FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Parallel GPU accelerated video transcoding service
Student:	Bc. Daniel Sedlák
Supervisor:	Ing. Tomáš Kvasnička
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

Nowadays, live streaming is gaining popularity and thus becoming a standard service that is integrated within social networks like Facebook, TikTok, and Twitch. However, current implementations of video transcoding services like Wowza, Flussonic, Elemental, or cloud solutions provided by eg. Amazon have either suboptimal end-to-end latency, are not optimized for scale, or are quite expensive.

Moreover, most of these solutions are built on top of FFmpeg libraries, which are not easy to work with. Furthermore, they are written in C, do not provide any memory-safe guarantees, and often introduce new bugs due to new functionality added.

This thesis aims to implement a parallel GPU accelerated video transcoding service. Moreover, the end-to-end latency should be minimal to achieve the best interaction between the audience and the streamer. Use Nvidia libraries to interact with the hardware accelerator (Nvenc, Nvdec, Cuda). To achieve memory safety, write the application in Rust, implement safe memory wrappers and use FFI to interact with the low-level Nvidia libraries. The application should balance among all available GPUs to accomplish an equal workload across all cards. Currently, we expect to work with Nvidia A16 or RTX A4000s but this is subject to change based on availability, testing, etc. All required hardware (servers, GPUs, networking, colocation, ...) will be provided by the supervisor.

In the thesis, apply standard SE methods and consider the following requirements:

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 11 October 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

- Implement proof of concept of Nvidia accelerated video transcoder using Nvenc, Nvdec, and Cuda.
- Write it in Rust to achieve better memory safety. And use FFI to interact with the low-level code.
- Balance incoming streams across multiple GPUs within a single node.
- Support video codec H264.
- Use RTMP on the input.
- The output format is to be considered by the student.
- Set up proper CI/CD environment and implement tests.
- Create user/programmer documentation.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 11 October 2022 in Prague.

Master's thesis

PARALLEL GPU ACCELERATED VIDEO TRANSCODING SERVICE

Bc. Daniel Sedlák

Faculty of Information Technology
Katedra počítačových systémů
Supervisor: Ing. Tomáš Kvasnička
April 19, 2023

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Sedlák Daniel. *Parallel GPU accelerated video transcoding service.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of code listings

I would like to express my deepest gratitude to my supervisor, Tomas Kvasnicka, for his persistent support, guidance, and encouragement throughout the entire process of writing this thesis. His insightful feedback and constructive criticism have been invaluable in shaping this work.

Furthermore, I would like to thank my family and friends for their love, encouragement, and understanding during this challenging journey. Their tireless support and encouragement have been a constant source of motivation for me.

Last but not least, I would like to express my gratitude to all coworkers at the CDN77 for the astonishing feedback during writing this code and code reviews. Without their cooperation and willingness to share their experiences, this research would not have been possible. Namely, in no particular order: Jiri Vebr, Vojtech Stafa.

Thank you all for your support and encouragement.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived its right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60(1) of the Act. This fact shall not affect the provisions of Article 47b of the Act No. 111/1998 Coll., the Higher Education Act, as amended.

In Prague on April 19, 2023 .....

Abstract

Nowadays, live streaming is gaining popularity and thus becoming a standard service that is integrated within social networks like Facebook, TikTok, and Twitch. However, current implementations of video transcoding services like Wowza, Flussonic, Elemental, or cloud solutions provided by eg. Amazon have either suboptimal end-to-end latency, are not optimized for scale, or are quite expensive.

Moreover, most of these solutions are built on top of FFmpeg libraries, which are not easy to work with. Furthermore, they are written in C, do not provide any memory-safe guarantees, and often introduce new bugs due to new functionality added.

This thesis aims to implement a parallel GPU accelerated video transcoding service, addressing some of these issues.

Keywords video, transcoding, GPU, NVIDIA, Rust

Abstrakt

V současné době získává živé vysílání na popularitě, a stává se tak standardní službou, která je integrována do sociálních sítí, jako jsou Facebook, TikTok a Twitch. Nicméně, současné implementace služeb pro transcoding videa, jako jsou Wowza, Flussonic, Elemental, nebo cloudová řešení poskytovaná např. společností Amazon mají buď suboptimální latenci mezi koncovými stanicemi, nejsou optimalizovány pro škálování nebo jsou poměrně drahé.

Většina těchto řešení je navíc postavena na knihovnách FFmpeg, které není snadné používat. Kromě toho jsou napsána v jazyce C, neposkytují žádné paměťově bezpečné funkce, a proto je nelze použít a často přinášejí nové chyby v důsledku přidání nových funkcí.

Cílem této práce je implementace paralelní služby překódování videa akcelerované pomocí GPU, která řeší některé z těchto problémů.

Klíčová slova video, transcoding, GPU, NVIDIA, Rust

List of abbreviations

GPU	Graphics processing unit
FFI	Foreign function interface
ABI	Application binary interface
FLV	Flash video
DASH	Dynamic Adaptive Streaming over HTTP
HLS	HTTP Live Streaming
CUDA	Compute Unified Device Architecture
API	Application programmable interface

Introduction

Firstly I would like to thank you for choosing my master's thesis as a source of information. I will do my best to explain all the details about video engineering best practices, slang, and the current state-of-the-art implementations used worldwide.

In recent years, live streaming is gaining more popularity. It is integrated into many social platforms like YouTube, Twitch, and Facebook. Thus, live streaming as a service is becoming a part of every public social network and seeks presence from the target audience.

This thesis aims to settle down an overview of the video engineering processes and seeks to set a knowledge baseline and terminology, possible state-of-the-art solutions used, and some protocols and hierarchical outlines. Moreover, this thesis discusses potential bottlenecks in this live-streaming process and possible improvements.

Most of the practices used nowadays are not very well documented, or they do not have documentation at all, and it is hard to get into video engineering for newcomers. Therefore, there will be a dedicated chapter that should go through all the necessary details.

Furthermore, we aim to implement the proof of concept of the parallel GPU-accelerated video transcoding service. Moreover, the goal is to implement it in Rust. Concepts used in implementing should serve as a solid foundation for implementing more robust systems.

Chapter 1
About this thesis

In this chapter, we will discuss our goal for this thesis, and we will talk about the structure of this thesis. Moreover, this chapter is an extension of the introduction chapter to ensure that we understand the situation to lay down a solid knowledge foundation.

1.1 Expected results

Roughly speaking, the goal of this thesis is to implement a proof-of-concept video transcoder service that is optimized for low-latency streaming. Moreover, the transcoder should be memory safe, where memory safety will be achieved by implementing it in Rust. That is a lot of words that need to be clarified. We will dig deeper into each step in the following chapters.

We will not be implementing a transcoder from scratch. Instead, we will implement safe FFI bindings that interact with NVIDIA libraries. These NVIDIA libraries will communicate with a GPU accelerator to handle all videos for transcoding.

The implementation part is a proof of concept; thus, we may take some shortcuts to achieve the desired goal. By proof of concept, we do not mean that it will not work, but it means we will implement only limited input and output formats as well as encoding codecs compared to the paid production-ready software. Also, the error handling will not be optimal. Ultimately, the steps we will discuss should serve as a firm base for someone who wants to implement something similar.

Moreover, the theoretical part of this thesis should lay stable bases for introduction to video engineering. There are few up-to-date materials that serve as beginner-friendly to the video engineering world. Therefore we will define the whole transcoding pipeline as well as we will describe each individual step.

1.2 Thesis structure

In the chapter 2, we will go through definitions of video engineering and all other concepts we will discuss later. These concepts consist of basic transport definitions like what is a format, muxing, demuxing, what we mean by transportation, and certain guarantees from the transport. Next, we will take a look at the transcoding process itself, like decoding, scaling, encoding, etc. This chapter should be enough to begin with the implementation part, where we will use all the knowledge we gain here. Furthermore, as we mentioned earlier, this chapter should serve as a necessary foundation for video engineering introduction for newcomers. This chapter should

serve as learning material to clarify all video engineering concepts that are partially defined or not at all in the real world.

We will begin with the implementation part in the chapter 3. We will discuss why the Rust programing language is the right choice. We will talk about its performance, reliability, and productivity. Also, we will show how we can benefit from using the Rust programming language. Deeper into the chapter, we will talk about NVIDIA SDK, how it is structured, what APIs we can use, and how context management works. Furthermore, we must discuss why NVIDIA libraries and GPUs are good for video transcoding. We will take a look at how we can take advantage of the GPU that can help us with decoding video, scaling frames, and encoding it back again.

In the chapter 4, chapter 5, chapter 6, and chapter 7, we will draft and discuss Rust-related wrappers around NVIDIA libraries. We will implement a wrapper around CUDA and context management functions. Next, we will take a look at NVIDIA NVDEC wrappers that will handle video decoding, where it will output decoded frames into the GPU memory. For the video encoding, we will wrap the NVIDIA NVENC, which will take frames from the GPUs memory and output the encoded byte stream. And for the frame resizing, we will implement wrappers around NVIDIA NPP.

In the chapter 8, we will use our developed libraries from the previous chapter and glue them together to implement a transcoding service that will use multiple GPUs for video transcoding and load-balance among them, respectively. The primary goal of the transcoding service is to accept incoming streams, schedule them among available GPUs as fast as possible, and implement video playback. We will set up each card to transcode the streams quickly to achieve the lowest latency.

In the last chapter 9, we will do a summary where we will sum up all results and achievements and possible plans for the future and improvements.

Chapter 2

Video engineering introduction

In this chapter, we will grasp video engineering terminology and its definitions. These definitions are significantly shattered. Video engineering involves designing, developing, and implementing video systems and technologies. It surrounds a wide range of activities, including creating and manipulating video content, developing video codecs and formats, and designing video distribution systems. There are many specializations in video engineering. We can observe some of them in the following list, which is not exhaustive[1].

1. **Video compression** – Video compression techniques or algorithms are used to reduce the size of video files by removing redundant data and reducing the amount of data that needs to be transmitted or stored. Common video compression algorithms include H264[2], H265[3], AV1, VP9, and many more.
2. **Video transportation** – Video transportation involves delivering video content over the network. Technologies such as HTTP Live Streaming (HLS)[4] and Dynamic Adaptive Streaming over HTTP (DASH)[5] are commonly used for video streaming or even the FLV, which behaves like an endless file. However, there are also more old-school transport protocols that are noncacheable, like RTMP.
3. **Video quality and perception** – The quality of a video is determined by a variety of factors, including resolution, frame rate, and bitrate. Video engineering also involves the study of how these factors affect the perceived quality of a video by viewers.

We will use the following Definition 2.1 for the Streaming. Moreover, we will be using streaming and live-streaming interchangeably. Some pieces of literature distinguish between these two things. For example, media such as video-on-demand or YouTube videos are technically streamed but not live-streamed.

▶ **Definition 2.1** (Streaming)**.** *Live-streaming or just streaming is streaming media simultaneously recorded and broadcast in real-time over the network.*

▶ **Definition 2.2** (Stream)**.** *The stream refers to a continuous flow of data. These data do not have to have precise boundaries where they begin or end. The stream represents the video, audio, or subtitles data flow.*

The video processing pipeline can be seen in the Figure 2.1. This will be our alpha omega in the implementation part. This pipeline is similar to the TCP/IP model[6], where each layer has a defined API for the lower or upper layer. Our view perspectives will be really similar. Moreover, it is a goal to map this terminology onto an already widely known model because then it is much easier to understand. The following sections define each layer.

The highest layer is the Transport layer. It handles all IO operations and ensures audio/video data delivery. Most transport protocols transport data in an arbitrary high-level structure, or often the audio or video data are interleaved together in a well-known format. Multiple tracks often represent the stream. The track can be video, audio, or subtitles. In the most typical scenario, we can have a stream where a track represents video in a certain quality. The stream can have multiple languages as audio; each language's audio is a separate track. The demuxer or muxer layer is responsible for extracting particular video or audio packets for the *given track*.

Remember that some stream formats support only a single track for audio and video. Also, some formats support multiple tracks, where each track can be the same video but in a different codec, or similarly for the audio, it can contain multiple audio tracks where each track is the same audio but in a different codec.

Packets are the input for the decoder. Packets may contain multiple encoded frames. Frames are produced by the decoder and are required by the encoder. Frames can then be post-processed; it includes operations like resizing or color tuning.

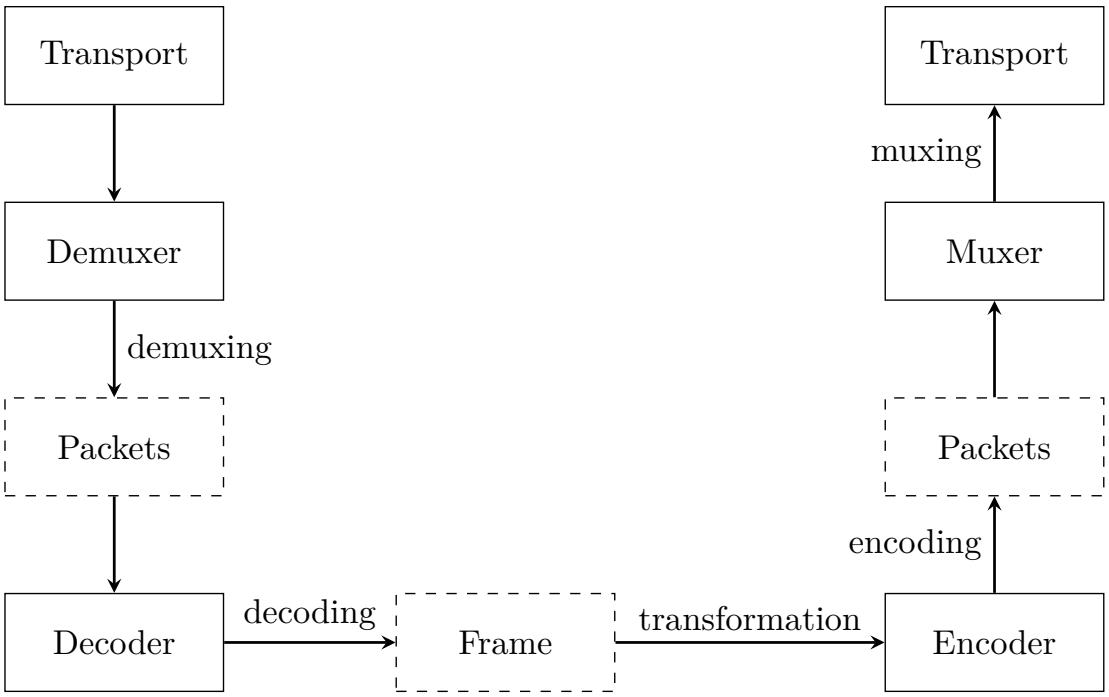

Figure 2.1 Video processing model

2.1 Transport layer

▶ **Definition 2.3** (Ingestion)**.** *Ingestion refers to the process of receiving or consuming data from a source, such as video streaming software.*

▶ **Definition 2.4** (Egestion)**.** *Egestion is the opposite of the Ingestion. The egestion refers to the process of sending data out over the network to the consumer like video player.*

The main goal of the transportation layer is to retrieve or deliver audio or video data. The most used transportation medium is the internet. However, in the imaginary world, we do not have to be limited by the internet. The transportation medium can even be a USB stick or a pigeon[7]. A considerable amount of work has been done in the last decade to make live streaming more reliable, stable, accessible, and efficient. The current trend shows that for the ingestion

part, the RTMP or WebRTC is mainly used, and for the delivery part (or the egestion), the HTTP-based formats[8] are used. HTTP because they are more accessible to the cache.

We will take a look at the most used ingest protocol RTMP in the subsection 2.1.2. Where ingest protocols are used to transport data from the publisher (or the publishing software) to the transcoding server, where it is processed.

The processed audio and video must then be delivered to the player. These delivery methods (or egress) are often encapsulated with HTTP-supported formats like HLS and DASH, which we will discuss in the subsection 2.1.1.1 and the subsection 2.1.1.2. These two delivery methods are famous for the accessibility of easy caching. They have well-defined boundaries of segments and are decodable independently of previous segments. These two formats go in hand with Adaptive bitrate streaming, which the viewer uses to detect appropriate video quality based on the bandwidth. We will discuss this in the subsection 2.1.1.

2.1.1 Adaptive bitrate streaming

Adaptive bitrate streaming is a technique used to provide high-quality video streaming over the internet. It works by adjusting the bitrate of the video stream in real-time based on the available network bandwidth and the device's processing power. The video player itself monitors these metrics. The Player then decides which video track is appropriate based on these monitored metrics[9].

Adaptive bitrate streaming uses multiple versions of the same video, each at a different bitrate and resolution. Moreover, versions are segmented into smaller clips so viewers do not need to wait for an entire video to load before they can begin watching it[9]. The player dynamically switches between these renditions based on the current network conditions. Some video players can use or rely on different predicates that determine appropriate renditions. This allows the video to be played smoothly even when there are fluctuations in the network bandwidth or the device's processing power. One of the key benefits of adaptive bitrate streaming is that it provides a consistently high-quality viewing experience for the user. It ensures that the video is played at the highest possible quality, given the available resources, and it minimizes buffering and other interruptions[10]. Adaptive bitrate streaming is commonly used in online video platforms such as YouTube, Netflix, Twitch, and Hulu. It is also used in live-streaming events, such as sporting events and concerts, to ensure a smooth and uninterrupted viewing experience for the audience.

