Tampere University

Jussi Iho

LOW-LATENCY AND HIGH-BANDWIDTH VIDEO STREAM DELIVERY

Master of Science Thesis
Information Technology and Communication
April 2021

ABSTRACT

Jussi Iho: Low-Latency and High-Bandwidth Video Stream Delivery
Master of Science Thesis
Tampere University
Information Technology
April 2021

Most conventional video formats have been designed to produce high compression ratios through the use of sophisticated video coding methods, resulting in dramatic reductions of required bandwidth. The complexity of these standards is however also their weakness, as the inherent compute intensity adds to the video latency. Therefore, low-latency video delivery continues to be a challenging problem, especially for low-power mobile and IoT devices.

An alternative approach has been proposed, where certain texture compression formats, commonly used in computer graphics, would be used for real-time video compression at a very low latency. These formats would allow the use of fast, hardware-accelerated render-time decoding, while also lending themselves well to low-latency GPU encoding and parallel computation in general.

The main drawback would come in the form of low compression ratios, but it has become a lesser issue, thanks to the recent advent of new high-bandwidth wireless networking technologies, such as the 802.11ax (WiFi 6) and 5G standards. Regardless, they still have only an enabling role, as general-purpose platforms still depend on largely software-based network I/O solutions.

The objective of this work, is in studying the technical aspects of texture-compressed stream delivery, and finding the best strategies for the performance optimization of the Linux kernel network stacks using general-purpose hardware. A video streaming pipeline optimized for texture formats is proposed, utilizing multi-threading, GPU acceleration, and network stack performance tuning.

As a result, the pipeline was found to be capable of reaching very low latencies in the case of high-bandwidth networks, when extrapolating from performance measurements in localhost TCP and UDP tests. As an example, a 2160p frame encoded in the BC1 format, could be delivered with a total end-to-end latency of under 10 ms, although it would require a 10 Gbit/s network and a high core count -CPU. The achieved bandwidths were 31.7 Gbit/s and 25.3 Gbit/s for the proposed TCP and UDP implementations respectively. As the latency is roughly proportional to the frame size and network bandwidth, using a higher compression ratio format or more bandwidth could easily bring the 4320p performance to a similar level with the 2160p results.

The use of texture compression in video delivery was concluded to be on the edge of viability for the aforementioned low-power systems in wireless networks. The limiting factors are the network performance and the CPU-overheads in the Linux network stack. While significant improvements in device compute performances are unlikely to be seen in the near future, advancements in the networking capabilities of consumer hardware could, however, be enough to make texture compressed video delivery a reality.

Keywords: texture compression, latency, performance, networking, Linux

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Jussi Iho: Matalan viiveen ja korkean kaistanleveyden videovirran siirto
Diplomityö
Tampereen yliopisto
Tietotekniikka
Huhtikuu 2021

Useimmat konventionaaliset videoformaatit on suunniteltu tuottamaan korkeita kompressiosuhteita käyttäen hyödyksi hienostuneita videokoodausmenetelmiä johtaen dramaattisiin vähennyksiin tarvittavassa kaistanleveydessä. Näiden standardien kompleksisuus on kuitenkin myös niiden heikkous, koska niiden laskentaintensiivisyys kasvattaa videon viivettä. Näin ollen matalaviiveinen videon siirto on edelleen haastava ongelma erityisesti matalatehoisissa mobiili- ja IoT-laitteissa.

Vaihtoehtoista menetelmää on ehdotettu, missä usein tietokonegrafiikassa käytettyjä tekstuuriformaatteja voitaisiin käyttää reaaliaikaisessa videokompressiossa erittäin matalalla viiveellä. Nämä formaatit mahdollistaisivat nopean laitteistokiihdytetyn renderöintiaikaisen dekoodauksen ja samalla soveltuisivat hyvin matalan viiveen GPU-enkoodaukseen ja yleisesti rinnakkaiseen laskentaan.

Ensisijainen haittapuoli olisi matalissa pakkaussuhteissa, mutta tästä on viime aikoina tullut vähäisempi ongelma kiitos viimeaikaisten kehitysten korkean kaistanleveyden langattomissa verkkoteknologioissa kuten 802.11ax- (WiFi 6) ja 5G-standardeissa. Niillä on kuitenkin ainoastaan mahdollistava rooli, sillä yleiskäyttöiset laskenta-alustat ovat edelleen suurelta osin riippuvaisia ohjelmistopohjaisista verkko-I/O-ratkaisuista.

Tämän työn tavoite, on tutkia tekstuurikompressoidun videovirran siirron teknisiä näkökohtia ja löytää parhaimmat menetelmät Linux-järjestelmien verkkopinojen suorituskykyoptimointiin käyttäen yleiskäyttöisiä laitteistoja. Lisäksi ehdotetaan tekstuuriformaateille optimoitua videovirtajärjestelmää, jossa hyödynnetään monisäikeistystä, GPU-kiihdytystä ja verkkopinon hienosäätöä.

Lopputuloksena järjestelmän havaittiin olevan kykenevä saavuttamaan hyvin matalia viiveitä korkean kaistanleveyden verkoissa, kun ekstrapoloidaan paikallisten TCP- ja UDP-testien tuloksista. Esimerkkinä BC1-formaatissa enkoodatun 2160p-videokuva pystytiin siirtämään alle 10 ms kokonaislatenssilla, vaikkakin se edellytti 10 Gbit/s -verkkoa ja korkeaa CPU-ydinmäärää. Saavutetut kaistanleveydet olivat 31.7 Gbit/s TCP:lle ja 25.3 Gbit/s UDP-toteutukselle. Koska viive on suurinpiirtein verrannollinen videokuvan kokoon ja verkon kaistanleveyteen, korkeampi pakkaussuhde tai kaistanleveys voisi tuoda 4320p-suorituskyvyn samankaltaiselle tasolle 2160p-tulosten kanssa.

Johtopäätöksenä tekstuurikompression käyttö videon siirrossa todettiin olevan toteutettavuuden rajoilla edellä mainituissa matalatehoisissa järjestelmissä langattomissa verkoissa. Rajoittavat tekijät olivat verkon suorituskyky ja korkea CPU-käyttö Linuxin verkkopinossa. Vaikka huomattavat parannukset laitteistojen laskennallisessa suorituskyvyssä ovatkin epätodennäköisiä lähitulevaisuudessa, edistysaskeleet kuluttajalaitteiden verkko-ominaisuuksissa voisivat kuitenkin olla tarpeeksi tekemään tekstuurikompressiosta videon siirrossa todellisuutta.

Avainsanat: tekstuurikompressio, viive, suorituskyky, tietoverkot, Linux

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This master's thesis is dedicated to my brother's dog Nasu, who kindly posed for some images used here. She has always been a good sport. Additionally, Jakub Zadnik needs to be acknowledged for his contribution in the development of a GPU encoder and for sharing his expertise in matters related to video coding.

Developing the described video delivery system was a project of over six months, which required arduous research on many sub-topics unfamiliar to the author. Some of these, such as Linux kernel networking, GPU-accelerated stream processing, or special considerations related to texture formats in video compression, were in themselves questions extensive enough to write a master's thesis on them alone. Bearing that in mind, the information condensed to this work should still be a useful and practical introduction to any reader studying texture-compressed stream delivery, or high-performance networking in general-purpose Linux systems.

Admittedly, it is easy to question whether limiting the scope to only software-based solutions is sensible, as some specialized networking hardware could make them irrelevant. However, such investments are not always within the realm of possibility. Furthermore, the author sees that the essence of engineering lies in innovating to push the envelope of the limited tools and resources currently available.

The work for this publication was funded by ECSEL Joint Undertaking (JU) under grant agreement No 783162 (FitOptiVis). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Netherlands, Czech Republic, Finland, Spain, Italy.

Tampere, 26th April 2021

Jussi Iho

CONTENTS

1 Introduction 1
2 Texture-Compressed Video Streaming 3
2.1 Texture Compression Formats 3
2.2 Technical Considerations 7
3 High-Performance Networking 12
3.1 Linux Network Stack 12
3.2 Transport Layer Protocols 17
3.2.1 User Datagram Protocol 17
3.2.2 Transmission Control Protocol 19
3.3 Network Performance Tuning 22
3.3.1 Multi-Threaded Sockets 22
3.3.2 Socket Buffers 24
3.3.3 Packet Fragmentation 25
3.3.4 Busy-Poll Receiving 26
4 Proposed System 28
4.1 Architecture Overview 28
4.2 Video Coding 31
4.2.1 Encoder and Decoder 32
4.2.2 Video Formats 33
4.3 Stream Delivery 35
4.3.1 UDP Stream Delivery 37
4.3.2 TCP Stream Delivery 41
4.4 OpenGL Buffer Transfers 43
5 Results and Evaluation 47
5.1 Performance Measurements 47
5.2 Future Work 54
6 Conclusions 57
References 59

1 INTRODUCTION

Media streaming has become a significant and prevalent application of the modern Internet, especially over the past decade. The same time span has also seen the popularization of cloud and remote computing concepts as a method of building scalable and accessible Internet applications, but so far little of this has translated into improvements in latency. On the contrary, achieving ultra-low latency in streaming of high-bandwidth media, such as video, continues to be a non-trivial challenge.

A number of interesting latency-sensitive streaming applications have been emerging in recent years. Most challenging are those that aggregate other requirements, such as compute- or bandwidth-intensity, mobility and energy efficiency. Besides the delivery of real-time video in general, some points of interest include interactive remote rendering, virtual or augmented reality, machine vision, autonomous transportation and, teleoperation applications [1].

Other broader technological trends and concepts also link to advancements in low-latency communications. Most notable are mobile computing and Internet of Things (IoT) systems, that are required to operate on thin power budgets or modest compute capabilities. To counter this, considerable efforts have been directed at distributed computational offloading solutions over networks. One of these is the concept of multi-access edge computing (MEC), which aims to bring compute resources to the immediate vicinity of mobile radio access networks (RAN) for the use of client devices with minimal latency [2].

In the recent years, compute performances have mostly been increasing only through increased parallelization and hardware-offloading capabilities. General-purpose computing on GPUs (GPGPU) has especially become the prevalent solution for processing large volumes of data [3]. I/O performances on the other hand have seen much more improvement in terms of not only memory speeds, but also network technologies [4]. Consequently, a strong case could be built to say that the future of information technology is more-and-more in interconnected and distributed compute systems [5].

Data centers and other industrial and enterprise settings have enjoyed over 10–100 Gbit/s wired networking for a long time now [6]. Wireless networks are also being boosted by the introduction of the new 802.11ax (Wi-Fi 6) and 5G millimeter-wave technologies. These new standards are now promising comparable (theoretical) bandwidths of close to 10

Gbit/s and network latencies in the order of milliseconds. [7] A gap has formed between the compute and network performance, where the network bandwidths have exploded while the compute side has stagnated [4].

The fundamental issue of latency lies in the fact that it is not a singular problem area, but a sum of many separate challenges concerning compute, transfer and display capabilities [7]. Despite of the ongoing technological advancements in networking, they only have an enabling role in the latency-optimization of the media streaming applications.

The focus has to also shift towards optimizing end-host performance and especially software stacks, which besides the application code itself, also extend to operating system level [6]. In MEC contexts this is further underlined by the fact, that although servers may make use of various specialized hardware acceleration and interconnect solutions, low-end consumer devices currently still have to rely on more simplistic general-purpose hardware.

The objective of this work is to study methods of maximizing low-latency high-bandwidth stream performances in general-purpose Linux devices and high-speed wired and wireless networks. For the purposes of the work, a real-time video streaming system is developed utilizing kernel network stack optimizations, heavy multi-threading and GPU acceleration. Additional considerations include GPU buffer management schemes and portability to mobile platforms.

As a secondary goal, practical considerations of using texture compression formats in ultra low-latency video streaming are also discussed. These simple formats have a great deal of computational benefits over conventional video codecs, but come at the cost of very low compression ratios generating extremely high bandwidth requirements at high resolutions. In other words, texture compression in frame-by-frame video streaming applications makes an ideal candidate for demonstrating high-performance networking in practice.

Low-latency and high-bandwidth video stream delivery is studied in this work from both theoretical and practical aspects. Chapter 2 discusses texture compression formats in the context of video coding including their special properties and their practical significance. Chapter 3 looks into performance aspects of networking, focusing on transmission medium, transport protocol and Linux network stack impacts to end-to-end performance. The information is then applied in Chapter 4, where the proposed video streaming system is walked through starting from its architecture and moving into video coding, stream handling and buffer management aspects. The performance results are presented in Chapter 5 with additional analysis of their causes, implications for any real-world applications, and future work proposals. The final conclusions of the work are included in Chapter 6.

2 TEXTURE-COMPRESSED VIDEO STREAMING

Traditionally, most video compression formats have been designed to maximize their compression ratio for a given level of quality. While this has made streaming and storing large amounts of video data over Internet much more economically viable, the main drawback comes in the form of compute time required in coding of video.

As a response to the compute bottleneck issue, much work has been done to optimize software codecs, and some alleviation has also been found from GPU and hardware acceleration. Unfortunately, especially low-cost and low-power systems, such as some mobile and IoT devices, remain problematic.

An alternative solution has been proposed, where instead of investing in local compute resources, high-performance networking capabilities are utilized in conjunction with lightweight texture compression formats to deliver video at the lowest possible latency. The goal of this work is to develop a video streaming pipeline, which demonstrates the use of texture compressed video in practice.

2.1 Texture Compression Formats

In modern computer graphics, large numbers of high-resolution textures are used in rendering complicated 3D environments on a normal basis. Because GPUs have only a limited amount of VRAM for storing these textures, various compression schemes have been developed to reduce their memory footprint.

The drawback of using compressed textures manifests in requiring extremely fast decompression during rendering. Due to the fast decoding requirement, texture compression primarily relies on relatively simple adaptive color quantization-based approaches for lossy compression. The contrast is stark when compared to e.g. frequency domain analysis or motion compensation –based coding schemes utilized in conventional video compression formats such as H.264 [8]. The resulting compression ratios of the texture formats are much lower, typically in the range of 1:3 to 1:6 for the formats covered here.

Despite the low compression ratio, texture compression has other significant advantages. One such benefit is the fact that the compression ratio is constant, because these formats utilize constant-dimension pixel blocks that get compressed into a fixed number of bits.

The compressed blocks are also completely independent of each other. These properties result in predictable memory use –patterns, that have great favourable effects in both compute parallelization and memory management.

Decompression is also accelerated in hardware, with virtually all graphics hardware supporting many different formats [9]. From an application standpoint, there is little difference between using compressed or uncompressed textures, as the decoding occurs transparently at render-time.

The aforementioned benefits of uniform block formats also apply to real-time encoding, although there is no hardware support available for it. Despite of that, a well-optimized encoder running on a moderately powerful GPU can achieve very low encode latencies [9]. Even software encoders running on modern mid-range CPUs have been shown to be capable of encoding 4320p frames in the order of milliseconds [10]. It should however be noted, that a low-latency encoder may produce lower quality results compared to an offline encoder, even though the compression formats would be the same [11].

One of the more widely adopted texture compression formats at least in PC systems is the S3 Texture Compression (S3TC) family of formats. Somewhat confusingly, they are known either as Block Compression (BC) formats in the Direct3D API [12], or DXT formats in the OpenGL API [13].

As mentioned, texture compression formats operate on pixel blocks that are compressed into a fixed number of bits. In the case of the S3TC BC1 (DXT1) format, blocks have a constant size of 4-by-4 pixels, that are encoded into 64 bits. The first 32 bits contain two 16-bit color values in RGB 565 format representing the local minimum and maximum color values of the block. The last 32 bits in the compressed block store sixteen 2-bit values, which are used to linearly interpolate in the local color space defined by the minimum and maximum. [13] Therefore, all color values inside the block effectively get quantized to four distinct color levels between the minimum and maximum values. Since an uncompressed 24-bit RGB pixel block of equivalent size has a memory footprint of 384 bits, the resulting compression ratio of the BC1 format is 1:6.

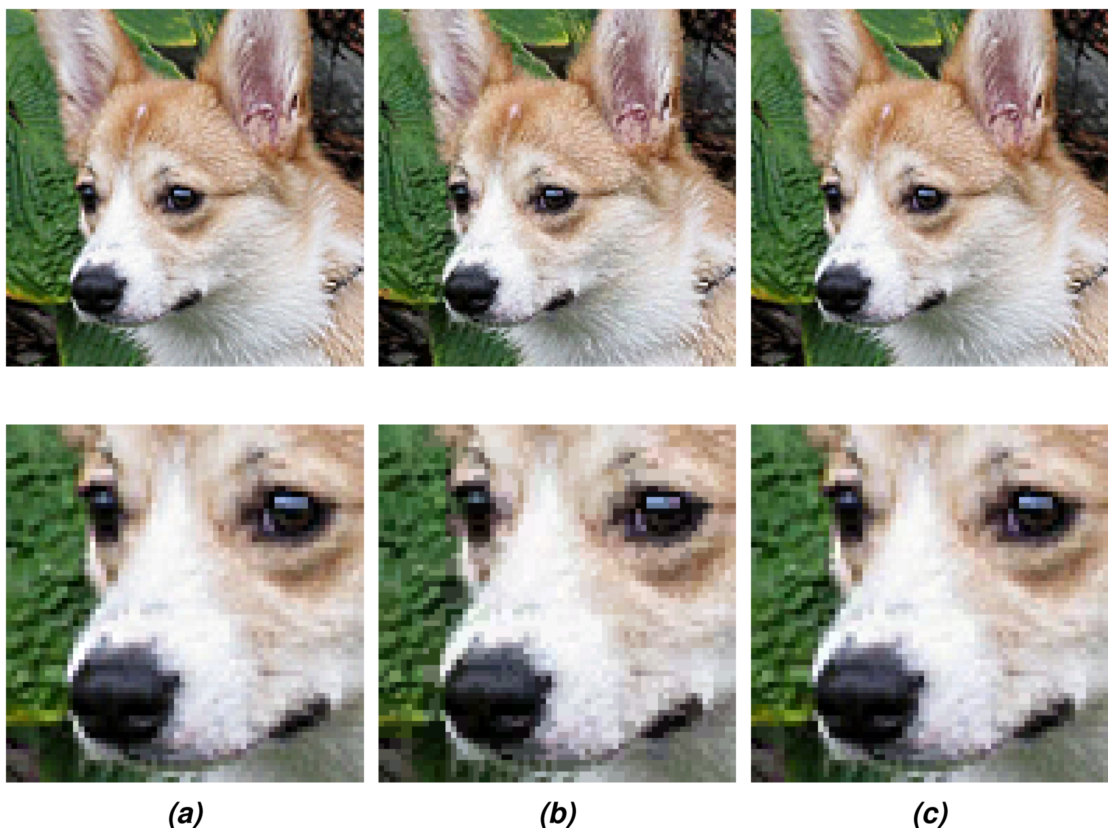
S3TC BC3 (DXT5) extends the BC1 format by including more complete alpha channel support, where BC1 supported only optional 1-bit alpha channel by modifying the way encoded data is interpreted. BC3 uses 128 bits per 4x4 pixel block by encoding the color data to a standard 64-bit BC1 block and by encoding the alpha channel to a separate 64-bit block. This alpha-block includes a 16-bit block storing two 8-bit minimum and maximum alpha values, and a 48-bit block containing sixteen 3-bit values used to once again linearly interpolate between the stored minimum and maximum. [13] The block size is twice as large and thus has a compression ratio of 1:3.

Since video formats do not under normal circumstances have any need for alpha channel

data, BC3 format is not directly relevant from the perspective of this work. However, it has been previously proposed, that the BC3 format could be repurposed to deliver YCoCg data instead of RGBA with a quality improvement over the standard BC1 or BC3 formats. This can be done by converting RGB data to YCoCg color space, storing the Y channel into the 64-bit alpha block, and using the other 64-bit block to store the two chrominance channels. The decoding and conversion back to RGB could then performed render-time using a simple fragment shader. [11]

As a complicating factor, the latter 64-bit block is expected to store 3-channel RGB data, which is what a texture sampler would support. To avoid wasting the third channel completely when storing two-channel CoCg data, the remaining channel can be used to store a separate scale factor, which is used to reduce quantization errors in the chrominance channels during the coding process. [11]

Rudimentary quality comparison of an RGB 24-bit image encoded in BC1 and YCoCg-BC3 formats is provided in Figure 2.1 using a real-time low-latency GPU encoder. It should however be noted, that there does not exist any singular standard texture encoding method, and different encoders will yield better results at the cost of encoding time. In conventional computer graphics applications, texture compression is typically even done by a much slower offline encoder for the best possible quality.



