universität wien

# MASTERARBEIT | MASTER'S THESIS

Titel | Title

Evaluation of the Feasibility of Implementing Vulkan Video Extensions on CPUs

verfasst von | submitted by

Ing. Bernhard Clemens Schrenk BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

| | |
|---|---|
| Studienkennzahl lt. Studienblatt \| Degree programme code as it appears on the student record sheet: | UA 066 921 |
| Studienrichtung lt. Studienblatt \| Degree programme as it appears on the student record sheet: | Masterstudium Informatik |
| Betreut von \| Supervisor: | Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs |

# Acknowledgements

This thesis would not have been possible without the continuous support of my supervisor Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs throughout my studies, research projects and this work. His engaging courses, particularly on cloud gaming, provided the inspiration for the topic of this thesis. I am especially thankful for the opportunity to collaborate with him as a co-speaker at two Vulkanised conferences, where I had the chance to present my research on Vulkan Video.

I am also grateful to the open source community behind Mesa3D, whose contributions to Lavapipe provided the essential foundation for this work.

# Abstract

Vulkan is a modern, low-overhead, cross-platform graphics and compute API and an open standard primarily designed for use with graphics processing units (GPUs). Through its extensible architecture, it supports the integration of additional technologies. One of the most recent additions is the set of Vulkan Video Extensions, which enable hardware-accelerated video encoding and decoding for formats including H.264, H.265 and AV1. These extensions are essential in domains such as cloud gaming, CAD rendering, and remote visualization, where real-time video streaming and low latency are critical. However, many systems – including legacy hardware, embedded platforms or testing environments – lack access to dedicated video acceleration hardware, limiting the applicability of these extensions.

This thesis evaluates the feasibility of implementing Vulkan Video Extensions entirely in software on general-purpose CPUs. The goal is to extend Vulkan's usability to hardware-constrained environments by enabling software-based video encoding while maintaining conformance with the Vulkan specification. The implementation builds upon Mesa3D's Lavapipe, a software rasterizer that provides Vulkan API support via LLVM-based CPU rendering. In this work we add support for Vulkan Video Extensions to Lavapipe and integrate a lightweight H.264 encoder targeting the Constrained Baseline Profile, which is suitable for latency-sensitive applications. Three research questions guide the work: Can Vulkan Video Extensions be implemented on CPUs using Lavapipe? How can a video codec be interfaced with the Vulkan API? And does the implementation conform to the Vulkan and H.264 specifications?

Our implementation demonstrates that Vulkan Video Extensions can be successfully realized in software on CPUs, producing standards-compliant output as verified using the Vulkan Conformance Test Suite. Furthermore, we show that performance requirements of real-world applications can be met and analyze how performance depends on instruction sets and compiler optimizations. This thesis lays the groundwork for CPU-based Vulkan Video support and contributes to the broader effort of making Vulkan a platform-independent solution for video processing.

# Kurzfassung

Vulkan ist eine moderne, plattformübergreifende Grafik- und Berechnungs-API und ein offener Standard, der primär für den Einsatz mit Grafikbeschleunigern (GPUs) entwickelt wurde. Durch die erweiterbare Architektur ermöglicht der Vulkan-Standard die Integration zusätzlicher Technologien. Unter den jüngsten Erweiterungen sind die Vulkan Video Extensions, die hardwarebeschleunigte Videokodierung und -dekodierung für Formate wie H.264, H.265 und AV1 unterstützen. Diese Erweiterungen sind besonders in Bereichen wie Cloud-Gaming, CAD-Rendering und Remote-Visualisierung essenziell, in denen Echtzeit-Videostreaming und geringe Latenz entscheidend sind. Viele Systeme – darunter ältere Hardware, eingebettete Systeme oder Testumgebungen – verfügen jedoch nicht über dedizierte Video-Beschleunigerhardware, was die Anwendbarkeit dieser Erweiterungen einschränkt.

Diese Arbeit untersucht die Machbarkeit einer vollständigen Softwareimplementierung der Vulkan Video Extensions auf generischen Prozessoren (CPUs). Ziel ist es, die Nutzbarkeit von Vulkan auf Umgebungen ohne dedizierte Video-Hardware zu erweitern, indem softwarebasierte Videokodierung unter Einhaltung der Vulkan-Spezifikation ermöglicht wird. Die Implementierung basiert auf Lavapipe, einem Software-Rasterizer des Mesa3D Projektes, der Vulkan-API-Unterstützung über eine LLVM-basierte CPU-Renderpipeline bietet. In dieser Arbeit wird Lavapipe um Vulkan Video Extensions erweitert und ein H.264-Encoder integriert, der das auf latenzkritische Anwendungen optimierte Constrained Baseline Profile unterstützt. Drei Forschungsfragen leiten die Arbeit: Können Vulkan Video Extensions auf CPUs mithilfe von Lavapipe implementiert werden? Wie kann ein Videocodec mit der Vulkan-API verbunden werden? Und entspricht die Implementierung den Spezifikationen von Vulkan und H.264?

Die in dieser Arbeit entwickelte Lösung zeigt, dass Vulkan Video Extensions erfolgreich auf CPUs in Software umgesetzt werden können und dabei validiert durch die Vulkan Conformance Test Suite standardkonforme Ergebnisse liefern. Darüber hinaus wird gezeigt, dass die Leistungs-Anforderungen realer Anwendungen erfüllt werden können und wie die Leistung von der Wahl der Befehlssätze und der Compiler-Optimierungen abhängt. Diese Arbeit legt das Fundament für Vulkan-Video-Unterstützung auf CPUs und trägt zur Weiterentwicklung von Vulkan als plattformunabhängige Lösung für Videokodierung bei.

# Contents

# List of Figures

# Listings

# 1. Introduction

The Vulkan API is a programming interface for accessing modern graphic cards (GPUs) for graphics and compute tasks. It is an open standard [KVWG25] and is supported by a multitude of platforms like PCs, game consoles and mobiles [KGa]. To be extensible and be able to adapt to different feature sets of a target platform, Vulkan has introduced the concept of extensions. Extensions are optional parts of the API, which only need to be supported by target platforms that have the necessary hardware support. A rather new set of extensions are the Vulkan Video Extensions. They enable applications to use hardware accelerators for video compression and decompression often build into modern GPUs. The set contains extensions for different video compression standards.

The use of dedicated hardware for rendering graphics or compressing/decompressing video has advantages for performance and energy efficiency, but is not always feasible. There are needs for handling these tasks in software on general purpose CPUs. Those needs can arise in software testing scenarios for having a reference implementation or in production system due to unavailability of dedicated hardware. In the industry there is often the need for supporting legacy or embedded systems which are not providing all functionalities in hardware. The combination of rendering computer graphics and then compressing the result as video stream is often found in cloud gaming, but also in professional tools, like CAD or physics simulation software. Supporting those systems can be achieved by implementing multiple solutions in the application, but this would increase the complexity and reduce the gain achieved by using Vulkan as the one API that supports all target platforms. For those applications it makes sense to use software emulation running on general purpose CPUs which expose the same Vulkan API as GPUs do.

With a Vulkan driver emulating missing hardware, application developers can focus on developing the application. All the platform dependency will be handled transparently by loading the correct driver. For the task of rendering graphics on the CPU, software rasterizers are available. Even if some of them support the Vulkan API there is none which supports video extensions yet. As mentioned above the combination of rendering graphics and video compression has a wide field of applications. Therefore, it would be beneficial if there would be CPU based Vulkan API implementations supporting the Vulkan Video Extensions. Existing software implementations of the Vulkan API demonstrate that it is possible to provide Vulkan support by using the CPU only. We want to analyze if it is possible to provide also Vulkan Video Extensions on CPU only. Therefore, we focus on hypothesis H1:

- **H1:** It is possible to implement Vulkan Video Extensions completely on CPUs.

For evaluating if this is true we plan to answer three research questions. A full Vulkan

implementation is needed as base for our research. We therefore plan to build on top of Mesa's Lavapipe, which is an open-source state-of-the-art rasterizer with Vulkan API. The Vulkan Video Extensions are designed to support low-level video compression hardware. They are not designed to wrap a full video codec, but to provide the elements to build a codec. Therefore, we have to clarify if we can use an existing codec and connect it to our API implementation. Finally we need to clarify if the implementation and the connected codec are working correctly. Therefore, we formulate our three research questions as:

- **RQ1:** Can we implement Vulkan Video Extensions on top of Mesa's Lavapipe driver?
- **RQ2:** How can we connect a video codec with the API?
- **RQ3:** Does the implementation conform to the Vulkan specification and industry standards for video encoding?

To summarize, this thesis is motivated by the desire to extend the portability and applicability of Vulkan-based applications to CPU-only environments by exploring the feasibility of implementing the Vulkan Video Extensions in software. The growing importance of video processing in areas like cloud gaming, professional visualization, and embedded systems highlights the need for flexible, platform-independent solutions that can operate without dedicated video hardware. The central aim of this research is therefore to investigate whether it is technically and practically possible to emulate the Vulkan Video Extensions on general-purpose CPUs, without relying on GPU hardware acceleration. By using Mesa's Lavapipe driver as the base, we aim to develop a prototype that integrates a software video codec into the Vulkan API in a manner consistent with the official specifications. This provides insights not only into the technical challenges of such an implementation, but also into the degree of conformance that can be achieved relative to Vulkan's expectations for video functionality.

Through answering the posed research questions, we assess the feasibility, integration complexity, and standard compliance of this approach. This lays the groundwork for potential future development of CPU-based Vulkan video drivers and inform developers and researchers interested in platform-agnostic graphics and video APIs. To address the hypothesis and research questions, this work follows a structured approach:

- **First**, we analyze the Vulkan Video Extensions in detail, focusing on their design, intended use cases, and requirements for correct implementation.
- **Second**, we evaluate the architecture and capabilities of Mesa's Lavapipe driver, identifying extension points and areas that need modification or enhancement to support video encoding functionalities.
- **Third**, a prototype implementation is developed, connecting Lavapipe with a CPU-based video codec. This prototype serves as a proof of concept for supporting video compression through Vulkan on CPU-only systems.

- **Finally**, the prototype is evaluated for correctness, compliance and performance using conformance tests and application-level validation.

This structure ensures that each research question is systematically addressed and that the findings are grounded in practical implementation experience. In doing so, this thesis contributes to a better understanding of how Vulkan can be extended to serve a broader range of systems, including those that cannot rely on dedicated GPU resources.

# 2. Related Work

Video compression and decompression are computationally intensive algorithms. The computational load depends not only on the video resolution but also on the chosen video coding format. Modern formats such as H.264, H.265 and AV1 are composed out of techniques like discrete cosine transformation, spatial prediction, motion compensation and entropy encoding which must be calculated for each pixel of the video [ITU03, ITU13, AfOM19]. Applications using video compression and decompression must fulfill, apart from correct video encoding and decoding, additional requirements like bandwidth or latency constraints. For example, video players need to decompress videos at least as fast as they get played back. Video conferencing and cloud gaming applications need to ensure a low latency. This means video frames need to be captured, compressed, transferred and decompressed within a reasonable short time frame. The exact limits depend on the application and are for video conferencing about 150ms [RDM+09] and for cloud gaming about 100ms [KPS07].

Given those requirements and the fact that many techniques used in video compression can be parallelized and executed in pipelines [RGM06, SFLB07], video encoders and decoders (together called video codecs) are often implemented in hardware. On the one side special purpose hardware allows to reach those goals with less cost and better energy efficiency [NBL+14]. On the other side software codecs can be more versatile. They can be executed on general purpose hardware making them independent from the target system. Furthermore, they can implement better heuristics. Heuristics are used while compressing videos for tasks like motion estimation. Improved heuristics in the encoder can achieve better video quality with less bandwidth without the need to replace the decoder [BS23]. This implies that no single codec is universally superior. Even for one video coding format, several implementations (codecs) coexist. This multitude of coding formats and codecs makes it harder for applications to use them. A way to tackle this complexity is to build, according to the facade pattern, APIs combining access to a collection of codecs, as shown in Figure 2.1. Such APIs allows an application to support multiple video coding formats on different hardware or software, without the added complexity of interfacing with each codec. On one side there are high level APIs, like on libraries as FFmpeg. Those are good for quick evaluation. On the other side there are low level APIs, which are also necessary to implement high level libraries. They provide better possibilities to influence the encoding process as stated by Nvidia in "FFmpeg should be used for evaluation or quick integration, but it may not provide control over every encoder parameter" [Nvib]. Additionally, only low level APIs allow tight integration with the graphic functionality of hardware accelerators.