The following diagram [11] graphically represents what Adaptive Bitrate Streaming is. Overall there are several benefits to using adaptive bitrate streaming.

1. **Reduced bandwidth costs** – Adaptive bitrate streaming can help reduce bandwidth costs for content providers by only delivering the necessary amount of data to the viewer based on their connection speed[12].

2. **Compatibility with a wide range of devices** – Adaptive bitrate streaming can be used to deliver content to a wide range of devices, including computers, smartphones, tablets, and smart TVs[12].

3. **Improved scalability** – Adaptive bitrate streaming can help content providers scale their operations more easily by allowing them to deliver content to a larger number of viewers without overloading their servers or network infrastructure[12].

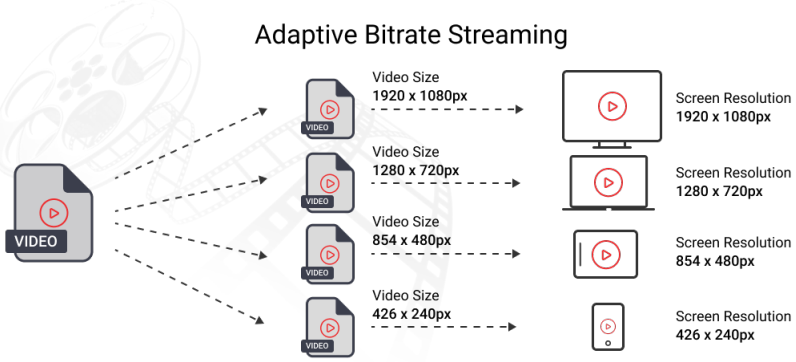

Figure 2.2 Adaptive Bitrate Streaming [11]

2.1.1.1 HLS

HTTP Live Streaming (HLS) is a protocol developed by Apple for adaptive bitrate streaming of audio and video content over the internet. It works by breaking the content into small chunks, called segments, and storing them on a web server. The player retrieves these segments and plays them in a continuous stream. One of the critical features of HLS is that it can adapt to changes in network conditions by switching between different bitrates of the same content. This allows the player to maintain a high-quality viewing experience even when there are fluctuations in the network bandwidth or the device's processing power. HLS is widely supported on various devices and platforms, including iOS, Android, and desktop web browsers, but it is primarily used only in iOS devices[4].

HLS is entirely a text-based format where the information is encoded via `m3u8`[13]. On the top level, the main HLS manifest consists of the main `master.m3u8`, it specifies the locations of various media files that are available for streaming. The file typically contains a list of URLs for the media files, along with metadata about the media, such as the video codec, resolution, bitrate, and many more. Furthermore, the file serves as the "master" playlist for the media being streamed, and it is used to index the individual media files that make up the stream when a viewer accesses the `master.m3u8` file, their device will download and parse the file to determine the locations of the media files and the metadata associated with them. The device will then use this information to request and play the media files in sequence, creating a continuous streaming experience for the viewer. The file itself is encoded in plain text [14, 8].

```
#EXTM3U
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=150000,RESOLUTION=640x360
https://domain.com/360p/index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=240000,RESOLUTION=1280x720
https://domain.com/720p/index.m3u8
#EXT-X-STREAM-INF:PROGRAM-ID=1,BANDWIDTH=440000,RESOLUTION=1920x1080
https://domain.com/1080p/index.m3u8
```

Code listing 1 Example of `master.m3u8`

Each sub playlist of the mentioned `master.m3u8` contains a list of URLs for a set of media segments, along with metadata about the segments, such as segment duration length, timing information, etc. The sub-playlists are typically organized into different tiers or qualities. For example, in the provided listing 1 we can see that each tier corresponds to a different bitrate

and resolution. This allows the viewer's device to select the appropriate sub-playlist based on its available bandwidth and device capabilities. An example of a sub-playlist can be seen in the following listing 2.

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:4
#EXT-X-MEDIA-SEQUENCE:4
#EXTINF:10.00,
fileSequence4.ts
#EXTINF:10.00,
fileSequence5.ts
#EXTINF:10.0,
fileSequence6.ts,
#EXTINF:10.0,
fileSequence7.ts,
#EXTINF:10.0,
fileSequence8.ts,
#EXTINF:10.0,
fileSequence9.ts
```

Code listing 2 Example of `index.m3u8`[15]

2.1.1.2 DASH

Dynamic Adaptive Streaming over HTTP (DASH) is a protocol for adaptive bitrate audio and video content streaming over the internet. It is similar to the HLS protocol. It is a text-based format where the pieces of information are stored as XML. Apart from the HLS, the DASH playlist an only a single file. It does not require fetching individual sub-playlists. Other than that, it has the same properties as HLS. DASH is an open, standardized protocol that is supported by a wide range of devices and platforms, Android and desktop web browsers. It is not well supported on iOS devices[5].

2.1.2 RTMP

RTMP is the live-streaming transport protocol that Adobe developed, where it was initially used to stream audio and video content from Flash Player to media servers for playback in web browsers.

It transports video, audio, and metadata over the internet. The protocol itself has a turbulent history, the specification has not been not fully released, so we have only partial specification[16]. The protocol is no longer maintained and lacks support for newer media formats and codecs[17]. The specification was reverse-engineered before its documentation was partially released. Therefore, there exist implementations of the RTMP that do not have to understand each other. So it, unfortunately, leads to multiple, incompatible implementations.

The protocol itself is based directly on the TCP, where the transported data have binary representation. Consequently, when the client initiates the connection to the server, it establishes bidirectional connections between the client and the server. After the client connects, the server initiates the handshake process. Once the handshake is finished, the client and the server initialize the session. This includes the client sending pieces of information about the session, like if the client wants to receive data or publish data. RTMP internally uses FLV for the encapsulation of the audio of video data[16].

RTMP has been widely used for live streaming of events as of today, but it is being replaced by newer, more modern streaming protocols such as HLS and DASH. RTMP is used the most for the ingestion of the source data. It is not used for the delivery part because it does not scale well. Transported data cannot be effectively cached, and it does not support adaptive bitrate streaming[17].

2.2 Muxer and demuxer layer

The muxer and the demuxer are often tightly coupled with the transportation layer.

The transportation layer determines the input for the demuxer. Demuxing is the process of reading a stream and saving each part of audio, video, and metadata – as a separate stream[18] for a given track. Therefore, the responsibility of the demuxer is to produce packets for each stream. More broadly speaking, the demuxer ensures that the incoming stream is correctly deinterleaved into separate tracks. The track can be video, audio, or subtitles data. Each deinterleaved packet corresponds to the given track.

The muxer accepts packets for each track that was predefined. Videos can contain multiple video or audio tracks. Moreover, the videos can contain multiple subtitles tracks. Each packet that we put into the muxer corresponds to each track. Muxer ensures that these packets are correctly interleaved and serialized for the transportation part[19].

▶ **Definition 2.5** (Demuxing)**.** *Demuxing is the process of separating a multiplexed stream of data into its individual components. Moreover, it refers to the process of separating the audio, video, and other data streams that are multiplexed together in a single file or stream[18].*

▶ **Definition 2.6** (Muxing)**.** *Multiplexing is a technique used to combine multiple data streams into a single stream for transmission or storage[19].*

▶ **Definition 2.7** (Remuxing)**.** *Remuxing is a combination of demuxing 2.5 and muxing 2.6. It separates the input multiplexed stream from one format/container, and then it combines these separated streams into another format/container.*

2.2.1 FLV

Flash Video (FLV) is an audio/video format developed by Adobe. It was tightly incorporated within the Adobe Flash Player[16].

FLV consists of tags. Tag is the elementary unit of the FLV format. The concatenation of many tags represents the whole FLV file. There is the Video tag, Audio tag, and Data tag. The first two mentioned tag types are used to store video and audio data. The Data tag is used to store helper metadata pieces of information. The most commonly used Metadata tag is the `OnMetaData`. It contains meta information like video width and height, audio sampling, encoder, etc[20].

The FLV officially supports only a single video track and can have multiple audio tracks. However, the binary representation of the format allows numerous tracks of the same type. FLV also supports a limited amount of codecs that are predefined. The most commonly used are H264 for the video and AAC for the audio[16]. However, some companies/individuals also requested unofficial support for the H265 video codec[21], nevertheless this feature was rejected.

2.2.2 MP4

MP4 is one of the most popular formats. YouTube uses it, and its versatility makes it the most common video format[22]. MP4 has universal support for many devices like mobile phones, web browsers, consoles, etc.

The elementary unit of each MP4 is called Atom. The whole MP4 format consists of multiple Atoms that are sequentially stored. Each Atom has a tree data format, where each child of the parent is another Atom. There are countless options for generating these Atoms for a given video because it depends on whether we want to focus on low latency, immense video fragmentation, backups, etc[23].

2.3 Decoder and encoder layer

The decoder and encoder layer aims to accept packets on the input and generate new packets at the output. This transformation into other packets can include a change of the codec itself, frame resizing for the video part, or other byte stream tuning like adding additional metadata to the video frames.

To be precise, the decoder accepts packets. When the decoder receives enough packets for the decoding, it starts the decoding part. The number of incoming packets to start the decoding is specific to a given video codec. A single packet can contain multiple video frames. On the decoded frames, we can do various types of transformations. It includes transformations like resizing (or sometimes called scaling), color tuning, etc. All of these transformations are optional and can be arbitrarily chained. We generally want to scale down frames in the resizing transformation because it is a lossless process.

The encoder accepts raw frames on the input. The mentioned decoder typically produces these raw frames. The objective of the encoder is to do the encoding on incoming frames and generate new packets that the muxer can process.

▶ **Definition 2.8** (Encoding)**.** *Video encoding is the process of converting raw video data into a compressed digital format that can be stored, transmitted, and played back on a device or system. Video encoding involves applying a specific algorithm or set of algorithms to the raw video data in order to reduce the file size and make it more suitable for transmission or storage.*

▶ **Definition 2.9** (Decoding)**.** *Video decoding is the process of converting a compressed digital video format into raw video data that can be displayed or processed by a device or system. Video decoding involves applying a specific algorithm or set of algorithms to the compressed video data in order to reconstruct the original raw video data.*

▶ **Definition 2.10** (Transcoding)**.** *Transcoding is a combination of the Definition 2.9 and the Definition 2.8 process. Optionally process can also include the resizing part.*

2.3.1 Codec

A codec is a software algorithm that compresses and decompresses digital data. It is primarily used to reduce the size of the data so that it can be stored and transmitted more efficiently over the network. There are many different video and audio codecs (like H264, H265, AAC, and AV1), each with its unique features and compression methods. For example, some are more encoding-heavy than decoding-heavy, and vice versa[24, 25].

2.3.1.1 Bytestream

Some codecs, like H264 and H265, need to be wrapped into a format more suitable for transportation. For example, there is a *Annex B*[2] or AVCC[23]. Therese transportation formats differ for various video codecs. However, we will focus only on these two mentioned.

The fundamental building block is the NALU which stands for Network Abstraction Layer Unit, where each NALU contains information about the type of data it carries and a payload, which is the actual compressed video data[2].

The *Annex B* specifies that each NALu[2], is prefixed with specified start code `0x00_00_00_01` or `0x00_00_01`, there is also a safety mechanism called emulation byte prevention. Emulation byte prevention is the mechanism to prevent misinterpretation of start codes in the compressed video byte stream. It is achieved by adding an extra `0x03` byte when encountering the `0x00_00` pattern to the byte stream[2]. This is true for both the H264 and H265[3] codecs.

The `AVCC` specifies that each NALu is prefixed with its size rather than the specified start code. However, emulation byte prevention is still used[2]. This byte stream format is used for the MP4 and FLV format[23].

2.3.2 Frame and pixel format

Frame refers to a single still image from the video sequence. Keep in mind that a series of frames that are rapidly displayed form the illusion of seamless motion. The number of frames displayed per second is referred to as the frame rate and is usually expressed as frames per second (FPS)[25].

The frame's byte representation can often be viewed as a 2D array, where the width of the frame is the number of columns, and the height is the number of rows for the array, where each element in this 2D array is a pixel. The pixel of a frame is often represented by more than a single byte. The pixel format specifies the arrangement of bits to represent the color information for each pixel. Each pixel is specified by its color depth, e.g., 8, 10, or 12 bits[26, 27]. The most common pixel formats are listed below.

- **RGB** – Red, Green, and Blue. This pixel format is widely used in computer graphics and video displays because of its simplicity. It is an additive color model in which the color components are added together in various ways to produce a broad array of colors[28].
- **YUV** – Luma and chroma, where the brightness information (luma) is represented by one channel, and two additional channels represent the color information (chroma).
- **CMYK** – Cyan, Magenta, Yellow, and Key (black), which is used in color printing.

We also distinguish between *Packed*, *Planar*, and *Semipacked* variants. These three mentioned formats describe how the pixel is stored in the mentioned imaginable 2D array.

- **Packed** – Pixels are stored and clustered consecutively. For example, for the RGB, it means that pixels are stored `R G B, R G B, ...`[29].
- **Planar** – It is the opposite of the *Packed* variant, where each pixel is made of individual bits stored in their own plane. For example, the RGB would be represented as `R R R ..., G G G ..., B B B ...`[29].
- **Semipacked** – Semipacked is a combination of the `Packed` and the *Planar* variant. There are multiple combinations; however, it is most commonly used in the YUV pixel format. For example, the *NV12* pixel format is *Semipacked* and is represented as `Y Y Y ..., U V U V U V ...`[29].

2.3.3 Scaling or resizing

Scaling or resizing is a process of changing the size of the frame, either by increasing or decreasing the frame's dimensions by adding pixels or removing pixels, or stretching the existing pixels[30].

Chapter 3

Implementing safe wrappers

Starting from this chapter, we have sufficient knowledge to start implementing the practical part. Our goal is to interact with the NVIDIA libraries to accelerate the video encoding process, decoding process, and scaling of the frames. Moreover, we will be using Rust programming language to implement safe wrappers. The safe wrapper is a layer of code that encapsulates and abstracts away potentially unsafe or error-prone functionality, providing a more straightforward and safer interface that we can interact with.

To recapitulate the assignment. We will be using Rust programming language [31] to achieve memory safety. The more detailed reason why Rust is in the section 3.1. Our main goal is to implement a parallel GPU-accelerated video transcoding service. Building this application will be in chapter 8; however, before we start building it, we first need to implement the three main components that form the video transcoder. And that is the decoder (chapter 5), encoder (chapter 6), and frame resizing (chapter 7). The NVIDIA libraries will do all the heavy lifting of the transcoding. Therefore, we need to gain a better understanding of these libraries, what they do, and how they operate. We will discuss this in the section 3.2. Also, we aim for the best end-to-end latency. To achieve this, we need to utilize a whole pipeline of the NVIDIA GPU. This means that we will store raw decoded frames in the GPU memory. This will help us with the latency because GPU's ram is much "closer" than the host memory. Also, transporting raw decoded frames through the PCI bus from the GPU to the host memory is not wise because it could lead to congestion.

However, using NVIDIA libraries casts another burden upon us. Since we will be using a higher-level language than C, so we need to create some interoperability between our "happy" Rust world, where we have a more robust API, and the depths of the C world, where most of the functions can shoot us in the leg or provide us with undefined behavior. Luckily NVIDIA has C-compatible ABI. Therefore, we can use FFI calls to interact with these libraries. Rust offers us to utilize FFI, but there is a catch, the compiler is not able to guarantee anything, but more on that in the section 3.3.

3.1 Why Rust is the right choice

Let's elaborate a little bit on why Rust should be the right programming language and how it will help us. The `rust-lang.org` points out the following things Rust can offer.

- **Performance** – Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages [31].

- **Reliability** – Rust's rich type system and ownership model guarantee memory-safety and thread-safety enabling you to eliminate many classes of bugs at compile-time [31].

- **Productivity** – Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling, an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more [31].

The strongest reason why to use Rust is its memory management approach. This memory management attitude is called **Ownership**, where Ownership is a set of rules that govern how a Rust program manages memory. Most well-known programming languages like Go, C#, and Java have garbage collection that regularly looks for no longer-used memory as the program runs. On the other side of the memory management spectrum, where the programmer has to manage how they use the memory by hand, we have programming languages like C, C++, Assembly, etc. Rust uses a third approach. Memory is managed through a system of ownership with a set of rules that the compiler checks. If any of the rules are violated, the program won't compile. None of the features of ownership will slow down the program while it's running [32].

These Rust ownership rules are [32]:

1. Each value in Rust has an owner.

2. There can only be one owner at a time.

3. When the owner goes out of scope, the value will be dropped.