(a) ***(b)*** ***(c)***

Figure 2.1. *Quality comparison of (a) uncompressed 24-bit RGB, (b) BC1 and (c) YCoCg-BC3 formats using a 192x192 pixel image and a zoomed 64x64 pixel section.*

Nevertheless, the comparison is still sufficient for demonstrating the level of quality that could be expected from a reasonable real-time implementation. Most compression artifacts are rather subtle, but some artifacting can be seen in edge areas of the BC1 image. Especially blocks, that contain three or more distinct shades of color, are most affected, because the range of possible color values is limited into a few levels in the interpolation axis defined by its two endpoint color values. In the case of BC1 using RGB 565 format, the red and blue shades naturally compress more poorly than green ones, as the latter channel gets one additional bit of precision over others.

Some images with softer color gradients compress quite well in the BC1 format, but sharp edges remain a problem. The YCoCg-BC3 format is the better fit for most natural images and could even be argued to have an imperceptible difference in quality. This is to be expected, as BC3 uses more bits per pixel than then BC1 by a factor of two. Both formats also benefit from very large resolutions, because the block sizes remain constant and become smaller in respect to the image as a whole making the artifacts less noticeable.

Other formats, which could be considered in the context of this work, include the Ericson Texture Compression (ETC) [14, 15] and Adaptive Scalable Texture Compression (ASTC) formats [16]. These are extensively used by mobile platforms and are consequently supported by the OpenGL ES API [17]. The regular OpenGL API also includes support for ETC2 [18], which together with ETC1 shares similar properties with the S3TC BC formats. ASTC introduces other interesting properties, such as variable block sizes resulting in adjustable quality levels and compression ratios ranging from 1:3 to 1:27.

In addition to actual texture formats, it would of course also be possible to stream other pixel formats, such as the already mentioned 16-bit RGB 565, or any 4:2:0 or 4:2:2 -subsampled luminance-chrominance formats as well. Technical properties would in this case be very similar, meaning low, but constant compression ratio, predictable memory footprints and very fast video coding or rendering.

Due to the pixel block sizes being fixed, it is not generally possible to directly support arbitrary resolutions that are not multiples of the block size. While most widely used resolutions of today are already divisible by the most common block size of four, such as 720p or 1080p, a more general method is still desirable. An encode time padding –approach was chosen as the solution, which is further discussed in Chapter 4.2.

Texture compression formats were of course originally designed for storing static image data, and not for coding live video streams. It is in principle possible that encoded output could include some instability in some edge cases when movement is present in frame. Visual inspections done in connection with this work, or other literature, have fortunately not indicated this to be a prevalent issue.

Instead, the remaining problems are more technical in nature, and involve the practical

performance limitations of general-purpose hardware when developing high-bandwidth video pipelines based on texture compression. Assuming sufficient network I/O performance, the other properties of the texture compression formats can luckily be made use of to offset some of the increase in bandwidth.

2.2 Technical Considerations

The primary technical issues of applying texture compression to video stream delivery, rise from the low compression ratios, requiring very high stream bandwidths. While a very high-resolution, high-frame rate H.264 video stream may typically at most have a bitrate in tens or hundreds of megabits per second, the stream bandwidths studied here can exceed those by two or three orders of magnitude.

Examples of these frame byte sizes and stream bandwidths of various texture formats are presented in Tables 2.1 and 2.2 respectively. The latter table shows that a compression ratio of 1:3 or more would be required to deliver 4320p video at 30 Hz frame rate in a 10 Gbit/s network. Raising the frame rate to 60 Hz would double the bandwidth and make BC1 and some of the ASTC formats the only viable alternatives.

The discussion on minimum bandwidth requirements is naturally assuming optimal conditions with minimal bandwidth losses due to packet retransmissions, processing overheads, and similar effects. The rest of the hardware and software stack involved also has to also match the performance of the network, which can be regarded as the true problem.

Table 2.1. *Frame sizes (in MB) corresponding to some frame formats and resolutions.*

	Ratio	**360p**	**720p**	**1080p**	**1440p**	**2160p**	**4320p**
RGB 24-bit	1:1	0.69	2.76	6.22	11.1	24.9	99.5
YUV (4:2:0)	1:2	0.35	1.38	3.11	5.53	12.4	49.7
S3TC BC3	1:3	0.23	0.92	2.07	3.69	8.29	33.2
S3TC BC1	1:6	0.12	0.46	1.04	1.84	4.15	16.6

Table 2.2. *Required minimum bandwidths (in Gbit/s) for the delivery of 30 Hz video.*

	Ratio	**360p**	**720p**	**1080p**	**1440p**	**2160p**	**4320p**
RGB 24-bit	1:1	0.17	0.66	1.49	2.65	5.97	23.9
YUV (4:2:0)	1:2	0.08	0.33	0.75	1.33	2.99	11.9
S3TC BC3	1:3	0.06	0.22	0.50	0.88	1.99	7.96
S3TC BC1	1:6	0.03	0.11	0.25	0.44	1.00	3.98

It needs to be noted, that the key metric here is not the bandwidth itself, but the end-to-end latency, which is determined as the time from a frame entering the pipeline encoder to the frame being fully delivered, decoded and rendered on a display device. The end-to-end latency is of course a further combination of the encode and delivery latencies, with the delivery part turning out to be the most dominant of all, and therefore the focus of this work too.

Real-world applications will of course have to deal with additional sources of latency, such as video source delays in the forms of video capture or rendering, and also display refresh latencies. Additionally, interactive applications would unavoidably have to handle delays in user inputs. For the purposes of the work, these are however considered to be separate issues, although many of the latency optimization strategies discussed in the following sections would be applicable to them as well.

Regardless, the bandwidths calculated in Table 2.2 are only minimum values needed to reach the given frame rate of 30 frames per second. Since the delivery latency is inversely proportional to the pipeline throughput, the truly ideal network bandwidth would be as “high as possible". If, for example, a 4320p frame was encoded in the BC1 format and thus had a byte size of 16.6 MB, it would result in a minimum delivery latency of 13.3 ms in a 10 Gbit/s network. Quadrupling the network capacity to 40 Gbit/s would in such case cut the network delay to only 3.3 ms.

The actual pipeline throughput and the resulting delivery latency is going to be affected by a wide range of factors besides the raw network bandwidth. In practice, the I/O capacities of end-host systems will be limited by factors such as CPU performance, which underlines the need for pipeline parallelization. Luckily, texture compression formats lend themselves extremely well for just that.

The basic data units in texture compression are the fixed-size pixel blocks, which are explicitly designed for parallel decoding, fully independent of each other. The benefits are not felt in only the decoding end either, but in all stages of the pipeline from video coding to transfer.

Assuming the network I/O is largely CPU-bound, utilizing multi-threading becomes pivotal, as it is the only way to reduce the delivery latency. This can for example be accomplished by dividing the video stream into multiple sub-streams with each thread being given a separate tile of the frame in a divide and conquer –fashion. The most sensible basis for partitioning these tiles would then be to follow pixel block boundaries.

Thanks to the constant compression ratio, encoded data can be stored in simple array formats of pre-determined sizes and topologies. Managing and partitioning such arrays using memory offsets in the various stages of the streaming pipeline would therefore be very easy and efficient.

Besides the decoding being trivial, fixed block –formats also make the writing of real-time GPU encoders very straightforward. At its simplest the encoding stage can be implemented in just a compute kernel or shader pass over the input frame. The only potential problem area rises in managing buffer transfers between the GPU and host system memory, where data has to be streamed to from the network. The PCIe-bus therefore becomes a potential bottleneck in discrete GPU systems, as even compressed frames can be up to tens of megabytes in size.

The CPU is left with the task of managing the various memory operations and GPU buffer transfers. A principle-level summary of the data flow through all the possible devices using memory mapping is presented in Figure 2.2. The presented pipeline comprises all stages of a complete pipeline, although the scope of this work has been further narrowed down to the delivery and decoding stages of it.

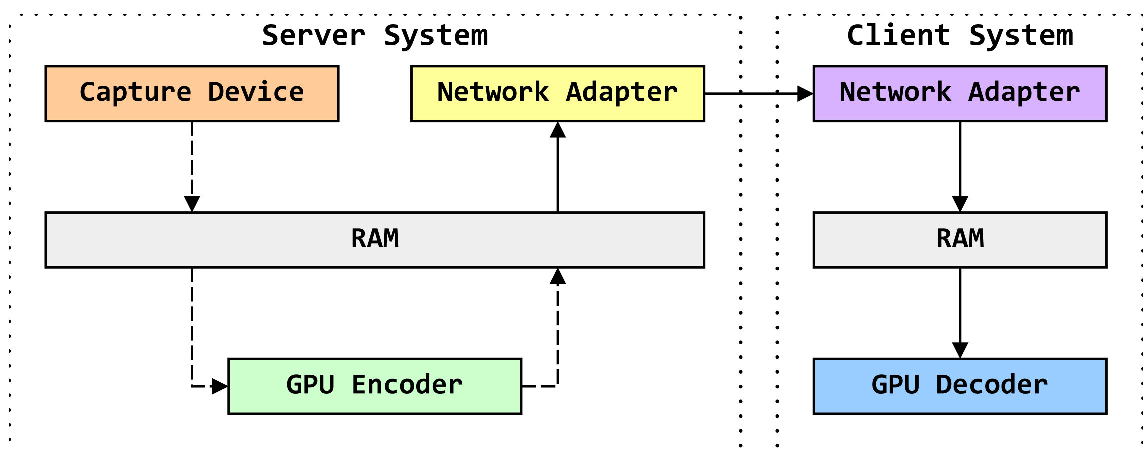


Figure 2.2. *The data flow in a hypothetical complete pipeline showing relevant devices, with the non-dashed section being the narrowed-down pipeline covered later in the work.*

The buffer transfers have potential to be the largest bottleneck after the network I/O. Besides the fact that they could limit the bandwidth of the entire pipeline, they can also add to its latency, unless proper measures are found to counteract it.

As demonstrated in Figure 2.3, the end-to-end latency of the pipeline does not have to be the sum of its parts. Even without increasing transfer bandwidths or compute capacities, it is possible to reduce the latency by introducing overlaps in the various processing and transfer stages of the pipeline.

Allowing overlaps in video coding and transfers is yet another major benefit of using texture compression formats. Ideally, the encoding application would encode new blocks while others were being transferred and sent over network. Similarly, the decoding application would receive new blocks while transferring others to GPU for rendering.

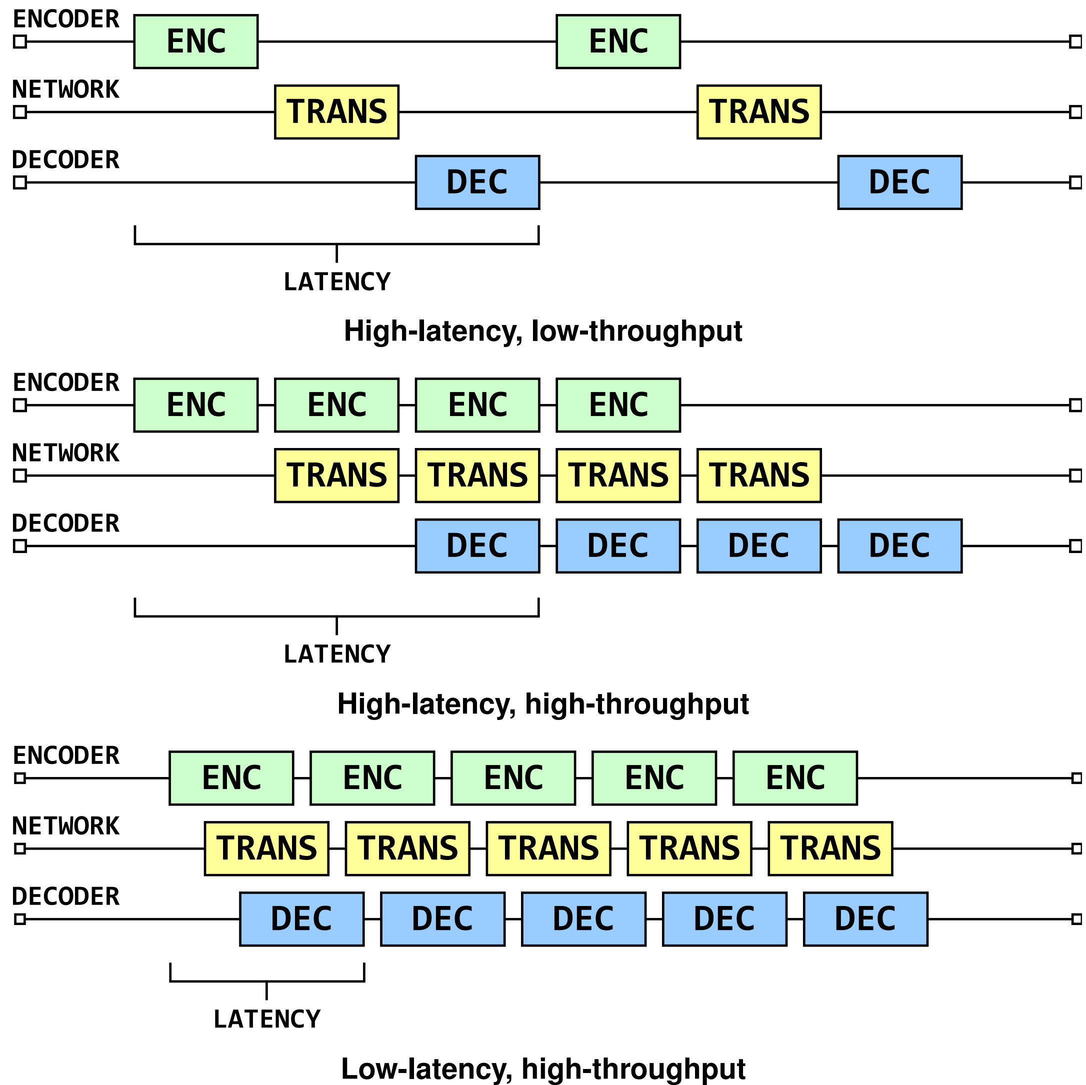


Figure 2.3. *Latency vs. throughput – increasing overlap in coding and transfer can help to reduce throughput without needing any increases of network bandwidth.*

With the addition of the overlapped transfers to the assemble of optimization schemes, the texture formats enable the proposed video streaming pipeline to work on impressively many levels of parallelization. Although this is convenient, it is also very necessary due to the extreme bandwidths required.

Also, as important as the performance optimizations are, the most effective method of reducing end-to-end latency would still be to use higher compression ratio formats. This is because the delivery latency would not be impacted by the used format itself, but only by the frame byte size and thus the compression ratio. Formats such as BC1 could directly halve the delivery latency when compared to the YCoCg-BC3 format, for example.

Finally, texture formats also have some favourable properties for network streaming applications through their error-resiliency, which is also in contrast to the previously mentioned H.264 standard and similar formats. Once again, the relevant factor here is in the block-

based nature of the formats. Relevant to especially lossy wireless networks, it may be preferable to instead decode a frame with some blocks missing or corrupted.

When encountering such bit errors, the effects of the errors would at worst be limited to only singular blocks in the frame. Additionally, in formats like BC1, for example, a singular bit error has a 50 percent probability to be limited to only one pixel in the block. Therefore, it may be possible to further optimize a texture streaming pipeline by doing away with error handling and retransmissions entirely, because even frames missing entire blocks due to packet losses would remain fully decodable.

Whether compromising quality to reduce latency is really an option would depend on requirements of the application in question. In human-interactive systems the answer would be up to the preferences of the user. Machine-vision systems on the other hand could continue functioning nearly normally and e.g. keep identifying objects even if some blocks in the stream were missing or faulty.

In any case, moving several gigabits of data per second through a video streaming pipeline with a desired frame latency of milliseconds is still a demanding task. When compared to conventional video streaming pipelines, texture compression makes the list of priorities turn on its head, as video coding –related compute time is decreased and video delivery becomes dominant instead.

3 HIGH-PERFORMANCE NETWORKING

Typically, video streaming applications, that utilize higher compression ratio formats, are not impacted by the network I/O performance. The bandwidths involved are low enough, that the end host network performance is only of secondary importance to the overall end-to-end latency. However, in the case of this work, where texture compression results in drastic increase in bandwidth use, network stacks become a crucial point of optimization.

An overwhelming majority of local or global network infrastructures and services are built on top of Linux systems. As could be expected, the network stack of the Linux kernel has been a target of continuous development over th years. The scope of this work is mainly concerned with the modern versions of the kernel, which implements the so-called "New" API (NAPI) in their network stacks [4].

The network stack is an extensive and complicated assembly of different protocols, sub-systems and hardware drivers, that have their own processing overheads. Most of these can be divided into classes of per-packet and per-byte costs [4], both of which are significant here. In order to gain a better understanding of how networking applications can be optimized for the best bandwidth and latency, a holistic look on the stack is required.

3.1 Linux Network Stack

Linux kernel network stack follows a layered model, where packets sent or received by the host system are passed through many different sub-systems. The described separation of concerns can be analysed through the OSI model, which divides the stack into seven distinct layers. The OSI model names these layers as physical, data link, network, transport, session, presentation and application layers. [19]

While the OSI model is a useful tool for higher-level software architectural design, a more condensed model is preferred in the context of the work. The so-called five-layer Internet Model combines the topmost three layers in the OSI model into one, called application layer, which represents the endpoint user space application in its entirety [19]. The functions of these five TCP/IP model layers are summarized in Table 3.1.

Referring to the Internet Model layer separation, Linux kernel implements the transport, network and some parts of the data link model as separated sub-systems. The bulk of

the layer 2 on the other hand, is implemented by the network interface card (NIC) and its driver, and the exact details will vary between different vendors. [20]

Table 3.1. *The layers of the Internet Model, their roles and some examples of protocols.*

Layer	Protocols	Role
L5: Application Layer	HTTP, FTP, SSH, etc.	Actual end-user application and other high-level controls, e.g. session management.
L4: Transport Layer	TCP, UDP DCCP, SCTP	End-to-end host connections and tracking their state across Internet.
L3: Network Layer	IPv4, IPv6	Packet routing and forwarding between separate networks based on IP addresses.
L2: Data Link Layer	Ethernet, IEEE 802.11	Packet delivery between neighbouring devices located inside the same network.
L1: Physical Layer	(N/A)	Physical hardware and electrical signals used to carry data.

All of the layers use different protocols, that define their own packet header formats relevant to the layer functions. It should be noted, that the scope of this discussion mostly focuses on IPv4 networks with TCP and UDP as the transport protocols.

As a packet passes through the kernel network stack, each layer header information is attached to the packet as demonstrated in Figure 3.1. Lower layer headers are placed in front of upper layers in order to allow network devices, such as switches and routers, that do not implement all TCP/IP model layers, to access their relevant header fields more easily [20].

A packet may at any stage be modified by a layer implementation, or discarded due to limited resource availability, checksum mismatch, other header sanity check fail or any other error [20]. To better understand the process, a more detailed look on how packets traverse their send and receive stacks should be taken. A simplified flowchart of the network stack is also presented in Figure 3.2.