All currently standardized coding formats are working on similar entities. They subdivide video streams into frames. A frame is one image of the video at a given point

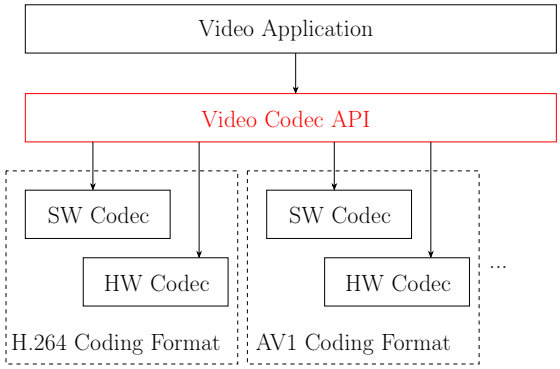


Figure 2.1.: The Video Codec API is located between the application and the codecs. It is the interface abstraction layer for codecs implemented using different technologies and for different coding formats.

in time in the stream. Some coding formats, like H.264, support interlaced video. In interlaced video the frame is divided into two fields, one containing the even, the other the odd lines of the frame. The encoder can choose to encode each field as a separate picture or encode the whole frame as one picture. In progressive video each frame is always one picture and so the terms can be used interchangeable. During encoding of a picture, previously encoded pictures need to be available as references for motion detection and compensation. It needs to be noted that the encoding order of pictures does not need to be the display order. This allows referencing pictures from before and after (seen in display order) of the currently encoded picture. Therefore, the codec must either reorder the pictures or the API needs to give the application control over the display and encode order. Apart from using reference pictures each picture gets encoded separately and therefore can also be decoded separately. Some formats subdivide the picture into smaller areas called slices or tiles. Slices can also be compressed separately as they do not depend on each other. Encoded slices can be seen as independent packets within the bitstream of the encoded video. The next subdivision used in current video coding formats are macroblocks (or coding tree units). Due to spatial prediction they are dependent on each other and multiple macroblocks are combined in one entropy coded stream [ITU03]. Therefore, macroblocks are not a good choice as entity on the interface level. Even if this stayed the same for decades [ISO93, ITU20], it must be noted that the requirements for Video Codec APIs can change in the future. Current research shows that advances in machine learning allow to represent complete videos as neural networks [CHW+21]. This can require new ways to interface with such codecs in the future. Considering only the currently standardized coding formats allows to combine their codecs behind one API based on:

- codec (configuration) parameters,
- picture slices representing decoded video data,
- data packets representing encoded video data and

- reference picture buffers.

Different coding formats use different terms to describe those components. We stick with this terminology for better comparability. Even if it would be possible to have one API supporting everything, there are different APIs supporting subsets of hardware, software, coding formats and other parameters. For every application it is therefore necessary to evaluate which API can be used or if multiple APIs need to be combined or extended. Especially risks like performance trade-offs and vendor lock-in must be considered and possibly avoided.

We want to give an overview over the existing APIs and find possible room for improvement to make them more versatile. As nearly all applications can benefit from hardware accelerator support, at least for energy efficiency, we only consider APIs which allow access to hardware codecs. For evaluating Video Codec APIs we need to define criteria relevant for business and academic use cases. The most important criteria are based on compatibility, availability and interoperability. The compatibility with standardized and widely used video coding formats is important for the interoperability with other video senders and receivers. Different video coding formats have different features and have different computational and storage requirements. Additionally there are different licensing requirements due to software patents in some countries, which could hinder someone in using a specific format [LGK23]. Therefore, multiple coding formats are in use. A good Video Codec API should support the major ones. In this work we consider all video coding formats which are in use by video discs (starting with DVD), television broadcaster (terrestrial and satellite) and streaming platforms. Therefore, we evaluate the APIs for support of the formats:

- MPEG-2,
- MPEG-4 Part 2,
- H.264 / MPEG-4 AVC,
- H.265 / MPEG-H HEVC,
- AV1 and
- H.266 / VVC.

The most important format is H.264 with over 80% of developers and industry experts recently surveyed using it in production. Followed by its successor H.265 with about 50% usage. All other formats have less than 10% usage in production. The survey also shows that there is a large interest (about 30%) in adopting AV1 this year [Bit23]. The evaluation showed that no hardware encoders are yet supporting H.266, therefore we do not mention it in the detailed evaluation.

Many APIs are limited to support only a subset of platforms. Applications planned to be deployed on multiple platforms or in a heterogeneous environment need to consider using Codec APIs available on all platforms or alternatively use multiple APIs in parallel.

This increases the necessary effort. Therefore, APIs supporting multiple platforms have an advantage. Some APIs only support hardware accelerators from one vendor. This limits applications to hardware of this specific vendor. Vendor-neutral APIs on the other side are more versatile, but still can only support hardware if the vendor supports it. An API supporting multiple hardware vendors has similar advantages like APIs supporting multiple platforms. It decreases the implementation and maintenance effort on the application side.

Supporting multiple hardware accelerators is important if applications are planned to be deployed on a wide field of hardware. To be even more versatile it is important to have fallback options to software codecs. Those codecs do not need special hardware and can run on general purpose CPUs. Even if this has disadvantages regarding performance it is often better or sometimes even necessary to have at least minimum support for such systems. An example for such software fallbacks is scientific (physics) simulation software. Software (CPU) based rendering is used as fallback for visualization tasks to support hardware which is designed for fast simulations, but without dedicated graphics hardware for visualization [TNI+24]. For such use cases graphic APIs often have fallbacks to software renderer. Examples are WARP for Direct3D [Micb], Mesa's LLVMpipe for OpenGL and Lavapipe for Vulkan [Pau]. For encoding the visualization results as video (either for storage or for streaming) it is advantageous to have similar software fallbacks in Video Codec APIs instead of requiring the application code to have multiple code paths to support software codecs. APIs need language bindings to be available for development in specific programming languages. If they are not written in the same language or do not have bindings to the language of the application code, additional effort is necessary for bridging to a different language. Therefore, supporting multiple or at least the most common languages is preferable.

Video encoding and decoding tasks often come with the requirement to integrate them with graphic tasks. The simplest form is the need to present the decoded video either directly on the screen or as a texture on a 3D surface in a game. Other integration requirements can arise from applications like cloud gaming where graphics rendering and video encoding must go hand in hand. The rendering task is providing the frames which then must get encoded as video stream. For such combined applications the video API needs interoperability with the graphics API. A basic integration allows sharing memory buffers. This enables to use zero-copy techniques for video frames. More advanced variants support tight coupling of the synchronization primitives. Offloading the synchronization between video and graphics tasks to the hardware accelerator is important for allowing the hardware to better utilize its components by using fine-grained scheduling. On the example with the video texture from above, we can show how tight API integration can reduce latency. Using fine-grained scheduling the graphics pipeline and the video pipeline can run in parallel for the same frame until the texture unit needs the decoded video data. Without shared synchronization primitives the CPU needs to wait until the video frame is decoded, before it can queue the frame rendering on the graphics hardware.

We are limiting the comparison to APIs which are available on personal computers, high-performance computers and mobile platforms. Apart from these platforms there are

special purpose platforms like embedded systems for set-top boxes or Internet of Things devices. They often use specialized hardware tightly coupled with the software stack, tailored for specific applications like video playback or surveillance. By contrast, personal computers, high-performance computers, and mobile platforms operate in heterogeneous environments, requiring more versatile APIs that support a range of hardware accelerators and software codecs. This survey focuses on these platforms to address the needs of developers working on multi-platform applications, such as cloud gaming, video streaming, or video conferencing, where flexibility, cross-platform compatibility, and wide codec support are critical. Additionally we only consider the most recent APIs which support both video encoding and decoding and exclude legacy and deprecated APIs. The use of legacy APIs in new developments is constrained by their lack of support for modern technologies and the discontinuation of vendor maintenance. In this work we introduce each API separately, followed by Table 2.1 summarizing all APIs and their evaluation according our criteria.

Vendors of hardware accelerator usually provide their own APIs. In general they provide the best support for the specific hardware, but are not compatible with other hardware. They also do not provide any fallback to software codecs. We list the vendor APIs in alphabetical order. As already stated we only list vendors providing APIs on desktop and mobile platforms. We exclude vendors which are only available on embedded devices. AMD Advanced Media Framework (AMF) is the API provided by AMD for accessing the Video Codec hardware integrated in AMD GPUs. Depending on the hardware it provides access to all current video coding formats. It has integration into all current graphics APIs with the possibility to share data buffers, but still is a separate API using its own API objects. The API is accessible on Linux and Windows platforms using the C and C++ programming languages. It only supports AMD hardware containing Video Coding Engine or Video Core Next accelerators. There is no fallback to a software codec included [AMD]. The Apple Core Video framework exclusively supports Apple hardware. It supports both the desktop and mobile platforms of Apple. This API supports all current coding formats. Support for AV1 was added recently [Bit23]. The supported programming languages are Objective-C and Swift. The API is tightly integrated with Apple's media framework AVFoundation and over that with the Metal graphics API [App]. Intel Video Processing Library (Intel VPL) is supporting video accelerator in integrated and dedicated Intel GPUs. This library provides an API for accessing codecs for MPEG-2, H.264, H.265 and AV1. VPL is available on Linux and Windows and can be programmed using C. The API supports interoperability with Direct3D, VA-API, OpenCL and Vulkan. The OpenCL interoperability allows indirectly to connect to the OpenGL graphics API [Inta]. The Nvida Video Codec SDK gives access the NVENC and NVDEC video hardware accelerator. This API is limited to Nvidia hardware. All current video coding formats are supported except MPEG-4 Part 2. The SDK is provided on Linux and Windows and can be programmed using C. Interoperability with graphics APIs is possible, but needs one indirection via CUDA (the Nvidia computing API). Therefore, three technologies are involved in such a scenario [Nvia].

In contrast to hardware vendor specific APIs also operating system vendors provide

APIs which abstract the hardware and provide functionality for accessing Video Codecs. Those are supporting codecs of multiple hardware vendors. Android MediaCodec provides access to video codecs on the Android operating system for mobile devices. It supports all current video coding formats and can be programmed in Java and Kotlin. It is only available on Android platforms. One advantage is that MediaCodec automatically provides software codec fallbacks if hardware acceleration is not available. MediaCodec can share buffers with OpenGL and Vulkan [Gooa]. Direct3D 12 Video is the most recent evolution of DirectX Video Acceleration (DXVA) provided by Microsoft for video encoding and decoding within their Direct3D 12 API. With current drivers this API supports acceleration hardware from AMD, Intel and Nvidia. For decoding the API supports all current video coding formats. For encoding it is limited to H.264, H.265 and AV1. The API supports the C++ programming language and is tightly integrated with the Direct3D graphics API including synchronization. The API is available on Windows 10, with encoding support starting with Windows 11 [Mica]. Video Acceleration API (VA-API) is mainly targeting the Linux platform, but also supporting other Unix-like operating systems. It supports video decoding and encoding with hardware acceleration. It supports Intel hardware [Intc] and over the Mesa drivers also Nvidia and AMD hardware. For Linux installations running within the Windows Subsystem for Linux (WSL) the Mesa driver can translate VA-API to Direct3D 12 Video [Pau]. VA-API supports all important video coding formats and can be programmed using the C programming language. There are interoperability functions with OpenGL via OpenCL [Intb].

Interfaces which are developed independently from operating system and hardware vendors can be the most versatile ones, but independent APIs often do not get the necessary support from platform and hardware vendors to provide the necessary drivers. This causes that independent APIs are rare. Vulkan Video Extensions are part of the Vulkan Graphics and Compute API specification. It is provided by the Khronos Group, an industry consortium of the major hardware and software vendors. The Vulkan Video Extensions are API extensions providing video encode and decode tightly integrated in the Vulkan API. The supported video coding formats are H.264, H.265 and AV1. The Vulkan graphics API is in general supported on Android, Apple (iOS, macOS), Linux and Windows. But drivers supporting the Vulkan Video Extensions are only available on Linux and Windows for AMD, Intel and Nvidia hardware [Wil]. The API can be used in C and C++. As it is an extension to the Vulkan API it is tightly integrated with the graphics and compute parts of it. It shares device access and has a common architecture for memory sharing and synchronization [KVWG25]. Even if there are software fallbacks for the Vulkan Graphics API for systems without graphics accelerators there is no support in those fallbacks for the Vulkan Video Extensions [Pau].