```
// This example will not compile

// `s1` now owns the string.
let s1 = String::from("hello");
// Ownership is transferred to the `s2`.
let s2 = s1;
// `s1` no longer owns the string, thus we are not able to access it.
println!("{}, world!", s1);
```

Code listing 3 Example of violation of the Rust ownership [32]

To enrich the Rust compiler's power, the borrow checker operates by keeping track of the lifetime of data and the references to that data. When a reference is created, the borrow checker ensures that the data it references is still valid and that no other references to the same data exist. When a reference goes out of scope, the borrow checker ensures that the data it references is not accessed again [33].

The borrow checker also enforces rules for mutable references, which allow data to be modified. Only one mutable reference to a piece of data is allowed at a time to prevent data races and other concurrency-related bugs [34].

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
    // Here, s goes out of scope. But because it does not have
    // ownership of what it refers to; it is not dropped.
}
```

Code listing 4 Example of Rust borrow checker [33]

3.2 NVIDIA SDK

The NVIDIA Software Development Kit (SDK) is a collection of software development tools and libraries provided by NVIDIA for use with their GPUs. The SDK consists of libraries and tools for a wide range of tasks [35]. Namely, it includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and runtime libraries. We will be using only the GPU accelerated libraries [36].

The *kernel* is the code that can be executed on the NVIDIA GPU. It is a function executed in parallel by multiple threads on the GPU and designed to perform a specific computation. The CUDA is a platform and programming model for general-purpose GPU computing. NVIDIA is well known for its CUDA capabilities. Note that GPUs provide much higher instruction throughput and memory bandwidth than the CPU. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. This difference in capabilities between the GPU and the CPU exists because they are designed with different objectives in mind; therefore, using the GPU for programming requires adapting different programming approaches. While the CPU is designed to excel at executing a sequence of operations as fast as possible and has a limited amount of cores, the GPU is designed to excel at running thousands of cores in parallel. However, these GPU cores are much more lightweight than their CPU counterpart [37].

The GPU context (often just called the context) is defacto the allocator bookkeeper. When we allocate memory in the GPU, we need to tighten the allocation with a particular context. When the context goes out of the scope, it automatically deallocates everything tight with it. We will talk about context management in the subsection 3.2.3.

3.2.1 Library overview

Our primary interest in NVIDIA SDK is due to the dedicated chips on the NVIDIA GPU, which are used for video acceleration. NVIDIA GPUs contain one or more hardware-based decoders and encoders (separate from the CUDA cores), providing fully-accelerated video decoding and encoding for several popular codecs. With decoding/encoding offloaded, the graphics engine and the CPU are free for other operations [38].

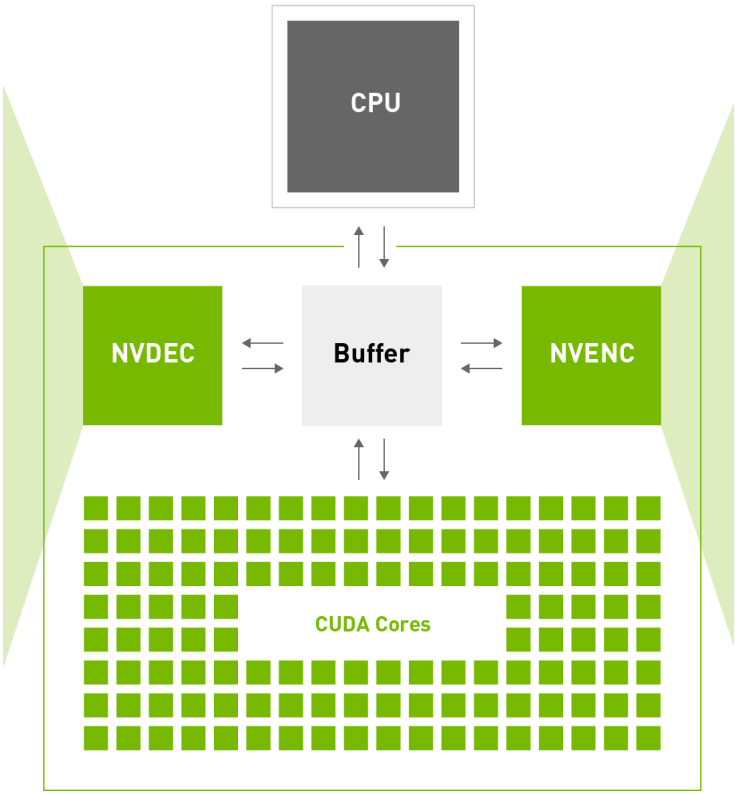

Figure 3.1 NVIDIA codec SDK [38]

We will be using a few libraries from the Nvidia SDK:

1. `libcuda` – This is the main building block for all libraries that NVIDIA offers. This library offers context creation on the GPU. Context is a runtime state that allows CUDA commands to be executed on a specific GPU. Moreover, CUDA context manages the GPU's memory and resources, such as CUDA streams, events, and textures [37]. Therefore, every other library is somewhat dependent on this library. We will dig more into this library in the chapter 4.

2. `libnvcuvid` – Library that provides API for the mentioned NVIDIA decoder. We will refer to this library as NVDEC. NVDEC decodes the compressed video streams and copies the resulting YUV frames to video memory [39]. This library consists of 2 main parts that we will be using, the video decoder and parser, more on that in the chapter 5.

3. `libnvidia-encode` – It provides API for NVIDIA video encoder. It will help us to offload video encoding from the CPU to the GPU. NVENC is a proprietary technology developed by NVIDIA, and is only available on NVIDIA graphics cards [40]. We will dig more into this library in the chapter 6.

4. `linpp*` – Collection of libraries for image manipulation. The NVIDIA Performance Primitives (NPP) library provides GPU-accelerated image, video, and signal processing functions. It can perform tasks such as color conversion, image compression, filtering, thresholding and image manipulation [41]. We will use this for image resizing. We will talk more about these libraries in the chapter 7.

3.2.2 Driver vs. runtime API

NVIDIA GPU has two types of programming API. The low level is called a driver API, and the higher level is called runtime API. The low level API is C ABI compatible, so it is used when we want to operate the GPU from other programming languages than C or C++. On the other hand, the runtime API is a C++ wrapper, so it is hard to use in other programming languages due to ABI incompatibility and name mangling. The driver and runtime APIs are very similar and can be used interchangeably. However, there are some differences worth noting.

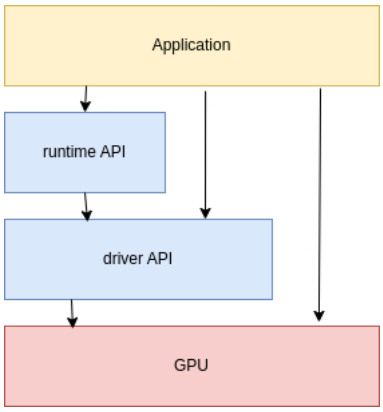

Figure 3.2 Runtime vs. driver API comparison.

Runtime API is built directly on top of the driver API, as can be seen in the Figure 3.2. Thus, it does not bring anything specific that cannot be achieved via the mentioned driver API.

- **Complexity vs. control** – In comparison, the driver API offers more fine-grained low-level control, particularly over contexts and module loading. Kernel launches in the driver API are much more complex to implement [42], as the execution configuration and kernel parameters must be specified with explicit function calls. In the runtime API, all the kernels are automatically loaded during initialization and stay loaded for as long as the program runs [43].
- **Context management** – Runtime API solves all context management shenanigans. Managing context can become very difficult in large-scale applications. We will uncover all the mist in the subsection 3.2.3.

We will not be dealing with the kernel launches on the GPU because all necessary functionality can be found in the libraries. However, when using NVIDIA's compiler `nvcc`, it automatically uses the runtime API. Thus, the kernel launches are much easier to read. Examples like code differences will be omitted for the sake of simplicity. Still, they can be found online in NVIDIA's documentation for the CUDA [44].

3.2.3 Context management

A single context represents a specific instance of the driver that is associated with a particular thread of execution. Each context maintains its own state and can be used to perform operations on the GPU, such as allocating memory, launching kernels, and synchronizing execution. A process can create multiple contexts, allowing different threads to perform operations on the GPU concurrently [37]. However, a single context can be active at the time and can be used only in a single thread; this thread synchronization should be done in the GPU driver itself.

However, it is poorly documented [45, 46], but more on that in the implementation chapter in the chapter 4. Context management has a structure of the stack data structure. Each time that we want to operate the GPU, we need to call `cuCtxPushCurrent()`. Only the context that is on the top of the stack is currently active. To make this context available to other threads, we need to pop it from our stack with `cuCtxPopCurrent()`, then we are able to push on the top of the stack of other threads [47].

Context can be created by the `cuCtxCreate()` function call. After creation, this context is already present at the stack, and we need to pop it right away [47].

Remember that multiple contexts (and their associated resources such as global memory allocations) can be allocated concurrently on a given GPU, but only one of these contexts can execute work at any given moment on that GPU; contexts sharing the same GPU are time-sliced. Thus creating additional contexts incur memory overhead for per-context data and time overhead for context switching. Therefore, it is best to avoid multiple contexts per GPU within the same CUDA application [48].

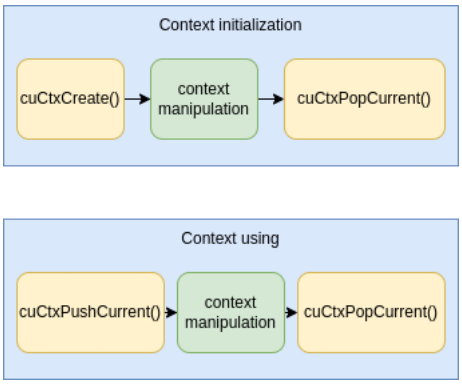

Figure 3.3 Context manipulation

It is said that context management can be done through the driver API but is not exposed in the runtime API. Instead, the runtime API decides which context to use for a thread. Since we are not using the runtime API, therefore we will omit the explanation of it. Keep in mind that contexts may consume significant resources, such as device memory, extra host threads, and performance costs of context switching on the device. It is recommended to create only a single context per application. But not only that, it is recommended to use only a single context per the GPU [43].

3.3 Rust FFI

FFI (Foreign Function Interface) is a mechanism that allows a program written in one programming language to call functions from libraries written in another programming language. Practically, it allows for interoperability between different languages by providing a bridge between them. This is often used when we need to use a library or function that is written in a different language but don't want to re-write the entire program in that language. To make this work, these two worlds need to have well-defined ABI (Application Binary Interface). To put this into perspective, when passing data between these two worlds, like structs, integers, etc., they need to have the same layout, size, and alignment. This can get complex because complicated or non-trivial objects or datatypes may be difficult to map from one environment to another, so we often rely on trivial types like numbers and pointers. Most of these FFI calls rely on C ABI [49].

3.3.1 Unsafe Rust

Rust itself has two modes, the ”safe, happy world” (we will call it the safe Rust) and the ”unsafe” world (which will just call unsafe Rust). In fact, the safe Rust does not allow dereferencing random pointers. In the safe Rust, we can only access references because, with references, the compiler can check the validity of the reference. References are generic over a type and over a lifetime. References have the following syntax `&'a T` (where the lifetime can be omitted in most cases [50]), where the ampersand signal the reference, the `'a` signals the lifetime `a`, and the `T` is a generic type. The lifetime refers to the duration for which a reference to a value is valid.

Rust pointers (`*const T` or `*mut T`) have no lifetime parameter. Thus compiler does not guarantee anything, which is why it is considered unsafe. However, keep in mind that storing pointers is not unsafe; only dereferencing is unsafe, thus when we want to access data on the pointer, we need to use the keyword `unsafe { /* dereferencing */ }`) to tell the compiler that we are aware of all risks [51, 52].

3.3.2 Creating bindings

We must create the bindings to communicate with libraries written in a different programming language. Because the NVIDIA libraries are C ABI compatible, we can call the function from these foreign libraries with the Rust types because Rust enables us to use the C byte-aligned types.

To create bindings, we can do it by hand, as can be seen in the listing 5. But writing it by hand is not error-prone and can lead to many attention errors, like writing a bad type or bad function name.

```rust
use libc::size_t;

#[link(name = "library_name")] // Library name that we want to link.
                               // Can be omitted.
extern {
    // Here, inside the extern, are functions exported from
    // the library.
    fn get_data(input_size: size_t) -> size_t;
}

fn main() {
    let x = unsafe { get_data(100) };
    println!("Returned number from the FFI: {}", x);
}
```

Code listing 5 Bindings written by hand [53]

However, these attention bugs can be easily eliminated. We can take advantage of commonly used tool `bindgen`[54]. This tool can be used as a library or as a CLI utility. It accepts the header file on the input a produces equivalent Rust code. One example of how it works can be seen in the listing 6. Also, these bindings are often generated automatically by the `build.rs`[55].

```c
typedef struct Example {
    int foo;
    char bar;
} Example;

void accept_example(Example* ex);

// Transforms into.
// #[repr(C)]
// pub struct Example {
//     pub foo: ::std::os::raw::c_int,
//     pub bar: ::std::os::raw::c_char,
// }
//
// extern "C" {
//     pub fn accept_example(ex: *mut Example);
// }
```

Code listing 6 Transformed C header file into Rust.

Chapter 4

CUDA wrapper

To use more specific hardware capabilities like NVENC or NVDEC, we first need to create an abstraction over the CUDA context management and memory allocation. We will not implement all the features that CUDA offers, so we will only focus on a minimal subset we need. This subset consists of context creation (and manipulation) and memory allocation. The following chapters section 4.1 and section 4.2, will talk about implementing each of mentioned parts. As we have mentioned earlier in the chapter 3 we will use the Driver API for abstracting calls to the `libcuda.so`. This library is distributed along the NVIDIA drivers. However, we will install the NVIDIA SDK, which is a superset to the NVIDIA drivers. The NVIDIA SDK contains drivers, development tools like CUDA memory check, NVIDIA compiler (NVCC), and development header files that we need for the binding generation. We need the header file to be able to create bindings with the `bindgen`[54]. We need the `cuda.h` for this particular CUDA library wrapper.

4.1 Context creation and destruction

For the sake of simplicity and line reduction, we will omit error checking for the returned errors by the NVIDIA driver. Also, we assume that the binary is correctly linking against the correct library, and bindings were generated for the Rust.

Rust has a very satisfying idiomatic pattern using marker types to restrict implementations that we will use for the context abstraction[56].

4.1.1 Context initialization

Before we start creating the context, we need to call the `cuInit(0)`, zero as an argument is the default value, and the parameter is reserved for future use. Furthermore, the call initializes the CUDA driver API and must be called before any other function from the driver API[57]. This call can be called multiple times without causing unexpected side effects.

To proceed further with the context creation, we must get all available devices. By device, the terminology means the GPU. The number, UUID, or PCI bus address identifies each GPU. We will stick with the number identification. We can retrieve all GPUs by calling the `cuDeviceGetCount(&mut local_variable)`. It will initialize the variable that we provided a reference to it. We can easily wrap this call to the `CudaContext`, where we will determine the device count and then return `Device` abstraction with its identifier. We will use this extracted identifier in the context creation.

```rust
/* imports omitted */
pub struct Device {
    pub id: CUdevice,
}

pub struct Initialized {
    pub(crate) cuda_context: *mut CUctx_st,
}
pub struct NotInitialized;

pub struct CudaContext<Stage = NotInitialized> {
    pub(crate) stage: Stage,
}

impl CudaContext {
    pub fn new() -> CudaContext<NotInitialized> {
        unsafe { cuInit(0)) };

        Ok(CudaContext {
            stage: NotInitialized {},
        })
    }

    /// Lists all devices that the NVIDIA driver recognizes.
    pub fn get_devices(&self) -> Vec<Device> {
        let mut count = -1;
        unsafe { cuDeviceGetCount(&mut count) };

        let mut result = Vec::<Device>::with_capacity(count as usize);

        for i in 0..count {
            result.push(Device {
                id: i,
            });
        }
        result
    }
}

```

Code listing 7 Cuda context initialization

If we are unsure if the index for the given GPU is correct, we can request additional information for the given GPU by calling the `cuDeviceGet(&mut device, index)`[58]. It enables us to extract more information from the GPU. Also, when using the `nvidia-smi`[59], the listed indexes for the GPU can be shown as well. However, remember that the index is not guaranteed to be the same for the GPU after the reboot. If we have only a single GPU, then we do not need to care about reindexing, but most of the time, we will have multiple GPUs in the machine.

4.1.2 Context building & destruction

After CUDA driver API initialization, we can proceed to the context creation itself. It can be done by calling the `cuCtxCreate_v2(&mut local_context_variable, 0, device_id)`. The call will initialize the provided pointer. As a second parameter, we will set it to zero, which means `CU_CTX_SCHED_AUTO`. This argument can tune the scheduler behavior. As a last argument, we specify the device on which we want to have the context allocated[60].

