantly affected by this fact.

Frame rate. In Figure 15, we compare the frame rates of systems using learned codecs on different GPUs and systems using traditional codecs on a CPU. LiFteR achieves consistent real-time frame rates across different hardware platforms. However, DVC and RLVC demonstrate worse frame rates than LiFteR on low-end hardware, such as "1080" and "2080". Despite optimizations in LiFteR, it does not outperform traditional video codecs, which are heavily engineered in speed.

Compression performance. In Figure 16, it is evident that LiFteR outperforms other baselines in terms of coding efficiency. Among the traditional systems, x265-veryslow performs the best, achieving a PSNR of 38.20 dB at 0.59 *bpp*. However, LiFteR achieves the same PSNR with less than half the bandwidth usage, requiring only 0.28 bpp. In comparison, the best-performing learned approach, RLVC, needs 0.39 *bpp* to achieve a PSNR of 38.50 dB, while LiFteR achieves a superior PSNR of 39.12 dB at the same *bpp*. LiFteR's superior coding efficiency stems from (i) the inter-frame dependency presented by LFR at the cost of increased GPU utilization, and (ii) ECC (§3.2.2), which effectively captures such dependency via the self-attention mechanism. These factors mitigate and outweigh the negative impact of the LFR video dispatcher (§3.1) and highly-parallelized motion estimation (§3.2.1), which potentially widen the gap between a frame and its reference.

Adaptability of LiFteR. LiFteR has the unique feature of adapting the subGOP size to achieve varying speeds. We evaluated LiFteR's performance across different subGOP sizes, including 2, 6, 14, and 30, corresponding to the full binary tree depths of 2, 3, 4, and 5, respectively. We also included a special case, "subGOP=1", representing LiFteR processing with TFR, similar to DVC and RLVC. In Figure 17, we can observe the frame rate of LiFteR as a function of the subGOP size and compare it with other learned approaches processing the same number of frames. The results indicate that LiFteR improves its speed by 2-3.2× as the subGOP size increases, while the other learned approaches' speed remains almost unchanged. It is worth noting that there is a slight increase in frame rates of other learned approaches. The reason is that the GPU is relatively slower when performing the first forward propagation. Then, it improves and gradually stabilizes for subsequent forward propagations. As a result, even if these approaches process frames one by one with TFR, their speed is slightly increased when the number of frames increases.

Figure 18 compares LiFteR's coding efficiency at different subGOP sizes. Initially, LiFteR is suboptimal when the subGOP size is 1 or 2. We speculate the time dimension of the input is not large enough to benefit from ECC for these small subGOP sizes. As the subGOP size increases, the coding efficiency improves and stabilizes. This result also indicates LiFteR's ability to adapt to larger subGOP sizes (14 and 30) than what it is trained with (6) with noticeable degradation.

5.3 Design Analysis

Impact of dependency graph and ECC. The dependency graph and ECC are the two key designs in LiFteR. To demonstrate their significance via comparison, we introduced three alternative designs, in addition to our original design ("Default"): 1) "w/o IFL": eliminating inter-frame dependency from the learned codec, i.e., removing IFL, 2) "Chain": substituting the binary tree in the dependency graph with a chain, and 3) "One-hop": substituting the binary tree in the depen-

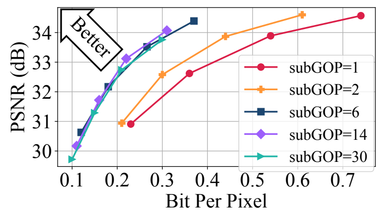


Figure 18: Coding efficiency of LiFteR initially improves with a larger subGOP size.

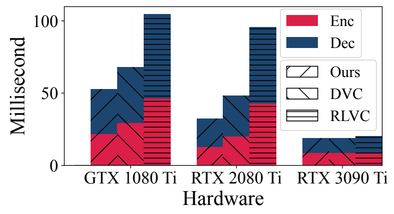


Figure 19: Time resource in learned codecs is generally allocated more to decoding than encoding.

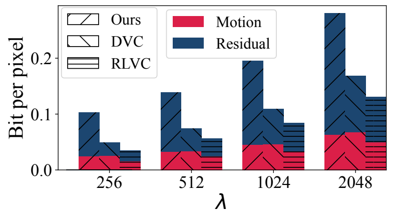
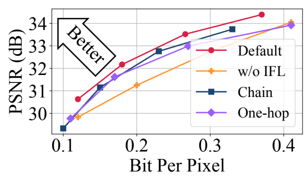


Figure 20: Bit resource in learned codecs is distributed more to residual than motion.