We showed that even if limiting the scope by not considering embedded systems and legacy APIs, there is not the one Video Codec API available, which can fulfill all requirements and is available on all platforms. An API which is versatile and supports all major platforms and major hardware architectures would be advantageous for application developers. Without such an API application developers need to implement multiple code paths and address multiple APIs, to achieve a wide platform support. There are efforts

Table 2.1.: Comparison of all evaluated Video Codec APIs according our criteria

| API | Coding Formats[1] | Platforms | Hardware Supp. | Software Codec | Prog. Languages | Graphics Interop.[2] |
|---|---|---|---|---|---|---|
| **AMD AMF** | all | Linux, Win | AMD | no | C, C++ | yes (buffer) |
| **Apple Core V.** | all | Apple | Apple | no | Obj-C, Swift | Metal (full) |
| **Intel VPL** | most | Linux, Win | Intel | no | C | yes (buffer) |
| **Nvidia Video C.** | most | Linux, Win | Nvidia | no | C | yes (buffer) |
| **Android MediaC.** | all | Android | Android Mobiles | yes | Java, Kotlin | yes (buffer) |
| **Direct3D 12 V.** | most | Windows | AMD, Intel, Nvidia | no | C++ | Direct3D (full) |
| **VA-API** | all | Linux | AMD, Intel, Nvidia | no | C | yes (buffer) |
| **Vulkan Video** | most | Linux, Win | AMD, Intel, Nvidia | no | C, C++ | Vulkan (full) |

[1] most: at least all recent formats (H.264, H.265 and AV1)
[2] buffer: allows sharing memory buffers; full: allows sharing buffers and synchronization primitives

to standardize independent APIs and with Vulkan Video Extensions there is an API which has already the widest platform and hardware support. But still it has limitations especially in the area of platform support and software fallbacks. There are multiple open questions if limitations are of technical nature or commercial interest or if they just were not addressed yet. Some of them were already investigated, like the incompatibility with Android as stated by a Google employee in the issue tracker with "The Vulkan Video extensions are incompatible with Android's graphics driver architecture and multimedia security architecture" [Goob].

We focus on the open question of a software fallback for Vulkan Video Extensions and evaluate if it is feasible to implement Vulkan Video Extensions on CPUs. This is significantly influenced by the existence of CPU based Vulkan drivers, commonly known as software rasterizers. Their capabilities in implementing the Vulkan API are crucial for our work. Software rasterizer implement the complete rendering pipeline in software without relying on GPU hardware acceleration. Software rasterizers can be used for research [Mir13, dBGR+06]. Some are providing also a Vulkan API, like Vulkan-Sim which allows to analyze each rendering cycle in detail and was designed for ray tracing research [SCL+22]. Software rasterizer are also used for educational purposes as they can easily be analyzed and adapted by students [FWW13]. As our goal is to provide a solution which can be used in real-world scenarios, we focus on rasterizer which were developed for GPU replacement as their main intend and which are providing recent Vulkan support. Two projects are currently providing Vulkan 1.3 support. Google's SwiftShader project implements an high-performance software rasterizer with Vulkan and OpenGL interface [Gooc]. Mesa's Lavapipe project has a similar goal and already reaches Vulkan 1.4 support while being about 42% faster in popular benchmarks compared with SwiftShader [Fry25]. Lavapipe is based on Mesa3D, which not only supports software rasterization, but also hardware accelerators. It has a solid base library Gallium3D, which already provides a good framework for extensions [Pau] and which contains basic video support routines due to their usage in hardware accelerators. This makes it a good candidate to be used as base for our work. In the field of general GPU emulation, much research is done in command conversion between different processor architectures, including intermediate representations and caching [CGXJ14, MID+23]. Command conversion is an important part of the shader execution and just-in-time compilation done by Lavapipe. GPU emulation is heavily used in the area of high performance computing, like simulators for CUDA and OpenCL workloads [BYF+09, PHO+15, UJM+12] to gain more insights into the execution, allowing better optimizations. While those are mainly for compute tasks, these can still provide valuable input for emulating graphic pipelines.

One of the goals of our work is to interface with an existing software video codec. We need to modify it, so that we can a adapt its API to match our requirements. Therefore, we focus on open-source H.264 codecs. One of the most advanced encoder for H.264 is x264, which supports nearly all H.264 features [x26]. Integrating x264 can be hard due to its complexity and also its licensing requirements. Cisco open sourced its H.264 encoder OpenH264. It is optimized for video streaming applications and can freely be integrated in any software [Cis]. A very simple, but easily adaptable encoder is minih264. It is freely

available as one source file on GitHub and can also be integrated in every software. It brings support for SSE2 and ARM Neon instruction sets [lie]. Parallelization is a key aspect in developing competitive codecs. Using SIMD instruction sets allow data-level parallelism on instruction level. Many building blocks of modern codecs, like interpolation filter, cost function and transform function can benefit from SIMD instruction sets, like SSE2 and ARM Neon [AHSH14]. More sophisticated parallelization instruction sets like AVX-512 should be enabled conditionally depending on their availability, but also on bitrate/resolution presets and the number of parallel streams, to maximize the throughput of the whole processor. Those instruction sets can cause reduced processor frequencies especially on server processors and should only be used if the amount of computation per pixel is high and the memory bandwidth can be fully utilized. This could be further improved by monitoring CPU frequencies and dynamically switching instruction sets [TMM+]. On top of data-level parallelism, thread-level parallelism is used to utilize multiple processor cores. The high compression ratio of modern video coding standards with intra- and inter-frame encoding rely on reuse of already encoded data both in spatial and in time dimension. These data dependencies need to be analyzed before parallelization can be applied. Decomposing a frame into slices can break those dependencies and allow to achieve higher speedups on multi-core CPUs [Her11], but with negative effects on the compression ratio. An alternative is parallelizing the encoding of macroblocks and considering multi frame parallelism with the downside of higher latency [GTC03]. Another optimization applied in video encoders is algorithmic pruning. Early decisions for encoding modes, like intra/inter mode or for used blocksizes reduces the number of modes which need to be evaluated making them more efficient for general purpose CPUs, which can not run all algorithm paths in parallel. Fast intra mode selection algorithms can reduce the encoding time substantially with only a little loss of bit rate and visual quality [RVTS09]. These optimizations of codecs are not only useful for increasing the performance, but also for reducing the energy consumption, especially in mobile applications. Energy-efficient video encoding algorithms can also help to reduce cost and safe energy in server applications. Modeling the energy consumptions of those algorithms can help to reach this goal [RKH24]. FFmpeg [Bel] and GStreamer [fre] are libraries building on top of these highly optimized codecs and wrapping them. They allow easy integration of audio and video codecs into applications without knowing all the details of the different video coding standards. Internally FFmpeg and GStreamer can use hardware acceleration via APIs like Vulkan Video.

Video codec testing and result validation consists of both checking the correct functionality of the codec itself and the validation that the resulting bitstream can be correctly decoded and is equivalent to the original video. This validation must consider that lossy compression standards can not regenerate the exact video, therefore a a reconstruction quality metric must be calculated [RLCW00]. Apart from peak signal-noise-ratio (PSNR) many more metrics were developed and tested. Most of them rely on perception-based models considering the specifics of human vision [WBSS04, ITU23, WM08, SQ15]. Apart from validating the output, video codecs can also be checked for correct functionality. An emerging technique for finding faults in software is fuzzing. The tool TwinFuzz can

be used to test video hardware acceleration stacks for faults and vulnerabilities. It is based on a differential oracle utilizing FFmpeg as white-box proxy to allow testing of black-box hardware implementations of video codecs [LCW+25]. Due to its design it also allows testing of the software path and therefore testing of unmodified open-source video codecs and video stacks. Due to the black-box design of hardware and the effort needed to adapt and design new hardware, software stacks, like FFmpeg for video en- and decoding or Lavapipe for software rasterization are often used to research and develop new functionalities and extensions or for developing test cases for upcoming extensions. As an example Lavapipe was used to build the test cases in the Vulkan Conformance Test Suite for the `VK_EXT_device_generated_commands` extension before it was implemented by hardware vendors, enabling them to test and validate the upcoming extension already before release [Fry25]. Lavapipe was also used in the past for emulating hardware implementations on the CPU for research and compatibility purposes, like the Vulkan ray tracing extensions [Seu23]. This work shows that Lavapipe is capable of being used for emulation research and therefore makes it an ideal test-bed for our work.

# 3. Background

To implement and evaluate extensions for a Vulkan driver many base technologies are necessary. Apart from the standards (H.264 and Vulkan) which we want to support we also describe LLVM and Mesa3D's LLVMpipe/Lavapipe as these are the software components we use as base for our implementation and evaluation. Additionally, we show in this chapter how Vulkan drivers like Lavapipe get loaded and integrated with the operating system, so that it is possible to develop and evaluate those drivers without touching operating system kernel code. On top of these base technologies, which are necessary to reach basic functionality, we also explore the possibilities of optimizing H.264 on CPUs using SIMD (Single Instruction, Multiple Data) instruction sets, as this is the base for reaching competitive performance compared with hardware accelerators.

To validate the output of our implementation and the video encoder we introduce a conformance test suite and a technique to calculate a metric of image similarity to be able to compare lossy compressed images with their source. Furthermore, we discuss which Vulkan Video features and API extensions are necessary for a successful implementation. We finish the chapter with an introduction into the `minih264` video encoder API and our redesign to fulfill our requirements. These technologies and software components provide the base for this work, allowing us to show that Vulkan Video can be implemented on CPUs reaching both standard conformance and satisfactory performance.

## 3.1. Advanced Video Coding (AVC, H.264)

Advanced Video Coding, standardized by the International Telecommunication Union (ITU) as H.264 [ITU03], is the most important video coding standard [Bit23]. It was also the first standard which was supported by Vulkan Video Extensions [KVWG25]. H.264 standardizes a big set of video coding features together with an exact bitstream definition. Codecs implemented on simpler hardware could not support all features. Therefore, the standard defines profiles. Each profile is a subset of features. A codec which supports a specific profile only needs to support that subset of features. Videos can then be encoded with a specific target profile to support a specific subset of codecs and video players. This allows to develop simple codecs for applications with limited requirements and advanced codecs which support more features. Those features can therefore be subdivided in base features, which always need to be supported, features for more advanced applications (e.g. monochrome, higher color resolution, bit error redundancy) and features which allow higher compression ratios with similar quality, but needing more computing power. For our evaluation we plan to support the minimum profile, which is the Constrained Baseline Profile (CBP). For proofing our statement we do not need to implement higher

profiles. Features of higher profiles can easily be added later by using more advanced codecs without changing the interface substantially. The features we need to support are:

- Videos with 8 bits per sample, encoded in YCbCr with 4:2:0 subsampling,
- Intra-picture prediction (spatial prediction) with variable block sizes,
- Inter-picture prediction (with multiple reference pictures),
- Quarter-pixel motion compensation,
- Integer discrete cosine transform (iDCT),
- Hadamard transform (for DC components),
- Adaptive quantization,
- In-loop deblocking filter,
- Context-adaptive variable-length entropy coding (CAVLC) and
- Exponential-golomb coding.

The Constrained Baseline Profile is often used in latency-sensitive applications like video conferencing as it does not encode B(idirectional predicted)-frames. As B-frames allow referencing future frames, the order of frames in the data stream does not correspond to the display order. The decoder has to wait before it can present a frame. Therefore, B-frames increase the latency. Furthermore, CBP only supports progressive video, therefore we do not need to support splitting a frame into separate fields and can define that one frame is one picture.

Modern CPUs are equipped with a number of architectural features that are relevant for efficient software-based video encoding and decoding. Unlike GPUs, which are optimized for massively parallel workloads, CPUs prioritize general-purpose execution and low-latency control flow. However, several CPU features can still be leveraged to accelerate video processing, particularly for formats like H.264. A key feature in this context are SIMD instructions. They allow the CPU to perform the same operation on multiple data points simultaneously. This is especially useful in video processing, where many operations such as pixel transformations, motion compensation, or entropy encoding can be vectorized. Early SIMD instruction sets include Intel's MMX and SSE (Streaming SIMD Extensions), with SSE2 being supported on all x86-64 CPUs. Later extensions such as AVX, AVX2, and AVX-512 offer wider registers and more complex instructions, but are only available on more recent CPU generations. While SSE and AVX provide instructions for vectorizing floating point calculations, SSE2 and AVX2 are designed for integer operations. Many video encoders make heavy use of SIMD instructions to optimize performance-critical paths. Especially in the context of H.264, which is designed for integer arithmetic, SSE2 can be perfectly used for:

- **Motion Estimation:** During motion estimation, the encoder searches for blocks in reference frames that closely match the current macroblock. This involves computing the sum of squared differences across blocks of pixels. This operation is often approximated by calculating the sum-of-absolute-differences (SAD). SSE2 provides packed integer instructions (e.g., `movdqu`, `paddq`) that allow processing multiple pixel values in parallel. Furthermore, SSE2 includes a dedicated instruction for calculating the sum of absolute differences (`psadbw`). This reduces the number of loop iterations and memory accesses.