Context management has a structure of the stack data structure. Only the context that is on the top of the stack is currently active. For example, when we want to allocate memory on a specific device, then the device context must be active in the executing thread. Moreover, when the stack is active, then we have access to its address space; otherwise, when we access allocated memory on the noncurrent active context, then we would get an error. When we create the context, it is made active in the current thread automatically. So we need to pop it right away to make it floating. It can be achieved by calling the `cuCtxPopCurrent_v2(std::ptr::null_mut()))` where the argument is a destination where we want to pop it[60].

When we allocate the context, we also must not forget to deallocate the allocated context. It can be simply implemented in the `Drop`[61] trait, which is Rust's destructor. Context is destroyed by calling the `cuCtxPopCurrent_v2(cuCtxDestroy_v2(pointer_to_the_context))`.

```
/* rest of the previous example */

impl CudaContext {
    /// Creates context that is linked with a given GPU.
    pub fn create_context(self, device: CUdevice)
        -> CudaContext<Initialized> {
        let ctx = unsafe {
            let mut ctx: *mut CUctx_st = mem::zeroed();
            cuCtxCreate_v2(&mut ctx, 0, device);
            cuCtxPopCurrent_v2(std::ptr::null_mut());

            ctx
        };

        Ok(CudaContext::<Initialized> {
            stage: Initialized { cuda_context: ctx },
        })
    }

}

// Destructor
impl Drop for Initialized {
    fn drop(&mut self) {
        cuCtxDestroy_v2(self.cuda_context);
    }
}
```

Code listing 8 Cuda context creation & destruction

To make working with context easier, we can implement `run_in_context` function, which will accept a lambda function that will be executed when the context is made active in the current

thread. This helper function will ensure that we always correctly pop the context from the currently active.

```
/* rest of the previous example */

impl CudaContext<Initialized> {
    /// Runs operation in cuda context.
    pub fn run_in_context<F, O>(&self, func: F) -> O
    where
        F: FnOnce() -> O,
    {
        unsafe {
            cuCtxPushCurrent_v2(self.stage.cuda_context);
        };

        let result = func();

        unsafe {
            cuCtxPopCurrent_v2(std::ptr::null_mut());
        };

        result
    }
}
```

Code listing 9 Run in the Cuda context

A process can create multiple contexts, allowing different threads to perform operations on the GPU concurrently. However, it is recommended to use a single context per GPU and thus when we want to use a single context for a single GPU then we need to share that one context across multiple threads. Context can be shared by multiple threads where the GPU driver will solve the synchronization. Because each request for the GPU needs to be serialized, as of the time writing this, the context is thread safe[37]. This information is hard to find because it is not explicitly written because, in the earlier version of the CUDA, it probably was not thread-safe[45, 46].

4.2 GPU memory allocation

NVIDIA driver API offers many memory allocation primitives like 1D, 2D, and 3D arrays. Moreover, there are aligned and unaligned versions of them. For our purpose, the 1D array will be enough. We need to implement allocation, deallocation, and moving between the device memory space and the host memory space. Keep in mind that this implementation completely omits error handling for the sake of simplicity[62].

To simplify the wrapper, we will define the `SlabType`, which is our user-defined type. The `SlabType` represents arbitrary memory allocation. For now, it only represents the `ContinuousMemory`. This abstraction represents the 1D array that we have talked about. However, in the future, it can also represent the 2D and 3D allocations.

This abstraction is needed because most of the other memory structures have different function calls for copying between each other. The `ContinuousMemory` is probably the simplest one to copy and manipulate. In 2D and 3D allocations, we need to consider the alignment and padding, and that would make the wrapper messier.

```rust
/* imports omited */

/// Memory that is allocated on the particular device
#[derive(Debug)]
pub struct DeviceSlab {
    pub(crate) cuda_context: Arc<CudaContext<Initialized>>,
    pub(crate) ptr: CUdeviceptr,
    pub(crate) slab_type: SlabType,
}

#[derive(Debug)]
pub enum SlabType {
    ContinuousMemory(u64),
}

impl CudaContext<Initialized> {
    /// Allocates continuous slab of memory.
    /// It is the same like `malloc`, but rather
    /// in the device memory.
    pub fn mem_alloc(self: &Arc<Self>, size: u64) -> DeviceSlab {
        let ptr = unsafe {
            self.run_on_context(|| {
                let mut ptr: CUdeviceptr = mem::zeroed();
                cuMemAlloc_v2(&mut ptr, size);
                ptr
            })
        };

        Ok(DeviceSlab {
            cuda_context: self.clone(),
            slab_type: SlabType::ContinuousMemory(size),
            ptr,
        })
    }
}

impl Drop for DeviceSlab {
    fn drop(&mut self) {
        self.cuda_context.run_on_context(|| unsafe {
            let result = cuMemFree_v2(self.ptr);
            if result != cudaError_enum::CUDA_SUCCESS {
                tracing::error!(?result, "Unable to free cuda memory.");
            }
        }
    }
}
```

Code listing 10 Cuda memory allocation structures

Copying data between address spaces is simple as well. It can be achieved by the `cuMemcpyHtoD_v2`, `cuMemcpyDtoH_v2` and `cuMemcpyDtoD_v2`. Where the *H* stands for the Host and the *D* stands

for the device[62].

CUDA also offers Unified Memory, where the Unified Memory is a single memory address space accessible from any processor in a system. So the GPU driver allows applications to allocate data that can be read or written from code running on either CPUs or GPUs. When code running on a CPU or GPU accesses data allocated this way, the CUDA system software and the hardware takes care of migrating memory pages to the memory of the accessing processor[63]. But we will not use the managed memory because it would be inefficient for our use case. The Unified Memory introduces unnecessary overhead and higher latency due to synchronization between the host and GPU memory.

```rust
impl CudaContext<Initialized> {
    /// Copies data from the host to the device memory. Device memory must
    /// be same size as host input array to prevent undefined behavior.
    pub fn copy_host_to_device(
        &self,
        host: &[u8],
        device: &mut DeviceSlab,
    ) {
        unsafe {
            self.run_on_context(|| {
                match device.slab_type {
                    SlabType::ContinuousMemory(size) =>
                    cuMemcpyHtoD_v2(device.ptr, host.as_ptr().cast(), size),
                }
            })
        }
    }

    /// Copies data from the device memory into the host memory.
    /// Host memory must be same size as
    /// device memory to prevent undefined behavior.
    pub fn copy_device_to_host(
        &self,
        device: &DeviceSlab,
        host: &mut [u8],
    ) {
        unsafe {
            self.run_on_context(|| {
                match device.slab_type {
                    SlabType::ContinuousMemory(size) =>
                    cuMemcpyDtoH_v2(host.as_mut_ptr().cast(), device.ptr, size),
                }
            })
        }
    }

    /// Copies data from source slab into destination slab.
    pub fn copy_device_to_device(
        &self,
        src: &DeviceSlab,
        dst: &mut DeviceSlab,
    ) {
        unsafe {
            self.run_on_context(|| {
                match src.slab_type {
                    SlabType::ContinuousMemory(size) => {
                        cuMemcpyDtoD_v2(
                            dst.get_inner_ptr(),
                            src.get_inner_ptr(),
                            size
                        )
                    },
}})}}}
```

Code listing 11 Cuda memory copying

Chapter 5

NVIDIA decoder wrapper

The decoder is responsible for decoding the encoded frames. These encoded frames are mostly encoded with the H264, H265, or AV1 codec. Therefore decoder will extract the encoded frames in the incoming byte stream.

The NVIDIA GPU has, in most of the newer GPUs, a dedicated hardware processor for accelerating video decoding. It is mainly referred to as NVDEC. To access the NVDEC, we will need to use a library named `libnvcuvid.so` which exposes API to the decoder and the needed parser. Moreover, we will need the CUDA abstraction that we have implemented in the chapter 4 to be able to store the decoded frames in NVIDIA's GPU. We will limit our scope primarily to the H264[2] video codec. We assume that bindings to the library were correctly generated by the `bindgen`[54].

5.1 Video parser

Encoded packets are often stored in something that is called the byte stream. For the H264[2] and H265[3], this byte stream is called *AnnexB* (there is also *AVCC*[23] more on that in the chapter 2). This byte stream contains encoded frames with their corresponding metadata. Some metadata are necessary for decoding, and some only store helper pieces of information from the encoder. We will use the video parser provided in the `libnvcuvid.so`, which will provide correctly initialized structs for the decoder, which we will receive via callbacks.

The callbacks run on the calling thread. They do not create additional threads. However, they are called from the C world, therefore, they need to be marked as `extern "C"`; otherwise, they would be mangled. Data are injected into the parser via the `cuvidParseVideoData` call, which accepts two arguments, one is an instantiated parser, and the other is populated struct `CUVIDSOURCEDATAPACKET`. This structure contains points to the byte array, its size, and optional flags, like the end of the data stream, etc. For H264 and H265 codecs, it secretly expects video data in the *AnnexB* format, which we have mentioned earlier. Once the library encounters the picture parameter set and sequence parameter set, it triggers the `pfnSequenceCallback` callback. This callback has enough information for the decoder initialization. We are not able to instantiate the decoder itself without processing the input data first. It will tell us the width, and height of the input, how many decoding planes we need to allocate, etc. Once the parser has enough data, then it triggers the `pfnDecodePicture` callback, where we pass the callback data into the encoder.

Based on our pieces of information, we can effortlessly hide the parser logic behind the safe interface. We need to have the sequence, decode, and display callback. Moreover, we want to

be able to take advantage of the question mark operator to propagate errors downstream. Also, we need another helper method to report errors. Keep in mind that the interface will be called from the C world. Therefore, we need a way to send these errors to the Rust world before we eventually return execution to the C world.

```rust
pub trait ParserCallbacks<Error>
where
    Error: std::error::Error,
{
    /// This callback is called only once at the beginning once the picture
    /// parameter set and sequence parameter set is encountered. In this
    /// function, you must initialize the frame decoder. You cannot do it
    /// earlier because you need more information to initialize the
    /// decoder.
    ///
    /// It can be called later again when a format change is detected.
    fn sequence_callback(&mut self, format: CUVIDEOFORMAT)
        -> Result<SequenceOutput, Error>;

    /// It is triggered when we have enough data to call the decoder.
    /// Thus we need to submit the frame to the decoder.
    fn decode_callback(&mut self, params: CUVIDPICPARAMS)
        -> Result<(), Error>;

    /// Once we are ready to retrieve decoded frames from the decoder,
    /// this callback will be called.
    fn display_callback(&mut self, picture: CUVIDPARSERDISPINFO)
        -> Result<(), Error>;

    fn report_error(&mut self, error: Error);
}
pub enum SequenceOutput {
    /// Correct value is located in the [`CUVIDEOFORMAT`]
    /// in `min_num_decode_surfaces`.
    DecoderSurfaceCount(u8),
}
```

Code listing 12 Parser interface

When creating the new parser, we require that the passed state type to the parser must satisfy the necessary trait bounds, where the error type will be hardcoded to a well-known error type.

The mentioned state will represent our business logic in a plain Rust world without knowing about the parser implementation. The state needs to implement the parser callback trait. Therefore the state will receive a callback notification once the parser is populated with more data.

The passed state needs to be allocated on the heap. We need to guarantee that the value will not move. Otherwise, the provided opaque pointer to the user data in the initialization phase of the parser would not be valid in the long run. However, we cannot use the Box primitive directly because Box has strict-aliasing rules. We must assert uniqueness over its content. We must ensure that raw pointers derived from a box after that Box has been mutated through,

moved, or borrowed as mutable is not allowed[64].

```
pub struct Parser<T>
where
    T: ParserCallbacks<NvidiaDecoderError>,
{
    /// Newtype around *mut c_void that
    /// is Send and Sync.
    handle: ParserHandle,
    state: *mut T,
}
```

Code listing 13 Parser structure

In summary, we have the parser abstraction. This parser abstraction will own the state. This will represent the decoder instance, where the state needs to implement the required trait bound. So when we feed the parser with more data, the parser will call the callbacks on the state when deemed appropriate.

5.2 Video decoder

The video decoder has an independent API from the parser. Therefore, it has its initialization process. The decoder's configuration options include parameters like: codec, chroma format, output frame format, deinterlace mode, output surfaces, and counting. Most of these options are obtained from the sequence callback, where the parser triggers that. Parser callback provides everything in the `CUVIDEOFORMAT` structure. Decoder is then created by calling the `cuvidCreateDecoder(&mut local_handle, options)`.

Therefore, the best way to continue is to declare a new structure for the Decoder part since it is an independent unit. So the first step is to implement the `ParserCallbacks`, where we will instantiate the Decoder in the sequence callback. We are obliged to define codec in the sequence callback. NVDEC itself supports multiple codecs. However, we will narrow it down to just the H264 for simplicity. Also, we need to specify the output format, which is the pixel format of the outputted frames. We need to put the NV12 pixel format. Others are supported in rare situations, unlike the mentioned NV12, which is supported for all input data. NVDEC also supports dynamic resolution change, which means that if input data changes during the stream, then the decoder reconfiguration is triggered, and we do not have to allocate the new Decoder so we can reuse the existing one. It is also not used in many situations. To fulfill our predefined trait contract, we need to return the `SequenceOutput` from the sequence callback. This is to let the parser know which value we determined.

5.2.1 Submitting decoding

We will implement the decoding of the frames in the decode callback. The parser triggers this callback when it has enough data. The decoding is initiated by calling the `cuvidDecodePicture`, where we get the arguments for this function in the callback. We also need to execute the decoding in the context of the GPU.

Also, we need to remember that the decoding is blocking operation. Therefore, we must feed data into the parser on a separate thread or blocking task. We will be using tokio[65] for the multithreading part. Tokio offers us to offload the blocking operation to the individual thread by calling `tokio::task::spawn_blocking(...)`.

```rust
pub struct NvidiaDecoder {
    parser: Arc<Mutex<Parser<Decoder>>>,
}
tokio::task::spawn_blocking(move || {
    let mut inner = inner.blocking_lock();
    // Decode operation is blocking.
    inner.parse_new_data(&data)
})
.await
.expect("Video decoding cannot panic.")?;
```

Code listing 14 Parser offloading

However, it has problems. The function `tokio::task::spawn_blocking(...)` requires that the passed value must have a static lifetime. Therefore we must create an atomic reference count (ARC) and also put it inside the mutex because we require a mutable reference. Every time we call spawn blocking, we must acquire the lock even though we know that we are the only one contesting the lock.

5.2.2 Retrieving submitted frame

Once the frame is ready to be retrieved from the decoder, the parser triggers the display callback. The decoded frame resides in the decoder buffer. To retrieve the frame, we need to allocate a separate memory slab in the GPU, and then we need to copy it from the decoder buffer into the pre-allocated memory slab.

When the callback is triggered, the decoder expects that we will map into the memory the frame ready to be decoded. The function `cuvidMapVideoFrame64` maps video frames corresponding to the given index for use in the CUDA world. It returns the device pointer and associated options for the frame. Once we successfully map the video frame, then we can retrieve the decode status of the frame by calling the `cuvidGetDecodeStatus`. Decode status tells us whether the decoding process was successful. If it is successful, then we can copy the data from the buffer to the preallocated slab in the GPU memory. Where in our case, the decoded frame would be in the NV12 format. This output format depends on the decoder configuration. However, we hardcoded this value always to be the NV12 pixel format because others are not well supported, and they do not work for all input types. Keep in mind that once we successfully mapped the video frame, we also need to correctly unmap it by calling the `cuvidUnmapVideoFrame64`, so we can create a straightforward abstraction over it. We can create a helper structure that will map the frame inside the constructor and unmap it in the destructor. Thus we do not forget to deallocate it correctly. Once the frame is copied to the separate slab, we are done and ready to transport decoded frame to the next stage for preprocessing, which can be resizing/scaling or encoding into a different codec.

5.3 Abstraction summary

Our abstraction consists of a parser abstraction wrapper that handles all callbacks. Parser has two functions: a generic constructor over a parameter requiring callbacks implemented and it also has a function from which we can feed new data into the parser. From this, we must remember that the parser is blocking and pushing new data triggers callbacks on the generic parameter. Therefore it can be blocking. Next, we implemented abstraction over the decoder, where the

decoder implements callbacks required from the parser trait bound. Therefore parser triggers a function on the decoder and initiates decoding.