(a) Coding efficiency

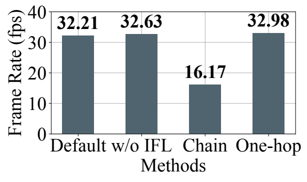
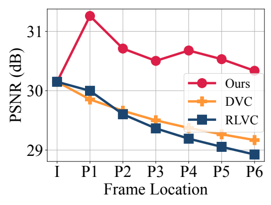

(b) Frame rate

Figure 21: Binary tree and inter-frame dependency optimally balance coding efficiency and frame rate.


(a) $\lambda = 256$

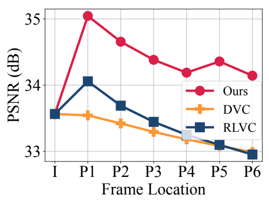

(b) $\lambda = 2048$

Figure 22: LiFteR's quality is more stable than others.

dency graph with a one-hop tree. We examine the impact of each design on coding efficiency and frame rate. For coding efficiency, Figure 21(a) shows both "Chain" and "One-hop" modes cause a PSNR drop of approximately 0.5 dB at the same *bpp*, compared to "Default". Additionally, the "w/o IFL" mode experiences a more significant loss of coding efficiency than the "Default" mode, as its PSNR drops by around one dB at the same *bpp*. These results highlight the importance of the combination of binary tree and inter-frame dependency in achieving satisfactory coding efficiency. Regarding frame rate, the "Default" mode has roughly twice the frame rate of the "Chain" mode, demonstrating a significant advantage of the binary tree. The "One-hop" mode exhibits a similar frame rate as the "Default" mode, indicating that is it not necessary to parallelize the processing of all frames to achieve substantial decode speed improvement. It is also found that the "w/o IFL" and "Default" modes have similar frame rates, which suggests that the inclusion of the inter-frame dependency has negligible impact on the frame rate. Overall, the binary tree and the leverage of inter-frame dependency strike the optimal balance of coding efficiency and frame rate for LiFteR.

Impact of the virtual buffer. To illustrate the significance of the virtual buffer, we remove it from LiFteR and perform streaming experiments with the "1080" hardware on two network traces. Figure 23 reports the average normalized QoE, video quality, and rebuffer rate. We observe a decline in both the learned approaches' QoE and video quality. Even when the bandwidth condition changes from limited to adequate, the quality and QoE of the learned approaches do not improve as significantly as the traditional approaches. This is because, without the virtual buffer, the real buffer in learned approaches remains consistently low, causing the ABR algorithm to download low-quality segments, irrespective of the bandwidth condition. Additionally, the downloaded segments have low bitrates, so the contrast in rebuffer rates between traditional and learned approaches is less pronounced than in Figure 13(c) and Figure 14(c).

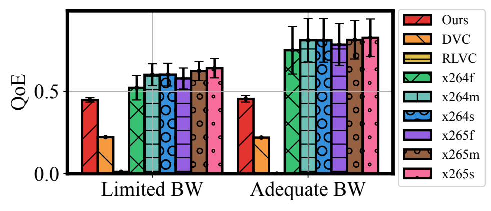
5.4 Micro Benchmark

Resource allocation. Time (computation time) and space (bitstream size) are two critical resources for learned codecs. We analyze the allocation of these resources in our approach, DVC, and RLVC using the UVG dataset. Figure 19 displays the time spent on encoding and decoding across various hardware. All approaches require more time for encoding than decoding, as the decoder modules are a subset of the encoder modules. Figure 20 illustrates the allocation of bits for motions and residuals over λ values of 256, 512, 1024, and 2048. We observe that our approach allocates a higher percentage of bits to the residual bitstream than the other approaches. We speculate that this is because LFR increases errors in ME and MC, thereby necessitating more bits in the residual to compensate for those errors.