- **Transformation:** The Discrete Cosine Transform (DCT) is used to convert spatial-domain pixel data into frequency-domain coefficients, which are more suitable for compression. H.264 uses an approximated version of DCT called integer DCT. SSE2 packed instructions can be used to implement fast integer DCT algorithms by parallelizing the computation over multiple blocks using packed integer instructions.

- **Pixel Filtering and Deblocking:** H.264 includes an in-loop deblocking filter to reduce compression artifacts at block boundaries. This filter involves conditional operations and averaging across neighboring pixels. SSE2 supports 128-bit wide logical operations (e.g. `pxor`), packed comparison (e.g. greater than `pcmpgtb`), packed minimum, packed maximum and packed averaging instructions (`pavgw`) that can be used to parallelize filtering logic efficiently.

- **Quantization and Inverse Quantization:** Quantization reduces the precision of DCT coefficients to improve compression. SSE2 allows batch quantization of coefficients using instructions such as `psraw` (arithmetic shift) and `pmullw`, `pmulhw` (low/high word multiplication), which are useful for fixed-point scaling.

- **Entropy Coding Preprocessing:** Before final entropy coding (CABAC or CAVLC), coefficients and motion vectors are rearranged and transformed. SSE2 is often used for zig-zag scanning (using shuffle operations `pshuflw`, `pshufhw`) and zero-run detection using comparison and mask instructions (e.g. `pmovmskb`).

SSE2 code is often written using compiler intrinsics, which expose the instructions directly in C/C++ code. For example, the `_mm_add_epi16` intrinsic adds 8 packed 16-bit integers across two 128-bit registers, equivalent to operating on 8 pixels in parallel. It gets compiled into the SSE2 instruction `paddw`. This allows developers to retain control over low-level data manipulation while still benefiting from the compiler's register allocation and instruction scheduling. On one side SIMD instructions significantly reduce the number of instructions and memory accesses required, which is crucial for real-time encoding scenarios. On the other side encoders which use specialized instructions are not easily portable between CPU architectures. Even if similar SIMD instructions are available on other platforms (e.g. ARM provides the NEON instruction set), core parts of the encoder need to be implemented for each architecture separately. In addition to SIMD, CPU cache hierarchies play an important role. Modern CPUs typically include a multi-level cache structure (L1, L2, and L3). Efficient video codecs aim to keep frequently accessed

data—such as motion vectors, frame references, and quantization tables—within the L1 or L2 caches to avoid latency from main memory access. Poor cache locality can quickly become a bottleneck, especially when processing high-resolution frames.

Multithreading is another important optimization technique. H.264 can be parallelized on the slice level with the downside of decreasing the compression ratio as motion vectors and intra-prediction can not cross slice borders [ITU03]. Parallelization on the frame level is a further optimization technique, which can be exploited if latency is not important, especially in offline video encoding. These architectural features, SIMD, cache hierarchies, and multithreading, enable CPUs to perform video encoding at competitive speeds, especially when GPU-based solutions are not available or practical. They form the technical basis for evaluating whether Vulkan Video Extensions can be implemented efficiently in a CPU-only context.

## 3.2. Mesa3D/Gallium3D Architecture

Mesa3D started as an open source implementation of OpenGL. The history page in the project's documentation [Pau] says that the initial Mesa3D version did all rendering on the CPU. It took some years until Mesa3D started to support hardware-accelerated rendering on GPUs and became the standard user-space component of the Linux Direct Rendering Infrastructure (DRI). Mesa3D is accessing the GPU hardware via kernel-space drivers, mainly via the Direct Rendering Manager (DRM) of the Linux kernel. Vulkan support was later added to Mesa3D and the library was ported to multiple platforms. Depending on the platform capabilities it can render in software or translate to other graphics APIs (e.g. on Windows to Direct3D). For each API/hardware/platform combination a separate Mesa3D driver was implemented. As there is much common code in those drivers (e.g. OpenGL state tracking, platform interface support, ...), Gallium3D was introduced as an abstraction layer. Most modern Mesa3D drivers are based on the Gallium3D abstraction layer. The Gallium3D infrastructure gives Mesa3D a modern modularized software development approach and allows code reuse and faster driver development. Gallium3D's main structure is documented in the Mesa3D source tree documentation [Pau] and consists of the key components:

- **drivers**: The (GPU dependent) rendering driver
- **frontends**: The graphics API frontend
- **winsys**: The platform dependent windowing system integration
- **auxiliary**: Common code (state tracker, llvm, shader compiler, ...)

Mesa3D contains multiple software rasterizer, which allow rendering on the CPU instead of using GPUs. While *softpipe* is optimized for portability, *LLVMpipe* is optimized for speed. LLVMpipe uses the LLVM compiler infrastructure to generate native CPU instructions for the rasterization pipeline and shader code. This is done on-the-fly during runtime [Pau]. During pipeline creation LLVMpipe generates the exact pipeline commands and provides

them in LLVM intermediate representation (IR), a language internally used by LLVM to represent parsed code. The shader compiler uses an LLVM frontend to translate the shader code into LLVM IR. Finally the code parts get linked and LLVM can run optimizations over the whole pipeline before the LLVM backend generates native CPU instructions. On execution of the pipeline, LLVMpipe calls the dynamically generated and optimized code. LLVMpipe has an OpenGL interface and provides a good alternative if GPU based rendering is not available.

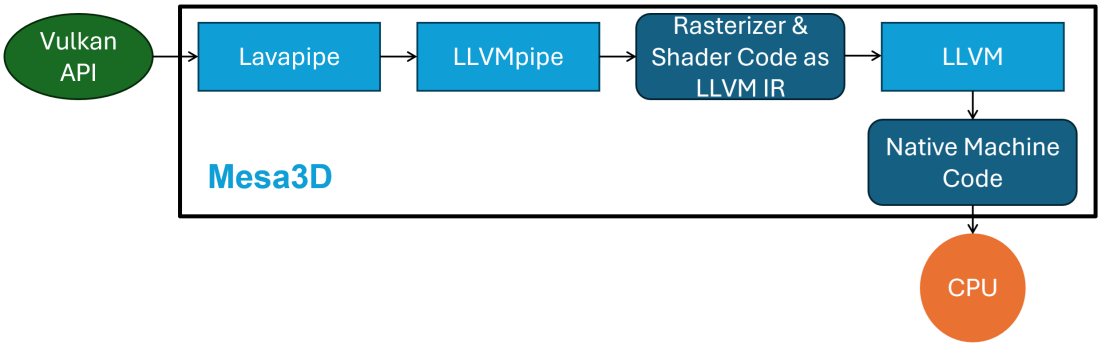


Figure 3.1.: Architecture of the software rasterizer LLVMpipe with Vulkan API Lavapipe in Mesa3D

Lavapipe is a Gallium3D frontend to LLVMpipe adding Vulkan API support to the LLVMpipe software rasterizer. This allows Mesa3D to be used as Vulkan driver without the need for specialized hardware. Lavapipe is certified for Vulkan 1.3 conformance, but already supports all Vulkan 1.4 requirements [Fry25]. Lavapipe can be used as fallback if there is no GPU available or a necessary extension is not supported. Apart from this, Lavapipe can be used as reference platform for testing graphics code especially in continuous integration systems and for researching and developing new Vulkan extensions. Figure 3.1 shows the architecture of Lavapipe and LLVMpipe in Mesa3D. Lavapipe provides the Vulkan API and uses LLMVpipe internally. LLVMpipe is responsible for generating the rasterizer and shader code as LLVM intermediate representation. Finally LLVM generates native machine code which then gets executed on the CPU. LLVM is a collection of modular and reusable compiler and toolchain technologies. It is widely used as infrastructure for building compilers, JIT (Just-In-Time) compilation and execution engines, static analyzers, and code generators. The project is open source and maintained by the LLVM Foundation. Even if the C/C++ compiler Clang is the best known LLVM project, LLVM at its core is not a compiler in the traditional sense, but a framework for building compilers. It provides a flexible Intermediate Representation (IR), a rich set of optimization passes, and code generation backends for a variety of hardware targets, both for CPUs and GPUs. The LLVM IR is a low-level, strongly typed, SSA (Static Single Assignment)-based representation that abstracts away most of the details of the target architecture, allowing frontends and backends to evolve independently [Fou]. LLVM

consists of the the main components:

- **Frontend:** Translates source code into LLVM IR. For example, Clang is the C/C++/Objective-C frontend of LLVM. Other languages like Rust (via rustc), Swift, and Julia also use LLVM-based frontends. Each frontend performs parsing, semantic analysis, and some early transformations before emitting IR.

- **IR and Optimizer:** The IR is the central data structure used by LLVM. It exists in both a textual and a binary form (bitcode), and supports multiple optimization passes such as dead code elimination, inlining, loop unrolling, constant folding, and vectorization. Optimizations can be applied at compile-time, link-time, or runtime (in JIT scenarios).

- **Backend (CodeGen):** Translates LLVM IR into machine code for a specific architecture, such as x86-64, ARM, RISC-V, WebAssembly or even GPU instructions. The backend includes instruction selection, register allocation, and scheduling. Backends are modular, so support for new targets can be added without modifying the frontend.

- **JIT Compilation:** LLVM includes JIT execution engines such as MCJIT and ORC (On-Request Compilation). These can be used to compile and run code at runtime, allowing dynamic code generation. This is useful in applications like language runtimes (e.g. Julia, LuaJIT), GPU drivers, and emulators. JIT compilation and execution is used in optimized software rasterizers.

- **Tooling Infrastructure:** LLVM also includes reusable tools like `opt` (optimizer), `llc` (IR to native code), `clang-format` (code formatter), `lld` (linker) and `lldb` (debugger). These tools are built on the same libraries and can be used standalone or embedded in larger toolchains.

- **Runtime Libraries:** LLVM provides many runtime libraries, like compiler runtime and standard C and C++ libraries, but also specialized libraries for OpenCL and OpenMP.

One of the key advantages of LLVM is its modularity. Frontends and backends can be developed independently, and many components (such as the IR optimizer) are shared across languages and targets. This architecture enables code reuse, consistent optimization quality, and easier maintenance compared to monolithic compilers. LLVM's IR is also human-readable, which simplifies debugging and experimentation. Another major strength of LLVM is its support for aggressive, architecture-aware optimizations. For CPU-intensive workloads such as graphics rendering or scientific computing, LLVM can generate highly optimized machine code by exploiting features like SIMD instructions and specific microarchitectural properties. Projects like LLVMpipe use this capability to dynamically generate fast CPU code for graphics shaders, while retaining flexibility and portability.

Despite its advantages, LLVM also has limitations. The size and complexity of the infrastructure can be a challenge for smaller projects. The compilation overhead of LLVM-based systems (especially JITs) can be significant at runtime, which is not ideal for latency-sensitive applications. Additionally, integrating LLVM into custom toolchains requires familiarity with its internal APIs, which are extensive and can change between versions. LLVM is used in a broad range of applications, including general-purpose programming languages (e.g., Rust, Swift), GPU drivers (e.g., Mesa3D's shader compiler), embedded systems, static analyzers (e.g., Clang Static Analyzer), and binary translation tools. In graphics software like LLVMpipe, it enables runtime compilation of shader code to optimized native code, allowing shaders to run efficiently on general-purpose CPUs. This makes LLVM a powerful foundation for implementing software rendering and video processing on platforms without GPU support.

## 3.3. Operating System Integration of Vulkan Drivers

The integration of Vulkan drivers with the operating system is done through a combination of kernel interfaces, window system protocols, and user-space libraries. While the Vulkan API is standardized and cross-platform, its actual implementation depends on OS-specific components that manage hardware access, memory, and display surfaces. On Linux systems, Vulkan drivers typically rely on the Direct Rendering Manager (DRM) subsystem in the Linux kernel. DRM is responsible for providing low-level access to GPU resources, including memory allocation, command submission, and display control via kernel mode-setting. Vulkan drivers access DRM via the user-space `libdrm` library, which wraps `ioctl` calls to the kernel and provides abstractions for buffer and synchronization management. On top of this, the Mesa3D graphics stack provides an implementation framework for various Vulkan drivers, including RADV (for AMD GPUs) and Lavapipe (for software rendering on the CPU). The Mesa winsys layer abstracts kernel interfaces, so that drivers like Lavapipe can run independently of DRM, using system memory instead of GPU memory. Window system integration is handled via WSI (Window System Integration) extensions, which are part of the Vulkan specification. On Linux, this includes support for both X11 and Wayland via extensions such as `VK_KHR_xlib_surface` and `VK_KHR_wayland_surface`. These allow applications to create presentation surfaces, which the Vulkan implementation uses to present rendered frames to the display. Even software drivers like Lavapipe can use these extensions, since they simply provide a mechanism to copy rendered images from CPU memory into the compositor's display buffer. On Windows, Vulkan drivers interface with the Windows Display Driver Model (WDDM), the kernel interface used for GPU management. Drivers are provided as Installable Client Drivers (ICDs), which are dynamically loaded DLLs. Surface presentation uses the Win32 API. Vulkan integrates with the DirectX Graphics Infrastructure (DXGI) and the Graphics Device Interface (GDI) to manage swapchains and window surfaces through the `VK_KHR_win32_surface` extension. Lavapipe is also available on Windows, implemented as a software-only ICD, and performs rendering entirely on the CPU without relying on WDDM or GPU hardware.