To wrap this up, we can implement a glue that glues these two components together and handles blocking operations. Our decoder has a contract that requires sending a channel. Therefore, we can take advantage of the channel implementation in the `tokio`[66] library. When we push data into the glue, we call `spawn_blocking` as mentioned earlier and await finish. Once the newly decoded frame is ready, we can expect it on the receiving part of the channel.

```
/* imports omitted */
pub struct NvidiaDecoder {
    parser: Arc<Mutex<Parser<Decoder>>>,
}
impl NvidiaDecoder {
    pub fn new(
        cuda_context: Arc<CudaContext<Initialized>>,
        codec: VideoCodec,
    ) -> Result<
        (
            Self,
            mpsc::UnboundedReceiver<
                Result<NV12DecodedGpuFrame,
                NvidiaDecoderError>>,
        ),
        NvidiaDecoderError,
    > {
        /* initialization omitted */
    }
    pub async fn push_new_data(&mut self, data: Vec<u8>)
        -> Result<(), NvidiaDecoderError> {
        let inner = self.parser.clone();
        // Decoding itself is a blocking operation.
        {
            tokio::task::spawn_blocking(move || {
                let mut inner = inner.blocking_lock();

                // Decode operation is blocking.
                inner.parse_new_data(&data)
            })
            .await
            .expect("Video decoding cannot panic.")?;
        }
        Ok(())
    }
}
```

Code listing 15 The decoder glue

Chapter 6

NVIDIA encoder wrapper

Encoder is responsible for encoding individual frames. The output is in the form of a byte stream. Namely, NVIDIA supports H264 and H265. The latest generation of the GPU newly supports AV1. Encoder also ensures that the appropriate metadata for the byte stream is generated. This metadata can include arbitrary SEIs or even subtitles.

The NVIDIA GPU has a dedicated hardware accelerator for the encoding part. It is commonly known as the NVENC. To access the NVENC, we need a library `libnvidia-encode.so` that includes all functions that we need. Moreover, we will need the CUDA abstraction that we have implemented in the chapter 4 to be able to submit frames for encoding. Because in our processing pipeline, we store decoded frames in the GPU memory, therefore, we also supply the frames to the encoder from the same GPU memory. We limit our use case only to the H264 codec for simplicity. We assume that bindings to the library were correctly generated by the `bindgen`[54].

6.1 Binding generation caveats

Generating bindings for the NVENC has some caveats. `Bindgen` has a problem with C macro expansions[67]. This is rather unfortunate because NVENC API requires specifying the version of parameters that are supplied with each call, where the version is specified by the mentioned macro expansion. However, the stock `bindgen` configuration is not able to generate this expansion. Therefore, we need to help the `bindgen` a little bit. We can tweak our `bulid.rs` to generate bindings only for the necessary functions and to fix the macro expansion.

```rust
// Related https://github.com/rust-lang/rust-bindgen/issues/753
#[derive(Debug)]
struct Fix753;
impl bindgen::callbacks::ParseCallbacks for Fix753 {
    fn item_name(&self, original_item_name: &str) -> Option<String> {
        Some(original_item_name.trim_start_matches("Fix753_").to_owned())
    }
}
fn main() {
    println!("cargo:rustc-link-lib=nvidia-encode");
    let bindings = Builder::default()
        .clang_args(["-I", env!("CARGO_MANIFEST_DIR")])
        .header("wrapper.h")
        .parse_callbacks(Box::new(Fix753 {}))
        .allowlist_type("^CU.*")
        .allowlist_function("Nv.*")
        .allowlist_type("NV.*")
        .allowlist_var("Fix753.*")
        .allowlist_var("NVENC.*")
        .default_enum_style(bindgen::EnumVariation::Rust {
            non_exhaustive: false,
        })
        .derive_debug(true)
        .impl_debug(true)
        .generate()
        .expect("Unable to generate bindings");
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings
        .write_to_file(out_path.join("nvenc.rs"))
        .expect("Couldn't write bindings!");
}
```

Code listing 16 NVENC `build.rs` bindgen helper

Unfortunately, it is not the only caveat. The GUID profiles need to be correctly exported as well. GUID profiles are used in the encoder configuration step. GUID profiles tell which profiles and parameters we would like to use, for example, which quality preset we want. Fortunately, we can handle this easily with the Rust macros.

6.2 Video encoder

To get started, we firstly need to check for the API compatibility version and initialize the encoding API. The check for the max supported version is required to avoid runtime errors of incompatible calls. The current API version is defined in the header file because every NVENC header file is associated with a particular NVENC version. Where the current version is equal to the `NVENCAPI_MAJOR_VERSION << 4 | NVENCAPI_MINOR_VERSION`, so the current API version must be lower than the maximally supported version that can be retrieved by calling the `NvEncodeAPIGetMaxSupportedVersion`. If this check passes, we can obtain the mentioned API function list. The API function list is required to call individual functions performing NVENC operations. We do this by calling the `NvEncodeAPICreateInstance` where the argument is the `NV_ENCODE_API_FUNCTION_LIST`. This structure has a property version, and we, as the caller, must

populate the version field with the version defined in the header file. That is why we needed a working macro expansion from the bindgen to avoid random numbers in our source code.

6.3 Video encoder configuration

After we successfully obtain the function list, we can proceed with the initialization of the encoder itself. To do so, we need to create a valid configuration. The configuration is represented by the `NV_ENC_INITIALIZE_PARAMS` and the `NV_ENC_CONFIG`. We must specify the codec. The codec is defined by the GUID that we mentioned earlier. Each codec has its own GUID. Currently, the NVENC supports three codecs H264, H265, and newly the AV1. As we mentioned earlier, we will focus only on the H264 codec. Next, we need to specify the preset. This preset is only NVIDIA-specific, where each preset will generate predefined defaults that can be overridden later. The defaults are handy because we do not have to tune every property, so we can focus only on those that are interesting to us. Presets can go from P1 up to P7, where P1 means defaults that consume the lowest amount of resources up to P7, which can be hungry for resources.

After that, we are obliged to specify the width and the height. NVENC supports changing the width and the height in runtime. Therefore it allows us to specify the encoding width and max encode width and vice versa with the height. These three properties are mandatory, and others are optional. We can also specify the encoded framerate. However, the encoder itself does not do frame dropping and frame adding. If we are about to lower the frame rate or increase the frame rate from the input, we must do the frame adding or dropping by ourselves. Moreover, we can also edit the rate control options that include the variable bitrate and the constant bitrate, where we can hint at the average bitrate. Also, we can specify the keyframe interval by GOP length.

After we assemble our desired configuration, we can call the `nvEncInitializeEncoder` from the function list. It initializes provided pointer with the encoder.

6.4 Encoder input buffers

NVIDIA supports submitting frames for encoding from the userspace memory. However, we want optimal utilization by keeping the decoded frames in the same GPU memory. It is much faster than copying decoded frames from the GPU back to the user space memory and sending them again to the GPU memory for encoding. NVIDIA supports submitting frames from the GPU memory. However, it requires extra steps. We need to register new input buffers that will hold the frames that are ready to be encoded.

These buffers require extra bookkeeping that we can handle with helper wrappers. We can model them as a simple state machine. One state is the mapped state, and the second state is the unmap state. We can generally start with the following Listing 17, where we have mapped and unmapped buffers that share a standard structure.

Input buffer, in general, needs to hold a pointer to the encoder. Because when creating the input buffer, we must also register it to the encoder. Also, the encoder owns the buffers. Therefore it needs to be wrapped in the atomic reference counter to prevent leaking the memory. Also, the input buffer needs to allocate GPU memory through our CUDA implementation that we did earlier. When we register the buffer, it first starts in the unmapped state. When the buffer is unmapped, we can only expose a single function called map that requires self-reference. Therefore we can guarantee in compile time that the map is not called more than once because calling it performs a transition to the other state. We can simulate this in the Listing 18.

Also, we need to implement the destructor for the input buffer, which will unregister the resource eventually.

When the buffer is mapped, we must implement a special destructor for the `MappedInputBuffer` that will unmap the resource. We must do it because we cannot unregister resources that

```rust
pub(crate) struct InputBuffer {
    encoder: Arc<EncoderWrapper>,
    pub slab: DeviceSlab<ContinuousMemory>,
    pointer: *mut c_void,
}

pub(crate) struct MappedInputBuffer {
    input_buffer: Option<InputBuffer>,
    mapped_resource: MappedResourcePtrWrapper,
}

pub(crate) struct UnmappedInputBuffer(InputBuffer);
```

Code listing 17 NVENC buffers

```rust
impl MappedInputBuffer {
    pub(crate) fn get_mapped_resource_ptr(&self) -> *mut c_void {
        self.mapped_resource.0
    }

    pub(crate) fn unmap(mut self)
        -> Result<UnmappedInputBuffer, NvidiaEncoderError> {
        #[allow(clippy::expect_used)]
        let input_buffer = self.input_buffer.take().unwrap();

        unsafe {
            input_buffer
                .encoder
                .function_list
                .nvEncUnmapInputResource
                .unwrap()(
                input_buffer.encoder.encoder_ptr,
                self.mapped_resource.0,
            ).unwrap()
        };

        Ok(UnmappedInputBuffer(input_buffer))
    }
}
```

Code listing 18 NVENC unmap buffer transition

are mapped. Therefore, this nice abstraction will solve all possible logical bugs.

The transition from the mapped state is similar to the mapped buffer. We expose the unmap function that behaves like a destructor, so it unmaps the buffer, and the function returns `UnmappedInputBuffer`.

6.5 Encoder output buffers

We also need to prepare output buffers for the encoder. NVENC supports the output of the encoded frames into the GPU memory. However, we do not need this feature. Therefore, we can use the direct simplified API to download the encoded frame. For this purpose, we need to create an abstraction over the buffer. We call `nvEncCreateBitstreamBuffer` in the constructor of the OutputBuffer, and in the destructor, we will perform the correct deallocation. Once we want to retrieve the encoded data from the NVENC, this can be done only after successful encoding. Then we must lock the output buffer. Once it is in the locked state, we can retrieve the encoded data.

This can model this abstraction in a way where the `OutputBuffer` has a single method `lock` that will return a drop guard that will hold the reference to the parent. This drop guard pattern[68] will ensure that we do not hold the locked buffer for too long as well as due to the borrow checking rules, we cannot call the `lock` more than once, and the compiler will guarantee this for us.

In the `LockedBuffer<'_>`, we then can write data into an arbitrary type that implements the `Write` trait. Once the buffer is locked, we can download the NVENC buffer. When the buffer is locked successfully, it hints us the size of the encoded byte stream. So as long as the buffer is locked, we can download the output data. We can do it by resizing our buffer to the expected size and calling the memory copy from the exposed buffer into the output preallocated buffer.

```rust
pub(crate) struct OutputBuffer {
    encoder: Arc<EncoderWrapper>,
    // Place where encoded data are held.
    temporary_buffer_space: Vec<u8>,
    pointer: OutputPtrWrapper,
}
impl OutputBuffer {
    pub(crate) fn lock(&mut self)
        -> Result<LockedOutputBuffer, NvidiaEncoderError> {
        /* calling the nvEncLockBitstream */
    }}
pub(crate) struct LockedOutputBuffer<'a> {
    buffer: &'a mut OutputBuffer,
    expected_data_size: usize,
    bitstream_ptr: *mut c_void,
}

impl LockedOutputBuffer<'_> {
    pub fn write_into<T>(&mut self, destination: &mut T)
        -> std::io::Result<()> {
        /* writing downloaded data into the destination */
    }
}
```

Code listing 19 NVENC output buffer

6.6 Encoding process

Once we have done all the configuration and wrapper shenanigans, we are ready to send frames for encoding. We can make a high-level structure `NvEncoder` that will accept the CUDA context and the NVENC configuration. In the constructor, we can create all the necessary buffers. Typically we want to create the same amount of input buffers as well as output buffers. We can make this number of buffers configurable.

Next is the function that accepts the frame itself. We need to define a wrapper structure that will represent the frame. This wrapper structure needs to contain a GPU memory slab. And we can pass other meta pieces of information, for example, SEIs or if we want to force the keyframe. These parameters are stored in the `NV_ENC_PIC_PARAMS` structure.

Initially, we need to obtain one input and output buffer from our buffer of free ones. If we do not have any, we cannot continue. We need to copy the input frame to the mapped input buffer, and after that, we can get the mapped resource pointer. Also, we need to get the underlying pointer to the output buffer. We need to do the bookkeeping for these buffers because when we run the encoding function `nvEncEncodePicture` it can return two non-error status codes.

If the encoder returns with `NV_ENC_ERR_NEED_MORE_INPUT` we need to put both the mapped input buffer and the output buffer to hold into the FIFO queue. This status code can be used when we set up the encoder to use the B-frames. Therefore, we need to wait for more user input.

If the encoder returns the `NV_ENC_SUCCESS` we can iterate through the FIFO queue of the output buffers. In order, we need to lock the output buffer and copy the data from the locked buffer to our desired destination for each of them. After we copy the data from the output buffer, we can drop the lock, unmap the corresponding input buffer, and return both buffers to the pool of free buffers.

6.7 Encoder wrapper summary

Our abstraction involves implementing multiple buffer wrappers for input and output frames and the encoder itself. Moreover, it handles all possible weird cases, whereas our API is still simple enough. We only expose the constructor that prepares everything and the encoding function, where the encoding function will write output data (if any) into the provided mutable reference that implements the well-known `Write` trait.

Chapter 7

GPU frame resizing wrapper

Frame resizing is responsible for doing multiple output qualities. Typically we want to create multiple output resolutions for the incoming stream. Moreover, we often want to downscale the incoming stream.

A classic scenario is that we decode the incoming video stream. By decoding the stream, we obtain individual frames of the video. Then we downscale each frame to multiple resolutions and send this downscaled frame into the particular encoder for the given resolution. Because each encoder expects only a single video track, or at least our implementation of NVENC expects that.

We will perform this resizing in the NVIDIA GPU with its CUDA capabilities. There are already existing NVIDIA primitives that are widely popular. Therefore, we will go with the NVIDIA Performance Primitives (NPP); this library provides GPU-accelerated image functions that are faster than the CPU-only implementations[69].

Everything is located in the `npp.h` header file and we will be using the `libnppig.so` and the `libnppicc.so`.

We will use only a subset of these primitives because we need only three functions. We assume that bindings to the library were correctly generated by the `bindgen`[54].

7.1 Error handling

In order to ensure correct memory handling, we need to create a special error type. We need to encapsulate the general NPP error and our custom errors. For example, for us, it does not make sense to resize frames that do not have width or height divisible by two.

So we can create the following error type. We will also write a helper macro that will help to check for successful returns from the FFI function.

```rust
#[derive(Error, Debug, PartialEq, Eq)]
pub enum NppError {
    #[error("Npp error: {:?}.", .0)]
    Library(crate::ffi::NppStatus),
    #[error("Cuda error: {0:?}")]
    Cuda(#[from] rust_cuda::CudaError),
    #[error("Height is not divisible by two.")]
    OddHeight,
    #[error("Height is not divisible by two.")]
    OddWidth,
    #[error("Width and heigh do not mach the slab capacity.")]
    InvalidDimensions,
    #[error("Slab is too big to resize.")]
    SlabSizeTooHigh,
}

macro_rules! check_npp {
    ($err:expr) => {{
        if $err != $crate::ffi::NppStatus::NPP_SUCCESS {
            return Err($crate::NppError::Library($err));
        }
    }};
}
pub(crate) use check_npp;
```

Code listing 20 NPP Error type

Also, we will use the `thiserror`[70] crate, which provides convenient derive macros for the enum type. This will implement the `std::error::Error`[71] for our enum types.

To narrow down our custom errors in each of the following functions, we need to validate whether the input width and height of the NV12 frame are equal to the allocated slab. Moreover, we need to check if the input height or input width is not odd. Otherwise, we cannot process it because our encoder implementation requires even width and height.

7.2 Deinterleaving

Our use case is very narrow in this library. Therefore, we will have a limited public API. We will define a structure that will hold the CUDA-allocated device slab with continuous memory. Define the width and height as `u16`. And that will be our only parameter for the function. The Rust programming language currently does not support named arguments of the function. Therefore when combining multiple arguments with the same data type in sequential order, we can accidentally switch the position of the arguments. This can be fixed by a mentioned structure where each property has its name, and we cannot mismatch the height and the width of the frame. We will use this pattern for the upcoming function wrappers as well.

Our input is a raw frame with NV12 pixel format encoding, which we mentioned in the Video Engineering chapter. However, NPP does not support NV12 as the interleaved format. Instead, it wants to be in the planar format, so each plane is separate. So we need to separate the luma plane and the chrominance plane. Therefore, we first need to begin with deinterleaving the input NV12 frame.

We will wrap the `nppiYCbCr420_8u_P2P3R`[72] function from the NPP library that does the mentioned conversion. We must not forget to execute the function in the CUDA context. Oth-

erwise, we are not able to access the slab pointer. The first parameter to the mentioned NPP function is a pointer to the GPU-allocated luma plane. This one is easy because NV12 starts with the luma plane. Therefore we can give it the initial pointer to the DeviceSlab. The second argument is the width of the frame. Next follows the interleaved chroma plane. The offset of this plane is the product of the width and the height. Next follows the output arguments, where we need to specify to which position we want to save the deinterleaved planes.

```
let ctx = slab.get_context();
let carry_slab_for_deinterleaving = ctx.mem_alloc(slab.capacity())?;
let y = slab.get_inner_ptr();
let cb_cr_offset = u64::from(width) * u64::from(height);
let cb_or_cr_size = u64::from(width) * u64::from(height) / 4;
let cbcr = slab.get_inner_ptr() + cb_cr_offset;

ctx.run_in_context(|| {
    check_npp!(unsafe {
        nppiYCbCr420_8u_P2P3R(
            y as *const u8,
            width,
            cbcr as *const u8,
            width,
            [
                carry_slab_for_deinterleaving.get_inner_ptr(),
                carry_slab_for_deinterleaving.get_inner_ptr() + cb_cr_offset,
                carry_slab_for_deinterleaving.get_inner_ptr()
                    + cb_cr_offset + cb_or_cr_size,
            ]
            .as_mut_ptr()
            .cast(),
            [width, (width / 2), (width / 2)].as_mut_ptr(),
            NppiSize { width, height },
        )
    });
    Ok(())
})??;
```

Code listing 21 NPP deinterleave

7.3 Interleaving

Interleaving is the inverse process of deinterleaving. We need it for assembling back the correct NV12 frame. On the input, it expects data to be in the planar format.