L2	L3	L4	L5
ETHERNET HEADER (14 B)	IPV4 HEADER (20 - 60 B)	UDP/TCP HEADER (8 - 20 B)	PAYLOAD

Figure 3.1. *The full packet header structure in an IPv4 Ethernet network.*

User space applications mainly access the network stack through a POSIX-compliant socket API, which allows opening and closing communication endpoints bound to 16-bit

port numbers. Besides acting as file descriptors allowing read and write operations, sockets also offer a large number of configuration options that tweak their behaviour. The most important required ones are the socket communication domain and type, which determine the network and transport layer protocols associated with the socket. Additionally, a large number of optional socket options, are available, if not restricted by the chosen protocol stack, and some will also be relevant to the following discussion. [21]

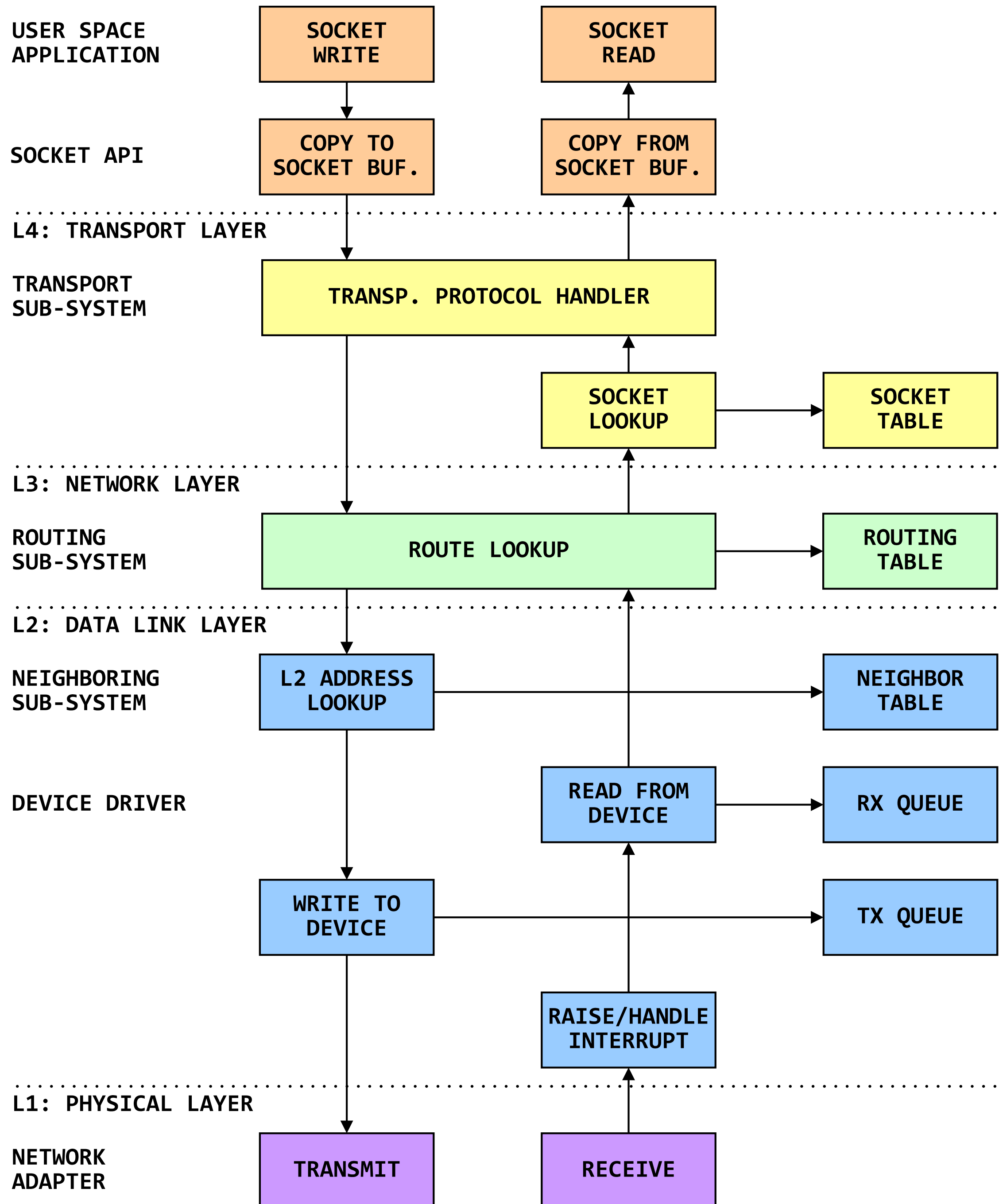


Figure 3.2. *Simplified network send and receive paths in Linux systems. [20]*

Each socket is allocated separate send and receive buffers by the kernel. The socket buffers, as they will be called for the remainder of this work, store packet data in a struct format, and act as a cache in which the packet is stored during processing. The struct data type, which for the sake of clarity is referred to here as a socket buffer object, contains all layer 2–4 packet headers, the packet payload, as well as various kinds of packet-specific meta information, such as the target network interface. [20]

When data is transmitted by a host, the application performs writes to a socket descriptor and the data gets copied to its send buffer in the kernel space. Likewise, as a packet is received, it is held in the socket receive buffer until it is processed and copied by a receive thread to some other user space memory location prepared for it [22].

Sitting right below the socket API itself is the transport sub-system, which is actually a collection of various transport protocol handlers that are interchangeably mapped to the relevant system calls [23]. The exact process depends on the protocol, but it would involve end-to-end communication management measures such as potential congestion control and packet loss or error detection and recovery.

The port number translation to socket instances is also performed in the packet receive path by the transport layer through a socket lookup [20]. An L4 header will therefore include the port number among other control information. Incoming packets with port numbers, that are not associated with opened sockets of the appropriate protocol, will get discarded.

The routing sub-system implements the L3 functionality in the Linux kernel, and it will process both incoming, outgoing, and loopback packets. A routing table lookup is performed for all these packets, and a checksum of the L3 header is also calculated in the case of both IPv4 and IPv6. It should be noted that L3 checksums do not include the payload itself, like L2 and L4 protocol checksums typically do. [20]

In the Linux kernels prior to version 3.6, a separate routing cache was maintained and lookups would revert back to the main table when a cache miss occurred. This functionality has since then been replaced by performance improvements to the main routing table lookups. [20]

The destination address may at this stage be subject to changes by the network address translation (NAT) system. If the destination address does not point to the local host, and no translation rule is present for the destination subnet, the packet is directed at the default gateway of the system, or discarded if the address is unreachable [20].

For outgoing packets, the source and destination addresses are encoded into the L3 protocol header, which is attached to the front of the transport layer header. The packet is then passed over to the appropriate network interface.

From the kernel point of view, the send path ends at the data link layer, where the kernel neighbouring subsystem is used to translate the target IP address into an L2 address needed in the layer header. This is accomplished through another lookup from the system neighbouring table, which is where L2 addresses of the adjacent nodes are cached for fast access [20]. In Ethernet networks it is likely to be the MAC address of a gateway router.

Network devices can have their own internal memory for storing incoming and outgoing packet data. In practice, some higher-end devices may even utilize multiple TX/RX queues, which are buffers mapped to system memory using a DMA ring buffering scheme. [22] Packets are therefore copied not only once, but twice in the send and receive paths, first to the kernel socket buffer, and then to a device buffer. After an L2 header is created, the outgoing packet is in the end handed over to a network interface driver to be transmitted over the physical layer to the next node in the network.

L2 protocols, like Ethernet, typically implement checksums for transmitted frames. If errors are detected in a packet, it will be (silently) dropped by an L2 device, i.e. a network switch, along the way [20]. It is then up to the L4 protocol implementation to detect a missing packet and deal with it assuming it can.

All L2 devices define a maximum transfer unit (MTU), which is the largest packet size they are able to receive or send. In other words, the packet payload and all L2–L5 headers combined must not exceed the MTU in any part of the packet path. In general, the network path MTU is arguably one of the most impactful restrictions posed by network infrastructures.

Unlike sending, which can be sequential to packet traversal in the stack, receiving is by-default interrupt driven. Upon arrival of a new packet, the NIC will move the data to its RX queue, and generate an interrupt signalling the system that new data is ready in the circular buffer. The interrupt handler will then trigger the kernel stack, which will move the data to its kernel space socket buffer for further processing. [22] If the RX queue is full, which can happen when upper stack fails to keep up with incoming traffic, then the NIC may either store it in its internal memory, or drop the excess packets [4].

In the old Linux kernels up to version 2.4, one interrupt would be generated per received packet [20]. This was expensive and infeasible in high-load use cases, such as the video streaming system presented in this work, where the stream width may be in millions of packets per second. The interrupt handling issue was resolved in the introduction of the NAPI system to the kernel, which changed the pipeline so that the entire RX queue could be flushed at once by the interrupt handler, disabling new interrupts while doing so [4].

The packet receive path is similar to the send path, but advances in the opposite direction and includes different lookups. Because network communication is always assumed to

be at least somewhat unreliable, all L2–L4 protocols are required to perform additional checks on header information and payload integrity. Luckily, the receive stack is able to utilize some additional hardware offloading capabilities, such as checksum computations in the NIC just before transmission [4].

Reducing the CPU involvement as much as possible is crucial. Packet traversal through the various layers introduces a number of system calls and processing steps leading to a massive per-packet CPU overhead, which is added to the per-byte copy and checksum overheads [4]. Here the significance of the network hardware is highlighted, as hardware offloading and large MTU allowances can either directly reduce the overheads, or the number of packets to process.

3.2 Transport Layer Protocols

As stated, transport layer protocols manage the end-to-end communications between two endpoint hosts. The two most popular L4 protocols of the Internet are by far the transmission control protocol (TCP) and user datagram protocol (UDP).

The protocols give very different guarantees to the application layer and are therefore the most visible part of the network stack to the user space application. Even more importantly, their performance characteristics are very different with far-reaching effects to any pipeline. The effects are also dependent on the transmission medium used.

3.2.1 User Datagram Protocol

UDP is known as the simplest L4 protocol, since it is in fact only a thin wrapper over the routing sub-system. It is a connectionless and stateless protocol, that provides no guarantees of packets being received in order, or at all. UDP is additionally regarded to be a message-oriented protocol, meaning that the delivered packets, or datagrams, represent atomic messages instead of continuous streams.

Because of its simplicity, UDP is a very lightweight protocol, which is reflected in its 8-byte header format, that is used to carry port number and packet size information. The 16-bit packet length field limits the maximum size of UDP packets to 65 536 B, which is plenty considering that in practice the network MTU is the true limiting factor. The header structure is presented in Figure 3.3.

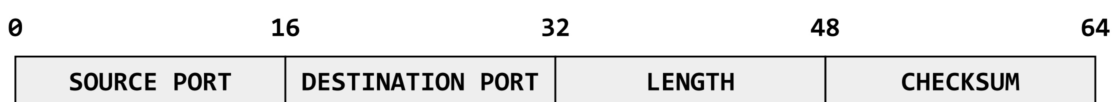


Figure 3.3. *The 8-byte L4 header used by UDP [24].*

As a stateless protocol, UDP does not implement many of the high-level features of other competing L4 protocols, such as congestion control mechanisms. Due to the 65 KB packet length limit, most implementations support packet fragmentation at send time, but there is no mechanism for L4 defragmentation at the receiving end.

Slightly against a common misconception, UDP does support error-detection through the 16-bit checksum field of the header. The checksum is acquired by summing two byte sections of data in the payload and a “pseudo header" comprising of the L4 and some L3 header information [24]. Because UDP implements only error detection and not error recovery, a packet that fails the checksum test is discarded by the receiver [20]. Since the checksum is only 16-bits, it is of course not fully impossible that checksum collisions could mask errors in large packets.

The checksum use is compulsory in IPv6, but not in IPv4 networks [25], where if checksumming is not in use, zero data gets written to the packet checksum field. If checksum is used, but its value is actually zero, it can be distinguished from the disabled value by coding it as a one’s complement of the result, i.e. all ones or “negative zero". [24] In practice, the computation of the checksum is almost always offloaded to the NIC, meaning there is little processing overhead involved [4].

Most notably UDP checksums are by default disabled when the target is the localhost address [20], as packet errors are not expected inside the network stack itself. Memory errors are of course still possible, but could be accounted for by other means, such as error correcting memory. Regardless, disabling checksumming in loopback communications is yet another relevant performance factor, since virtual loopback adapters do not have access to checksum hardware offloading.

Another, modified version of the UDP protocol is also available, called Lightweight User Datagram Protocol (UDP Lite) [26], which allows adjustments to the checksum behaviour as its main feature. While UDP only supports checksums for the whole packet, or not at all, UDP Lite allows extending the checksum to a freely adjustable offset, meaning that it can choose to support checksums only for the L4 header and some part of the payload.

Since many media formats already have at least partial resilience for bit errors in streams, UDP Lite has seen some success in such streaming applications. The main limitation of the protocol is the fact, that the L2 implementation of the pipeline has to be also configured to not drop packets, which fail the checksum tests, making it less practical in other than specialized systems.

Generally speaking, UDP streaming applications are capable of achieving lower latencies and higher raw throughput rates compared to other L4 protocols. The no-delay controllability of the UDP-based streams would make them especially viable for real-time applications. Theoretically the communication latency could also be reliably predicted in systems

optimized for hard real-time applications.

Depending on the application, while the L4 overhead of UDP is very low, it may however end up being offset by potential increases in processing overhead in the application layer. Whether this is true depends on the packet-loss and packet-reordering tolerances of the application. For use cases, where reliability is important, there exists other, more reliable transport protocols that often manage to be more efficient than any measures implemented in the application layer.

3.2.2 Transmission Control Protocol

TCP is the most widely adopted transport layer protocol of the Internet and typically the primary alternative for UDP. The main goals of TCP are to provide reliable connection-based stream communications over unreliable networks with some other L4 control capabilities as well.

The important guarantees of the TCP protocol include error-free and order-preserving packet delivery between two endpoints through loss- or error-detection, reordering, and retransmission schemes [27]. However, these guarantees make TCP a stateful and significantly more complicated protocol compared to UDP, with some additional overheads and performance shortcomings.

The protocol being connection-oriented means that one host must establish a connection with another listening host before data can be transmitted. Connections offer bi-directional communication, and both hosts maintain state information about the other side and state of the connection. [27] The benefit of the connection semantics is that the application layer is not required to be aware of the potential errors in transmission, as the L4 implementation can account for them internally. The connection requirement also makes TCP a strictly unicast protocol, unlike UDP.

As a stream protocol TCP does not technically require the application layer to be aware of any MTU limitations or packet fragmentation taking place behind the scenes. When a data stream is written to a TCP socket, it is broken down into segments, that the protocol implementation can operate on. These segments are added a variable size L4 header with a minimum length of 160-bits, as presented in Figure 3.4.

Besides including the port number for the socket table lookups, the TCP header also contains sequence and acknowledgement fields that play part in the TCP acknowledgement mechanism used to handle packet losses. For every successfully received packet, the receiving host is expected to send an ACK message informing the sender that the packet has been received. If a packet, identified by its sequence number, is not acknowledged by the receiving side, then it and all its preceding non-acknowledged packets are retransmitted. [28] Therefore, even unidirectional streaming involves two-way communication.

<table>
<tr><td>0</td><td>16</td><td>32</td><td></td><td></td><td>48</td><td>64</td></tr>
<tr><td colspan="1">SOURCE PORT</td><td>DESTINATION PORT</td><td colspan="4">SEQUENCE NUMBER</td><td></td></tr>
<tr><td colspan="2">ACKNOWLEDGEMENT NUMBER</td><td>DATA OFFST</td><td>RESRV</td><td>FLAGS</td><td>WINDOW SIZE</td><td></td></tr>
<tr><td>CHECKSUM</td><td>URGENT POINTER</td><td colspan="4">(TCP OPTIONS)</td><td></td></tr>
</table>

Figure 3.4. *The 20-byte (minimum) L4 header used by TCP [28].*

TCP packets also contain a 16-bit checksum computed similarly to UDP from the L4 header, some L3 header fields, and the payload itself. If a checksum indicates a bit-error in the packet, the receiving host can request a retransmission of the segment from the sender. Alternatively, retransmission occurs after a time-out if no acknowledgement is sent, indicating that the packet was dropped somewhere along the route. [28] The use of checksum is always required in TCP, but like with UDP, checksum computation is commonly handled in hardware.

Segments are also reordered by the TCP protocol handler so that the application layer will always receive them in correct sequence. As could be expected, packet reordering can add to the end-to-end latency of the application, since some segments may have to be retransmitted to make a complete payload.

By default TCP acknowledgements are cumulative, which means that the receiving host can acknowledge the receiving of multiple continuous segments by sending a singular ACK message with the sequence number of the last segment [28]. This can however be inefficient in circumstances where multiple segments are received simultaneously, but e.g. the first segment in the sequence is faulty. In such a case the receiver does not have a method to communicate to the sender which segment needs to be retransmitted, meaning all segments could end up being retransmitted.

The cumulative acknowledgement problem has luckily been solved by the selective acknowledgement (SACK) extension, which is a useful TCP header option, that can be used by the receiver to acknowledge a discontinuous set of segments and avoid large numbers of unnecessarily retransmissions. In order to use SACK messages, both hosts have to support it, which can be negotiated at connection time. [29]

A number of similar TCP performance tweaks have been introduced over years. Some of them are relevant especially for low-bandwidth applications, such as the Nagle's algorithm [30] or delayed acknowledgement [31], that can be disabled in socket options to improve the throughput and latency [32, 33]. For the purposes of this work however, these effects are not as relevant, since the bandwidth of the video stream is so large.

TCP does not only ensure reliable stream delivery, but it also implements a congestion control mechanism, which can help to avoid saturating a transmission channel with traffic

[34]. Congestion control is commonly motivated by the fact that in general-purpose contexts multiple applications are usually required to share limited network resources, and situations where one application consumes all bandwidth are to be avoided. For the proposed video streaming system, congestion control turns out to be a hindrance, since it actually aims to consume as much bandwidth as possible.

Congestion control is implemented in TCP through a sliding window mechanism, which limits the number of in-flight segments that can be transmitted without an acknowledgement being received yet. The window size is increased in an additive fashion for every received successful ACK message, or until a maximum window size is reached. Correspondingly, if a segment is detected to be lost due to packet loss or corruption, the window size is reduced in a more sharp multiplicative step. [34]

The TCP additive increase – multiplicative decrease (AIMD) congestion control mechanism results in a characteristic sawtooth effect in the overall end-to-end throughput. The pattern is a well-documented property of TCP, and it is one of the main performance issues of the protocol, besides the “slow start" problem, which is caused by the initial window size always starting from one. [19, 34] Both effects are visualized in Figure 3.5.

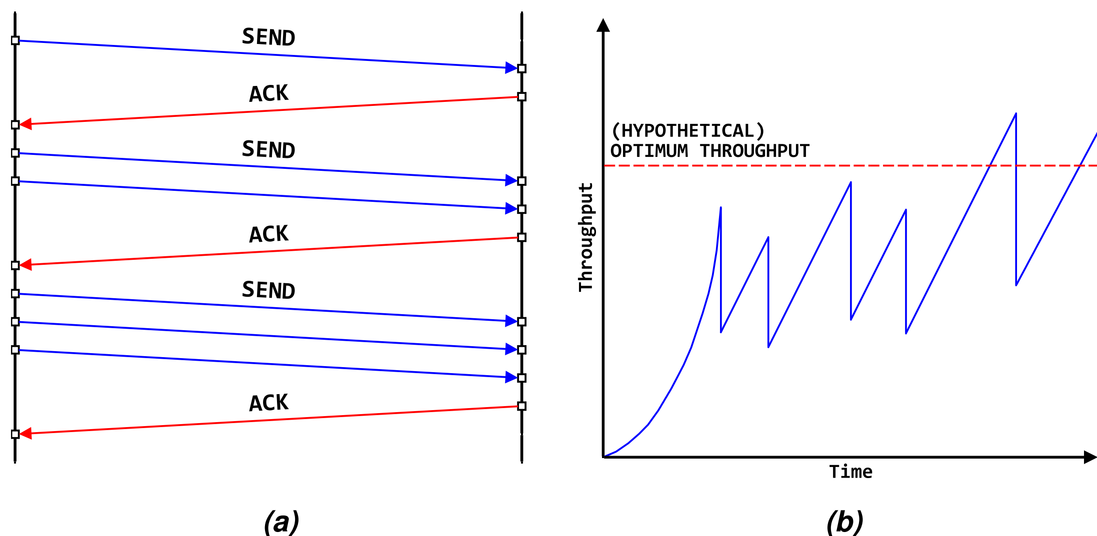


Figure 3.5. *(a) Acknowledgment-driven window scaling behavior of the TCP congestion control mechanism, and (b) the resulting characteristic sawtooth throughput pattern. [19]*

The congestion control mechanism fulfils its purpose well, in allowing fair sharing of limited bandwidth among multiple hosts in networks, where link saturation leads to dropped packets. Unfortunately, the window reduction mechanism does not distinguish between true capacity saturation and sporadic errors in lossy transmission mediums. Because of this uneven throughput behaviour, a singular TCP connection cannot realistically ever fully saturate network bandwidth alone [34].