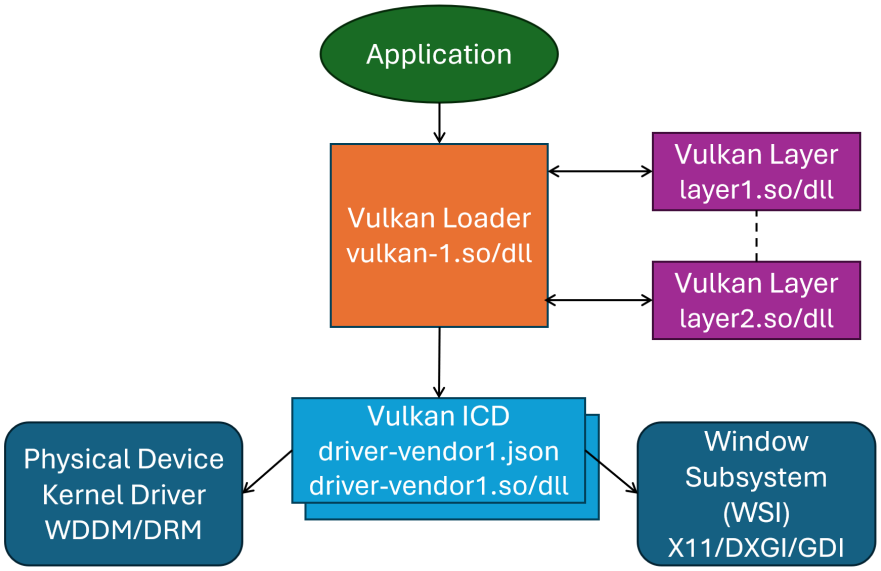


Figure 3.2.: The Vulkan loader & driver architecture and its operating system integration

A key part of Vulkan's portability and extensibility is the Vulkan loader. The loader is a central user-space library as shown in Figure 3.2. Applications dynamically link with the Vulkan Loader, which must be, depending on the platform, installed as `vulkan-1.dll` or `vulkan-1.so`. The loader then discovers and loads the appropriate Vulkan driver at runtime. Drivers are distributed as shared libraries (`.so` files) on Linux and `.dll` files on Windows. They follow a standardized interface defined by the Vulkan specification. The loader locates these drivers by reading configuration files or environment variables. On Linux, environment variables such as `VK_ICD_FILENAMES` can be used to override the default search path and point directly to a specific ICD JSON file, which in turn defines the path to the actual shared object. This allows users to easily test or install Vulkan drivers without modifying system directories. Similarly, on Windows, the registry is used for ICD discovery by default, but the `VK_ICD_FILENAMES` environment variable can also be used for custom driver paths during development or testing. In addition to ICDs, the Vulkan loader also supports optional layer drivers. Layers are shared libraries that can intercept and modify Vulkan API calls, typically used for debugging, validation, or performance tracing. For example, the Vulkan validation layers provided by the LunarG SDK are commonly used during development to check for API misuse. Layers are loaded based on configuration files or environment variables like `VK_LAYER_PATH`, and can be enabled or disabled at runtime without modifying the application or the core driver. Together, these components form a modular Vulkan architecture where the loader, ICDs, and layers work together to provide flexibility and portability. For drivers like Lavapipe, this means they can be deployed simply by dropping a shared library and corresponding JSON manifest into a directory and updating an environment variable. No kernel driver is required, and full Vulkan conformance can be achieved entirely in user space.

## 3.4. Vulkan Conformance Test Suite

Khronos Group is providing the Vulkan Conformance Test Suite (CTS) to allow driver vendors to validate the correct implementations of the Vulkan API. The CTS is an open-source project containing test cases for the Vulkan specification and all ratified extensions. Hardware and driver vendors must proof passing all tests before getting certified as a conformant Vulkan implementation. This ensures consistency within the Vulkan ecosystem [KGb]. The CTS contains test cases for many aspects of the Vulkan API, like graphics pipelines, shader operations, memory management, synchronization and recently also for Vulkan Video Extensions. Especially in the area of Vulkan Video the CTS checks aspects like:

- video decoding and encoding operations conform to Vulkan specifications.
- synchronization between video operations and other components work as expected.
- memory management works as specified.
- image format support is compatible to the specification.
- encoding and decoding produces correct images equivalent to the original input.

Peak Signal-to-Noise Ratio (PSNR) is a standard metric for measuring the quality of reconstructed images and videos after a lossy compression/decompression process [RLCW00]. It can be defined via the mean squared error (MSE) and the maximum possible pixel value (MAX) and expressed as decibels (dB):

$$PSNR = 10 \cdot \log_{10} \left( \frac{MAX^2}{MSE} \right)$$

The mean squared error can be calculated using the pixel values of the original ($I_{orig}$) and the reconstructed ($I_{rec}$) image of size $m \times n$:

$$MSE = \frac{1}{m \cdot n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} (I_{orig}(i,j) - I_{rec}(i,j))^2$$

The Vulkan CTS uses PSNR to validate the correct encoding and decoding of videos. Each image gets analyzed and compared with a threshold:

- **>30 dB** means the image was correctly encoded and decoded. The test **succeeded**.
- **10-30 dB** issues a warning, but the test is still counted as **successful**.
- **≤10 dB** means there is a critical problem. The test **failed**.

This technique employed in the Vulkan CTS allows us to validate the results of our implementation by comparing the video encoded using our Vulkan Video extension with the original content. With this we can show that the videos contain the same content and therefore are equivalent, even if they are, due to the lossy compression, not exact copies.

## 3.5. Required Vulkan Extensions

To show that Vulkan Video Extensions can be generally supported on CPUs, we show that video encoding can be supported on CPUs. Showing that video encoding works is sufficient to also show that video decoding can be supported as:

1. the API for decode is using the same data structures as the encode API and
2. decode is a sub-problem of encode because reference picture reconstruction is already required for inter-frame compression.

As H.264 is the most widely used video coding standard and also one of the possible standards supported by Vulkan Video Extensions, we decided to support H.264 video encoding. H.264 has many optional features. The set of necessary features is chosen by the supported profile. As already described the minimum set of H.264 features is contained in the Constrained Baseline Profile [ITU03]. Therefore, we must at least support the CBP. For supporting it we mainly need to provide I and P frame encoding and therefore also full reference picture handling. This allows us to utilize all basic Vulkan Video API functionalities and proof that they can be supported on CPUs. The Vulkan Video Extensions allow the application to query for supported features. Beside advertising the supported video coding standard and profile, there are also a couple of further features (mainly affecting the API usage) which can be supported. For our proof it is important to support all features, which are required by the Vulkan standard, but we can leave out all optional features. One large, but optional feature is rate control. The standard does not require to support any automatic rate control (constant/variable bitrate control modes). Therefore, we implement in this work only the disabled (manual) and default rate control modes. Some other features are given as a set of features, where at least one must be supported. One of the most important feature sets falling into this category are the supported image formats. The Vulkan driver specifies which image formats it supports. It must support at least one. The application needs to adapt to this and provide all images in one of the supported formats. Also for decoding the application needs to choose the output format from the list of supported formats. H.264 CBP is limited to encode videos with 8 bits per sample and YCbCr 4:2:0 subsampling. Therefore, we must support at least one Vulkan image format with these parameters. Based on this requirement and the availability of codecs with algorithms for separate planes we support images using the 3-plane format `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM`. Due to limitations in the Vulkan Conformance Test Suite (CTS) we also need to support 2 plane input images in the format `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM`. We handle 2 plane input images by converting them to a 3 plane image before encoding. For reference pictures we only support the 3 plane format, which is conforming to the Vulkan standard. We added support for this operation mode to the CTS and provided the improvement to the CTS maintainers.

The possible parameter space of video coding standards is huge. Decoders must support any combination of parameters of the supported profiles. This is possible as the standards specify in detail how video decoding must be done. Encoding is not completely

standardized. A standard conformant encoder must generate a standard conformant bitstream, but it is not specified how this must be done. Therefore, the encoder has more freedom in choosing the parameters and the subset of supported functionality. This gives us the possibility to choose an encoder which does not support all features. It only needs the features necessary to generate a bitstream, which a standard conformant decoder with CBP support can correctly decode. This is enough for a standard conformant encoder and is therefore enough to proof our hypothesis. Vulkan Video Extensions give the application much power in controlling the video coding parameters. This allows to optimize the video stream for each application individually. But this also means that implementations of Vulkan Video Extensions need to implement support for a wide range of parameters. As this is not possible, especially with limited hardware codecs, Vulkan Video Extensions allow the driver to override parameters chosen by the application. This is done in a handshake where the application suggests a parameter set and the driver returns either that it fulfilled the request or returns a modified parameter set. We use this feature to limit the parameters to those our codec supports, enabling us to use a codec with some limitations.

Extending Vulkan with new features can be done by releasing a new core version of the Vulkan standard. As standardizing a new version is a time-consuming effort, Vulkan has introduced the concept of Vulkan Extensions. This allows to add new features to Vulkan without changing the core standard [KVWG25]. It also allows to add features which are specific for one platform or vendor. A prefix in the extension name shows if the extension was ratified by the Khronos Group (`VK_KHR`) or if the extension is a vendor extension (e.g. `VK_AMD`, `VK_NV`), which do not need to be ratified [LH25]. Support for those extensions by the platform or driver is optional. An application must query the driver to check if an extension is supported before enabling it. Depending on the added functionality, extensions are classified as instance or device extensions. Most extensions are device extensions, but there are some important examples of instance extensions, like the surface extensions, which deal with the integration with the presentation and window system. Extensions are the usual way to start adding new features to the Vulkan standard. If an extension gets widely supported by vendors and is useful for many applications, it can be promoted to a core version of the standard. A promoted extension needs to be supported by all implementations which report support for that specific or a higher core version. It is important to note, that a supported extension does not always mean that all (or any) features of the extension are supported. To enable extensions Vulkan additionally requires to check and enable feature flags.

Vulkan provides a full set of Video Codec APIs for encoding and decoding video. Those APIs are standardized as a set of optional extensions to the Vulkan standard. These are official and ratified extensions with each extension providing a subset of the available APIs and together building a dependency tree as shown in Figure 3.3. The base extension `VK_KHR_video_queue` provides the central command queue for video operations together with the basic infrastructure, like video session, session parameter, reference picture and memory handling. Specific features which are just applicable to decoding or encoding videos are contained in the two extensions:

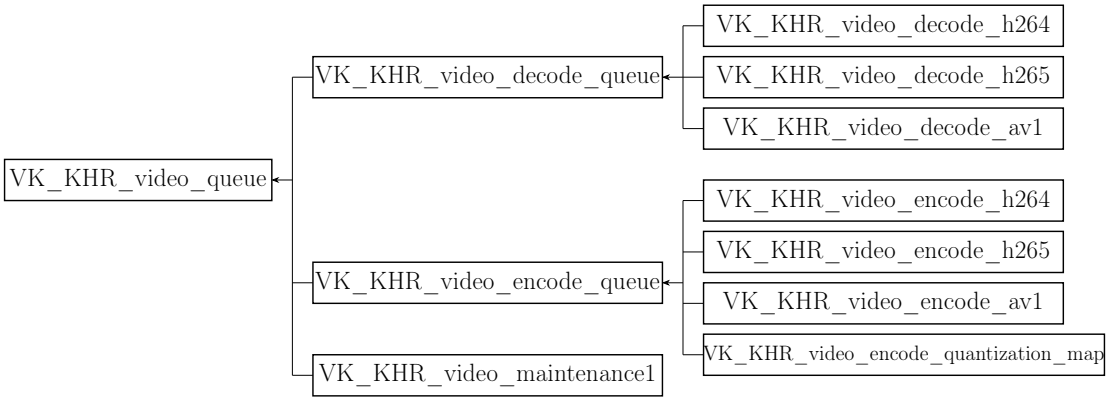


Figure 3.3.: The Vulkan Video Extensions and their dependencies

- `VK_KHR_video_decode_queue` for general decoding features and
- `VK_KHR_video_encode_queue` for general encoding features.