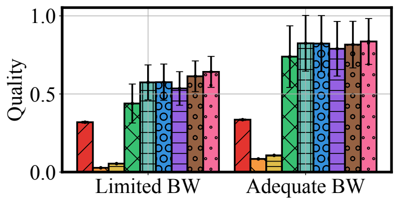
Error propagation. In Figure 22, we visualize the quality of frames reconstructed by our codec, DVC, and RLVC, based on their position in a seven-frame GOP using the UVG dataset ($\lambda = 256$ and 2048). For our codec, we set the subGOP size to 6. As the temporal distance between P and I frames increases, the PSNR of the P frame monotonically decreases for DVC and RLVC, which could lead to degradation of QoE when viewing frames distant from the I frame, particularly with large GOP sizes. In contrast, our approach with LFR can increase the PSNR of P frames even when their temporal distance to the I frame increases, potentially allowing for a smoother viewing experience.

6 Related Work

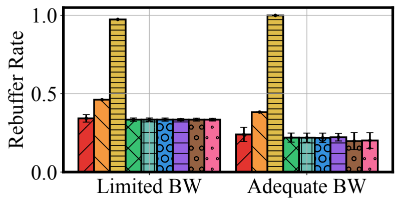
Video codecs. Video codec standards, such as MPEG-2 [33],



(a) Norm. QoE



(b) Norm. Video Quality



(c) Norm. Rebuffer Rate

Figure 23: Learned codecs do not outperform traditional codecs without the virtual buffer.

H.264 [57], and H.265 [52], rely on traditional handcrafted methods. Due to recent advancements in deep learning, researchers have started to replace operations in traditional video codecs with DNNs [1, 6, 12, 16, 18, 37, 40, 47, 59]. DNNs can also serve as hints to assist codecs in video analytics [13, 14, 22, 36]. Besides, researchers have proposed learned codecs built purely from DNNs [2, 25, 38, 39, 48, 62, 63]. Another category of learned approaches integrates DNN and progressive coding [21]. However, all existing learned codecs employ a TFR principle, suffering from slow decoding and low QoE problems. LiFteR addresses the low frame rate problem in video streaming systems with learned codecs with LFR. Further, LiFteR can be easily integrated into these TFR-based learned codecs for improved decoding speed. Although learned codecs with speed optimization like ELFVC [48] may achieve near real-time performance, e.g., 30 fps, on certain hardware, they cannot meet the real-time requirements when the hardware degrades or the frame rate requirements increase to 60 fps or 120 fps. In contrast, LiFteR flexibly handles these variations by adapting the subGOP size.

Video streaming. Enhancing the quality of video streaming has been the focus of numerous techniques, typically falling into one of three categories: push-based, pull-based, and video super-resolution (VSR). Push-based strategies analyze playback statistics from clients and push the appropriate bitrate of videos to each client from a central server. Several studies [8, 24, 28, 35] have investigated the effectiveness of these approaches. In contrast, pull-based strategies guide clients to download videos of appropriate bitrates from the server based on the predicted bandwidth or buffer level. Several studies, such as [41, 51, 61, 65, 67, 68], have explored the efficacy of these strategies. VSR techniques can also be applied to improve video streaming quality, where super-resolution models are used to increase the video resolution of downloaded segments, enhancing QoE [30, 64, 66]. Our approach innovates the video codecs of video streaming systems and can be integrated into these orthogonal designs without requiring any changes to the network protocol or application details.

Parallelism in video codec. Parallel processing techniques have been widely exploited in video compression to circumvent the speed limit of processors. There have been sophisticated hardware designs that parallelize vector quantization [43, 45], discrete cosine transform [17, 29, 53], variable length coding [11], and motion estimation [15, 27, 49]. Software-based techniques are categorized into spatial and temporal parallelism. Spatial parallelism [4, 5, 32] processes different regions in a frame concurrently. Temporal parallelism [3, 50] allocates several video frames to each processor. In contrast to these works focusing on parallelism at the GOP level, we allow parallelism at the frame level.

7 Discussion

Hardware requirements. As LiFteR trades the GPU utilization for its frame rate via LFR, the GPU memory might limit its decode speed. However, our experiments have demonstrated that everyday GPUs are sufficient for LiFteR to achieve a frame rate over 30 fps (Figure 15). It is also feasible to further reduce the hardware requirements via neural network compression [10, 34], which is orthogonal to our approach.
Applicable scenarios. As shown in Figure 19, the encoding time of LiFteR, like other learned codecs, is higher than that of decoding. However, in video-on-demand (VoD) streaming, the encoding speed is not critical since video segments are encoded before streaming. Therefore, LiFteR is highly suitable in the VoD streaming scenario.