The interleaving function will have the same function arguments as the mentioned deinterleaving function. It will accept the struct wrapper that wraps the slab, width, and height. At the beginning of the function, we also need to perform the same checks for input validation.

We will wrap the `nppiYCbCr420_8u_P3P2R`[73] function from the NPP library that does the conversion. This function also must be executed in the CUDA context. Otherwise, it will not have the correct memory access rights. The arguments are similar to the deinterleaving version. It expects an array of separated planes and their corresponding width. Also, we must specify the destination to which it will output the results.

```rust
let ctx = slab.get_context();
let carry_slab_for_interleaving = ctx.mem_alloc(slab.capacity())?;

let y = carry_slab_for_interleaving.get_inner_ptr();
let cb_cr_offset = u64::from(width) * u64::from(height);
let cb_or_cr_size = u64::from(width) * u64::from(height) / 4;
let cbcr = carry_slab_for_interleaving.get_inner_ptr() + cb_cr_offset;

ctx.run_in_context(|| {
    check_npp!(unsafe {
        let width = i32::from(width);
        let height = i32::from(height);
        #[allow(clippy::as_conversions)]
        nppiYCbCr420_8u_P3P2R(
            [
                slab.get_inner_ptr(),
                slab.get_inner_ptr() + cb_cr_offset,
                slab.get_inner_ptr() + cb_cr_offset + cb_or_cr_size,
            ]
            .as_mut_ptr()
            .cast(),
            [width, (width / 2), (width / 2)].as_mut_ptr(),
            y as *mut u8,
            width,
            cbcr as *mut u8,
            width,
            NppiSize { width, height },
        )
    });
    Ok(())
})??;
```

Code listing 22 NPP interleave

7.4 Resizing

Resizing is the process of changing the resolution of the frame. We will only do the downscaling, which means that we will only lower the resolution of the given frame. Upscaling is a challenging problem because we need to come up with arbitrary pixels to fill the gaps. On the other hand, lower resolution just means removing some already existing pixels, which is a relatively straightforward process.

Our resize function will tight together the previously implemented deinterleaved and interleave functions. Moreover, it will wrap the `nppiResizeSqrPixel_8u_C1R`[74] function. We will accept the frame information from a wrapper structure, where this previously mentioned wrapper structure holds together the allocated slab, width, and height of the frame. Moreover, we will specify the destination width and the destination height. At the beginning of the function, we must check for the even height and even width. Also, we must validate that the allocated slab capacity and its corresponding width and height are equal. Otherwise, we could expect weird behavior.

At the beginning of the function (apart from the input validation), we will call the previously

```
// This will be for resizing output.
 let carry_slab_for_resizing = ctx
    .mem_alloc((f32::from(destination_width)
        * f32::from(destination_height) * 1.5) as usize)?;

// Needed offsets for the deinterleaved plane.
let src_cb_cr_offset = u64::from(width) * u64::from(height);
let src_cb_or_cr_size = u64::from(width) * u64::from(height) / 4;
let dst_cb_cr_offset = u64::from(destination_width)
    * u64::from(destination_height);
let dst_cb_or_cr_size = u64::from(destination_width)
    * u64::from(destination_height) / 4;
```

Code listing 23 Resize offsets

implemented function for deinterleaving. The deinterleaving function will output a new GPU memory slab containing sequentially stored planes. After this, we need to calculate offsets for the given slab. We can see this in the Listing 23.

The resizing part needs to be done for each plane. This means that we will need to call resize for the luma plane and both chroma planes. However, the luma part will have different arguments than the chroma planes for the resize function. We can see this preparation in the Listing 24.

As mentioned before, we need to call the resize function multiple times. We have already prepared the `ResizeStruct` helper that we will use now. The structure encapsulates the given source pointer, source width, source height, destination width, and destination height. The source pointer will always be the deinterleaved slab we have received with the deinterleave function implemented before. We also need to have preallocated destination slab for the resizing part.

We can see the implementation in the Listing 25.

After the resizing part, we will call the previously implemented interleave function that will glue everything together to form a valid NV12 frame again.

```
let planes_to_resize = [
    ResizeStruct {
        src_ptr: deinterleaved_slab.slab.get_inner_ptr(),
        src_width: width,
        src_height: height,
        dst_ptr: carry_slab_for_resizing.get_inner_ptr(),
        dst_width: destination_width,
        dst_height: destination_height,
    },
    ResizeStruct {
        src_ptr: deinterleaved_slab.slab.get_inner_ptr() + src_cb_cr_offset,
        src_width: width / 2,
        src_height: height / 2,
        dst_ptr: carry_slab_for_resizing.get_inner_ptr() + dst_cb_cr_offset,
        dst_width: destination_width / 2,
        dst_height: destination_height / 2,
    },
    ResizeStruct {
        src_ptr: deinterleaved_slab.slab.get_inner_ptr()
            + src_cb_cr_offset
            + src_cb_or_cr_size,
        src_width: width / 2,
        src_height: height / 2,
        dst_ptr: carry_slab_for_resizing.get_inner_ptr()
            + dst_cb_cr_offset
            + dst_cb_or_cr_size,
        dst_width: destination_width / 2,
        dst_height: destination_height / 2,
    },
];
```

Code listing 24 Wrapper for resizing planes

```rust
for i in planes_to_resize {
    ctx.run_in_context(|| {
        check_npp!(unsafe {
            nppiResizeSqrPixel_8u_C1R(
                i.src_ptr as *const u8,
                NppiSize {
                    width: i.src_width,
                    height: i.src_height,
                },
                i.src_width,
                NppiRect {
                    x: 0,
                    y: 0,
                    width: i.src_width,
                    height: i.src_height,
                },
                i.dst_ptr as *mut u8,
                i.dst_width,
                NppiRect {
                    x: 0,
                    y: 0,
                    width: i.dst_width,
                    height: i.dst_height,
                },
                f64::from(destination_width) / f64::from(width),
                f64::from(destination_height) / f64::from(height),
                0.,
                0.,
                NppiInterpolationMode::NPPI_INTER_CUBIC as i32,
            )
        });
        Ok(())
    })??;
}
```

Code listing 25 Resize function

7.5 Frame resizing summary

In this chapter, we have successfully implemented the wrappers around NVIDIA NPP primitives. After all, we have three wrapper functions because our resize function generally consists of 3 underlying functions: deinterleave, resize, and interleave. We have also implemented some input validation checks to prevent resizing invalid frames for the decoder. For example, the odd height or width frames are considered invalid for the NVENC.

Chapter 8

Implementing transcoding service

In this chapter, we will implement and discuss individual subsystems of the transcoding service. For these subsystems, we will use the above-implemented wrappers and helpers. The video transcoding service will have the following subsystems.

- **Ingest part** – We will use the RTMP protocol for the ingestion part. The RTMP protocol is very straightforward for this use case because there are libraries that already implement the serialization protocol in the Rust programming language. Moreover, the FLV format that RTMP uses is also straightforward to implement.
- **Playback** – For the playback part, we will use the RTMP. However, the playback path will differ from the publishing path because our transcoding service can produce multiple qualities. FLV and, thus, RTMP support only a single video track. So we must make the stream available for playback on a different path. More on that later.
- **Scheduling part** – The critical component of a distributed transcoding service is to take advantage of multiple GPUs that are present in the host operating system. We will develop a cautious scheduling system that will distribute the transcoding jobs based on their cost.
- **Muxing and demuxing part** – So far, we have yet to implement higher-level abstraction for format handling. FLV is austere to implement. Therefore, we will implement a quick wrapper that will help us encapsulate the data from the encoder.

For the sake of simplicity, we will omit the audio handling. Therefore we will drop the incoming audio, and it will not be represented on the output. Moreover, we will not take care of exceptional cases like reconnecting the publisher, dynamic resolution change, adaptive bitrate streaming, etc. Also, we will implement tests that will be able to test everything that we have implemented in the CI environment.

8.1 Video ingestion

Video ingestion is related to the process where we deliver data into the transcoding server. We can split it into two categories, and that is the push model and the pull model. We will go with the push model, where we send new video data to the transcoding service. The pull model is used only in exceptional cases, for example, video restreaming. However, restreaming is different from our use case.

RTMP is a complex protocol. It is an RPC-based protocol with its own serialization mechanisms and control flow. Luckily we will not have to implement the publishing part. We can

already use existing software like OBS[75] or FFmpeg[76]. The tricky part is the receiving part from the server's perspective. Fortunately, there is already an existing library in Rust that will help us (more on that later).

8.1.1 RTMP

For the RTMP part, we will use a library called `rust-media-libs`[77]. This library supports the client part and the server part implementation of the RTMP protocol. However, we will be using only the server-side part.

Library has two levels of abstraction. It can act as a low-level serialization library that only guarantees serialization correctness, and the high-level abstraction generally encapsulates the RPC calls. Therefore, the high-level abstraction exposes functions like publishing requests and so on. It is implemented like a state machine. It is not bound to a particular socket. It just accepts bytes on the input, and as an output, it generates events to which we can react. When we want to react to a particular event, then we call the function on the state machine, and it serializes the messages and generates bytes that we are responsible for shipping to the end client.

When acting as a server, we must first listen to the specific port. RTMP is commonly bound to port 1935. We can use the `TcpListener`[78] from the `tokio`[65] crate. Once we accept the incoming client, we move the stream handle to a different task via `tokio::spawn`[78].

```
let mut listener = TcpListener::bind("0.0.0.0:1935").await?;
loop {
    let (stream, socket) = listener.accept().await?;
    tokio::spawn(async move {
        // Start RTMP state machine.
    });
}
```

Code listing 26 RTMP listening loop

The first RTMP-specific thing that we need to do is to do an RTMP handshake. Because before the client can process any RTMP traffic, both sides must perform a handshaking process. The handshake itself is implemented as part of the RTMP library that we are going to use[79].

Initially, we need to create the `Handshake` instance and specify the peer type that we want to act as a server. After that, we poll the socket for new bytes and try to progress with the handshaking. So, we read new bytes into the intermediate buffer. The read function returns how many bytes were read. If zero bytes are read, the client is disconnected, so we can stop processing the request and return. As long as we have new bytes incoming, we can feed it to the instantiated `Handshake` instance until the handshake completion. If we do not complete the handshake process in the first iteration, e.g., we do not have read enough bytes, we must continue reading until we have enough. The handshake instance keeps buffering the incoming data, so we supply it only with new data. Ultimately, we must return the bytes we have not used so far because they are necessary for the follow-up steps.

```rust
/// Performs handshake and returns remaining bytes.
async fn perform_handshake(stream: &mut TcpStream)
-> Result<Vec<u8>, RtmpBridgeError> {
    let mut buffer = [0; BUFFER_SIZE];
    let mut server = Handshake::new(PeerType::Server);
    loop {
        let read_bytes = stream.read(&mut buffer).await?;
        if read_bytes == 0 {
            return Err(RtmpBridgeError::ClientDisconnected);
        }
        match server.process_bytes(
            buffer
                .get(..read_bytes)
                .ok_or(RtmpBridgeError::UnableToIndexBuffer)?,
        )? {
            HandshakeProcessResult::Completed {
                response_bytes,
                remaining_bytes,
            } => {
                stream.write_all(&response_bytes).await?;
                break Ok(remaining_bytes);
            },
            HandshakeProcessResult::InProgress { response_bytes } => {
                stream.write_all(&response_bytes).await?;
            },
        };}}
```

Code listing 27 RTMP server handshake

The next step is to setup is the server session, where the server session encapsulates the parsing of the RTMP chunks coming from the client into the RTMP messages. It performs common sever side workflows to handle those messages. Moreover, it can provide pre-serialized messages that can be sent back to the client or events that other applications can perform custom logic against. However, we will use only the serialization part. Remember that the server session does not care how RTMP chunks are transferred or received. It leaves that space for us as the implementor of this library. During the initialization, we must pass the remaining bytes into the instantiated server session. It is required due to the properties of the header compression of the RTMP chunking protocol. Moreover, it requires that all responses generated by the server session are returned to the client **in order** as they were generated. Also, we must ensure that no additional bytes are sent to the client by us (other than those generated by the server session implementation) because any violation of these rules can lead to parsing errors by the client[80].

```rust
// Server initialization.
let mut events = VecDeque::new();
let (mut session, results) = ServerSession::new(ServerSessionConfig::new())?;
events.append(&mut session.handle_input(&remaining_bytes)?.into());
events.append(&mut results.into());
```

Code listing 28 RTMP server session initialization

After the server session initialization boilerplate, we must start the RTMP server event queue handler. During the server session creation, we created the `VecDeque`[81], which is the doubled-sided ring buffer. It serves the purpose of saving events that are ready to be processed. This event can be the output response of the raised event. The output response must be sent to the client via the create socket stream. Therefore we take the bytes and write them all to the socket. The raised event is something that we need to handle server side. Most of the time, these events are parsed client data. So it contains things like received audio data, video data, metadata changes, publish requests, connect requests, etc. When the whole event queue is processed, we just try to progress. This `try_progress` means that we are trying to read new client-side data and parse them to populate the event queue.

```
// RTMP event loop
loop {
    while let Some(i) = self.events.pop_front() {
        match i {
            RtmpServerStateMachineResult::OutboundResponse(p) => {
                self.stream.write_all(&p.bytes).await?;
            },
            RtmpServerStateMachineResult::RaisedEvent(event) => {
                self.handle_event(event).await?;
            },
        }
    }
    // Reads new data from socket.
    self.try_progress().await?;
}
```

Code listing 29 RTMP server event loop

During the event handling, we have multiple options for which events to react to. We will omit most of the events here for simplicity. In the beginning, the first event that the client sends is the connection request. The connection request is tied to the particular application name. We can imagine this as the request from the client that it wants can connect to the particular application name. We then have the possibility to accept the request or ignore it. When we call the accept, we must supply the request-id that we want to accept. We can do this by calling the `accept_request` function on the server session. It then generates events that we need to append to our event queue. After the connection is accepted, the client decides whether it wants to publish or play the particular stream name on a given application name. Keep in mind that there can be multiple stream keys on a given application name. However, only a single publisher is allowed for the combination of the stream key and the application name. If the client requests the publish stream request, it supplies the information to which the stream key it wants to publish. We need to check whether the streaming combination of the stream key and the application name is not already occupied and decline if so. Otherwise, we can accept the publish request. Once we accept the publish request, it starts sending the audio and video data that are represented as the video data event and audio data event. These data events are the FLV chunks. However, for our use case, we will feed this data into the video transcoder that will generate multiple qualities. Suppose the client decides after the connection request that it wants to play the data. In that case, it sends the play stream request that also contains the stream key that it wants to play from the particular application name. After we accept the request, it expects from us that we will send it the FLV audio and video chunks.

```rust
async fn handle_event(&mut self, event: ServerSessionEvent)
   -> Result<(), RtmpBridgeError> {
   match event {
       ServerSessionEvent::ConnectionRequested {
           request_id,
           app_name,
       } => {
           self.events
               .append(&mut self.server_session
                   .accept_request(request_id)?.into());
       },
       ServerSessionEvent::PublishStreamRequested {
           request_id,
           app_name,
           stream_key,
           ..
       } => {
               /* Many things omitted */
           self.events
               .append(&mut self.server_session
                   .accept_request(request_id)?.into());
       },
       ServerSessionEvent::VideoDataReceived {
           data, timestamp, ..
       } => {
               /* TODO: Send data to the transcoder. */
       },
       ServerSessionEvent::PlayStreamRequested { .. } => {
           /* Accept and send audio and video data. */
       },
       event => {
           tracing::debug!(?event, "Unhandled event.");
       },
   }Ok(())}
```

Code listing 30 RTMP event handling

8.1.2 FLV

We need a wrapper to help us deserialize and serialize FLV data. FLV, in general, is represented by chunks, with three types of chunks. The video, audio, and script data. For our purpose, we need only to handle video data. In general, we need a way how to extract H264 video data from the FLV video chunk. Moreover, we need to serialize the video data back once we successfully transcode the video because the encoder does not know how to encapsulate the video data to form the FLV chunk or the RTMP library.