Build on top of them are extensions for supporting specific coding standards:

- `VK_KHR_video_decode_h264` for decoding H.264 videos.
- `VK_KHR_video_decode_h265` for decoding H.265 videos.
- `VK_KHR_video_decode_av1` for decoding AV1 videos.
- `VK_KHR_video_encode_h264` for encoding H.264 videos.
- `VK_KHR_video_encode_h265` for encoding H.265 videos.
- `VK_KHR_video_encode_av1` for encoding AV1 videos.

General features for all video codec operations or improvements of the API for easier usage by applications are added by the extensions:

- `VK_KHR_video_maintenance1` adds missing features, which are too small to qualify for their own extensions.
- `VK_KHR_video_encode_quantization_map` adds a feature to allow specifying different quantization parameters for different image regions while encoding.

A specific hardware/driver combination is allowed to support a subset of these extensions depending on its own capabilities. Furthermore, the extensions themselves provide even more fine grained capability queries. The application needs to query the supported capabilities before it can choose the exact coding standard and the coding parameters for an operation. For a subset of parameters the Vulkan driver can also overwrite values chosen by the application. This allows to support a wide range of existing video codec hardware and enables to design Vulkan Video Extensions with just a subset of the possible

capabilities [KVWG25]. For implementing Vulkan Video Extensions on a CPU we need to support a set of Vulkan API extensions. Based on the features we want to support, we can now define the set of Vulkan extensions we need to implement and describe in detail which methods we need to support and how they interact. We implement them as an extension to Lavapipe as shown in Figure 3.4. To be able to support a wide field of hardware, Vulkan Video Extensions as a low-level, GPU-centric approach split up the video encode support into many extensions. For our CPU based implementation of Video Encode we focus on the Vulkan Video Extensions `VK_KHR_video_queue`, `VK_KHR_video_encode_queue` and `VK_KHR_video_encode_h264`. The first extension `VK_KHR_video_queue` defines the basic infrastructure for video encoding and decoding in Vulkan. For this it introduces the basic concepts, structures and operations for:

- **Video Profiles**: Defining the mode (encode or decode), coding standard, coding profile and high-level parameters (e.g. interlaced, luma and chroma bit depths).
- **Feature Queries**: Extending the existing Vulkan feature queries with video specific queries for codec and image format features.
- **Video Sessions**: Encapsulating a video codec instance.
- **Video Session Parameters**: Holding codec configuration parameters.
- **Memory Management**: Allocating and providing memory for codec internal use.
- **Video Commands**: Defining commands to be executed for starting and ending a video coding scope and controlling it (reset and rate control).
- **Video Queue**: Defining a Vulkan queue supporting the execution of video commands.

This is the basic infrastructure of video operations and integration with the Vukan API. It does not specify any operations or parameters specific to one coding standard or specific to encoding or decoding. This is done by the more specific extensions. As this is the common base for all further operations, we need to implement all the listed functionality of this extension to correctly support Vulkan Video Extensions. Furthermore, we need to adapt it to work with a software encoder rather than a hardware-accelerated codec. `VK_KHR_video_encode_queue` adds all functionality necessary for supporting video encode operations without going into detail about the coding standard. It mainly consists of the parts:

- **Encode Command**: Defines the video encode command to be executed in the command queue.
- **Session Parameter Feedback**: Supports the handshake for finding supported session parameters.
- **Session Parameter Encoding**: Provides bitstream header generation out of the session parameters.

- **Rate & Quality Control**: Enables control of the bitrate of the video stream.
- **Query Pool**: Allows to retrieving encode status information as an extension to Vulkan query pools.
- **Buffer & Image Usage Extensions**: Extends buffer and image usage flags for input, reference picture and output usage.
- **Encode Pipeline Stage**: Integrates video encode in the Vulkan pipeline concept.

These additions provide video encode specific controls and allow to integrate video encoding into the core Vulkan concepts and pipeline. We need to implement all these functions to be conforming to the specification. As already discussed before the only optional parts are the advanced rate control modes. `VK_KHR_video_encode_h264` is specifying all coding standard dependent functionalities. In our case these are the configuration parameters for H.264 encoding. This extension does not add new commands, but it adds 26 data structures and a couple enums and bit-fields for specifying all the possible parameters for H.264 video encoding from the stream down to the slice level. A big part of these also control the reference picture management. We only need to support those data fields in these structures which are required by the CBP of the H.264 standard. Using the session parameter override feature of Vulkan we can limit the supported parameter set even further. The Vulkan Video Extensions specify many additions to data structures and enumerations of the core Vulkan API. This means that we not only need to add new functions, but also extend the existing functions of the Vulkan API implementation in Lavapipe. The affected core functionalities are:

- **Feature Queries**: Adding more device features and image format details.
- **Device Creation**: Adding additional queue types.
- **Query Pools**: Adding additional options for query pools.

Figure 3.4 shows this as extended Lavapipe. We describe the changes in detail in the next chapter. The final design is very strictly guided by the Vulkan Specification. We need to implement all the required functionality to pass the conformance tests. We do not implement optional features, which are not necessary to proof our hypothesis, but, as seen above, most parts of the API are required. In this work we implement the functions, structures and enumerations exactly as specified in the Vulkan specification [KVWG25]. We do not have dedicated GPU memory, therefore our API looks like a graphics system with unified memory architecture to the application. This is completely conforming to the Vulkan standard and already used by Lavapipe. Applications can benefit from this by reducing memory copy operations. This design allows us to implement the Vulkan Video Extensions on top of Mesa's Lavapipe driver, which answers our research question RQ1.

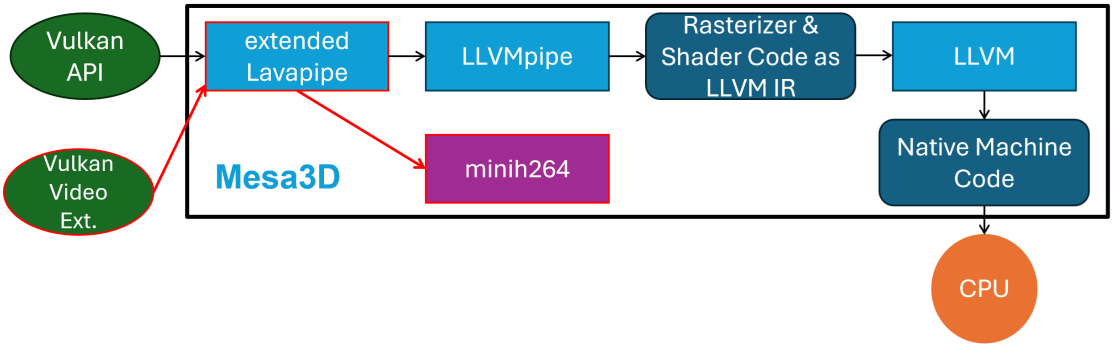


Figure 3.4.: Necessary extensions (in red) for Vulkan Video in Mesa3D

## 3.6. Codec Interface Design

After extending Lavapipe with all necessary interfaces required by the Vulkan Video Extensions, we can get input frames from the application and can provide back buffers which should contain the encoded bitstream. We can also handle encoding parameters, reference pictures and synchronization. Finally, a codec is needed to encode the frames and receive the video bitstream. We plan to integrate a third-party codec via an interface as shown in Figure 3.4. We design this codec interface based on the functionality needed to provide to the application and the functions required by the Vulkan Video Extensions API. We plan to use the encoder of a third-party codec. The main requirement we have is that we need to be able to adapt the interface to make it compatible with Vulkan Video Extensions. Especially the explicit control of encoding parameters and reference picture handling by the application needs to be supported. To be able to adapt the codec to our needs, we need access to the source code of the codec. Additionally, we prefer codecs with a license compatible with Mesa3D, so that we can distribute the result. As we only need to support a minimal subset of the H.264 features, we can use a codec with a small code base allowing easy modifications and adaptions.

Based on these requirements we selected to use minih264 as third-party encoder. The minih264 project is only providing H.264 encoding with support for I and P frames. This is exactly matching our requirements. The encoder is implemented in a single C source file and the Creative Commons license allows us to modify and integrate it with our project. Beside the C implementation, minih264 also provides optimized assembler code for SSE(x86) and NEON(ARM) architectures. We use the implementation and the interface of minih264 and adapt both to our needs. [lie]. We need to add the functionality required by Vulkan Video Extensions. Even if the interface is taken from minih264 we adapt it in a way so that other codecs can be used as well. This interface design allows us to connect a codec with the Vulkan Video Extensions API as required by our research question RQ2.

An application starts video encoding by creating a Vulkan video session and binding memory to it. During the video session creation all parameters affecting possible memory

usage must be specified. These are essentially the video coding standard, the profile, the color format and the maximum frame size. Using these parameters the codec can calculate the necessary needed memory. To map this functionality to the codec interface, we specify two methods. One for retrieving the necessary memory amount and one for initializing the codec. As parameters we only need the maximum frame size, as we already restricted our implementation to H.264 CBP. These parameters are given in a struct of type `H264E_create_param_t` as seen in Listing 3.1.

```
typedef struct H264E_create_param_tag
{
    // Frame size: must be multiple of 2 (due to 4:2:0 encoding)
    int width;
    int height;
} H264E_create_param_t;

// Types for persistent and scratch memory
typedef struct H264E_persist_tag H264E_persist_t;
typedef struct H264E_scratch_tag H264E_scratch_t;

// Get necessary memory size for persistent and scratch memory
int H264E_sizeof(const H264E_create_param_t* par, int* sizeof_persist,
    int* sizeof_scratch);

// Initialize codec
int H264E_init(H264E_persist_t* enc, const H264E_create_param_t* opt);
```

Listing 3.1: Codec Interface - Initialization

The codec can specify two different kinds of memory. The first one called `persist`, which must be retained during the whole video session and the second one called `scratch`, which is only necessary during the encode operation. The persistent memory needs to be provided during codec initialization as parameter `enc`. As we do not have dedicated GPU memory we can use host memory for all allocations, so we use a short-cut here. Host memory can be allocated by the Vulkan driver without support by the application. Therefore, we report zero memory usage on the Vulkan API and the application does not need to bind GPU memory explicitly. We internally reserve all necessary codec memory as host memory during video session creation.

Our implementation only needs to support progressive video (1 frame equals 1 picture) and only supports one slice per picture. Therefore, one frame gets encoded as one slice. This allows us to implement the Vulkan slice encoding using the codec's frame encoding function. Our codec needs to support 3-plane images with separate planes for YCbCr, often also abbreviated as YUV in codec interfaces. We use a data format for transporting those images, which also allows to specify a stride for each data plane. The stride value gives the distance between two rows in the frame data in bytes. This value allows us to skip data in each row, which does not get encoded. We use this for data alignment of each row and for adding a border around the image data, which does not get encoded. The border is important for the codec as it simplifies the motion detection algorithm on the frame edges. Listing 3.2 shows how the minimum border is defined in its header file together with the structure for transporting the frame data. The size of each plane

memory buffer is determined by the height, the stride and the border padding. The U and V planes are each a fourth of the size of the Y plane, due to the 4:2:0 encoding.

```
/**
*   Border padding size
*/
#define H264E_BORDER_PADDING        16

typedef struct H264E_io_yuv_tag
{
    // Pointers to 3 pixel planes of YUV image
    unsigned char* yuv[3];
    // Stride for each image plane
    int stride[3];
} H264E_io_yuv_t;
```

Listing 3.2: Codec Interface - Frame Data Structure

To be able to encode a frame we also need to define additional to the frame data the frame parameters. These parameters mainly define the type of the encoded frame (key, I or P frame) and the encoding quality. Additionally we also need to specify which stream (SPS) and picture parameter set (PPS) gets used for this specific encoded frame. The SPS and PPS referenced here must be inserted into the encoded data stream before referencing them. According the Vulkan Video standard, this has to be done by the application, but the standard also requires to provide an API for generating the SPS and PPS data to support the application. The structure for defining the parameters is specified in Listing 3.3.

```
/**
*   Frame type definitions
*/
#define H264E_FRAME_TYPE_KEY  6 // Random access point: IDR frame
#define H264E_FRAME_TYPE_I    5 // Intra frame
#define H264E_FRAME_TYPE_P    2 // Predictive frame

/**
*   Speed preset index
*/
#define H264E_SPEED_SLOWEST   0  // All coding tools, including denoise
#define H264E_SPEED_BALANCED  5
#define H264E_SPEED_FASTEST   10 // Minimum tools enabled

typedef struct H264E_run_param_tag
{
    // Variable, indicating speed/quality tradeoff, see H264E_SPEED_*
    // 0 means best quality
    int encode_speed;

    // Frame type override: one of H264E_FRAME_TYPE_* values
    int frame_type;

    // Quantizer value, 10 indicates good quality
    // range: [10; 51]
```

```
26      int qp;
27
28      // PPS quantizer value (as specified in active PPS)
29      int pps_qp;
30
31      // PPS id for the next slice (including SPS id)
32      int pps_id;
33  } H264E_run_param_t;
```