Besides the acknowledgement/retransmission overheads of TCP communications, the

congestion control mechanism can easily produce an even greater impact to the throughput performance. Especially wireless networks, which are inherently prone to packet errors and losses, have been found to be the most susceptible to throughput deterioration [19].

Since all networks have some upper capacity limit, some window scaling function is always needed, assuming that the real network bandwidth is not known in advance. The impact of congestion control can luckily be reduced by using a more aggressive window scaling function. In the case of this work, the chosen scaling function is the CUBIC algorithm, which has been default for Linux systems since around kernel version 2.6 and is currently the most widely adopted TCP congestion control algorithm in the world [35].

TCP is also vulnerable to the so-called buffer bloat problem related to wireless networks. Due to the unpredictability of a wireless medium, some router vendors have made attempts to even out throughput fluctuations by increasing buffer space in their network hardware. Somewhat paradoxically, the addition of new invisible buffer space can prevent TCP implementations from adjusting to the network congestion correctly, resulting in performance losses. [36]

All long round-trip time (RTT) networks are vulnerable to these issues, as latency reduces the TCP scheduler ability to adapt, because of the need for two-way communication. Related to RTT, is also the bandwidth-delay-product property of the network, which represents the intrinsic maximum amount of inflight-data for an L4 protocol. [29] In case of the video delivery system studied in this work, the congestion avoidance is less of an issue due to the fact that it naturally targets relatively low-latency networks.

3.3 Network Performance Tuning

An important goal of this work is to find ways to optimize the standard Linux network stack as far as possible for the best bandwidth and latency. Unfortunately, despite the efforts of the kernel development community, a number of performance pitfalls remain present in the stack.

The relevant question becomes, how to work around the limitations through application design and resource management. These optimization efforts are not limited only to the transport layer protocol options, which were already discussed in the previous sections, but to the fundamental mechanisms in the stack itself common to all L4 protocols.

3.3.1 Multi-Threaded Sockets

When taking a look at either the send or receive paths of the kernel network stack, a high degree of CPU overhead is very apparent throughout the different stages of packet

processing. One natural question that follows, is whether the network I/O could be upscaled through the use of multi-threading.

It has been found in multiple instances [23], that the kernel stack does not accommodate multi-threaded I/O to the extent one could expect from the world's most popular server operating system. The reason is that the implementation of the socket interface was originally developed in a time where the network was still the primary performance bottleneck instead of the end-host performance. Ever since the introduction of the NAPI system, work has been ongoing to change the state of affairs. [4]

Therefore, for an individual socket instance, the multi-threaded performance of the socket buffer data structures leaves a lot to be desired [23]. Since the bulk of the processing is handled by the kernel threads behind the interface, increasing the number of user-space threads is also of little help.

Instead, the workable solution could be considered obvious in the light of the fact, that many ordinary data center applications manage to utilize massive amounts of bandwidth just fine on a daily basis, assuming they run large numbers of processes in parallel. In other words, the winning strategy is in multi-socket multi-threading solutions.

In essence, large bandwidth networking applications, such as video pipelines, can overcome some of the performance limitations of the network stack by splicing their data into smaller sub-streams [34]. The described multi-socket streaming strategy enables the effective scaling of the application–kernel interface, and has produced throughput improvements for both the TCP and UDP protocols in many studies, such as the performance tests done as a part of this work.

While the associated CPU overheads themselves still remain, multi-socket streaming does at least allow making use of modern multi-core processors to their highest potential, aside from possible restrictions in e.g. memory and bus performances. If the network adapter happened to support multiple RX queues, or there were entirely separate adapters, they could also get assigned different queues thus distributing the interrupt handling overhead between the stream [4].

Transport protocol implementations can also benefit from the multi-socket approach directly. The most obvious example is of course the TCP protocol, where multi-connection TCP has shown improved performance scaling properties over single-connection alternatives [37]. It especially helps to address the bandwidth under-utilization problem caused by the sawtooth throughput effect, and partially the slow-start issue as well [34].

Multiple sockets could be further combined with various load-balancing systems designed to even out throughput differences between socket instances. The only drawback is, that these would likely have to be implemented completely by the application layer.

Alternatively, numerous entirely new transport layer implementations have also been proposed, like the Multi-Path TCP (MPTCP) [37], which attempt to integrate the multi-connection TCP scheme into the kernel itself. These protocols can then be abstracted into the existing socket API making them more-or-less interchangeable with existing TCP applications without the extra involvement of the application layer.

Parallel multi-socket streaming is not however the be-all and end-all of kernel networking performance. Even pushing aside any revisions to the system architecture and hardware stack, there is potential for improvement in the software stack too. While these issues are internal to the kernel [4], there also exists performance overheads related to system calls, memory conservation, and thread synchronizations that could be reduced through configuration options available to the end-application or its user.

3.3.2 Socket Buffers

Besides acting as the main programming interface between the kernel and application layer, sockets also have an important role in network stack resource management. The socket send and receive buffers act as temporary storage spaces for incoming and outgoing traffic, linking the user and kernel space programming together [23].

The socket buffers have a limited size, which can be adjusted through some socket options in the application. However, there also exists system-wide maximum buffer size values, which the application cannot exceed, unless the process is executed with root permissions. These maximum sizes are defined separately for the TCP and UDP protocols, and extend system-wide for all processes.

In the test system that was used in the work, and presumably in most Linux environments in general, the UDP socket buffer maximum size was by default set to 208 KB, while TCP buffers were up to 6 MB. For the UDP protocol the problem is severe, as it turns out that the socket buffer size can severely limit the effective network bandwidth for the system [38]. The issue is worsened by the fact, that UDP has no congestion control measure to counteract it.

One can surprisingly easily generate more than enough traffic to saturate the socket buffer space completely. Especially traffic sent or received in bursts will quickly overflow the buffer space, as the kernel-side threads are no longer able to process pending packets fast enough.

Saturating the send or receive buffer is an unsolvable problem to the kernel. In such cases, the only solution is to start discarding new packets [22]. Running out of receive buffer space will also impact the network adapter RX queue, which can similarly overflow in situations where the software stack is unable to keep up with the incoming packet flow, resulting in packet drops in the adapter [4].

From the perspective of the transport layer, the dropped packets will then appear as any normal transit-time packet losses. For UDP stacks, the buffer saturation will manifest as any missing packets. For the TCP, the connection throughput will be limited by congestion control, just like in the case of any network maximum bandwidth excesses.

3.3.3 Packet Fragmentation

Top network layers have to always take lower layer limitations into account. The MTU of the network is a hard limit to the packet maximum size imposed by the L2 frame size, which includes the payload and all headers across upper layers. In cases, where a large packet happens to cross an L2 node with an MTU less than the packet size, the packet may get simply discarded, preventing communication entirely [20].

In Ethernet networks the typical maximum MTU is 1 500 B, while the lower limit that a network device has to be able to handle, is 576 B for IPv4 networks and 1 280 B for IPv6 networks [25]. In specialized networks some high-end Ethernet routers and switches are able to expand this up to 9 000 B, also referred to as "jumbo frames".

Other L2 networks may impose larger MTU values, such as wireless IEEE 802.11 -networks, which support an MTU of 2304 B. Unfortunately, bridging the wireless connections with standard Ethernet networks would still limit the route maximum MTU to that of the Ethernet network. [39]

Some L4 or L5 implementations take the MTU limitations into account by performing MTU discovery in advance to find the maximum of the network path. L3- and L4-capable devices may also fragment packets automatically to an MTU-compliant size. [20] Other protocols do not make any MTU guarantees at all, and it is up to the application to limit its message sizes to a value that is known to be supported. Fragmentation operations requiring reordering of header data can also be a performance overhead, which is why most network adapters provide hardware fragmentation offloading capabilities to the kernel [4].

Packet size can have implications for the performance of the entire application. A significant per-packet overhead is involved throughout the different parts of the stack, caused by factors from interrupt handling to the number of required system calls. Ideally packet streams should follow the pipeline maximum MTU, and while fragmentation can allow a network application to at least function, it is not ideal from the performance standpoint.

The route MTU is likely one of the most universal limitations to end-to-end networking, since it impacts the entire network path between the hosts. Both network layer protocols, as well as transport protocols, also restrict the maximum packet or segment size typically to a value of 65 536 B, but in practice the data link MTU remains the limiting factor.

However, as discussed earlier, packets routed locally to the same host are never passed

through the layer 2 of the stack. This means that in local contexts, such as the localhost video stream performance tests, or any inter-process communication in general, MTU does not limit the packet size allowing at least testing within the full 65 KB range.

3.3.4 Busy-Poll Receiving

The weakness of the NAPI system is the interrupt handling in the receive path of the stack. When new packet data is written to the RX queue of the NIC, the interrupt generated has to be handled in a separate thread within kernel, which then will flush the queue of pending data [20]. In the meanwhile, application threads attempting to read the socket will be blocked until the data is processed and ready to be pulled from the socket buffer.

The interrupt-driven receive scheme is motivated by resource efficiency, since in most applications sockets can remain inactive for long periods of time, freeing CPU time for other tasks [4]. The primary problem of the interrupt-driven system is in latency caused by both the interrupts and the required context switching.

An alternative setup, designed to tackle the latency issue, has been proposed, where the application thread can actively poll the RX queue directly [40]. Referred to as busy-poll sockets, the approach allows optimizing for the best possible latency at a significant cost in CPU time, since the application thread can no longer sleep while waiting for packets. A comparison of the two types of sockets is presented in Figure 3.6.

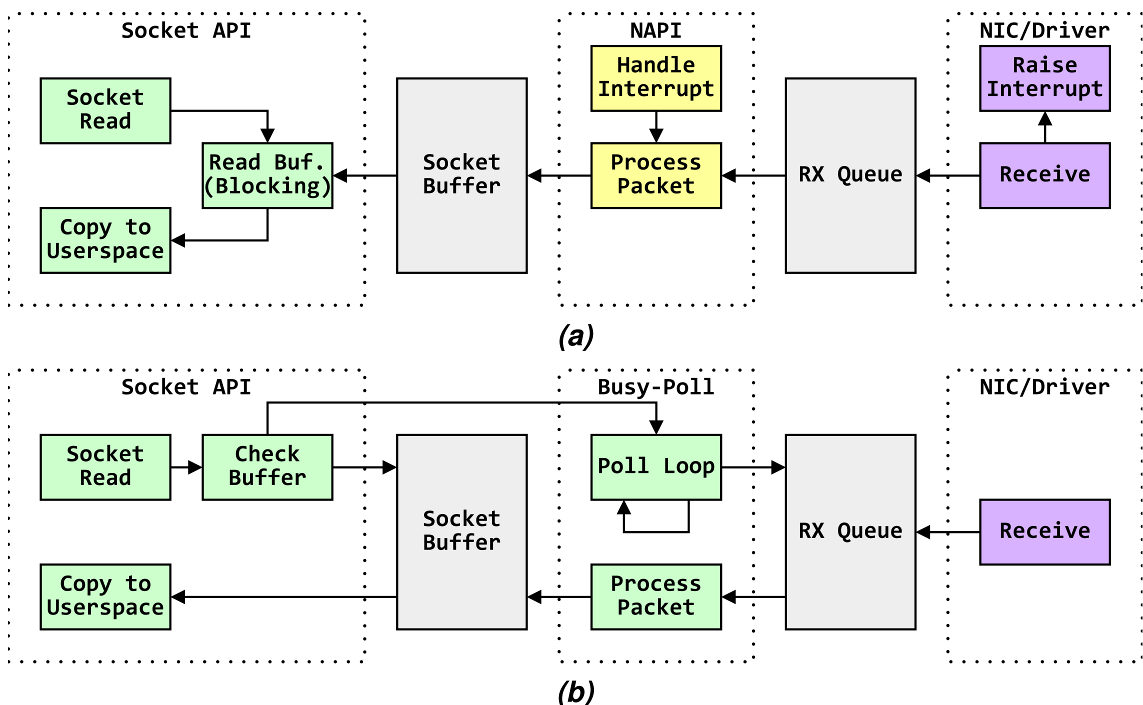


Figure 3.6. *(a) The normal NAPI receive process and (b) the busy-poll version, with separate threads highlighted in different colors [41].*

Busy-poll sockets are an interesting alternative to more advanced networking technologies, like the previously mentioned RDMA, which can be used to reduce latency using special high-end network hardware. Although busy-poll does admittedly also require explicit support from the physical network adapter, it has regardless become a common feature in consumer gear over the last decade. In other words, the hardware support requirement could be regarded mostly a non-issue in the case of busy-polling. [41]

However, consuming the CPU time of an entire processor core is only feasible if the core has no other tasks to perform at the same time. Although busy-poll sockets can in one-to-one comparisons outperform the ordinary NAPI sockets, as a whole they are mostly suitable for only those low-intensity use cases that require low-latency messaging but not high bandwidth [41].

The CPU-intensity makes busy-polling infeasible for high-bandwidth low-latency applications that depend on multi-threaded network I/O, where multiple sockets are used to stream data in parallel. One of these applications is arguably the video streaming pipeline discussed in the following Chapter 4.

Busy-polling is also limited to only real physical network adapters, since it explicitly targets the RX queue of the adapter [41]. A relevant example, where this does not apply, would be the Linux loopback adapter, which is a virtual interface that lives only in the layer 3 of the network stack. Since these receive ring buffers do not exist in loopback in the first place, busy polling cannot be used to speed up localhost communications.

4 PROPOSED SYSTEM

The video streaming system described in this work is primarily intended as a research platform for texture compression in low latency video coding. The goal of the following chapter is to study the technical challenges involved in achieving not only the lowest latencies, but also the very high bandwidths imposed by the texture formats.

The project also had multiple secondary design goals, which were influenced by a versatile set of research directions and use cases. Some important objectives include maintaining compatibility with general-purpose hardware and standard Linux systems. Both of these can be considered preconditions for porting the presented video streaming system to currently available mobile platforms, or in general any heterogeneous software and hardware environments. Likewise, some interest was directed at distributed compute applications.

The fundamental compromises between performance, quality and hardware were already discussed in Section 2.2. Other secondary factors related to software architectural questions also have to be taken into consideration, such as modularity and separation of concerns. These secondary considerations must not however compromise the latency or throughput performances.

4.1 Architecture Overview

The pipeline was designed to acquire video stream from a video source, encode and transfer it over network, and finally decode and render it at the receiving end. Two endpoint programs written in C++14 were implemented, following a classic client–server model where the encode program acts as the server and the decode program as the client. The programs are further divided into interchangeable modules and connect through a common main program, which also handles all runtime configuration and session management, such as initializations, handshake procedures and state transitions.

The encode program is split into three modules: video source adapter, video encoder and stream server. The decode program on the other hand consists of only two modules: stream client and video decoder. Multiple interchangeable versions of all these modules were developed, such as CPU and GPU implementations. The encoder and decoder

arguably form the active cores of both programs with video source and streaming modules acting as simple adapters to various video and network devices respectively.

Architectural decisions reflect the emphasis on compute and transfer performances. As discussed in Chapter 3, the high CPU overhead in the kernel network stack highlights the importance of multithreading, which acts as the main driver of the architectural choices. Scalability is pursued by splitting the video stream between the encoder and decoder into multiple parallel sub-streams. The corresponding I/O threads can then avoid unnecessary synchronizations between each other, with the only points of synchronization being the encoder frame input and decoder display output.

High-level block diagrams of the both the encode and decode programs are presented in Figures 4.1 and 4.2 respectively. In addition, the diagrams also display the main buffers and external devices of the pipeline.

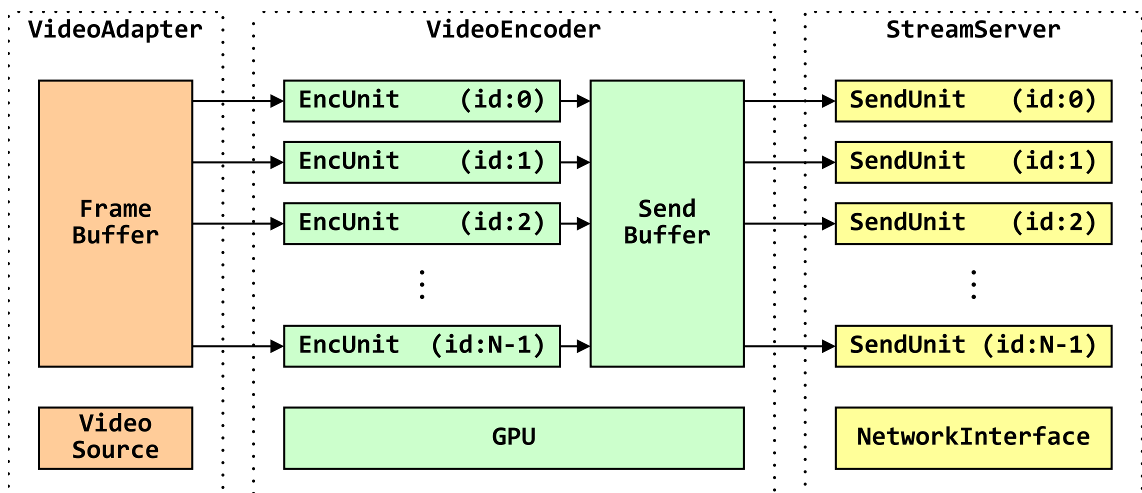


Figure 4.1. *The encode program block diagram.*

The current implementation of the presented model includes an additional point of synchronization at the encoder output, which blocks sending until the encoding is complete. This is not fully necessary, as the sub-streams are designated their own tiles of the frame. Ideally, the encoder input should therefore be the only point of synchronization besides decoder output.

Since the focus of the work is in studying the stream delivery, encoder output can still be regarded as the most relevant point of reference. The low encode latencies of the texture formats used here also make the encoding process impact to the end-to-end latency a lesser issue compared to the delivery.

Despite of the delivery latency forming the majority of the total end-to-end numbers, optimizing for maximum network throughput alone is not the end of story either. As proposed in Section 2.2, the end-to-end latency can further be reduced by introducing overlap in

the encoding, transfer, and decoding stages of the delivered frame. In practice these efforts were limited to the decoder implementation, though the same principles apply everywhere.

A number of abstractions were made while designing the architecture. Some of these terms should be further clarified as the following terminology in question will be referred to in the subsequent sections.

When implementing some of the tested video coding processes, it was decided to support splitting the data into separate channels, or grouping it in some other ways. These channels are managed as one or more distinct unit groups with each group handling some slice of the frame data.

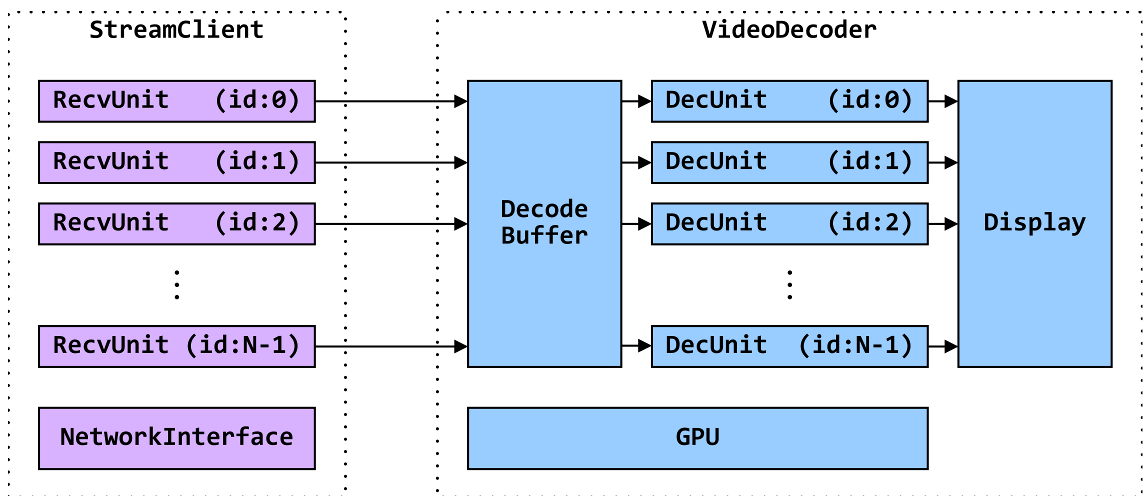


Figure 4.2. *The decode program block diagram.*

As might be expected, the groups further consist of variable numbers of units, which are abstractions for one end-to-end sub-stream. Each unit has a static number of resources attached to it — namely CPU threads, network sockets and buffers at both ends of the pipeline. Both the encode and decode programs have to negotiate together on how many units will be allocated for every groups.