We can observe the FLV packer structure in the following Table 8.1. It starts with packet type; we are only interested in packet type 9, which means it is a video packet. The next is the payload size, which defines how long is the given payload without the header. Stream ID is often hard coded to the number 0 or 1 because, for our use case, it does not make sense to interleave the incoming FLV. Also, it is not a widely used feature of the FLV. Then follows the payload itself. However, we need to parse deeper because this payload represents the FLV video

packet which differs from the FLV packet we are talking about right now. After the payload, there is the size of the packet as a whole. This property is valuable when we seek the FLV from backward. However, this is also not our use case.

Table 8.1 FLV packet header[16]

Packet type	`uint8`
Payload size	`uint24_be`
Timestamp	`uint32_be`
Stream ID	`uint24_be`
Payload	`variable`
Size of packet	`uint32_be`

After identifying the video packets, we can check the frame type and the video codec for the particular chunk. We can limit our scope to the H264 only. Therefore the frame type will always be the number 7, and the frame type depends on what the encoder produces. It will be either a keyframe (number 1) or an inter-frame (number 2).

Table 8.2 FLV video packet header[16]

Frame type	`4 bits`
Video codec	`4 bits`

Once we are sure it is the H264 video codec, we can check the last nested header, which is the H264 packet header. It contains two properties, and that is the packet type, which specifies whether it is the sequence header, NALu, or end of sequence signal. The sequence header contains all the necessary information for decoding. It contains things like sequence parameter sets, decoding parameter sets, and so on. The last property is composition time. This property is not used in the wild and is set to zero.

Table 8.3 FLV H264 packet header[16]

Packet type	`uint8`
Composition time	`int24_be`

To implement all this behavior in the Rust programming language, we will start with the demuxing. We will use the `nom`[82] crate, which is the efficient zero-copy library. It is a parser combinator library with a focus on safe parsing.

We can see the demuxing implementation in the following Listing 31. It uses nom for the deserialization where the function accepts a reference to the slice of `u8`, and it returns a parsed struct with the remaining bytes that were not parsed. The same goes for the video parsing and the H264 parsing. We are interested in the raw `AVCC` format of the H264 byte stream that we mentioned in the first chapters.

For the muxing back, we will implement for each struct the `Into<Vec<u8>>`[83] trait. We can see the example in the Listing 32. The implementation relies on each underlying structure that implements this interface as well. Otherwise, we are not able to serialize it. This method has caveats because it does unnecessary memory allocations.

We will omit the rest of the demuxing and muxing examples because they repeat themself and are really similar to the `Tag` implementation, only with different properties.

```rust
#[derive(Clone, Copy, Debug, PartialEq, Eq)]
pub enum TagType {
    Audio,
    Video,
    Script,
}
#[derive(Clone, Debug, PartialEq, Eq)]
pub struct Tag<'a> {
    pub tag_type: TagType,
    pub data_size: u32,
    /// Timestamp is often represented in milliseconds from the
    /// start of the FLV.
    pub timestamp: u32,
    /// Be aware of the fact that most FLV processing services like
    /// RTMP servers silently assume that only a single `stream_id`
    /// is used. Thus it is recommended to interleave all AV
    /// packets, including metadata, into a single `stream_id`.
    pub stream_id: u32,
    pub data: &'a [u8],
}
pub fn parse_complete_tag(input: &[u8]) -> IResult<&[u8], Tag> {
    flat_map(tuple((tag_type, be_u24)), |(tag_type, data_size)| {
        map(
            tuple((be_u24, be_u8, be_u24, take(data_size), be_u32)),
            move |(timestamp, timestamp_extended, stream_id, data, _)| {
                Tag {
                    tag_type,
                    data_size,
                    timestamp: (u32::from(timestamp_extended) << 24)
                        + timestamp,
                    stream_id,
                    data,
                }
            },
        )
    })(input)
}
```

Code listing 31 FLV demuxing with `nom`

```rust
impl Into<Vec<u8>> for Tag<'_> {
    fn into(self) -> Vec<u8> {

        let data_size = &u64::try_from(self.data.len())
            .expect("We assume that we are running on 64bit platform.")
            .to_be_bytes()[5..]; // u24
        [
            &[self.tag_type.into()],
            data_size,
            &self.timestamp.to_be_bytes()[1..], // timestamp upper
            &[self.timestamp.to_be_bytes()[0]], // timestamp lower
            &u64::from(self.stream_id).to_be_bytes()[5..], // u24
            self.data,
            &[0], // padding byte to fulfil u32
            data_size,
        ]
        .concat()
    }
}
```

Code listing 32 FLV muxing

8.2 Scheduler

The hard part in implementing distributed service is to stress each GPU in the system evenly. Let us call the transcoding process a job. The job represents the transcoding unit, e.g., it consists of decoding, resizing, and encoding. Each job typically has a different utilization because we may want to produce different output quality; therefore, we stress the encoder differently. Moreover, we can generate multiple resolutions, so we utilize the resizing part, or the incoming video has a high resolution; therefore, the decoder is utilized more.

We will go with the most straightforward scheduler implementation that we can implement. There were different experimentations with multiple implementations, and this one gave the most optimal results.

In the beginning, we will predefine transcoding templates. Each template specifies the properties of the transcoding part. The template specifies how many output qualities will be produced. Moreover, it can specify things like how many frames per second each quality should have, bitrate, keyframe duration, profile, and many more things that can be tuned. Furthermore, each transcoding template will have a given cost. This cost needs to be tested and optimized based on the GPU we will use because the cost can differ on each GPU type.

We can relax our condition by assuming that all servers are homogenous and have the same GPU. So each GPU has the same computing power. Each GPU will then have an assigned current capacity and maximum capacity.

In Rust, we can start implementing our idea of the scheduler. Our API will expose three main public functions. The constructor that initializes everything. The scheduler function, where we provide the function with the cost, and the function returns us the chosen GPU that we can use for the scheduling process. And the last function that will deallocate the cost for the chosen GPU once we are done using it.

The scheduler will generally hold the pool of available GPUs and their current usage. It can be represented as a vector of `SchedulableDevice`. We will initialize this pool in the constructor and assign a predefined maximum capacity for each GPU. Since we are in a homogenous system,

each GPU will have the same maximum capacity.

The following main thing is to expose the function to retrieve context for the chosen GPU. As we mentioned earlier, we will go with the simplest solution, and that is to select the least used GPU. In this case, we assume that the cost was carefully chosen and will mostly stay the same over time. It is better to guess the higher cost for the job than the lower cost and test the job utilization on the GPU itself.

```rust
#[derive(Debug)]
struct SchedulableDevice {
    name: String,
    initialized_context: Arc<CudaContext<Initialized>>,
    usage: usize,
    max_usage: usize,
    index: usize,
}
#[derive(Debug)]
pub struct Scheduler {
    devices: Vec<SchedulableDevice>,
}

impl Scheduler {
    pub fn try_new() -> Result<Self, SchedulerError> {
                /* initialization */
    }
    pub fn schedule_transcoder_job(&mut self, cost: usize)
    -> Result<ChosenGPU, SchedulerError> {
        let mut usable_gpus: Vec<&mut SchedulableDevice> = self
            .devices
            .iter_mut()
            .filter(|element| element.usage + cost <= element.max_usage)
            .collect();

        usable_gpus.sort_by(|element1, element2|
            element1.usage.cmp(&element2.usage));

        if let Some(the_chosen_one) = usable_gpus.get_mut(0) {
            the_chosen_one.usage += cost;
            Ok(ChosenGPU {
                index: the_chosen_one.index,
                context: the_chosen_one.initialized_context.clone(),
                usage: cost,
            })
        } else {
            Err(SchedulerError::SchedulerOverloaded)
        }
    } /* more stuff */ }
```

Code listing 33 GPU scheduler

8.3 Transcoder calls

We have the encoder and decoder implementation ready from the previous chapters. For simplicity, we can create an abstraction over the decoder, resizing part, and the encoder. We will call this abstraction transcoder. It will have a simple API where each transcoder instance accepts the hashmap of templates. The output width, height, and frames per second will represent our template. Also, we assume that each template has its own name.

In the following Listing 34, we can demonstrate the planned usage. We define a broadcast channel that will be used to receive new frames from the transcoder wrapper. Sender handlers will be cloned to each separate task, and we will receive the transcoded packets via the receiving handle, where the packets will have the corresponding template name to which they belong. Then we iterate over the templates and create the encoder for each template. We start a subroutine for each template, feeding it with decoded frames from the decoder.

Next, we need to expose a function that will feed new packets to the decoder. We can see the example in the Listing 35, where we submit data to the global decoder per job and await the results of the decoder. Once the decoding completes, resize the frame, and after successful resizing, send it to the corresponding encoder in a separate tokio task.

```rust
let (tx_packets, rx_packets) = broadcast::channel(256);
let mut encoders = Vec::new();
for (key, value) in templates {
    let (tx, mut rx) = mpsc::unbounded_channel();
    let mut encoder = EncoderContext::<NvidiaH264Encoder>::open_for_encoding((
        NvidiaH264EncoderConfig {
            /* template defaults + template values */
        },
        context.clone(),
    ))
    .await?;
    let tx_packets = tx_packets.clone();
    tokio::spawn(async move {
        while let Some((frame, timestamp)) = rx.recv().await {
            encoder
                .send_for_encoding(Some(frame), /* options */)
                .await.unwrap();
            while let EncoderStatus::Alive(PacketStatus::Ready(packets)) =
                encoder.retrieve_encoded_frames().await
            {
                let packets = packets.unwrap();
                // We do not really care whether the packet was delivered.
                tx_packets
                    .send(TranscoderItemResult {
                        packet: packets,
                        timestamp,
                        template: key.clone(),
                    })
                    .ok();
            }
        }
    });
    encoders.push(TranscoderInstance { config: value, tx });
}
```

Code listing 34 Service transcoder

```rust
pub async fn push(&mut self, data: H264AnnexB, timestamp: u32)
    -> Result<(), TranscoderError> {
    self.decoder.send_for_decoding(Some(data)).await?;

    while let DecoderStatus::Alive(FrameStatus::Ready(frame)) =
        self.decoder.retrieve_decoded_frame().await
    {
        let frame = frame?;
        for i in &mut self.encoders {
            let frame = frame.downscale(FrameDimensions {
                destination_width: i.config.width,
                destination_height: i.config.height,
            })?;
            i.tx.send((frame, timestamp)).ok();
        }
    }
    Ok(())
}
```

Code listing 35 Feeding new data into the service transcoder

8.4 Muxing and demuxing

The next step is to integrate the previous transcoder wapper into the RTMP server. We create the transcoder instance within the `PublishStreamRequest`, where we make sure that it is the first client that is currently streaming. We must find the template and calculate the cost for all the needed jobs. For that, we will utilize the RTMP application name that will match the predefined templates. We will reject the stream if the application name does not find any predefined template. Request the GPU from the scheduler based on the cost that we provide. After all of this, we are ready to create a ring buffer where each quality will be outputted. We will not go deeper into ring buffer implementation because it is unattractive. The pool of ring buffers has simple rules, we need to identify each ring buffer by the application name, stream name, and template name. We must ensure that we drop only NALu data from the ring buffer and not the sequence header. Therefore, the first entry in the ring buffer must always be the sequence header, and on top of that the next entry must be the keyframe. Otherwise, we cannot decode the following entries properly unless we reach another keyframe. The sequence header can be omitted when we do not care about players that connect mid-stream, so it will play adequately only for these streams where we requested playback for the given key before the publisher started publishing.

We can see the mentioned implementation in the Listing 36. This code represents the code we can fit into the mentioned `PublishStreamRequest` event handler.

The next crucial part is to supply data to the transcoder. In the previous chapter, we said that the transcoder expects data to be in the `AnnexB` format. However, the `VideoDataReceived` event supplies us with bytes that can be parsed into the FLV's `VideoData`. The FLV video data are in the AVCC format. The video data returns what the underlying frame type is. We do not care much about this value when deserializing. It also informs us about the code type. We must ensure that the codec is always the H264. Once we are sure that the data are in the H264 format, we can parse them and retrieve the `AVCVideoPacket`. There we have the packet type and the composition time. For us, the important thing is the packet type. The packet type can be sequence header, NALu, or the end of sequence signal. The sequence header is a

```
let (stream_key, args) = parse_stream_key_with_args(stream_key);
let Some(templates) = self
    .application_state
    .config
    .transcoding_templates
    .get(&app_name) else {
    return Err(RtmpBridgeError::TemplateNotFound);
};
let total_cost = templates
    .values()
    .fold(0, |item, template| item + template.cost);
let chosen_gpu = self
    .application_state
    .scheduler
    .write()
    .await
    .schedule_transcoder_job(total_cost)?;
let mut publishers = HashMap::default();
for template_name in templates.keys() {
    let stream_identifier =
        construct_unique_key(&app_name, &stream_key, template_name);
    let publisher = self
        .application_state
        .chunk_market
        .create_store(
            stream_identifier.clone(),
            StatikCache { args: args.clone() },
        )
        .await
        .ok_or(RtmpBridgeError::NotFirstPublisher)?;
    publishers.insert(template_name.clone(), publisher);
}
let (transcoder, rx) =
    Transcoder::new(chosen_gpu.context.clone(), templates.clone())
        .await
        .map_err(|error| {
            tracing::error!(%error, "Unable to crate transcoder.");
            RtmpBridgeError::InvalidData
        })?;
tokio::spawn(handle_transcoder_publishing(rx, publishers));
```

Code listing 36 Integrate transcoder into the RTMP server

special thing for the AVCC format. We can be sure that the publisher always sends this as a first FLV video atom. The sequence header contains the sequence parameter set and the picture parameter set. We cannot start the decoding without these two NALus, because they are needed to set up the decoder properly. The proper format of the sequence header is specified in the ISO specification[2, 23].

Once we converted the sequence header to the AnnexB compatible NALu, we are ready to parse the subsequent FLV video data we receive. Now we should expect only the packet types of type NALu. Once we encounter the packet type end of the sequence, then it does not contain any valuable data, so we can drop it and end the transcoding. The mentioned packet types of type NALu must also be converted to AnnexB. To make it AnnexB compatible, we need to traverse the current AVC packet data and replace the width of the atom size with `0x00_0x00_0x00_x01` as mentioned in the earlier chapters. After converting, we supply the data into the transcoder with the defined `push(...)` method.

The last thing is to retrieve the data from the encoders, convert it to the FLV AVCC and store it in the mentioned ring buffer. Our transcoding API return a channel receiver, in which we get the key and a value, where the key is the template name and the value is encoded packets. First, we must convert it to the AnnexB format. We can see naive implementation in the Listing 37. During the conversion, we return the vector of tuples where the bool signals whether the NALu contains the keyframe. After the successful conversion, we can check whether we have already sent the sequence header. If not, we have guaranteed from the NVIDIA encoder that the first generated packets are picture parameter set and sequence parameter set so that we can craft the sequence header from these pieces of information. We create the AVC video packet with composition time zero and the crafted sequence header. We then can create the FLV video header with H264 video codec, and as frame type, we can say that it is a keyframe. For the following packets, we can generate the same; however, it is not the sequence header anymore but NALu and the composition time will always be zero. For the frame type, it will depend on the output of the conversion. It can be only the keyframe or interframe type.

Once we successfully muxed everything, we can append the data into the mentioned ring buffer, where that client can request playback for the particular ring buffer.

```rust
pub fn convert_annexb_to_avcc(buffer: &[u8]) -> Vec<(bool, Vec<u8>)> {
    let mut result = Vec::new();
    let mut current = Vec::new();
    let mut index_current = 0;
    let index_end = buffer.len();
    let mut is_first = true;
    loop {
        if index_current >= index_end {
            let first = current[0];
            let is_keyframe = first & 0b00011111u8 == 5;
            let mut length = current.len().to_be_bytes()[4..].to_vec();
            length.append(&mut current);
            result.push((is_keyframe, length));
            return result;
        }
        if buffer[index_current] == 0 && buffer[index_current + 1] == 0
            && buffer[index_current + 2] == 1
        {
            if !is_first {
                let first = current[0];
                let is_keyframe = first & 0b00011111u8 == 5;
                let mut length = current.len().to_be_bytes()[4..].to_vec();
                length.append(&mut current);
                result.push((is_keyframe, length));
            } else {
                is_first = false;
            }
            current = Vec::new();
            index_current += 3;
        } else if buffer[index_current] == 0 && buffer[index_current + 1] == 0
            && buffer[index_current + 2] == 0 && buffer[index_current + 3] == 1
        {
            if !is_first {
                let first = current[0];
                let is_keyframe = first & 0b00011111u8 == 5;
                let mut length = current.len().to_be_bytes()[4..].to_vec();
                length.append(&mut current);
                result.push((is_keyframe, length));
            } else {
                is_first = false;
            }
            current = Vec::new();
            index_current += 4;
        } else {
            current.push(buffer[index_current]);
            index_current += 1;
        }
    }
}
```

Code listing 37 Converting AVCC to AnnexB

8.5 Continous integration

Let us talk about setting automatic testing environment. Nowadays, we mainly use GitHub[84] or GitLab[85] repositories for storing our code. Moreover, many companies host their GitLab instances on-premise. These online git repositories support pipeline integration. Therefore once we push our code into the online repository, we can run automatic pipelines that can run tests automatically. However, our code and tests rely on the NVIDIA GPU, so we encounter caveats. In my case, I used the company's GitLab instance, where we managed to configure the GitLab runner to use the GPU, which was provided for this thesis by the supervisor. It can be achieved by installing a docker engine that supports the passthrough of NVIDIA GPU, where we say in the GitLab runner that it should execute code in a docker container. The nvidia-docker[86] extends capabilities of the `Dockerfile` syntax[87], enabling us to specify the required capabilities like CUDA version that we require, which device to use, and driver capabilities like encode/decode capabilities, CUDA compute and many more[88].