Listing 3.3: Codec Interface - Frame Encoding Parameters

After defining all necessary structures we can define the frame encoding function as shown in Listing 3.4. To work correctly the function needs the initialized encoder memory and the uninitialized scratch memory. Further we need to provide the frame parameters and the frame data. The third set of parameters consist of the frame data of the reference picture (only necessary for P frames) and the output buffer for the newly generated reference frame (for the next P frame). The last set of parameters provide the buffer and buffer size for the generated video bitstream for this frame.

```
1  int H264E_encode(H264E_persist_t* enc, H264E_scratch_t* scratch,
2                   const H264E_run_param_t* opt, H264E_io_yuv_t* in,
3                   H264E_io_yuv_t* ref, H264E_io_yuv_t* dec,
4                   unsigned char** coded_data, int* sizeof_coded_data);
```

Listing 3.4: Codec Interface - Frame Encoding Function

All functions of the API return an error code to communicate possible problems with the given parameters. The list of possible errors is shown in Listing 3.5. In case the codec returns one of these errors, our implementation must map them to the corresponding Vulkan error codes.

```
1  /**
2  *   API return error codes
3  */
4  #define H264E_STATUS_SUCCESS                0
5  #define H264E_STATUS_BAD_ARGUMENT           1
6  #define H264E_STATUS_BAD_PARAMETER          2
7  #define H264E_STATUS_SIZE_NOT_MULTIPLE_2    5
8  #define H264E_STATUS_BAD_LUMA_ALIGN         6
9  #define H264E_STATUS_BAD_LUMA_STRIDE        7
10 #define H264E_STATUS_BAD_CHROMA_ALIGN       8
11 #define H264E_STATUS_BAD_CHROMA_STRIDE      9
```

Listing 3.5: Codec Interface - Error Codes

# 4. Implementation

With the design in mind we can delve into the concrete implementation of the Vulkan Video Extensions encode functionality within Mesa3D's Lavapipe driver and its integration with the minih264 software encoder. We need to implement new Vulkan feature queries, extend the command buffer handling, especially the code generator, add new query pool types, adapt the image buffer handling and finally implement video session and command support. After introducing our development environment we describe each extension in more details. Mesa3D and especially Lavapipe is developed to run on multiple platforms. Our development focuses on the two platforms Windows and Linux for:

- **primary development** using Windows 11 (x64) and Visual Studio 2022.
- **integration testing** on Windows Subsystem for Linux (WSL) with Docker.
- **performance testing** using Ubuntu 25.04 with GCC and Clang.
- **additional testing** on Raspberry Pi 5 (aarch64) with Debian Linux.

Beside Mesa3D, which we want to extend, we also need an application using the Vulkan Video Extensions for video encoding to be able to test our extended driver. We already developed a simple video encoder application `vulkan-video-encode-simple`, which is a perfect match for this use case. During development we could only test it with Nvidia GPUs, but it should be compatible with any Vulkan Video implementation, also with our CPU only implementation. The encoder application simulates the bare minimum necessary for rendering and encoding a graphics scene into a H.264 video, as it is used in applications like cloud gaming. It operates headless, which means that it does not present the output live. The recorded video can be viewed afterwards with a compatible video viewer, like VNC or ffplay. The headless mode and the independency from user input allows to render and encode as fast as the hardware allows, independent from real-time constraints. The main building blocks of our encoder sample application are:

- setting up Vulkan Graphics, Compute and Video queues and all necessary resources.
- rendering frames with a moving triangle via a Vulkan graphics pipeline, using offscreen rendering (no window, headless mode).
- converting the rendered RGB images into the YCbCr color format and adapting the memory layout to 2 or 3 planes using a compute shader.
- invoking Vulkan Video Encode commands to encode the converted images and generating an H.264 elementary stream.

- copying the bitstream to CPU buffers and writing it into a file, playable with standard tools.

This application contains all necessary components needed for using Vulkan Video for encode operations and is therefore a good base for developing a Vulkan Video capable driver. To be able to build Mesa3D and our encoder application we need to install as additional components:

- the **Vulkan SDK** as dependency of our encoder application,
- a **C/C++ compiler**, either MSVC, GCC or Clang,
- the **LLVM** library used by LLVMpipe,
- the **Flex** and **Bison** tools required by Mesa3D,
- **Python**, **Meson** and **Ninja** for the Mesa3D build system and
- **CMake** with **make** or **Ninja** as build system for the encoder application.

We provide a reproducibility package for setting up a development and evaluation environment similar to our environment. The package contains scripts, configurations and a description file for Windows. It additionally contains Docker container definitions for WSL. Additionally, we provide the description for setting up a Windows development environment in Appendix A, which also contains the GitHub links to our reproducibility package and software repositories for all platforms. With this development environment in place we can now go into the details of the implementation of our CPU-based Vulkan Video extension and describe which Lavapipe components need to be extended or newly developed to build a functional and standard conforming Vulkan driver for video encoding.

## 4.1. Vulkan Feature Queries

Due to the flexibility of Vulkan an important part of the API are feature queries. These API calls allow the application to query the Vulkan driver for the supported features of the device and adapt to it. This is crucial to support a wide range of platforms and hardware. Based on the result of the queries, the application has several options. It can choose to enable only the supported features, possibly degrading performance and/or functionality. Alternatively, it could inform the user which features are not available and terminate. Most applications have alternative code paths to dynamically adapt to the available feature set and emulate missing features. Adapting to the supported feature set is especially relevant in the context of Vulkan Video extensions, where hardware support varies widely. First the application has to query for the supported Vulkan extensions and then activating them during logical device creation. Lavapipe already has the necessary infrastructure in place to support this type of queries. In Listing 4.1 we show how we extend the list of supported extensions in the Lavapipe device code by those we additionally want to support.

```
static const struct vk_device_extension_table
    lvp_device_extensions_supported = {
    // ...
   .KHR_video_queue                          = true,
   .KHR_video_encode_queue                   = true,
   .KHR_video_encode_h264                    = true,
};
```

Listing 4.1: Feature Query - Extensions

After informing the application that we support the video encoding extensions for H.264, we also need to provide a command queue family for video encode. Lavapipe has a rather simple queue setup, as there is no need to adapt to hardware queues. It supports all queue capabilities on the same queue family. Therefore, we add the encode capability to the same queue family by setting the `VK_QUEUE_VIDEO_ENCODE_BIT_KHR` bit in the `queueFlags` bitmask. Additionally Vulkan Video extensions allow to query for properties specific to video queues using structures, which can be added to the `pNext` chain of the queue feature query. We must use these extension structures to report that our queue supports H.264 video encode. Furthermore, we also report that we support providing the encode result status via query pools. The function `lvp_GetPhysicalDeviceQueueFamilyProperties2` handles all this. Both the old `vkGetPhysicalDeviceQueueFamilyProperties` and the new API function `vkGetPhysicalDeviceQueueFamilyProperties2` gets forwarded by Lavapipe to this internal function. We need to check which of the optional structures the application has provided in the `pNext` chain. The type of the structure can be detected by the first member in the structure called `sType`. After evaluating the correct type the structure pointer can be casted to the correct type and the other members can be filled. Mesa3D provides convenient C macros for handling those chains. Listing 4.2 shows a simple code for supporting those 2 new structures.

```
vk_foreach_struct (ext, pQueueFamilyProperties[i].pNext) {
    switch (ext->sType) {
    case VK_STRUCTURE_TYPE_QUEUE_FAMILY_VIDEO_PROPERTIES_KHR:
        VkQueueFamilyVideoPropertiesKHR *propV = (
    VkQueueFamilyVideoPropertiesKHR *)ext;
        propV->videoCodecOperations =
    VK_VIDEO_CODEC_OPERATION_ENCODE_H264_BIT_KHR;
        break;
    case
    VK_STRUCTURE_TYPE_QUEUE_FAMILY_QUERY_RESULT_STATUS_PROPERTIES_KHR:
        VkQueueFamilyQueryResultStatusPropertiesKHR *propS = (
    VkQueueFamilyQueryResultStatusPropertiesKHR *)ext;
        propS->queryResultStatusSupport = VK_TRUE;
        break;
    }
}
```

Listing 4.2: Feature Query - Queue Properties

The function `vkGetPhysicalDeviceVideoCapabilitiesKHR` is added by the Vulkan Video Extensions to allow the application to query the driver for the capabilities of a video profile. In the context of Vulkan Video a video profile is the combination of the operation

(encode, decode), the coding standard (H.264, H.265, AV1), the chroma subsampling (monochrome, 4:2:0, 4:2:2, 4:4:4) and the bit depth (8, 10, 12 bits). Additionally it contains coding format specific parameters, like the coding standard profile (Baseline, Main, ...) for H.264 or film grain support for AV1. This function can be used to query the support for a specific profile and the supported capabilities. There is a set of general capabilities (e.g. minimum and maximum picture size) and a set of coding format specific capabilities (e.g. for H.264: maximum slice count, maximum P picture reference count, minimum and maximum quantization parameter). This API function was added by the Vulkan Video Extensions, therefore Lavapipe does not contain support for it, so we need to implement it. The function needs to check if the profile is supported before reporting the capabilities. Otherwise it needs to return one of the new error codes for unsupported profiles. We also accept profiles requesting the H.264 main profile as the CTS uses this profile. Even if we only support baseline features of H.264, we can still report the main profile and stay compliant with the standards. We use the returned capabilities and the possibility to override parameters to restrict the application to only use features we support. Our implementation of `vkGetPhysicalDeviceVideoCapabilitiesKHR` limits the capabilities to those supported by minih264 and those which seem to be sane values to reach our goal. We limit the maximum number of slices per picture to 1, as we do not support more slices on our codec interface. We limit the maximum number of reference pictures for a P frame to 1, as minih264 does not support more. Also the minimum and maximum quantization parameter is constrained by minih264 to 10-51 (due to its hardcoded quantization tables). We limit the maximum picture size to 4096x4096 as this is enough for running all our tests. The minimum picture size is constrained by the size of one H.264 macroblock (16x16). The implementation in `lvp_GetPhysicalDeviceVideoCapabilitiesKHR` is straight-forward. After checking the profile, it fills the capabilities into the structure members and returns success.

A second function `VkVideoEncodeQualityLevelPropertiesKHR` allows to query default values for the video encode rate control. This allows the application to get preferred default values which are in the allowed range of the device's capabilities. As this function is mandatory, we need to implement one which, similar to the "get capabilities" function, first checks if the profile is supported and then returns sane default values. We return that we prefer disabling rate control, which gives full control over the video quality to the application. We suggest using a GOP frame count of 30 (which is depending on the frame rate, roughly 1 second of video) and set all other values within the allowed range.