Unit allocation becomes an important element in the performance optimization of the pipeline, since it can be used to balance the resource allocation between different groups. Alternatively, the problem could be circumvented by utilizing load balancing patterns, but the static allocation scheme was deemed more likely to produce better performance.

On a higher level, the main modules of the encode and decode programs follow a state-machine-like behaviour. The aforementioned initial handshake process, which is presented in Figure 4.3, is implemented as a simple TCP-based protocol. Its main purpose is to allow the encode and decode programs to synchronize and agree on common configuration information before initiating a video stream.

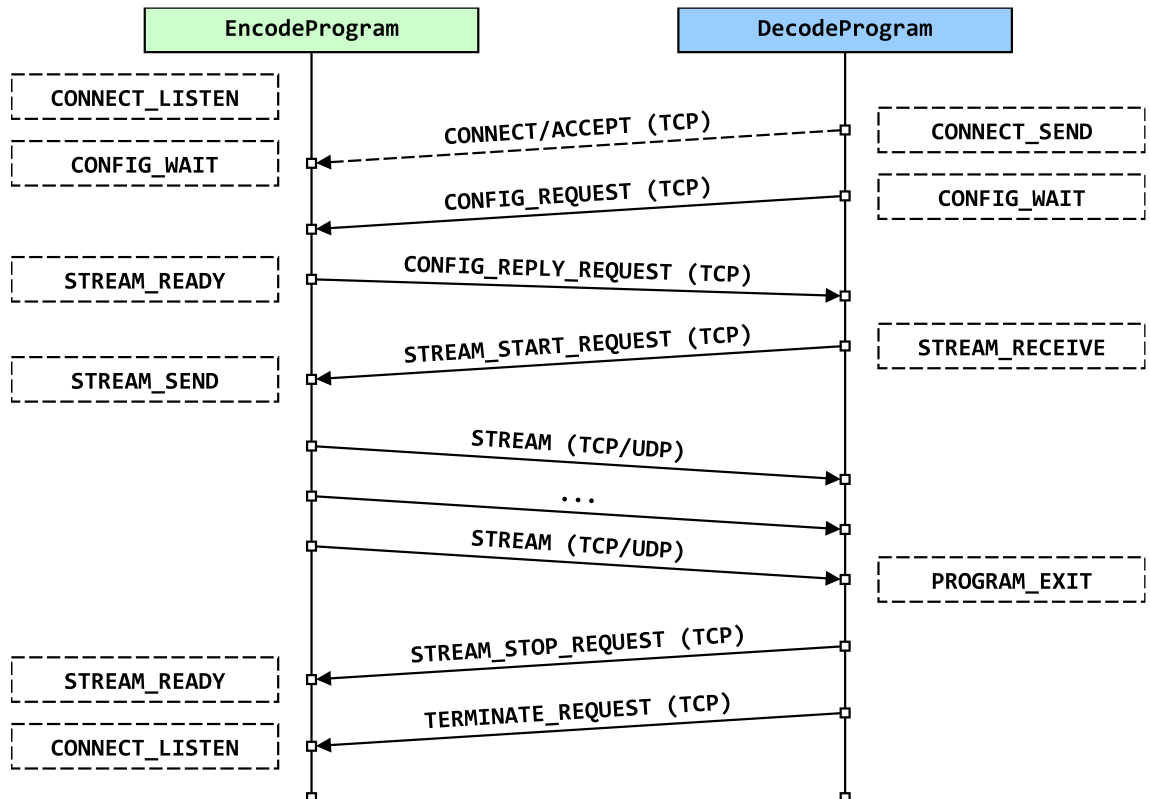


Figure 4.3. *The handshake process and state transitions.*

Ideally, the modularity of the architecture would allow full separation of the video coding and delivery parts of the program. If this was the case, the encoder and decoder would not be required to know how data is transported, and the network code would not need to care about video formats. In practice, only the latter was actually achieved.

Complete separation of concerns turned out to be not fully feasible. As an example, the transport layer protocols — UDP and TCP — provide completely different kinds of guarantees that can be used to further optimize the pipeline performance.

4.2 Video Coding

Texture formats lend naturally themselves to GPU-based coding, and the pipeline architecture was designed with this in mind. GPU-accelerated encoding and decoding would therefore be the obvious route for any commercial production system.

Keeping the previously outlined goals of the work in mind however, it was decided to implement a low-overhead CPU pass-through encoder in conjunction with a file system -based video adapter. The efforts were concentrated on the decoder implementation, although much of the presented solutions would in fact be more or less directly applicable to the encoder as well.

The decode program was was fitted with a feature-complete GPU decoder written for the OpenGL API. As one of the goals of the project was to port the decoder to a mobile platform, the chosen API version was in fact OpenGL ES 3.1, with the ES standard being largely backwards-compatible with the full OpenGL API. Thankfully, both versions share the EGL library for context management and allow using the ARM Mali GL ES emulation headers in desktop systems.

4.2.1 Encoder and Decoder

As the video data enters the pipeline, it is stored in a frame buffer of the video adapter, where it gets passed to the encoder through a callback function. The encoder then has to internally distribute the data among the designated number of units and unit groups for encoding. In the case of the CPU pass-through encoder, the frame buffer is passed as it is to a group of output threads. Here the frame data will be further split into packets before passing them onward to the relevant stream server send units. This is the key point of reference from where the delivery latency will actually be measured.

In the opposing host endpoint, the decoder assigns each receive unit a buffer address, which points to the main decode buffer. The exact details of how the data is received depend on the transport protocol used, and are further elaborated in Section 4.3. As the frame data is received in a RAM buffer, it will have to be transferred to the GPU using a double buffering scheme, which is likewise discussed separately in Section 4.4.

There still exist some complications related to the networking aspects, due to which the encoder and decoder were not completely equal problems. The root issue is that the encoder is not a target of any strict real-time requirements, as it can simply drop entire frames at once whenever it fails to keep up with the video source. This is made possible by the fact that the decoder is expected to handle any frame rate fluctuations by default.

The decoder on the other hand is required to operate on unreliable packet streams, which are prone to unpredictable timing, but also packet reordering and losses in the case of UDP. Moreover, the decoder does not have any easy way to drop frames, but only individual packets in case it is not able to keep up with the encoder. These are also not only video coding -related complications, but the effects have to be accounted for throughout the networking and buffer management code as well.

Like was mentioned, the encoder and decoder are the active cores of the endpoint programs. Even though the discussion on efficient real-time texture format encoders is not in itself part of the work, there still exists multitude of other fundamental problem areas regarding stream processing, and video format management.

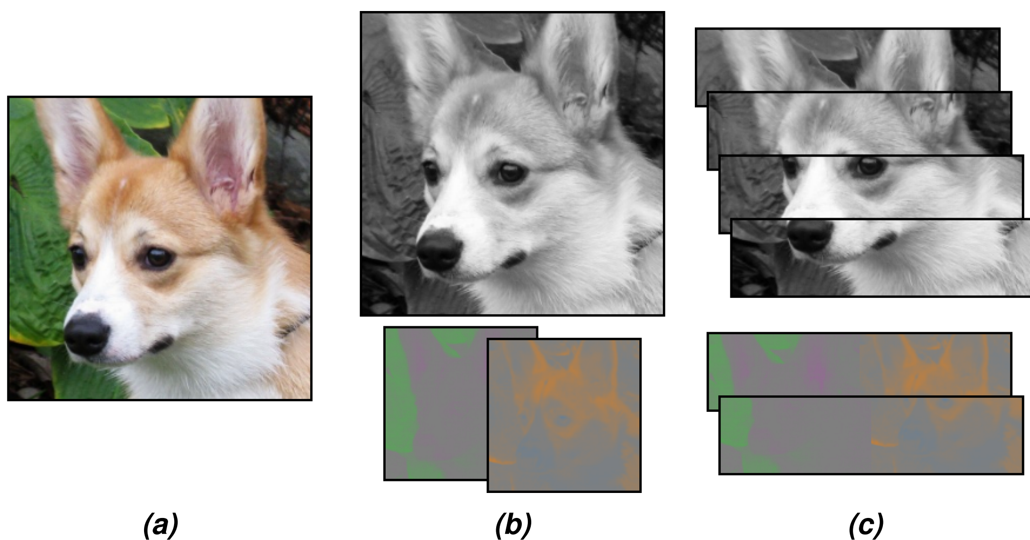
4.2.2 Video Formats

In the video coding stages, the unit and unit group -abstractions translate to numbers of tiles and bitplanes respectively. It should in fact be noted that the unit group allocations only impact video coding, and not stream delivery, since the tasks of separating and merging of the sub-streams are handled by the encoder and decoder. The management of these unit groups is done as part of an extensive video format system used by these components. The format negotiation itself occurs at the configuration exchange stage of the handshake process.

The pipeline was developed to support a wide range of video modes with both raw, compressed and sub-sampled formats as well as unlocked resolutions and frame rates. While some of the modes translate directly into single-channel compressed formats such as BC1 or YCoCg-BC3, others include multi-bitplane formats.

The number of unit groups is fixed and directly linked to the number of bitplanes in a video mode, while the number units and tiles in a group is arbitrary. However, the ratio of units allocated between different groups is not. Instead, the ratio should be balanced between the groups to match their memory footprint ratios, in order to balance the CPU time needed in stream delivery.

As an example, Figure 4.4 demonstrates a scenario where an RGB input frame is encoded into a two-bitplane YCoCg format with 4:2:0 subsampling. Here the Y-channel forms its own bitplane and the CoCg-channels are serialized into another two-channel bitplane with half the width and height. As a result, the CoCg-bitplane is only half the byte size of the Y plane. A sensible unit allocation pattern would therefore follow a 1:2 ratio to balance the computing and networking resource needs in all parts of the pipeline.



(a) ***(b)*** ***(c)***

Figure 4.4. *(a) A 24-bit RGB input frame, (b) sub-sampled YCoCg frame separated into luminance and chrominance bitplanes and (c) tile division following a 2:1 ratio.*

One practical limitation of the texture compression formats is that the geometries of both entire frames, as well as individual tiles, have to conform to the pixel block size of the format in question. In other words, the frame dimensions have to be multiples of the block size.

In the case of frame geometry, the issue is less prevalent, as most common video resolutions such as 1080p, 2160p and 4320p are already divisible by four, which is the block size in the S3TC and ETC formats. It is also supported by ASTC among many other values. Regardless, a general purpose video platform needs to be able to handle any arbitrary resolutions including odd frame dimensions.

A simple solution is to pad the input frame with additional data before encoding to a resolution complying with the block size. The complete padded frame can then be streamed to the decoder device and the corresponding padded regions may be cropped outside the visible area of the renderer viewport.

The padding process may easily be implemented using existing texture samplers during GPU encoding. One of the most sensible sampling modes in this case would be replicating the border row and column pixel values, also referred to as clamp-to-edge sampling in OpenGL [18].

The chosen tiling pattern utilizes horizontal slicing, which means that tile widths will always conform to the block size in a padded frame. Tile height however still requires a more best effort -style approach, where some tiles would inevitably have more pixels than others. Especially in high-resolution video modes this discrepancy luckily becomes very small.

Simultaneously another problem related to memory access patterns was resolved. Due to the horizontal slicing, both the tiles and the frame in its entirety form singular continuous blocks of memory both before and after encoding. All tiles may then be represented simply as memory offset and byte size values simplifying array accesses and therefore slightly improving the performance.

Together with the inherent uniformity that comes from block-based constant compression ratio texture formats, the continuity of the data in memory is another valuable advantage worth preserving. Besides the fact that it allows the optimization of the stream delivery functions presented in this work, it can also potentially greatly simplify the application of more specialized high-speed networking technologies, such as RDMA. This is the reason why the slight tile size discrepancies were in the end preferred over e.g. fairer, more square-like tiling methods.

While the Figure 4.4 presented a scenario where a frame was divided into even size tiles, Figure 4.5 shows a more generically applicable example. Here, both frame padding and variable size tiling are visualized.

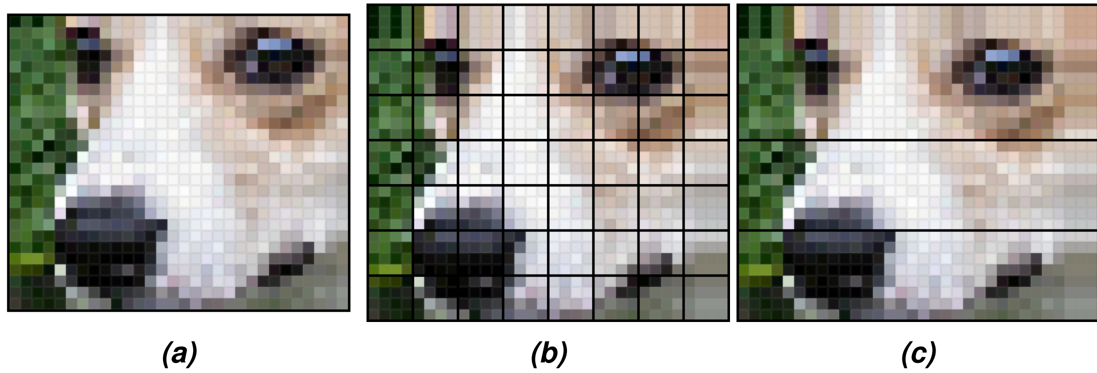


Figure 4.5. *(a) The visible area of a 30x26 frame, (b) the complete frame padded to a 32x28 resolution with a 4x4 block format and (c) tile division using a 3-unit split.*

The texture formats discussed in this work are decoded based on the maximum and minimum values of each individual block. Therefore, replicating the border row an column values should not alter these limits and thus have any visible effects in the border regions of the rendered frame.

In the current implementation of the pipeline, the padded data is streamed to the decoding device as any other data, which admittedly means a tiny increase in the consumed bandwidth. Since the width and height of the extra padded regions is limited to a remainder of less than block size, the extra data can however be considered insignificant especially in high resolutions.

All-in-all, these workarounds concerning the limitations of block-based texture formats are the only strictly video coding -related problem area discussed here. The main contribution lies in the delivery part of the high-bandwidth video stream.

4.3 Stream Delivery

The role of the stream server and client modules is to manage network sockets and act as their abstractions. As the encoder splits the encoded data into packets, it makes a synchronous callbacks to the stream module, which writes the data to the socket of the corresponding unit.

All stream state information, such as what frame packets belong to, which packets have already been transferred, and the correct ordering of the packets, is available to the encoder, making its task very simple. From the decoder point of view, this information becomes much more obfuscated, and some method has to be found to determine how the incoming packets should be processed. To further complicate the problem, the performance and scalability over an arbitrary number of threads and sub-streams remain central.

The decode units have to track two pieces of state information needed to tell what frame a packet belongs to and what its offset, or location, in the frame is. Of these two variables, offset is the more self-explanatory one needed to move the data on its correct place in the decoder frame buffer. Frame index on the other hand is a shared state variable used to track when a frame being received changes and subsequently control buffer updates.

To simplify the socket code and packet handling, the packet size itself is constant over a session. It is negotiated at handshake time and can be fully adjusted to match whatever the network capabilities happen to be. By default the value is set to 1412 bytes to conform to a typical network MTU of 1500 and avoid additional packet fragmentations.

In the very likely case that a compressed frame size is not evenly divisible by packet size, the last remainder packet of the frame is padded to the full packet size. The decoder is then expected to ignore any excess dummy data in the last packet. The main benefit of this arrangement is removing the need to separately code payload lengths to custom application layer headers and thus avoiding the need to make multiple separate system calls to read the payload from the socket.

The decoder is divided into a main thread and a number of unit threads. The main thread is responsible for managing the units and performing GPU buffer updates and rendering. The receive units are then assigned their own buffer addresses, which point to the offsets of the corresponding tiles in the decoder frame buffer. After this, each unit is free to move data to the buffer and then signal the main thread of their state changes whenever a frame is finished or a new one is being received.

The unit threads are designed to operate independently of each other, with their only point of synchronization being the point of frame change. As a frame update signal is received, the main thread will block the unit threads for a short period during which the frame buffer is replaced. The double buffering scheme then allows the main thread to perform a DMA transfer of the data to the GPU memory while the receive units proceed to receive the data of the next frame in parallel.

Each unit is guarded by its own mutex under which unit-specific state variables, such as the target buffer and current frame index or packet offset are maintained. Even though the current frame index is a shared property, each unit maintains its own record of it to avoid the need to synchronize for concurrent data accesses.

The arrangement was designed to improve performance by minimizing as many synchronizations between the main and receive threads as possible. Whatever other common state information that must to be shared between the receive units can also be cached separately by each unit and updated in conjunction with a frame update. The buffer update is also performed at this time, which would block any new receive operations at the moment in any case.

The only remaining problem is how buffer updates should be triggered, or in other words determining when a unit thread has finished receiving and moving frame data to its buffer. In the case of the TCP implementation, the update mechanism was built on top of a frame end signal, while the UDP version required both frame begin and end signals.

One fundamental issue is in separating packets of different sub-streams from each other. A simple solution was to allocate each unit its own receive socket, and use the port numbers as the basis of separation. Besides making it possible for the receive module to assign packets to their correct receive units, it also ensures the best possible scaling of the network I/O, due to socket performance scaling poorly across multiple threads.

In most commercial applications it is usually preferred to avoid reserving excessive number of ports, which could reach up to tens of sockets. However, the solution was deemed to be the best compromise in this case.

Besides the target unit, the rest of the packet meta-information, meaning the offset and frame index, have to be inferred from the packet stream itself. Thanks to the guarantees of the TCP protocol, the issue can in that case be solved by tracking and maintaining the amount of data received.

For UDP the unreliability makes the use of extra application layer headers unavoidable. Although frame changes could theoretically be identified by grouping packets by the time of arrival, the offsets would still have to be encoded into the header to account for packet reordering.

As the Linux kernel network stack already poses a significant per-packet CPU overhead, any additional overheads have to be considered carefully. Depending on the video format and packet size, the amount of incoming traffic can easily reach millions of packets per second.

4.3.1 UDP Stream Delivery

As a completely stateless and connectionless protocol UDP offers little guarantees of packet delivery to the application layer. In other words, the incoming video packets may arrive out-of-order or not at all, meaning that the application has to be able to handle all such errors in software.

In practice, this mandates inclusion of an application layer header to each packet of the video stream. The simple packet format used by the UDP version of the pipeline is presented in Figure 4.6. The proposed 32-bit header contains only two fields — frame identifier and offset.

The header structure is motivated by the desire to make it fast to decode. Frame identifier and packet offset fields fit into a single 32-bit integer variable and match byte boundaries

thus requiring potentially only one array access and two bitwise operations to separate them. After this, the decoder would have all the information required to locate the target position of the payload and which frame the packet is a part of.