8 Conclusion

Tight frame referencing has a long history of being adopted in video codecs. However, it is proven ineffective in video streaming with learned codecs, causing a low frame rate. To overcome its limitation, we design LiFteR, a video streaming system that employs a learned codec with loose frame referencing. Our experiments show that LiFteR delivers superior QoE compared to systems using existing traditional and learned codecs consistently.

Acknowledgments

We thank our shepherd Francis Yan and the anonymous reviewers for their valuable feedback, which greatly improves this paper. This work was supported by NSF under the award number NSF CNS 1900875, NSF CNS 2106592, NSF IIS 2140620, NSF IIS 2140645, NSF OAC 1835834, and NSF OAC 2144764 and Army Research Lab under the award number ARMY W911NF-17-2-0196.

References

[1] Eirikur Agustsson, Fabian Mentzer, Michael Tschannen, Lukas Cavigelli, Radu Timofte, Luca Benini, and Luc Van Gool. Soft-to-hard vector quantization for end-to-end learning compressible representations. *arXiv preprint arXiv:1704.00648*, 2017.

[2] Eirikur Agustsson, David Minnen, Nick Johnston, Johannes Balle, Sung Jin Hwang, and George Toderici. Scale-space flow for end-to-end optimized video compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8503–8512, 2020.

[3] I Ahmad, SM Akramullah, ML Liou, and M Kafeel. A scalable off-line mpeg-2 encoder using a multiprocessor machine. *Parallel Computing*, 2001.

[4] Shahriar M Akramullah, Ishfaq Ahmad, and Ming L Liou. A data-parallel approach for real-time mpeg-2 video encoding. *Journal of parallel and distributed computing*, 30(2):129–146, 1995.

[5] Shahriar M Akramullah, Ishfaq Ahmad, and Ming L Liou. Performance of software-based mpeg-2 video encoder on parallel and distributed systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4):687–695, 1997.

[6] Johannes Ballé, Valero Laparra, and Eero P Simoncelli. End-to-end optimized image compression. *arXiv preprint arXiv:1611.01704*, 2016.

[7] Fabrice Bellard. Bpg image format, 2018.

[8] Abdelhak Bentaleb, Ali C Begen, and Roger Zimmermann. Sdndash: Improving qoe of http adaptive streaming using software defined networking. In *Proceedings of the 24th ACM international conference on Multimedia*, pages 1296–1305, 2016.

[9] Gedas Bertasius, Heng Wang, and Lorenzo Torresani. Is space-time attention all you need for video understanding? *arXiv preprint arXiv:2102.05095*, 2021.

[10] Han Cai et al. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[11] Hao-Chieh Chang, Liang-Gee Chen, Yung-Chi Chang, and Sheng-Chieh Huang. A vlsi architecture design of vlc encoder for high data rate video/image coding. In *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 398–401. IEEE, 1999.

[12] Bo Chen, Zhisheng Yan, Hongpeng Guo, Zhe Yang, Ahmed Ali-Eldin, Prashant Shenoy, and Klara Nahrstedt. Deep contextualized compressive offloading for images. In *Proceedings of the 19th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 467–473, 2021.

[13] Bo Chen, Zhisheng Yan, and Klara Nahrstedt. Context-aware image compression optimization for visual analytics offloading. In *Proceedings of the 13th ACM Multimedia Systems Conference (MMSys)*, pages 27–38, 2022.

[14] Bo Chen, Zhisheng Yan, and Klara Nahrstedt. Context-aware optimization for bandwidth-efficient image analytics offloading. *ACM Transactions on Multimedia Computing, Communications and Applications*, 2023.

[15] Jie Chen and KJ Ray Liu. A complete pipelined parallel cordic architecture for motion estimation. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45(5):653–660, 1998.

[16] Tong Chen, Haojie Liu, Qiu Shen, Tao Yue, Xun Cao, and Zhan Ma. Deepcoder: A deep neural network based video compression. In *2017 IEEE Visual Communications and Image Processing (VCIP)*, pages 1–4. IEEE, 2017.

[17] Ching-Te Chiu and KJ Liu. Real-time parallel and fully-pinelined two-dimensional dct lattice structures with application to hdtv systems. Technical report, 1991.

[18] Hyomin Choi and Ivan V Bajić. Deep frame prediction for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(7):1843–1855, 2019.

[19] DASH Reference Client. Dash reference client, 2023.

[20] David Curry. Video streaming app revenue and usage statistics (2023), 2023.

[21] Mallesham Dasari, Kumara Kahatapitiya, Samir R Das, Aruna Balasubramanian, and Dimitris Samaras. Swift: Adaptive video streaming with layered neural codecs. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 103–118, 2022.