Chapter 9
Summary

The thesis had several objectives. The first want was to present the current state of video engineering. Moreover, to set up terminology and definitions. Then we slowly started talking about Rust and NVIDIA video-related stuff and started building Rust abstraction over the NVIDIA exported primitives. The abstractions were needed so we could react with the GPU. We successfully implemented abstractions over the CUDA context, NVDEC, NVENC, and NPP. In the latest chapter, we glued everything together to create a video streaming service. The service starts with a configuration file that specifies available transcoding templates and their cost. We can stream into the service via the RTMP protocol. Upon a new RTMP publish request, we allocate GPU for our transcoding purpose and create the transcoder itself. We can then play the requested video quality via other well-known RTMP names.

Significant knowledge has been gained during the creation of this thesis. Moreover, together with the supervisor, we have deployed this solution in production thanks to the infrastructure of a large CDN provider. The transcoder now powers hundreds of GPUs running inside tens of servers worldwide. When writing this thesis, the modified version of the application and libraries we implemented in this thesis handled thousands of video streams daily.

We have accomplished all goals that were set in the assignment. We have accomplished all goals that were set in the assignment. We implemented a proof of concept of NVIDIA accelerated video transcoder using NVENC, NVDEC, and CUDA. We wrote everything in the Rust programming language a used FFI to interact with the low-level libraries. We implemented support for load balancing across multiple GPUs in the system. We used H264 for the video encoding. We implemented RTMP support for publishing and playback. We set up the CI/CD with tests with corresponding documentation. As a bonus, the developed application is now running in production.

9.1 Plans for the future

In the near future, we plan to build extensive AV libraries that are optimized for live streaming in general. Moreover, we plan them to be written in Rust. The current implementation that we have does not support multiple video codecs or any audio codec. We plan to support audio resampling as well. Furthermore, we plan to implement more transport protocols that are used nowadays, like HLS, DASH, and WebRTC.

Bibliography

1. INGLIS, Andrew F.; LUTHER, Arch C. *Video Engineering*. 2nd. USA: McGraw-Hill, Inc., 1996. ISBN 0070317917.
2. STANDARDIZATION, International Organization for. *Coding of audio-visual objects — Part 10: Advanced video coding*. ISO/IEC 14496-10:2022. Vernier, Geneva, Switzerland: International Organization for Standardization, 2022. Available also from: `https://www.iso.org/standard/83529.html`.
3. STANDARDIZATION, International Organization for. *igh efficiency coding and media delivery in heterogeneous environments — Part 2: High efficiency video coding*. ISO/IEC 23008-2:2020. Vernier, Geneva, Switzerland: International Organization for Standardization, 2020. Available also from: `https://www.iso.org/standard/75484.html`.
4. APPLE. *HLS* [online]. [N.d.]. Available also from: `https://developer.apple.com/streaming/`.
5. STOCKHAMMER, Thomas. Dynamic Adaptive Streaming over HTTP –: Standards and Design Principles. In: San Jose, CA, USA: Association for Computing Machinery, 2011, pp. 133–144. MMSys '11. ISBN 9781450305181. Available from DOI: `10.1145/1943552.1943572`.
6. SOCOLOFSKY, T.; KALE, C. *A TCP/IP Tutorial* [Internet Requests for Comments]. RFC Editor, 1991-01. RFC. RFC Editor. Available also from: `https://www.rfc-editor.org/rfc/rfc1180.txt`.
7. WAITZMAN, D. *IP over Avian Carriers with Quality of Service* [Internet Requests for Comments]. RFC Editor, 1999-04. RFC. RFC Editor. Available also from: `https://www.rfc-editor.org/rfc/rfc2549.txt`.
8. KAUR, Amritpal; SINGH, Sarbjeet. A Survey of Streaming Protocols for Video Transmission. In: *Proceedings of the International Conference on Data Science, Machine Learning and Artificial Intelligence*. Windhoek, Namibia: Association for Computing Machinery, 2022, pp. 186–191. DSMLAI '21'. ISBN 9781450387637. Available from DOI: `10.1145/3484824.3484892`.
9. CLOUDFLARE. *What is adaptive bitrate streaming?* [Online]. [N.d.]. Available also from: `https://www.cloudflare.com/learning/video/what-is-adaptive-bitrate-streaming/`.
10. STREAMING LEARNING CENTER. *Adaptive Bitrate Streaming* [online]. [N.d.]. Available also from: `https://streaminglearningcenter.com/`.
11. BIANOR. *Adaptive Bitrate Streaming*. 2021. Available also from: `https://bianor.com/wp-content/uploads/2021/04/img_adaptive_bitrate_streaming.png`. [Online; accessed December 18, 2022].

12. YAQOOB, Abid; BI, Ting; MUNTEAN, Gabriel-Miro. A Survey on Adaptive 360° Video Streaming: Solutions, Challenges and Opportunities. *IEEE Communications Surveys & Tutorials*. 2020, vol. 22, no. 4, pp. 2801–2838. Available from DOI: 10.1109/COMST.2020.3006999.
13. R. PANTOS, Ed; APPLE, Inc; MAY, W. *HTTP Live Streaming* [Internet Requests for Comments]. RFC Editor, 2017-08. RFC. RFC Editor. Available also from: https://www.rfc-editor.org/rfc/rfc8216.txt.
14. APPLE. *HLS* [online]. [N.d.]. Available also from: https://developer.apple.com/documentation/http_live_streaming.
15. APPLE. *Live Playlist (Sliding Window) Construction* [online]. [N.d.]. Available also from: https://developer.apple.com/documentation/http_live_streaming/example_playlists_for_http_live_streaming/live_playlist_sliding_window_construction.
16. ADOBE. *RTMP* [online]. [N.d.]. Available also from: https://web.archive.org/web/20140821101147/https://www.adobe.com/devnet/rtmp.html.
17. NEUGEBAUER, Matthias. Nagare Media Ingest: A Server for Live CMAF Ingest Workflows. In: *Proceedings of the 13th ACM Multimedia Systems Conference*. Athlone, Ireland: Association for Computing Machinery, 2022, pp. 210–215. MMSys '22. ISBN 9781450392839. Available from DOI: 10.1145/3524273.3532888.
18. VIDEOLAN. *Demuxing* [online]. [N.d.]. Available also from: https://wiki.videolan.org/Demuxing/.
19. VIDEOLAN. *Muxing* [online]. [N.d.]. Available also from: https://wiki.videolan.org/Muxing/.
20. GAO, Mo; XIE, Weikai; LU, Chenping; SHEN, Ruimin. An Adobe Flash-Based Screencast Solution with Customized Screen Codec. In: *Proceedings of the Third International Conference on Internet Multimedia Computing and Service*. Chengdu, China: Association for Computing Machinery, 2011, pp. 187–192. ICIMCS '11. ISBN 9781450309189. Available from DOI: 10.1145/2043674.2043728.
21. FFMPEG. *Support H.265 over Adobe HTTP-FLV or RTMP* [online]. [N.d.]. Available also from: https://trac.ffmpeg.org/ticket/6389.
22. GOOGLE. *YouTube recommended upload encoding settings* [online]. [N.d.]. Available also from: https://support.google.com/youtube/answer/1722171/.
23. STANDARDIZATION, International Organization for. *Coding of audio-visual objects — Part 12: ISO base media file format*. ISO/IEC 14496-12:2022. Vernier, Geneva, Switzerland: International Organization for Standardization, 2022. Available also from: https://www.iso.org/standard/83102.html.
24. RASHEED, Zeeshan; BAIG, Waqar. Video codecs: An overview. *Journal of Multimedia*. 2010, vol. 5, no. 4, pp. 172–181.
25. RICHARDSON, Iain E. *Video codec design: developing image and video compression systems*. John Wiley & Sons, 2002.
26. BASLER. *Pixel format* [online]. [N.d.]. Available also from: https://docs.baslerweb.com/pixel-format.
27. PIXELINK. *Pixelink Knowledge Base* [online]. [N.d.]. Available also from: https://support.pixelink.com/support/solutions/articles/3000044298-pixel-format.
28. HIRSCH, Robert. *Exploring colour photography: A complete guide*. Laurence King, 2005.
29. INTEL. *Pixel and Planar Image Formats* [online]. [N.d.]. Available also from: https://www.intel.com/content/www/us/en/develop/documentation/ipp-dev-reference/top/volume-2-image-processing/image-color-conversion/pixel-and-planar-image-formats.html.

30. *Edge-Directed Interpolation* [online]. [N.d.]. Available also from: https://chiranjivi.tripod.com/EDITut.html.
31. RUST FOUNDATION. *Rust – A language empowering everyone to build reliable and efficient software.* [Online]. [N.d.]. Available also from: https://www.rust-lang.org/.
32. RUST FOUNDATION. *What Is Ownership?* [Online]. [N.d.]. Available also from: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.
33. RUST FOUNDATION. *References and Borrowing* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#references-and-borrowing.
34. RUST FOUNDATION. *Mutable References* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html#mutable-references.
35. NVIDIA CORPORATION. *Cuda zone* [online]. [N.d.]. Available also from: https://developer.nvidia.com/cuda-zone.
36. NVIDIA CORPORATION. *SDK Manager* [online]. [N.d.]. Available also from: https://developer.nvidia.com/drive/sdk-manager.
37. NVIDIA CORPORATION. *CUDA C++ Programming Guide* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
38. NVIDIA CORPORATION. *NVIDIA Video Codec SDK* [online]. [N.d.]. Available also from: https://developer.nvidia.com/nvidia-video-codec-sdk.
39. NVIDIA CORPORATION. *NVDEC video decoder API* [online]. [N.d.]. Available also from: https://docs.nvidia.com/video-technologies/video-codec-sdk/nvdec-video-decoder-api-prog-guide/index.html.
40. NVIDIA CORPORATION. *NVENC video encoding API* [online]. [N.d.]. Available also from: https://docs.nvidia.com/video-technologies/video-codec-sdk/nvenc-video-encoder-api-prog-guide/index.html.
41. NVIDIA CORPORATION. *NVIDIA Performance Primitives* [online]. [N.d.]. Available also from: https://developer.nvidia.com/npp.
42. NVIDIA CORPORATION. *Module* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#module.
43. NVIDIA CORPORATION. *Difference between the driver and runtime APIs* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-runtime-api/driver-vs-runtime-api.html#driver-vs-runtime-api.
44. NVIDIA CORPORATION. *Programming model* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.
45. ROBERT CROVELLA. *What kind of thread-safety is needed when using cuCtxSetCurrent* [online]. [N.d.]. Available also from: https://stackoverflow.com/a/48391221/7594900.
46. ARCHAEASOFTWARE. *CUDA streams and context* [online]. [N.d.]. Available also from: https://stackoverflow.com/a/6830506/7594900.
47. NVIDIA CORPORATION. *Context* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context.
48. NVIDIA CORPORATION. *Multiple contexts* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#multiple-contexts.
49. RUST FOUNDATION. *Module std::ffi* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/ffi/index.html.

50. RUST FOUNDATION. *Lifetime Elision* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/nomicon/lifetime-elision.html.
51. FRANCIS GAGNÉ. *What are the differences between a pointer and a reference in Rust* [online]. [N.d.]. Available also from: https://stackoverflow.com/a/62234967/7594900.
52. RUST FOUNDATION. *Unsafe Rust* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.
53. RUST FOUNDATION. *Foreign Function Interface* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/nomicon/ffi.html.
54. RUST FOUNDATION. *bindgen* [online]. [N.d.]. Available also from: https://github.com/rust-lang/rust-bindgen.
55. RUST FOUNDATION. *Build Scripts* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/cargo/reference/build-scripts.html.
56. GJENGSET, Jon. *Rust for rustaceans: idiomatic programming for experienced developers*. San Francisco: No Starch Press, 2022. ISBN 9781718501850;1718501854;
57. NVIDIA. *Initialization* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__INITIALIZE.html.
58. NVIDIA. *Device Management* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__DEVICE.html.
59. NVIDIA. *nvidia-smi* [online]. [N.d.]. Available also from: https://developer.nvidia.com/nvidia-system-management-interface.
60. NVIDIA. *Context Management* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__CTX.html.
61. RUST FOUNDATION. *Drop trait* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/ops/trait.Drop.html.
62. NVIDIA. *Memory Management* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/cuda-driver-api/group__CUDA__MEM.html.
63. NVIDIA. *Unified Memory for CUDA* [online]. [N.d.]. Available also from: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.
64. RUST FOUNDATION. *Considerations for unsafe code* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/boxed/index.html.
65. TOKIO TEAM. *Tokio* [online]. [N.d.]. Available also from: https://github.com/tokio-rs/tokio.
66. TOKIO TEAM. *Tokio channel* [online]. [N.d.]. Available also from: https://docs.rs/tokio/latest/tokio/sync/mpsc/fn.channel.html.
67. NICOKOCH. *Functional C macros are not expanded.* [Online]. [N.d.]. Available also from: https://github.com/rust-lang/rust-bindgen/issues/753.
68. AMANIEU, FAERN. *Crate lock_api* [online]. [N.d.]. Available also from: https://docs.rs/lock_api/latest/lock_api/.
69. NVIDIA. *NVIDIA Performance Primitives* [online]. [N.d.]. Available also from: https://developer.nvidia.com/npp.
70. DTOLNAY. *thiserror* [online]. [N.d.]. Available also from: https://docs.rs/thiserror/latest/thiserror.
71. RUST FOUNDATION. *Module std::error* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/error/index.html.

72. NVIDIA. *NVIDIA Performance Primitives* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/npp/group%5C_%5C_image%5C_%5C_color%5C_%5C_sampling%5C_%5C_format%5C_%5C_conversion.html#gae64ca2e03c4013142b5270b2dfec3d57.

73. NVIDIA. *NVIDIA Performance Primitives* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/npp/group%5C_%5C_image%5C_%5C_color%5C_%5C_sampling%5C_%5C_format%5C_%5C_conversion.html#ga6bb65404fa0de5e80462354441674f0f.

74. NVIDIA. *NVIDIA Performance Primitives* [online]. [N.d.]. Available also from: https://docs.nvidia.com/cuda/npp/group%5C_%5C_image%5C_%5C_resize%5C_%5C_square%5C_%5C_pixel.html#ga89f74fe8b5bc15df85241e534a12f173.

75. OBS. *OBS Studio* [online]. [N.d.]. Available also from: https://obsproject.com/.

76. FFMPEG. *FFMPEG* [online]. [N.d.]. Available also from: https://ffmpeg.org/.

77. KALLDREXX. *rust-media-libs* [online]. [N.d.]. Available also from: https://github.com/KallDrexx/rust-media-libs/.

78. TOKIO TEAM. *Tokio spawn* [online]. [N.d.]. Available also from: https://docs.rs/tokio/latest/tokio/fn.spawn.html.

79. KALLDREXX. *RTMP Handshake* [online]. [N.d.]. Available also from: https://docs.rs/rml_rtmp/latest/rml_rtmp/handshake/struct.Handshake.html.

80. KALLDREXX. *RTMP ServerSession* [online]. [N.d.]. Available also from: https://docs.rs/rml_rtmp/latest/rml_rtmp/sessions/struct.ServerSession.html.

81. RUST FOUNDATION. *Module std::collections* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/collections/struct.VecDeque.html.

82. RUST-BAKERY TEAM. *nom, eating data byte by byte* [online]. [N.d.]. Available also from: https://docs.rs/nom/latest/nom/.

83. RUST FOUNDATION. *Into* [online]. [N.d.]. Available also from: https://doc.rust-lang.org/std/convert/trait.Into.html.

84. GITHUB. *Landing page* [online]. [N.d.]. Available also from: https://github.com.

85. GITLAB. *Landing page* [online]. [N.d.]. Available also from: https://gitlab.com.

86. NVIDIA. *NVIDIA Container Toolkit* [online]. [N.d.]. Available also from: https://github.com/NVIDIA/nvidia-docker.

87. NVIDIA. *Environment variables (OCI spec)* [online]. [N.d.]. Available also from: https://github.com/nvidia/nvidia-container-runtime#environment-variables-oci-spec.

88. NVIDIA. *Constraints* [online]. [N.d.]. Available also from: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/user-guide.html#nvidia-require.

The contents of the included media

```
transcoding-service ..... Folder with compiled binary
src ..... Folder with latex source code
thesis.pdf ..... Compiled thesis into PDF
```