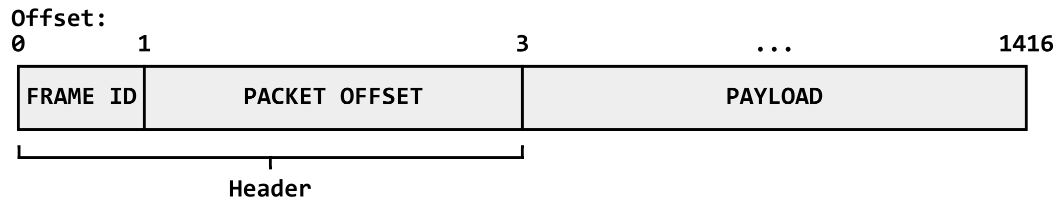


Figure 4.6. *Packet structure used in UDP streaming with a simple two-field 32-bit header and 1412-byte payload.*

The 24-bit packet offset is a relative offset counting from the beginning of the tile of the given unit. Naturally, the length of the offset field creates an upper limit for tile size. In total the address range becomes 2^{24} bytes, or around 16 MB, which is 1.5 times the size of an uncompressed 1440p frame in RGB 24-bit. When streaming uncompressed 8K video over UDP, the minimum unit and tile number would then become six tiles.

In practice, the address space is not a major issue since the streamed frames would be compressed and one would want to use a large number of units anyway to transport larger frames. Regardless, the offset field could also always be simply expanded to 32 bits, which should be enough for any currently available resolution.

In addition, there are alternative ways to code the offset, such as using a packet index instead in place of the byte offset. Since every packet is the same size, the actual offset could then be easily acquired by multiplying the packet index with packet byte size. This would make resulting address range and tile maximum byte size depend on used packet size. For example with only a 128-byte packet size and a 24-bit index the usable address range would already be over 2 GB.

It is the task of the encoder to actually attach the header information to the packets. Problematically, the encoder has to somehow reserve additional 4-byte gaps in the encoded frame data to account for the header, which complicates encoding especially in GPUs. Alternatively, the packet data could be copied to a separate extended buffer before passing it to the send module, but that would not be efficient either.

A compromise solution was developed, in which the encoder overwrites the packet header information on top of previous packet data that has already been sent. This would not be practical if the callbacks to the send module were not completely synchronous, or in general if there were any multiple threads working in parallel inside one unit. Luckily, for the current CPU pass-through encoder implementation, it is not the case.

Including of a header to the packets is however not sufficient to account for all the shortcomings of UDP. If and when packets are completely lost, there is no way to tell on the receiving side when all data has arrived. In principle, the chance of packet losses makes it impossible to achieve perfect timing in buffer updates, which would require triggering as soon as the last in-flight packet arrives.

Instead, a timer mechanism is used to trigger a buffer update in the main thread based on the average receive time across all units. Some padding is also added to account for variations in socket throughputs and system load.

Since the decoder is required to handle non-constant frame rates, the timer also has to adjust to any irregularities in the stream. In order to start the timer, unit threads therefore have to signal the main thread as they start receiving packets where the 8-bit frame index is newer than the current one, signifying that the encoder has begun sending another frame.

Only one unit can enter the frame begin notify -block, which, besides starting the timer, will also update the frame index in other units. Several units are likely to notice a frame change at the same time, but the other threads are instead forced to wait for a signal from the one thread inside the block.

Units that for some reason have not noticed a frame change before the first thread returns may not notice any change at all, as their frame indexes are silently updated. If the previous buffer update has not yet completed at the time, then the notifying thread will also block until buffer update is complete.

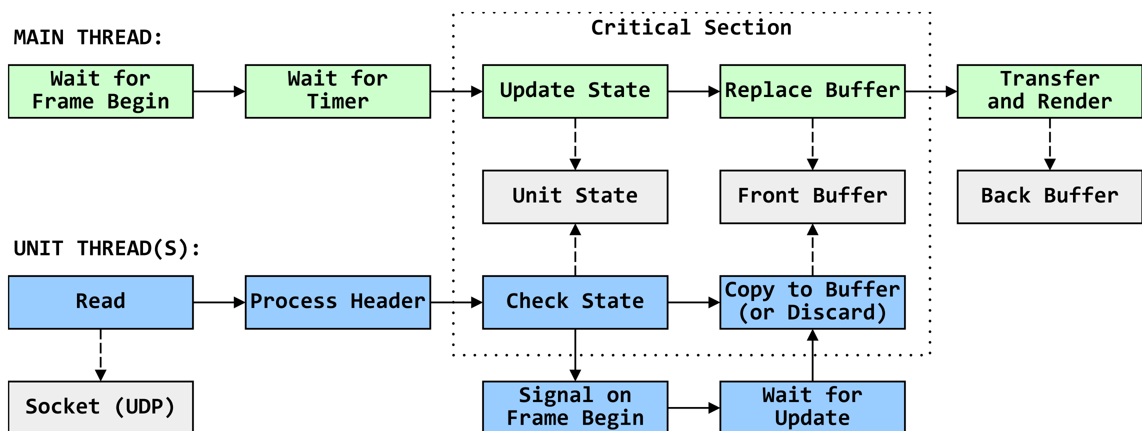


Figure 4.7. *A block diagram of the UDP receive loop and the critical section.*

The multitude of theoretically possible timing variations highlights the fact that parallel multi-socket streaming is by nature very chaotic, since only few timing guarantees can be made. The timing problems are further demonstrated in Figure 4.8, which shows a scenario where multiple conflicting operations occur in a system operating near its maximum

capacity. Despite of this, only one mutex lock in the begin block and a another one per each unit are required.

Other effects of UDP packet loss may be accounted for in an almost semi-passive fashion. If the packet frame index is older than the current frame, then the frame has already been rendered and the packet can simply be discarded. The corresponding segment of the image would not be updated and instead the old data in the buffer would remain there. Due to double buffering, the data would be from the second-last frame. If a packet is lost entirely, the effect will be the same. The resulting rudimentary visual packet loss -compensation mechanism is achieved for free as a side effect of the programming.

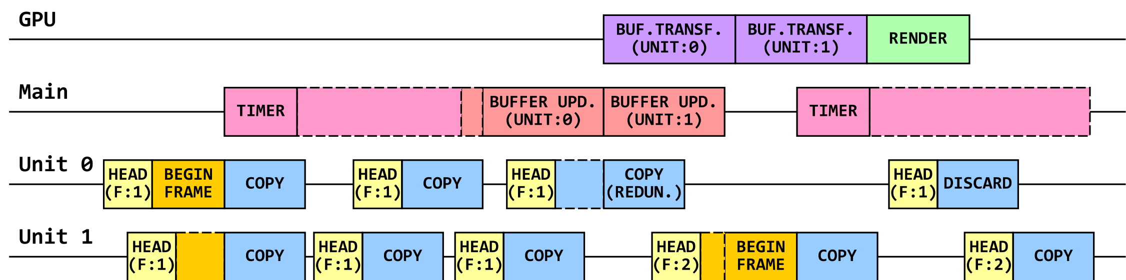


Figure 4.8. *Example of a timer-based multi-threaded UDP receive process highlighting some of the more critical timings and synchronization points.*

Header parsing is complicated by the fact that some risk of bit errors or other bad data is always present in network communications. In the case of wrong memory offsets, they could quickly lead to a segmentation fault in the decoder. To counteract any errors in the packet headers, a quick check has to be performed to ensure that the offset is smaller than tile byte size. Using offsets measured from main tile offset yields a small benefit here, since there is no need to check the parsed unsigned offset against a lower bound, as the lower limit is always zero.

There is a chance that a corrupted offset or frame index for that matter could still be in the valid range. In such case some visible artifact in the form of stuttering or misplaced data would be expected. The only way to try to counter such errors would be to add some kind of parity information to the header, or other similar solutions.

Errors may further accumulate in the payload itself, but luckily, texture compression formats are somewhat resistive to such errors. As discussed previously in Section 2.2, it could either affect only a singular pixel, or distort an entire block, depending on the video format used.

Other, more advanced error and packet loss compensation mechanism could of course in principle be implemented. For example an inpainting scheme could be a way to reduce quality loss. There is of course the question of whether these kind of methods could realistically be made lightweight enough not to impede overall performance.

It should be remembered that the decoder is a real-time system. If the video bandwidth exceeds the effective processing and transfer capacities of the pipeline, stream quality and user-experience will start to degrade one way or another no matter what compensation schemes are in use. More important is that the application can recover for any momentary disturbances.

4.3.2 TCP Stream Delivery

The TCP version of the receive process is somewhat simplified compared to UDP. Since data is guaranteed to arrive in order and without dropping packets, the frame indices and packet offsets previously coded into a header can now be determined by tracking the amount of received data.

Even more importantly, there is no longer a need to perform any packet reordering, which is now handled internally by the TCP implementation of the network stack. Where the UDP version was required to read a packet from a socket into a separate cache to first parse its offset from a header, the TCP units can place the data in its correct place already in the read system call. These guarantees therefore help to cut the required per-packet CPU time into a fraction of the UDP implementation.

Each of the TCP units can maintain its own private record of the stream state and know when all data in its tile has been received. At this moment the receive unit can then enter a frame end block and signal the main thread to perform a buffer update. Unlike in the UDP mode, there is also no need to synchronize at frame begin either, as that was only required for starting the buffer update timer, which is no longer present, and to update the frame indices, which can now be done independently. This TCP receive loop and the critical section, consisting of only the buffer pointer itself, is visualized in Figure 4.9.

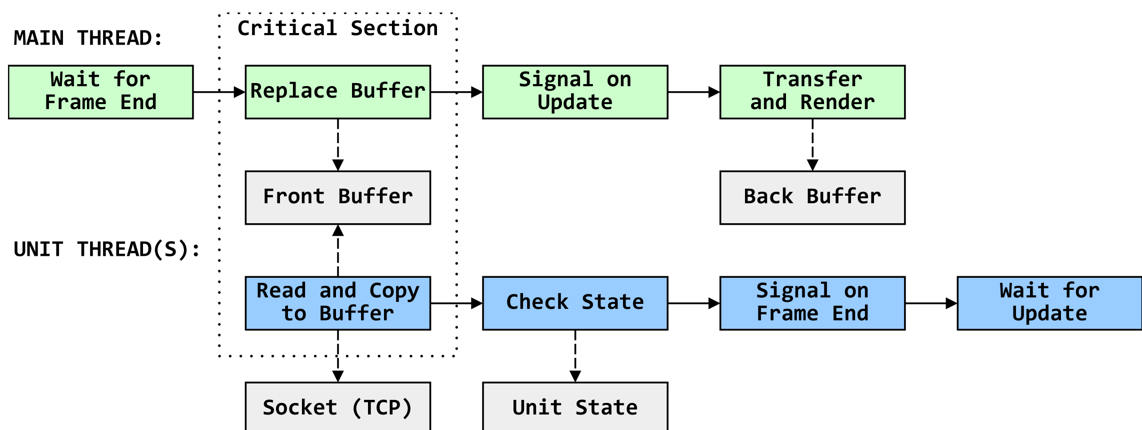


Figure 4.9. *A block diagram of the TCP receive loop and the critical section.*

Buffer updates may even be performed in some units while other threads are still in the middle of receiving data. Although rendering the frame is not possible until the last unit

buffer update is complete, it does at least allow distributing the DMA buffer transfers over a wider time span, which can avoid possible bus bandwidth saturation related to the transfers. This stepped buffer update scheme of the TCP implementation turns out to produce surprisingly large end-to-end latency benefits, which are analysed more in-detail in the results discussion.

The task of the main thread is simplified to waiting for frame end signals and performing the buffer updates in a FIFO order of the finished units. Once all units are finished, the frame can then be rendered. In the meanwhile, the double buffering scheme allows finished units to proceed to wait for the next frame data.

Most of the synchronization issues regarding the TCP mode are relatively minor compared to those of UDP, since the unit states are now internally maintained. The only cross-unit synchronization involves the protection of the frame end block in which units simply add themselves to a buffer update queue and then wait for the buffer update to complete, before proceeding to receive more data. The timing aspects of the TCP receive process are visualized in Figure 4.10.

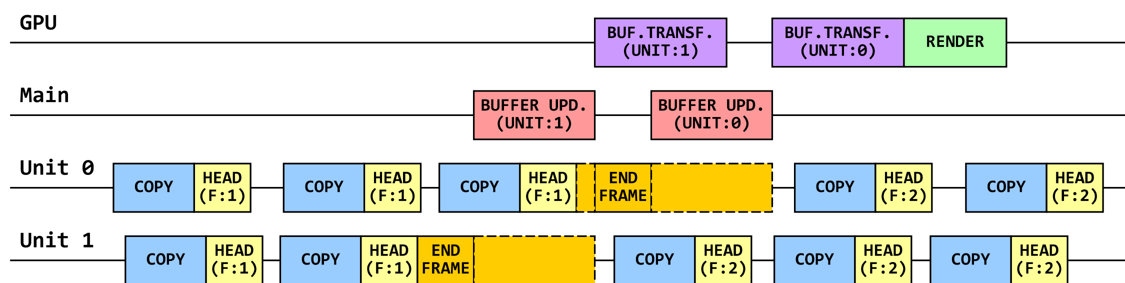


Figure 4.10. *Example of timings in the TCP receive process, demonstrating the stepped buffer update scheme.*

TCP creates an indirect synchronization mechanism between the encoder and decode programs, because the encoder send code is in its current implementation sequential and synchronous to the video encoding. In a case where a TCP connection bandwidth is less than the encoding bandwidth, the socket buffer space would be filled and the encoder would simply start dropping frames.

Frame dropping in the encoder remains, in fact, the only possible way to discard data and adjust to any throughput limitations in the TCP version. Individual packets cannot be dropped, as missing packets in the stream would cause the stream to fall out of sync, although some revisions to the network could maybe be done to counteract it. It is here where the TCP pipeline is very unlike the UDP implementation, in which packets may be dropped or discarded at any time for any reason. This limitation of the TCP implementation remains somewhat inconvenient in practice, as for example it makes any socket read or write errors irrecoverable.

Buffer updates of course still also block all copy operations related to any incoming packets of the next frame within the unit. If the main thread was performing slowly enough, incoming TCP traffic would also become partially blocked, reducing the throughput. The result is a self-regulating congestion control mechanism, meaning there is no need for other measures in cases, where the decoder is not able to keep up with the incoming traffic.

In ideal systems, or close-to-ideal such as highly reliable wired networks, it could be argued that TCP is the more sensible stream transport protocol. Even for less idealized conditions, it is unlikely, that any self-implemented UDP packet reordering and retransmission mechanism in the application layer could outperform the kernel TCP stack. While block-based formats can at least eliminate the need for retransmissions, data reordering in the buffer is still a necessary feature.

However, as previously discussed in Chapter 3, if network reliability or round-trip-time were to worsen, TCP would definitely begin to suffer from some negative performance characteristics. Therefore, based on application layer implementation alone, it is not completely clear how these two protocols would actually compare in various demanding real-world networks.

4.4 OpenGL Buffer Transfers

In the previous sections, stream delivery was discussed more from a CPU-oriented point of view, where data was being simply copied to some buffer in memory. Utilizing double buffering, the filled buffer would then be replaced in a buffer update event, while the contents were decoded and rendered on screen.

From the viewpoint of a receive unit, each unit gets allocated its “own" buffer, or a memory address, which is guaranteed to fit all the tile data by the main thread. These buffer addresses point actually to different offsets within one larger frame buffer, though the units do not need to be aware of it.

In the OpenGL decoder, the buffer addresses are part of a GPU buffer object mapped to RAM. For every received frame, a DMA buffer transfer operation is performed from RAM to GPU, followed by a render call to whatever display surface is in use. Thanks to the hardware-accelerated decoding, rendering the texture to a quad itself takes a time of only tenths of a millisecond even in 4320p resolutions and modest hardware. The buffer transfer latency on the other hand, is much more significant by an order of magnitude.

The received data is always expected to be in a valid texture format, which begs the question of why use GL buffer objects for storage instead of textures directly. The answer lies in the fact that the various memory mapping solutions available for buffers in the OpenGL (ES) API make them the obvious storage type of choice for the pipeline.

Unfortunately, the buffer data has to be translated into a texture object for rendering, and also to make texture samplers and parameters work in general. The chosen solution to the problem was to perform a separate copy operation from the buffer object to a texture object, which has to be done after the buffer transfer from RAM has completed. Although in any other part of the pipeline any additional copy operations would be highly problematic due to latency increases, the buffer to texture -copy is less so due to the fact that the operation takes place inside the VRAM and is thus extremely fast even for large uncompressed frames.

There does exist one special alternative to this, which is called a buffer texture object and has been available since OpenGL version 3.1. As per its name, it is a type of texture where the pixel data is retrieved from the buffer behind it. However, these buffer texture objects only support uncompressed texture formats, and the use of e.g. sampler parameters is also restricted, making their use infeasible. [42]

Regarding multi-bitplane video formats, the decoder is designed to create multiple buffers and textures for each unit group. Specially designed GLSL fragment shaders can then be used to combine the bitplane textures at render-time.

In OpenGL, a buffer map calls take as input the target buffer itself, and a number of required or optional access flags, which either act as performance hints, or cause drastic alterations to how the mapped buffer is used. In addition, the mapping can only target a specific part of the buffer, in which case offset and byte size values are also included. In the case of the decoder, there is never need to read from these buffers, which is why they are mapped as write-only. Reading from such buffers in RAM will thus constitute as undefined behaviour. [18]

Upon a successful return of the call, the OpenGL implementation will have allocated a new region for the mapped buffer somewhere in RAM. This is the address that will then be passed to receive units when offset appropriately.

Up to this point, no data has yet been transferred to the GPU. In the initial versions of the pipeline, the transfer would occur when an unmap call was made to the buffer, indicating that the data could now be transferred to the GPU. The transfer would then be followed by a buffer to texture copy call and finally the render call. Simultaneously, the second buffer in the double buffering scheme was mapped to RAM and the next frame would be received to it.

As could be expected, the buffer map and unmap calls are relatively costly operations [43]. Although the remapping calls were only needed typically only tens of times per second, which could be considered modest compared to some other rendering and stream processing applications, they still had a small measurable impact to the buffer transfer latency.

Moreover, by default the OpenGL buffer mappings are considered synchronous operations. Any pending operations still using the buffer are required to complete before mapping takes place, meaning that the decoder main thread is blocked and the relevant tasks effectively become serialized internally by the driver. In other words, this ensures that any pending render operations complete before new data can be written to the buffer. In some circumstances, it may however be preferable that the main thread is given more control over what tasks truly require synchronization and which can run in parallel.

The API provides a special unsynchronized-flag in the map call, which can remove the CPU–GPU synchronization point [18]. It is then up to the application to ensure that any applicable tasks related to the buffer are first completed in both the GPU and CPU.

Although unsynchronized buffers could on a surface level appear a prefect solution, they have a somewhat unintuitive drawback on the GPU driver side. These unsynchronized map calls can cause new task serializations within the driver itself, producing apparent stalls in driver communications. [43]

Here, it should be mentioned that other interesting buffer mapping flags include the explicit flush option, where an implicit complete buffer flush at the unmap call is replaced by explicit flush range -calls [18]. What this in effect does, is it allows controlling what parts of the buffer are transferred and when, which could in theory be used to make small progressive buffer transfers as the receive threads write data to the buffer.

The ultimate solution to the buffer transfer performance problem is to do away with buffer remapping in its entirety. Starting from OpenGL 4.3, or OpenGL ES 3.1, the standards include support for so called persistent and coherent buffer storage [44], which is the approach chosen in the current implementation of the pipeline.

Use of the persistent flag indicates to the OpenGL driver, that the mapping of the buffer is to be permanent over a longer time period, and it can be accessed by both the GPU and CPU without additional buffer mapping calls in between. The buffer transfers are then either handled by the same previously mentioned explicit flush calls, or by making the mapping coherent. [44]

The coherent flag is optional to persistent buffers and it locks the mapped section of the RAM in place preventing paging and making it effectively pinned. Such memory is then always accessible to the GPU device and therefore does not require performing any separate transfer operations. [44] Instead the transfer occurs implicitly when the data is copied from the buffer into the texture.

As a side effect, persistent buffering tends to simplify the programming to a degree, since both of the double buffers only have to be mapped once, and then they can be written to directly without need for flushing either. The buffers do however have to be both allocated and mapped using the persistent and coherent flags, and some type of a fence synchro-

nization or a finish call has to be done following rendering [44]. Luckily rendering itself is very fast, so in practice the chance on actual contentions happening should be near zero.