[22] Kuntai Du, Ahsan Pervaiz, Xin Yuan, Aakanksha Chowdhery, Qizheng Zhang, Henry Hoffmann, and Junchen Jiang. Server-driven video streaming for deep learning inference. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 557–570, 2020.

[23] FCC. Measuring broadband america - july 2012, 2012.

[24] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 131–144, 2015.

[25] Amirhossein Habibian, Ties van Rozendaal, Jakub M Tomczak, and Taco S Cohen. Video compression with rate-distortion autoencoders. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 7033–7042, 2019.

[26] Yanbin Hao, Shuo Wang, Pei Cao, Xinjian Gao, Tong Xu, Jinmeng Wu, and Xiangnan He. Attention in attention: Modeling context correlation for efficient video classification. *IEEE Transactions on Circuits and Systems for Video Technology*, 32(10):7120–7132, 2022.

[27] Zhong-Li He, Chi-Ying Tsui, Kai-Keung Chan, and Ming L Liou. Low-power vlsi design for motion estimation using adaptive pixel truncation. *IEEE Transactions on circuits and systems for video technology*, 10(5):669–678, 2000.

[28] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 393–406, 2017.

[29] Ig-kyun Kim, Jin-jong Cha, and Han-jin Cho. A design of 2-d dct/idct for real-time video applications. In *ICVC'99. 6th International Conference on VLSI and CAD (Cat. No. 99EX361)*, pages 557–559. IEEE, 1999.

[30] Jaehong Kim, Youngmok Jung, Hyunho Yeo, Juncheol Ye, and Dongsu Han. Neural-enhanced live streaming: Improving live video ingest via online learning. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 107–125, 2020.

[31] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[32] Pasi Kolinummi, Juha Sarkijarvi, T Hamalainen, and Jukka Saarinen. Scalable implementation of h. 263 video encoder on a parallel dsp system. In *2000 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 551–554. IEEE, 2000.

[33] Didier J Le Gall. The mpeg video compression algorithm. *Signal Processing: Image Communication*, 4(2):129–140, 1992.

[34] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[35] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. Gso-simulcast: global stream orchestration in simulcast video conferencing systems. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 826–839, 2022.

[36] Luyang Liu, Hongyu Li, and Marco Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[37] Zhenyu Liu, Xianyu Yu, Yuan Gao, Shaolin Chen, Xiangyang Ji, and Dongsheng Wang. Cu partition mode decision for hevc hardwired intra encoder using convolution neural network. *IEEE Transactions on Image Processing*, 25(11):5088–5103, 2016.

[38] Guo Lu, Chunlei Cai, Xiaoyun Zhang, Li Chen, Wanli Ouyang, Dong Xu, and Zhiyong Gao. Content adaptive and error propagation aware deep video compression. In *European Conference on Computer Vision*, pages 456–472. Springer, 2020.

[39] Guo Lu, Wanli Ouyang, Dong Xu, Xiaoyun Zhang, Chunlei Cai, and Zhiyong Gao. Dvc: An end-to-end deep video compression framework. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11006–11015, 2019.

[40] Wei Luo and Bo Chen. Neural image compression with quantization rectifier. In *ICML 2023 Workshop Neural Compression: From Information Theory to Applications*, 2023.

[41] Hongzi Mao, Ravi Netravali, and Mohammad 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.

[42] Alexandre Mercat, Marko Viitanen, and Jarno Vanne. Uvg dataset: 50/120fps 4k sequences for video codec analysis and development. In *Proceedings of the 11th ACM Multimedia Systems Conference*, pages 297–302, 2020.

[43] Toshiyuki Nozawa, Masahiro Konda, Masanori Fujibayashi, Makoto Imai, Koji Kotani, Shigetoshi Sugawa, and Tadahiro Ohmi. A parallel vector-quantization processor eliminating redundant calculations for real-time motion picture compression. *IEEE Journal of Solid-State Circuits*, 35(11):1744–1750, 2000.

[44] Hughes Perreault, Guillaume-Alexandre Bilodeau, Nicolas Saunier, and Maguelonne Héritier. Spotnet: Self-attention multi-task network for object detection. In *2020 17th Conference on Computer and Robot Vision (CRV)*, pages 230–237. IEEE, 2020.

[45] KS Prashant and V John Mathews. A massively parallel algorithm for vector quantization. In *Proceedings of the 1995 NASA Space and Earth Sciences Workshop*. Citeseer, 1995.

[46] Prajit Ramachandran, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. *Advances in neural information processing systems*, 32, 2019.

[47] Anurag Ranjan and Michael J Black. Optical flow estimation using a spatial pyramid network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4161–4170, 2017.

[48] Oren Rippel, Alexander G Anderson, Kedar Tatwawadi, Sanjay Nair, Craig Lytle, and Lubomir Bourdev. Elf-vc: Efficient learned flexible-rate video coding. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 14479–14488, 2021.

[49] Alexander Roach and Alireza Moini. Vlsi architecture for motion estimation on a single-chip video camera. In *Visual Communications and Image Processing 2000*, volume 4067, pages 1441–1450. SPIE, 2000.

[50] Ke Shen and Edward J Delp. A spatial-temporal parallel approach for real-time mpeg video compression. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 2, pages 100–107. IEEE, 1996.

[51] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. Bola: Near-optimal bitrate adaptation for online videos. *IEEE/ACM Transactions on Networking*, 28(4):1698–1711, 2020.

[52] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[53] M-T Sun, T-C Chen, and Albert M Gottlieb. Vlsi implementation of a 16* 16 discrete cosine transform. *IEEE transactions on circuits and systems*, 36(4):610–617, 1989.

[54] Suramya Tomar. Converting video formats with ffmpeg. *Linux Journal*, 2006(146):10, 2006.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[56] Haiqiang Wang, Weihao Gan, Sudeng Hu, Joe Yuchieh Lin, Lina Jin, Longguang Song, Ping Wang, Ioannis Katsavounidis, Anne Aaron, and C-C Jay Kuo. Mcl-jcv: a jnd-based h. 264/avc video quality assessment dataset. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 1509–1513. IEEE, 2016.

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

[58] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[59] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 416–431, 2018.

[60] Tianfan Xue, Baian Chen, Jiajun Wu, Donglai Wei, and William T Freeman. Video enhancement with task-oriented flow. *International Journal of Computer Vision*, 127(8):1106–1125, 2019.

[61] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Alexander Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *NSDI*, volume 20, pages 495–511, 2020.

[62] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with hierarchical quality and recurrent enhancement. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6628–6637, 2020.

[63] Ren Yang, Fabian Mentzer, Luc Van Gool, and Radu Timofte. Learning for video compression with recurrent auto-encoder and recurrent probability model. *IEEE Journal of Selected Topics in Signal Processing*, 15(2):388–401, 2020.

[64] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 645–661, 2018.

[65] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[66] Anlan Zhang, Chendong Wang, Bo Han, and Feng Qian. Yuzu:neural-enhanced volumetric video streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 137–154, 2022.

[67] Rui-Xiao Zhang, Tianchi Huang, Ming Ma, Haitian Pang, Xin Yao, Chenglei Wu, and Lifeng Sun. Enhancing the crowdsourced live streaming: a deep reinforcement learning approach. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 55–60, 2019.

[68] Yuanxing Zhang, Pengyu Zhao, Kaigui Bian, Yunxin Liu, Lingyang Song, and Xiaoming Li. Drl360: 360-degree video streaming with deep reinforcement learning. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1252–1260. IEEE, 2019.

A H.264 and H.265 Commands

We denote the frame width, the frame height, the frame rate, the GOP size, the compression quality, and the output filename as w, h, fps, GOP, Q, and $output$. The command for compressing a video from Pipe using 'veryfast', 'medium' and 'veryslow' modes of x264 are listed as follows, respectively.

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset veryfast -tune zerolatency -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output
```

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset medium -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output
```

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx264 -pix_fmt yuv420p -preset veryslow -crf Q -g GOP -bf 2 -b_strategy 0 -sc_threshold 0 -loglevel debug output
```

The command for compressing a video from Pipe using 'veryfast', 'medium' and 'veryslow' modes of x265 are listed as follows, respectively.

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset veryfast -tune zerolatency -x265-params "crf=Q:keyint=GOP:verbose=1" output
```

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset medium -x265-params "crf=Q:keyint=GOP:verbose=1" output
```

```
ffmpeg -y -s wxh -pixel_format bgr24 -f rawvideo -r fps -i pipe: -vcodec libx265 -pix_fmt yuv420p -preset veryslow -x265-params "crf=Q:keyint=GOP:verbose=1" output
```