The buffer transfers, or copies to texture, do still take substantial amounts of time, as they are limited by the PCIe bus throughput. This is where the previously discussed stepped transfer method of the TCP implementation can still be a substantial improvement.

While the discussion has mostly concerned the decoder, almost all of the buffer management strategies are also usable in any GPU encoder implementation as well. The encoder would either manage each encode unit separately and stream data from each unit's own output double buffer, or perform the encoding in a singular pass over the entire frame and manage one larger buffer similarly.

Whether either option is preferable would depend on how the encoder input is received. In its current form the encoder accepts as input only complete frames, but the implementation could be converted into a stream-based model instead. The encoder input may also become from a render buffer of a 3D application running within the same GPU, in which case the input data would already be available in GPU memory and there would be no need for any input buffer transfers at all.

There also exists some unknown factors that would need to be studied more closely. These include determining the optimal balance between using singular large buffer objects, or multiple smaller buffers and piecewise transfers resulting in larger numbers of API calls. Likewise, any vendor-specific driver and hardware differences in performance characteristics appear to be poorly understood. Regardless, the strategies approached here should be applicable for a wide range of GPU-accelerated stream processing applications.

5 RESULTS AND EVALUATION

Performance tests were performed in a localhost environment using the previously described CPU pass-through encoder and the OpenGL decoder. The main goal of these tests was to evaluate the performance of the technical solutions discussed throughout Chapter 4. These include the performance impacts of the UDP and TCP implementations, Linux kernel, and GPU buffer transfer strategies.

The primary metric was the delivery latency, which consists of the time between an encoded frame entering the send loop, and the frame being rendered on screen. Display device latency is excluded. Therefore, the two interesting sub-components of the delivery latency are the network and buffer transfer latencies.

Based on the performance measurements, it is apparent that the described implementation has some shortcomings. Therefore, a final set of future work proposals will also be presented to address the identified problems.

5.1 Performance Measurements

The key benefit of the localhost tests using loopback adapters is that the endpoint applications share the same system clock, which can be used as a timing reference. One drawback is that the network adapter queue transfers, receive interrupt handling, and hardware offloading are all left out, since the virtual loopback interfaces do not implement the layers 1 or 2 of the stack.

The approach is arguably not ideal, as some of the packet processing work covered in Chapter 3 has to be either performed by the CPU, or skipped entirely. On the other hand, concentrating on the software stack will at least make the results of the analysis universally applicable, because the effects of adapter-specific hardware differences get eliminated.

It should additionally be noted, that the encoder and decoder compete over the same system resources. While the used unit thread numbers have been chosen to reduce the need for context switching, memory access still has to be shared. In any other kind of application it would not be a major issue, but for the extremely intensive I/O workloads in question it is likely to play a role.

For similar reasons, using a feature-complete GPU encoder together with the GPU decoder would have also been infeasible, assuming that only one graphics card was installed. Instead, the pass-through CPU encoder was preferred due to its small performance overhead. Likewise, the video source adapter was made capable of pre-loading video data to RAM, removing the need for runtime file system operations.

The shortcomings are also partially offset by the fact that data is not required to cross any physical networks and only needs to be shifted around between RAM and VRAM instead. Therefore, the localhost tests are still able to give an approximation of the performance characteristics and capabilities of the pipeline in the high-bandwidth low-round-trip networks it was primarily designed for.

A high-end desktop work station was used to run the performance tests, where the hardware was optimized for both high single-thread performance as well as reasonable multi-threading performance. The relevant hardware specifications have been collected in Table 5.1.

Table 5.1. *The performance test computer specifications.*

	Model	**Notes**
CPU	AMD Ryzen 9 3950x @ 3500 MHz	16-core/32-thread
GPU	Nvidia GeForce RTX 2070 @ 1620 MHz	PCIe 3.0 x16
RAM	4x8 GB DDR4 CL16 @ 3600 MHz	Dual-channel

The objective of the first tests was to evaluate the frame size impact to the delivery latency. Tests were run using a variable frame resolution from 144p to 4320p, the uncompressed 24-bit RGB format, 16 stream units and a packet size of 1408 B. As the delivery latency is only affected by the byte size of the frame, and not its format, the results can be easily extrapolated to compressed formats by multiplying the results with the compression ratio.

Parts of the 4320p results of both UDP and TCP pipelines in time domain are presented in Figure 5.1, demonstrating the typical latency variation in the first 20 frames of video with a 10 Hz refresh rate — a time span of two seconds. The plotted values include the median latency of 1000 runs, and their 5th and 95th percentiles. The delivery latencies as the function of frame byte size excluding the first five frames are presented in Figure 5.2 for the same 1000 run series.

From the results, it becomes apparent that a small amount of time is required for the stream to stabilize. Firstly, the slow start effect of the TCP protocol is easily detectable in the first frame latency of the TCP version, which it is not present in the UDP data. Secondly, the UDP version instead suffers from more general instability where the measured latency can even go down.

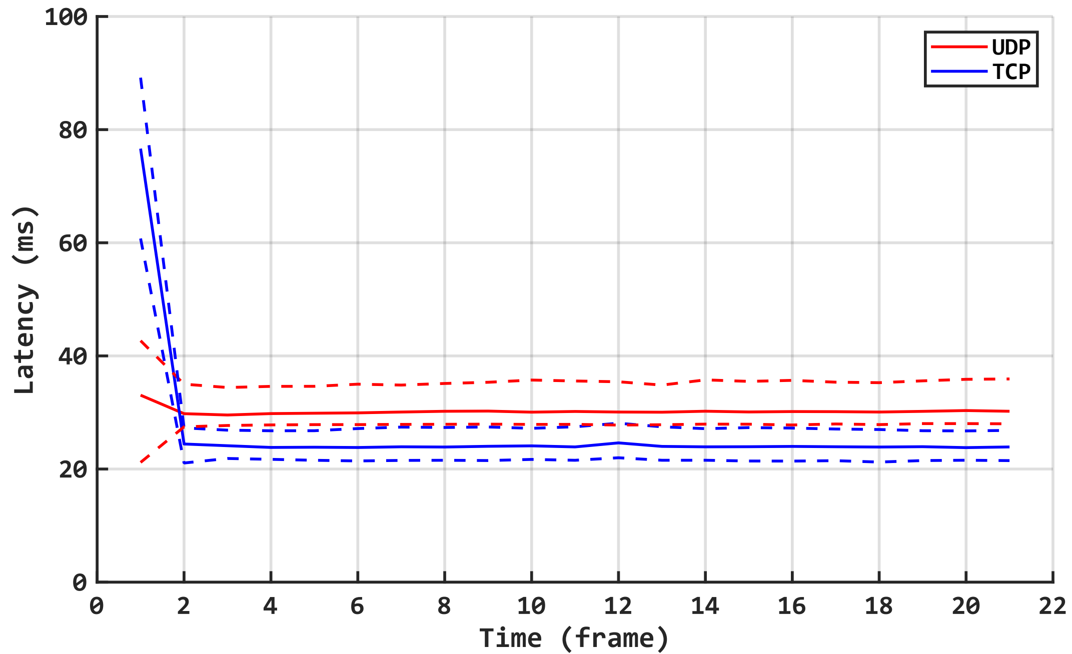


Figure 5.1. *The 5th, 50th, and 95th percentile latencies of the first 20 frames in 4320p resolution and RGB 24-bit format over a series of 1000 test runs using 16 units.*

The primary cause of the UDP fluctuations is the implementation of the frame update timer mechanism, which can take multiple frames to determine the optimal delay. In conjunction with the timing issues, the system thread scheduler is also likely to affect the socket startup performances of both protocols, as it also needs to train for the sudden jump in workload.

Previously, there also existed a version of the pipeline, where both the TCP and UDP implementations used two threads per decode unit, handling socket reading and packet copying separately. Tests run with the two-thread version showed a longer, approximately 35 frames long settling in -period and a few millisecond latency increase. The effect was determined to be caused by the thread scheduler, and it disappeared following the move to single-thread receiving.

The impact of the scheduling effects is challenging to isolate from the slow start and timer adjustment mechanisms, since the latter are more dominant. Likewise, at this stage it is difficult to fully rule out any other potential resource use -related adjustment effects within the kernel network stack.

The median delivery latency for the UDP pipeline is 30 ms, and for the TCP version 24 ms, with 5th and 95th percentiles being 28–36 and 22–27 ms respectively. The range of fluctuations being this large even in the localhost tests shows that the network stack, thread schedulers, and OpenGL drivers perform quite chaotically even in close-to-ideal conditions.

When studying the impact of the frame size to the median latency in Figure 5.2, the performance behaviour becomes more predictable. As expected, the delivery latency is roughly proportional to the frame resolution and thus its byte size, while the format itself has little effect. The video format only plays a significant role in the encoding times, which are not included in these numbers.

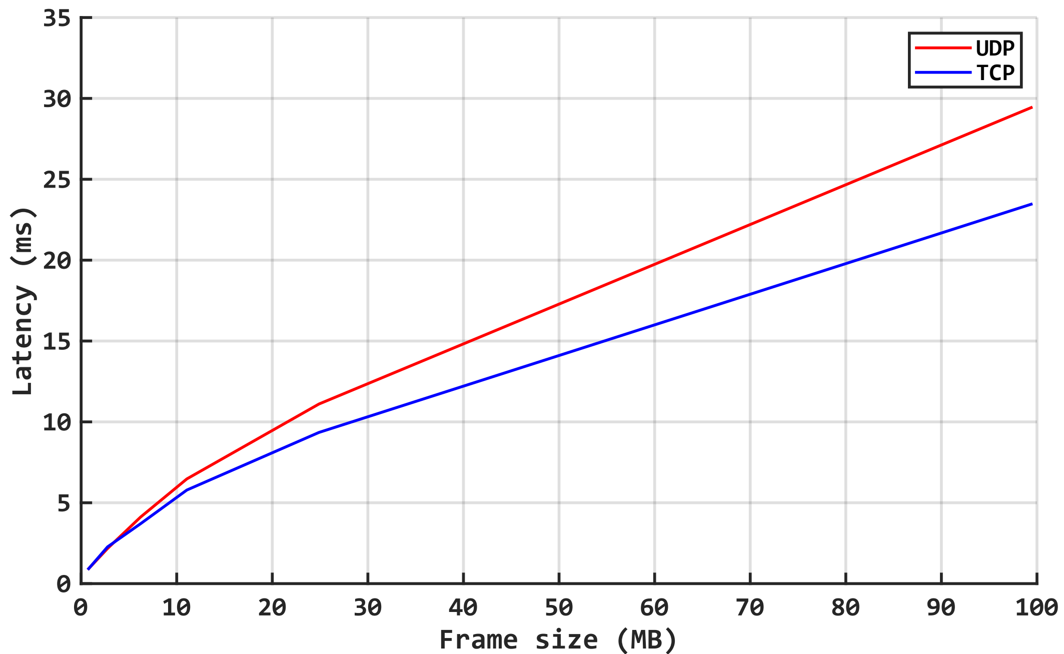


Figure 5.2. *Delivery latency measurements compared to frame size over a series of 1000 test runs using 16 units and 1408 B packets.*

In the case of an 4320p frame in 24-bit RGB format, the frame size is approximately 99.5 MB, which translates to 31.7 and 25.3 Gbit/s of throughput for the TCP and UDP versions respectively. Intuitively, the UDP latency curve has a slightly higher slope compared to the TCP due to packet processing overheads and one additional copy operation in the receive path. In smaller frame sizes, UDP still manages to match the TCP performance.

The results imply that even a small increase in compression ratio will produce a linear decrease in latency. When accounting for the network bandwidth and round-trip time, the total could easily reach tens of milliseconds for the uncompressed 24-bit RGB format in 4320p. In other words, this would give an advantage to higher compression ratio formats such as BC1 over BC3 for example, as halving the frame size means halving the delivery latency.

Concerning the results presented in Figure 5.2, the latency curve is directly related to the maximum frame rate of the system at a given frame size. For example, BC3-encoded 4320p video with a delivery latency of around 12 ms in UDP could be streamed just barely at 60 frames per second, or at 16.7 ms per frame.

These frame-based bandwidth measurements are of course not fully representative of the true overall throughput. Overlaps between decoder buffer transfers and encoding mean that the true maximum frame rate is slightly higher.

While the UDP implementation is at a disadvantage in the localhost environments, it does at least benefit from being able to discard late packets. TCP on the other hand forces the decoder to wait for all the data to arrive, meaning the slowest unit and TCP connection determine the latency of the entire pipeline.

If all units were streaming data at similar rates, this would not be a major factor, but further analysis of parallel socket performance variations indicates that this is not the case. More specifically, tests were run to measure how much time individual units were spending in the encoder and decoder send and receive loops on average. The results presented in Table 5.2 demonstrate that there is a lot of random throughput fluctuations between the sockets.

Table 5.2. *Average sub-timings for an uncompressed RGB 24-bit frame using a 4320p resolution and 1408 B packet size.*

	Send Loop (unit avg.)	**Send Loop (unit max.)**	**Recv Loop (unit avg.)**	**Recv Loop (unit max.)**	**Total**
UDP	13 ms	20 ms	17 ms	23 ms	31 ms
TCP	11 ms	19 ms	12 ms	20 ms	24 ms

It turns out, that the slowest unit is typically transferring data around 50 % slower than the average in the case of both protocols. It is certain, that at least some of the 5th and 95th percentile fluctuations seen in Figure 5.1 are caused by these throughput variations.

In the UDP implementation these socket performance differences manifest as some late data getting discarded. The latency itself is mostly unaffected, as the update timer is based on the average receive time of all units. In accordance with these results the UDP frame update timer is rather tightly optimized to trigger after a period of 1.5 times the average receive time from the first packet of a new frame arriving.

The measured socket throughput variations have a much greater effect on the TCP pipeline, as its latency is directly tied to the slowest sub-stream latency. However, thanks to the stepped buffer update mechanism, much of the waiting time is spent on buffer transfers, and it more than makes up for having to wait for the slowest units at least in these tests.

The impact of the stepped buffer update method was tested using a modified implementation of the TCP pipeline, in which buffer updates were instead blocked until the last receive unit had finished, much to the style of the update timer in the UDP version. While

the normal version had a delivery latency of 24 ms for a 4320p frame, the deferred transfer version scored only 29 ms.

Both the UDP and deferred TCP buffer transfer latencies were measured to be around 8 ms. This aligns with the numbers in Table 5.2, where the UDP total latency is around the sum of the measured buffer transfer time and the maximum receive time multiplied by 1.5.

These preceding tests were run using a unit number of sixteen, which should represent an optimal number of threads in a 16-core system with hyper-threading. In order to affirm this, a set of tests were also run using a variable number of units demonstrating the scalability of the pipeline. These results are presented in Figure 5.3, showing that both UDP and TCP performance curves level out at around 16 threads, which is expected.

For both protocols the socket read and write calls form a large part of the used CPU time. Therefore, an important method of reducing the time spent in socket-related system calls is to produce less packets overall. Thankfully, the pipeline was developed to support variable packet sizes, which makes it possible to directly measure the performance effect, as presented in Figure 5.4.

It should be noted, that for the UDP pipeline the packet size has a concrete meaning as the length of the actual datagram sent over network. For the stream-oriented TCP protocol, the packet size represents a more abstract concept as an atomic segment of a tile. Nevertheless, in both cases the packet size parameter is linked to the number of system calls and other operations required to transfer a complete frame.

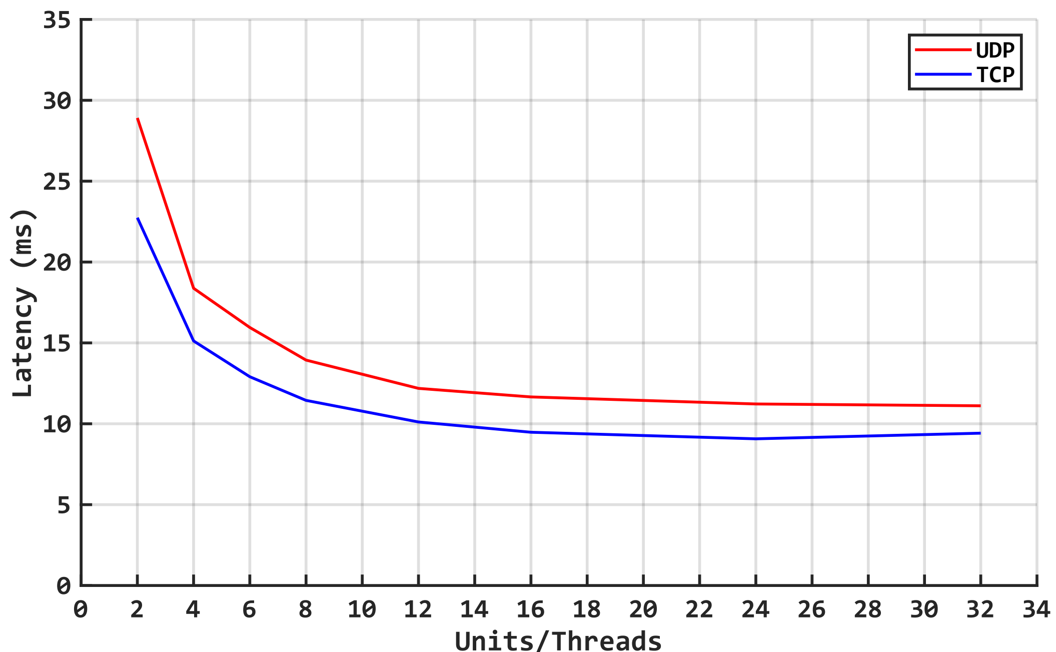


Figure 5.3. *Delivery latency as a function of units and threads using an RGB 24-bit 2160p format and 1408 B packet size.*

From Figure 5.4, it can be seen that the packet size reduction had an immense effect on both of the protocols. When comparing to the default 1408-byte packets, the performance difference shows that the limiting factor is indeed in the CPU processing overheads within the stack.

Besides CPU time, the second most important limitation is the simple memory bandwidth, which would not be impacted by packet size adjustments. As previously mentioned, it is effectively reduced by the fact that the encoder and decoder are competing of the same bandwidth in the localhost tests. The memory frequency in the test system is however higher than most platforms, which likely alleviates some of the problem.

The impact of the memory clock frequency was experimented by performing the same tests with a 3 200 MHz clock rate instead of the 3 600 MHz used in other tests. The 400 MHz decrease in the frequency produced a clearly measurable 10–15 percent increase in latency for both protocols, which makes sense considering that the workload is extremely I/O intensive. The CPU clock frequency, on the other hand, was not modified in any of the tests discussed here.

UDP does not seem to benefit proportionally from the packet size increase any more than TCP, despite its higher per-packet CPU overhead in the application layer. The notion could be explained by the fact, that although TCP abstracts packet-based messaging behind the socket API, the L4 implementation still has to perform packet reordering and retransmission internally.

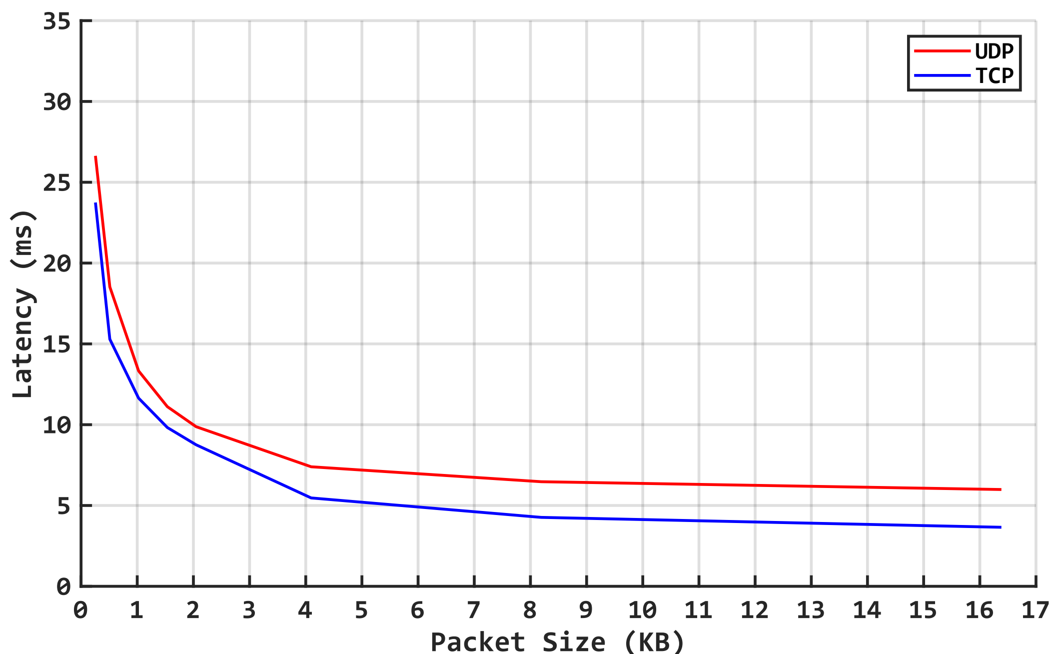


Figure 5.4. *Latency as the function of packet size using the 24-bit RGB format, 4320p resolution and 16 units.*

It can be inferred that the pipeline could easily benefit from large MTU -technologies such as the Ethernet jumbo frames. In the case of the stream-based TCP implementation the packet size could also be increased just to reduce the number of needed socket-related system calls.

Regarding kernel network stack configuration, the UDP and TCP receive and send buffer size limits were increased to a more reasonable value of around 16 MB per socket, following the discussion in Section 3.3. The tests were able to verify that this had a considerable effect on the throughput in the TCP pipeline and the packet drop rate in the UDP version, which could exceed 50 % in some circumstances. After the increase, no more packet losses were observed outside the startup adjustment phase.

The rest of the discussed network stack or socket options had more minimal impacts. Testing busy polling in localhost was not possible due to loopback interfaces lacking the support for it, though as previously discussed, it is likely that the results would have been significantly worse. All the TCP tests were run using the cubic window scaling function due to its favourable performance characteristics. TCP socket options such as Nagle's algorithm or delayed acknowledgement, did not have measurable effects to performance at all, as was expected for low-round-trip connections.

Due to some of the discussed complexities of the network stack, and the operating system environment as a whole, it is in general difficult to determine the perfect combination of tweaks to reach the absolute minimum latency. Instead, the most favourable path would likely be to direct these efforts to improve the code instead of the configuration.

Regardless, the pipeline in its current form is more than enough to assess the feasibility of texture compression formats for video streaming. In addition, the lessons learned in this work should have a wide range of applications in the fields of high-performance network communications and stream processing.

5.2 Future Work

Despite the extensive efforts, the pipeline presented here still has a number of performance shortcomings, that could be fixed. The purpose of this section is to propose solutions to these issues, and also to point out possible future work topics, including some outside the area of stream delivery.

Several of the performance enhancements concern the fact, that both TCP and UDP sockets presented similar non-ideal performance behaviour in the context of multi-socket streaming. For both protocols, the delivery latency becomes at least partially constrained by the slowest socket in the pipeline. The results showed that the socket throughput discrepancy was a major contribution of 4–8 ms to the 4320p latency.

In the localhost tests the issue could be solved by adjusting the system thread scheduling and possibly moving towards harder real-time environments. However, thread affinity still does not address issues like packet-loss-induced congestion control in the transport layer. A more general solution would be to directly balance socket workloads and resources.

Ideally, both UDP and TCP pipelines would use a load-balancing scheme to balance the socket throughput variations. While multi-threaded socket accesses have been shown to be inefficient, an equivalent system could instead be implemented by using finished units to take and transfer some of the pending data of the slow-performing threads. As units would not be required to share socket accesses, the strategy should be more efficient than e.g. a more typical thread-pool implementation.

Since units currently identify their packets by the port numbers, any load balancing system would require encoding the packet unit into its header. This is not necessarily a problem for the UDP pipeline, since the existing header offset field could be re-purposed to count from the beginning of the frame, thus carrying the unit information with it.

For TCP, the implementation of load balancing through unit re-allocation would require the use of additional control messages, or inclusion of application layer headers. TCP could also benefit from being able to discard late data, like the UDP version, at least in the case of significant disturbances in individual connections.

Additionally, an already proven solution would be to implement the stepped buffer transfer mechanism in the UDP pipeline. Such change should then produce a latency improvement of around 4 ms in the 4320p tests — or 50 % of the buffer transfer time — like in the TCP version.

Another point of improvement would involve the superfluous copy operation in the UDP receive path, which is not present in the TCP implementation. The copy operation in question was originally added to account for packet reordering the UDP protocol, and involves reading the packet into an intermediary buffer to allow header processing before copying to the correct frame buffer offset.

However, considering that the pipeline was designed for low-round-trip networks, it may not be necessary to reorder all of the UDP packets in the first place. In such short-distance networks it is likely that packets arrive to the receiving device in the order they were sent in. Therefore, it could be interesting to develop an UDP implementation which predicts the offsets of incoming packets, checks the header offset afterwards and then re-orders the few out-of-order packets within the frame buffer.

The only problem would be in handling the header information, as receiving it directly into the frame buffer would be destructive to existing data there. A straightforward mitigation would be to move the header section to the end of the UDP packet, removing most overwrites of valid data, assuming that the encoder sends packets in order. Some neigh-

bouring blocks could still be corrupted by out-of-order packets. Whether this is acceptable would depend on the application.

The estimated latency save of removing the excess copy from the UDP path would be in the range of multiple milliseconds. When added together with the other potential performance optimizations, it should be fully possible to bring the UDP pipeline latency to the level of TCP in the localhost environments.

Even though the current UDP implementation performed consistently worse than TCP in all of the tests, there is strong reason to believe that UDP could even surpass the performance of the TCP pipeline. The case for UDP is even stronger in lossy real-world networks, due to reliable networks tending to favour TCP streams.

There are of course other topics besides the transport protocols and socket optimizations, which have been left outside the scope of this work. Even concerning transport and application protocols, there exists a variety of alternatives to UDP and TCP, that should also be acknowledged. Other similarly excluded topics would include multi-path delivery and adaptive streaming, or in other words, dynamic runtime adaptation to network condition deterioration through quality adjustments.

If the requirement of relying on only general-purpose compute platforms was loosened, there would also be immense potential for utilizing more specialized networking solutions. The end-host network I/O performance could, for example, be improved by technologies such as RDMA and GPUDirect, or other kinds of acceleration, like FPGA-based smart NICs.

Some other topics outside stream delivery include video encoding and packet loss compensation, which have been covered here barely at all. Suffice to say, that although texture compression has shown great potential in very low-latency video streaming, a great deal of work remains.

6 CONCLUSIONS

The growing interest towards remote and distributed computing, and the recent emergence of ever more ambitious edge computing paradigms, has motivated great developments in the field of high-performance networking. At the same time, compute performance has seen only modest improvements, which underlines a fast-growing gap between the networking and end-host performances. In other words, the software stack has become a major problem for high-performance network connectivity, especially in the consumer-grade mobile and IoT devices.

One particularly challenging area is in real-time video delivery, where the problems of latency and bandwidth are combined. Against the described background, the texture compression formats have become an interesting option for real-time video delivery. These formats offer extremely fast GPU video coding at the cost of low compression ratio.

Conventional video formats involve bandwidths of typically tens, or at most hundreds of megabits of traffic per second. Texture-compressed video, on the other hand, would increase high-resolution video stream bandwidth by a factor of two to three orders of magnitude. This makes the delivery of the video a massive challenge, even in modern high-performance networks.

The objective of this work, was to study the technical aspects of texture-compressed video stream delivery in general-purpose Linux systems, with additional interests in mobile devices and wireless networking. A video streaming pipeline was therefore developed, utilizing heavy multi-threading, standard Linux socket interfaces, and GPU-acceleration.

Besides their low compression ratio, texture formats have a multitude of other, more interesting properties discussed as part of the work. Most of these are consequences of the associated fixed-size block formats, that allow performance optimizations by simplifying stream processing greatly. The formats are also highly error- and loss-resistant making them well suited for streaming over unreliable networks.

Due to the aforementioned fact that the software stack is the limiting factor, an overview of the Linux kernel network stack followed, concentrating on a layered view of the send and received paths. Multi-socket streaming was concluded to be the most scalable method for very high-bandwidth video delivery, among other optimizations such as socket buffer size tuning. Two different transport layer protocols, TCP and UDP, were also studied, with

a focus on their latency and throughput performance characteristics.

The need for network I/O scalability reflected into the proposed pipeline architecture through the use of multiple sub-streams, or delivery units. These units would also be relevant to GPU buffer management, where modern OpenGL features were made use of to efficiently stream received data into GPU memory.

The latency and throughput performances of the proposed video delivery pipeline were evaluated in extensive localhost tests. The TCP version of the pipeline proved to perform the best in the close-to-ideal conditions of the localhost environment, reaching 31.7 Gbit/s, while UDP reached 25.3 Gbit/s. In terms of latency, these bandwidths translate to 24 ms and 30 ms for uncompressed RGB 24-bit video in 4320p resolution. These latency values do not however include the network latency, only that of the network stack.

Both protocols experienced problems of uneven socket throughput rates between different units, which appears to be an inherent problem of multi-socket streaming using the network stack and the kernel thread scheduling. The performance imbalance was a problem for both protocols, where the delivery latency became limited by the slowest unit in the pipeline. A stepped buffer update scheme was developed for the TCP implementation in order to partially counteract the, allowing partial buffer transfers while data was still being received.

While the performance of the UDP pipeline was consistently worse, there is reason to believe that when properly optimized, it could even surpass the TCP implementation. TCP especially benefited from the localhost test setup, which could not replicate a real-world lossy network environment.

Since the delivery latency is proportional to only the frame size, the results could be extrapolated for compressed formats. When factoring in the latency of the network, results imply, that it should be within the realm of possibilities to use wireless 802.11ax-networks to deliver BC1-compressed 2160p video at 60 Hz with a latency of 10 ms.

The delivery of texture compressed video stream remains a demanding problem. In the case of the latest wireless standards, like millimeter-wave 5G or 802.11ax, texture compression can be argued to be on the edge of feasibility. The kernel CPU overheads, and the network bandwidth remain the main issues. For high-performance wired networks and devices, texture compression would be viable already.

Wireless networking at bandwidths of over 10 Gbit/s may not be realistic in the near-future, but the issue of processing overheads is solvable already today. Hardware-offloaded network I/O, or any other zero-copy networking schemes could for example be enough. Considering the potential market for the applications, and the great efforts involved in the development of edge computing technologies, this may be enough to pave way to the future use of texture compression in real-time video delivery as well.

REFERENCES

[1] Lema, M. A., Laya, A., Mahmoodi, T., Cuevas, M., Sachs, J., Markendahl, J. and Dohler, M. Business Case and Technology Analysis for 5G Low Latency Applications. *IEEE Access* 5 (2017), 5917–5935. DOI: 10.1109/ACCESS.2017.2685687.

[2] Porambage, P., Okwuibe, J., Liyanage, M., Ylianttila, M. and Taleb, T. Survey on Multi-Access Edge Computing for Internet of Things Realization. *IEEE Communications Surveys Tutorials* 20.4 (2018), 2961–2991. DOI: 10.1109/COMST.2018.2849509.

[3] Zhang, K., Hu, J. and Hua, B. A holistic approach to build real-time stream processing system with GPU. *Journal of Parallel and Distributed Computing* 83 (2015), 44–57. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2015.05.002.

[4] Hanford, N., Ahuja, V., Farrens, M. K., Tierney, B. and Ghosal, D. A Survey of End-System Optimizations for High-Speed Networks. *ACM Comput. Surv.* 51.3 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3184899.

[5] Mayberry, M. The Future of Compute: How the Data Transformation is Reshaping VLSI. *IEEE Symposium on VLSI Technology*. 2020, 1–4. DOI: 10.1109/VLSITechnology18217.2020.9265068.

[6] Kissel, E., Swany, M., Tierney, B. and Pouyoul, E. Efficient Wide Area Data Transfer Protocols for 100 Gbps Networks and Beyond. *Proceedings of the Third International Workshop on Network-Aware Data Management*. Association for Computing Machinery, 2013. ISBN: 9781450325226. DOI: 10.1145/2534695.2534699.

[7] Jiang, X., Shokri-Ghadikolaei, H., Fodor, G., Modiano, E., Pang, Z., Zorzi, M. and Fischione, C. Low-Latency Networking: Where Latency Lurks and How to Tame It. *Proceedings of the IEEE* 107.2 (2019), 280–306. DOI: 10.1109/JPROC.2018.2863960.

[8] Wiegand, T., Sullivan, G., Bjøntegaard, G. and Luthra, A. Overview of the H.264/AVC video coding standard. *IEEE Trans. Circuits Syst. Video Technol.* 13 (2003), 560–576.

[9] Holub, P., Šrom, M., Pulec, M., Matela, J. and Jirman, M. GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video. *Future Generation Computer Systems* 29.8 (2013), 1991–2006. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.06.006.

[10] Nah, J.-H. QuickETC2: Fast ETC2 Texture Compression Using Luma Differences. *ACM Trans. Graph.* 39.6 (Nov. 2020). ISSN: 0730-0301. DOI: 10.1145/3414685.3417787.

[11] Waveren, J. M. P. and Castaño, I. *Real-Time YCoCg-DXT Compression*. Tech. rep. Nvidia, 2007.

[12] *Texture Block Compression in Direct3D 11*. Microsoft. URL: https://docs.microsoft.com/en-us/windows/win32/direct3d11/texture-block-compression-in-direct3d-11 (visited on 03/01/2021).

[13] Brown, P., Stewart, I., Haemel, N., Pooley, A., Rasmus, A. and Shah, M. *S3TC Texture Compression*. Ed. by S. Grajewski. Khronos Group Inc. July 15, 2013. URL: https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_texture_compression_s3tc.txt.

[14] Ström, J. and Akenine-Möller, T. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. *Graphics Hardware*. Ed. by M. Meissner and B.-O. Schneider. The Eurographics Association, 2005. ISBN: 1-59593-086-8. DOI: 10.2312/EGGH/EGGH05/063-070.

[15] Ström, J. and Pettersson, M. ETC2: Texture Compression using Invalid Combinations. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. Ed. by M. Segal and T. Aila. The Eurographics Association, 2007. ISBN: 978-3-905673-47-0. DOI: 10.2312/EGGH/EGGH07/049-054.

[16] Nystad, J., Lassen, A., Pomianowski, A., Ellis, S. and Olson, T. Adaptive Scalable Texture Compression. *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. Ed. by C. Dachsbacher, J. Munkberg and J. Pantaleoni. The Eurographics Association, 2012. ISBN: 978-3-905674-41-5. DOI: 10.2312/EGGH/HPG12/105-114.

[17] *OpenGL ES 3.1 Reference Pages*. Khronos Group Inc. URL: https://www.khronos.org/registry/OpenGL-Refpages/es3.1/ (visited on 03/01/2021).

[18] *OpenGL 4.5 Reference Pages*. Khronos Group Inc. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/ (visited on 03/01/2021).

[19] Narayan, S. Improving Network Performance An Evaluation of TCP/UDP on Networks. PhD thesis. 2014.

[20] Rosen, R. *Linux Kernel Networking: Implementation and Theory*. 1st ed. Apress, 2014. ISBN: 978-1-4302-6196-4. DOI: 10.1007/978-1-4302-6197-1.

[21] *socket(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/socket.2.html (visited on 03/01/2021).

[22] Wu, W., Crawford, M. and Bowden, M. The performance analysis of linux networking – Packet receiving. *Computer Communications* 30.5 (2007). Advances in Computer Communications Networks, 1044–1057. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2006.11.001.

[23] Rivera, D., Blasco, S., Bustos-Jimenez, J. and Simmonds, J. Spin lock killed the performance star. *34th International Conference of the Chilean Computer Science Society (SCCC)*. 2015, 1–6. DOI: 10.1109/SCCC.2015.7416588.

[24] Postel, J. *RFC 768: User Datagram Protocol*. 1980. DOI: 10.17487/RFC0768.

[25] Deering, S. and Hinden, R. *RFC 2460: Internet Protocol, Version 6 (IPv6) Specification*. 1998. DOI: 10.17487/rfc2460.

[26] Larzon, L.-A., Degermark, M. and Pink, S. *RFC 3828: The Lightweight User Datagram Protocol (UDP-Lite)*. Ed. by L.-E. Jonsson and G. Fairhurst. 2004. DOI: 10.17487/rfc3828.

[27] Postel, J. *RFC 793: Transmission Control Protocol*. 1981. DOI: 10.17487/RFC0793.

[28] Wetherall, D., Spring, N. and Wetherall, D. *RFC 3540: Robust Explicit Congestion Notification (ECN) Signaling with Nonces*. 2003. DOI: 10.17487/RFC3540.

[29] Jacobson, V. and Braden, R. T. *RFC 1072: TCP Extensions for Long-Delay Paths*. 1988. DOI: 10.17487/RFC1072.

[30] Nagle, J. *RFC 896: Congestion Control in IP/TCP Internetworks*. 1984. DOI: 10.17487/RFC0896.

[31] R. Braden, ed. *RFC 1122: Requirements for Internet Hosts - Communication Layers*. 1989. DOI: 10.17487/rfc1122.

[32] Minshall, G., Saito, Y., Mogul, J. C. and Verghese, B. Application Performance Pitfalls and TCP's Nagle Algorithm. *SIGMETRICS Perform. Eval. Rev.* 27.4 (Mar. 2000), 36–44. ISSN: 0163-5999. DOI: 10.1145/346000.346012.

[33] *tcp(7) - Linux manual page*. URL: https://man7.org/linux/man-pages/man7/tcp.7.html (visited on 03/01/2021).

[34] Altman, E., Barman, D., Tuffin, B. and Vojnovic, M. Parallel TCP Sockets: Simple Model, Throughput and Validation. *Proceedings IEEE INFOCOM. 25TH IEEE International Conference on Computer Communications*. 2006, 1–12. DOI: 10.1109/INFOCOM.2006.104.

[35] Mishra, A., Sun, X., Jain, A., Pande, S., Joshi, R. and Leong, B. The Great Internet TCP Congestion Control Census. *Proc. ACM Meas. Anal. Comput. Syst.* 3.3 (Dec. 2019). DOI: 10.1145/3366693.

[36] Jiang, H., Wang, Y., Lee, K. and Rhee, I. Tackling Bufferbloat in 3G/4G Networks. *Proceedings of the Internet Measurement Conference*. Boston, Massachusetts, USA: Association for Computing Machinery, 2012, 329–342. DOI: 10.1145/2398776.2398810.

[37] Barré, S., Paasch, C. and Bonaventure, O. MultiPath TCP: From Theory to Practice. *NETWORKING 2011*. Ed. by J. Domingo-Pascual, P. Manzoni, S. Palazzo, A. Pont and C. Scoglio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, 444–457. ISBN: 978-3-642-20757-0.

[38] Gu, Y. and Grossman, R. Optimizing UDP-based Protocol Implementations. *Proceedings of the Third International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*. 2005.

[39] Gast, M. S. *802.11 Wireless Networks: The Definitive Guide, Second Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596100523.

[40] Cummings, J. and Tamir, E. *Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll*. Tech. rep. Intel, 2013.

[41] Xie, J., Li, X., Meng, Q., Fan, X., Bo, N. and Ren, F. Comparing Busy Poll Socket and NAPI. *IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. 2019, 318–325. DOI: 10.1109/ICPADS47876.2019.00052.

[42] P. Brown, ed. *GL Texture Buffer Object*. Khronos Group Inc. June 4, 2015. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_buffer_object.txt.

[43] Everitt, C. and McDonald, J. Beyond Porting - How Modern OpenGL can Radically Reduce Driver Overhead. Steam Dev Days. Feb. 11, 2014.

[44] Bolz, J., Koch, D., Leech, J. and Kilgard, M. *GL Buffer Storage*. Ed. by G. Sellers. Khronos Group Inc. Apr. 20, 2015. URL: https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_buffer_storage.txt.