The Vulkan specification describes a wide range of image formats, but only a subset of them are normally supported by a specific device. From those supported formats, each format only needs to support a subset of features (e.g. color attachment, transfer source, video encode source, video reference picture). The supported features also depend on the tiling arrangement of the image data. Furthermore, it is even possible that only a specific combination of features can be activated on the same image at the same time. Even if multiple features can be activated, images must be transitioned between image layouts if they get used in different ways. This allows the driver and the hardware to optimize the image data layout for specific use cases. To support querying for all these

combinations, the Vulkan API splits the feature queries into 2 functions (and multiple versions of them). The function `vkGetPhysicalDeviceFormatProperties2` allows to get the supported features for buffers with a specific image format. On top of buffers, Vulkan specifies "images", which have additional parameters. Images have usage flags which correspond to some extent to the feature flags of the underlying buffers. The function `vkGetPhysicalDeviceImageFormatProperties2` can then be used to query a combination of image format, usage, video profile and further parameters for more details and limitations. As already discussed, we enable 3-plane images to be used for input and reference pictures. Additionally we support 2-plane images to be used as input pictures for the video encoding process. Reference pictures are stored in decoded picture buffers (DPB) by Vulkan Video, therefore the feature flag contains DPB as abbreviation as shown in Listing 4.3.

```
if (format == VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM)
    features |= (VK_FORMAT_FEATURE_2_VIDEO_ENCODE_DPB_BIT_KHR |
                 VK_FORMAT_FEATURE_2_VIDEO_ENCODE_INPUT_BIT_KHR);
if (format == VK_FORMAT_G8_B8R8_2PLANE_420_UNORM)
    features |= (VK_FORMAT_FEATURE_2_VIDEO_ENCODE_INPUT_BIT_KHR);
```

Listing 4.3: Feature Query - Format Properties

The code for handling `vkGetPhysicalDeviceImageFormatProperties2` is located in `lvp_get_image_format_properties` and is extended to check if the given video profile is supported. Furthermore, it prevents reference pictures to be used for any other usage. As these pictures can be in an internal memory layout of the codec, and we do not support layout transitions, we have to disable all other usages. For any image format specifying a video profile we enforce that it is either used as video encoder input or reference picture. Our extension as shown in Listing 4.4 ensures those conditions and uses the same syntax as Lavapipe (including jumping to the `unsupported` label).

```
if (video_profile_list) {
    // Check profile support
    // ...

    // DPB pictures do not support any other usages
    if (info->usage & VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR && info->
    usage != VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR) {
        result = VK_ERROR_IMAGE_USAGE_NOT_SUPPORTED_KHR;
        goto unsupported;
    }
    // picture must be either DPB or SRC
    if (!(info->usage & VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR || info->
    usage & VK_IMAGE_USAGE_VIDEO_ENCODE_SRC_BIT_KHR)) {
        result = VK_ERROR_IMAGE_USAGE_NOT_SUPPORTED_KHR;
        goto unsupported;
    }
}
```

Listing 4.4: Feature Query - Image Format Properties

An application using these functions would need to try all possible combinations of image formats, tiling arrangements, video profiles and features until it finds the supported ones. As this is not manageable for the number of possible combinations introduced by Vulkan Video, the Vulkan Video Extensions add another API call allowing to get a list of all supported combinations for a specific video profile by calling `vkGetPhysicalDeviceVideoFormatPropertiesKHR`. Our implementation of this function is again straight-forward. We first check if the given profile and image usage combination is supported. We return, depending on the requested usage either the supported 3-plane format (for DPB usage) or an array of the 2 and 3-plane formats (for encode source usage). For encode source usage we also enable two additional flags (mutable and extended usage). These together allow the application to create image views for single planes of the multi-plane image and enable usage flags on those views, which are not allowed for the multi-plane image. We do not need to write any extra code to support those views and usages, but applications can save copy operations by directly writing into the memory of the image planes (for example in the RGB->YCbCr conversion shader). As the function returns a variable sized array the Vulkan specification requires the application to call the function twice. Once to determine the number of array entries and a second time providing the memory for the array and getting the data filled in. Mesa3D has convenient macros for handling this correctly and type-safe. Listing 4.5 show how the array gets declared first, then the number of entries get counted and optionally the entries get filled and finally the correct status code gets returned.

```
1  // declare array
2  VK_OUTARRAY_MAKE_TYPED(VkVideoFormatPropertiesKHR, out,
       pVideoFormatProperties, pVideoFormatPropertyCount);
3
4  // generate entries
5  const VkFormat supportedFormats[] = {VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM,
       VK_FORMAT_G8_B8R8_2PLANE_420_UNORM};
6  for (int i = 0; i < sizeof(supportedFormats) / sizeof(supportedFormats
       [0]); ++i) {
7    // count the entries
8    vk_outarray_append_typed(VkVideoFormatPropertiesKHR, &out, p)
9    {
10      // and optionally fill the entry
11      p->format = supportedFormats[i];
12      // ... fill all other members of this entry
13
14   // DPB only supports 3-plane
15   if (pVideoFormatInfo->imageUsage &
       VK_IMAGE_USAGE_VIDEO_ENCODE_DPB_BIT_KHR)
16     break;
17 }
18 // return status code (VK_SUCCESS or VK_INCOMPLETE)
19 return vk_outarray_status(&out);
```

Listing 4.5: Feature Query - Video Format Properties Array

## 4.2. Vulkan Image Handling & Reference Pictures

We need two different kind of images for video encoding. The first type are input images. They contain the video frames which we receive from the application. We need to encode them and return the encoded bit stream. Input images are specified in Vulkan as images with encode source usage. The second type of images contain reference pictures. They are created within the encoder by decoding the last encoded picture again. Therefore, they exactly match the pictures the decoder generates including all compression artifacts. They are stored in decoded picture buffers (DPB). DPBs are specified in Vulkan as images with encode DPB usage. The application needs to store reference pictures until they get used again during encoding of a later frame.

As the application, Lavapipe and our H.264 encoder are running on the CPU, all three can access the same memory. This allows us to simply reference the internal buffers of Lavapipe images when calling the H.264 encoder eliminating all copy operations. The Gallium architecture already provides a convenient interface function `pipe_context::texture_map()` for getting access to image data and LLVMpipe implements it as direct access to the internal image buffers. The function also provides the image stride. A simplified form of `map_image_to_ioyuv()` used to map the three planes of our Vulkan image to our codec interface is shown in Listing 4.6.

```
// the image structure for the encoder
H264E_io_yuv_t src_yuv;
// stores additional information; necessary for unmap
struct pipe_transfer *transfers[3];
// fill with image size
struct pipe_box box = { /* ... */ }
const VkImageAspectFlagBits planes[] = {
  VK_IMAGE_ASPECT_PLANE_0_BIT,
  VK_IMAGE_ASPECT_PLANE_1_BIT,
  VK_IMAGE_ASPECT_PLANE_2_BIT,
};
for (int p = 0; p < 3; ++p) {
    // get the plane number for each plane
    uint8_t plane = lvp_image_aspects_to_plane(image, planes[p]);
    // plane content must be itself a 1-plane 8-bit image
    assert(image->planes[plane].bo->format == PIPE_FORMAT_R8_UNORM);
    // get the data pointer and fill transfers structure
    yuv->yuv[p] = state->pctx->texture_map(state->pctx,
                                           image->planes[plane].bo,
                                           0, // level
                                           PIPE_MAP_READ,
                                           &box,
                                           &transfers[p]);
    // get the stride from the transfers structure
    yuv->stride[p] = transfers[p]->stride;
}
```

Listing 4.6: Vulkan Image Handling - Map Image Buffers

The image format used for video encoding is critical to both performance and qual-

ity. Most modern video encoders, including minih264, work with planar YUV formats, typically YUV 4:2:0. Vulkan supports this via multi-planar image formats such as `VK_FORMAT_G8_B8R8_2PLANE_420_UNORM` for encoding the image as 2-planes with Y separately and UV combined or `VK_FORMAT_G8_B8_R8_3PLANE_420_UNORM` for encoding the image as 3-planes with each component separately. In these formats, the luminance (Y) component is stored at full resolution, while the chrominance (U and V) components are subsampled by a factor of two in both horizontal and vertical directions. Using planar YUV formats enables efficient compression because the human eye is more sensitive to brightness than to color details, allowing encoders to discard more color information without noticeably affecting visual quality. Choosing the correct 3-plane Vulkan image format ensures proper alignment and memory layout to maintain compatibility with our video encoder. Each plane of the 3-plane YUV image can be seen as its own image containing exactly one component. Vulkan's use of the image format `PIPE_FORMAT_R8_UNORM` for each plane of the multi-plane image reflects this structure. Applications can use image views to access each plane individually. Also Gallium's internal abstraction handles mapping each plane independently. This allows straightforward mapping between the Vulkan image representation and the codec interface, which expects raw 8-bit buffers for each component. This also simplifies stride handling, as each plane can be treated as a linear 2D buffer of bytes and both Lavapipe's internal structure and our encoder interface work with the same stride concept. This works for all 3-plane images, regardless if they are used as encoding source or for reference pictures. For 2-plane images, which we support as encoding source, we need to separate the second (UV) plane into separate U and V planes. The first plane (Y) can simply be mapped as shown above. For the separation we use temporary buffers, which we reserve already during initialization. This is done in `convert_image_to_ioyuv()`. This process uses an extra copy operation and additional memory. Therefore, using 3-plane images is the preferred option to reduce overhead.

A special handling is also necessary for reference pictures. They are used during the motion detection and compensation process. Content which is already contained in the reference picture does not need to be encoded again. Only small differences, if existing, need to be encoded. This reduces the necessary space for the encoded video. To be able to compensate for motion and therefore movement of content between pictures, the encoder needs to detect this motion and encode it as motion vectors into the resulting bitstream. H.264 allows motion vectors to point outside of the picture. This enables better optimization for objects entering the video on the frame edge. H.264 defines in clauses 8.4.2.2.1 and 8.4.2.2.2 that if a referenced pixel is outside of the reference picture, the X and Y coordinates are clipped to the nearest valid coordinate [ITU03]. As the motion detection algorithm has to scan the whole picture this edge condition would introduce additional calculation overhead, which is unnecessary for most of the other pixels. To circumvent this overhead, the algorithm operates on a reference picture which is slightly larger than the input image with a centered content. We fill the border with the same pixel values as the nearest pixels of the original reference picture. This means we effectively extend the edge of the reference picture to build a border around it. This

allows the algorithm to reach the same effect as clipping the coordinates at the edge by just addressing the new border pixels. Filling the border is done by the encoder, but we need to adapt Lavapipe such that it allocates more memory for images, which are created with DPB usage enabled. In `lvp_image_create()` we check the video profile and the usage flag, if we detect DPB usage, we add extra space for the border. The width of the border is specified by the encoder interface in `H264E_BORDER_PADDING` and is for minih264 16 bytes. We also make sure that the rows are aligned in memory on border size boundaries. As we only allow DPB usage, we do not need to hide the border memory on layout transitions. The flexibility of the 3-plane formats allow us to easily extend our implementation later, if we want to support further image formats with different chroma subsampling (4:2:2 or 4:4:4) for better color resolution or more bits per sample for better color depth as used in high dynamic range (HDR) videos.

## 4.3. Vulkan Query Pools

Vulkan uses Query Pools for providing feedback from the command execution to the application. This feedback can come from different sources. One of the query types is the video encoding status. Each encoded slice creates a query entry. This entry can contain the status of the encoding process, mainly if success or failure, the reason for a failure (e.g. output buffer too small) and the number of bytes written into the output buffer. The application can select during pool creation which information must be stored in the entry. To manage and use a query pool, an application needs to:

1. Create a query pool for a specific query type.
2. Execute a begin query command on a queue.
3. Execute a command which provides query data (e.g. encode video).
4. Execute an end query command on the same queue.
5. Wait until the command execution is finished.
6. Get query pool results.
7. Destroy the query pool.

Lavapipe has already the necessary infrastructure for handling query pools. We just need to add support for video encode feedback queries. During pool creation we need to store which information is of interest for the application. As the internal query pool structure has already members for other types of queries, which are unnecessary for video queries, we can write our information as union into the same memory space. Therefore, we do not need to change the internal pool structure. Vulkan Video does not allow to have multiple video status queries activated at the same time on the same command buffer. This allows us to store a pointer to the query pool in the internal queue state if the application executes a begin query command and delete it again if the application

executes an end query command. This pointer is used by the encode command to find the query pool and add an entry after encoding a frame. Finally, we need to extend the function `lvp_GetQueryPoolResults()` to reach full query pool support. Applications can get the query results in either 32 or 64 bit and they can enable or disable the status byte. Therefore, we need to convert our internal query pool entries while copying them into the application buffer. This functionality allows the application to get feedback from the video encode process, which is necessary to detect failures and to know the size of each encoded frame, for knowing how much date the application needs to read from the output buffer.

According to the Vulkan specification, applications must ensure that the results of queries are available before reading them. This typically requires synchronization using fences or waiting for the queue to complete execution. The query availability status byte, when enabled by the application, provides a lightweight mechanism to poll for query completion without the need for full synchronization. This can reduce CPU overhead and latency in some scenarios. In the context of video encoding, detecting availability is important, especially when operating in a low-latency environment. For our CPU only solution this is much easier compared with a heterogeneous system including hardware accelerators. Lavapipe's query pool is already designed to correctly synchronize with the queue command execution. While the synchronization is correct, there is still room for improvement by providing a more fine-grained synchronization on frame level for multi-frame command buffers. We expect that encoding multiple frames in one command buffer increases latency even more than we can gain with better synchronization. Therefore, this is not a viable solution for low-latency applications. For latency insensitive applications it is anyway not important to have tighter synchronization, as they normally optimize for throughput, where synchronizing less often is advantageous. So we do not consider fine-grained synchronization of query pool results for our solution.

## 4.4. Vulkan Video Sessions

Before video encoding commands can be recorded to a command buffer both a video session and a video session parameter object need to be created and bound to the command buffer. The video session object stores information that is immutable during the session, like the codec and the maximum frame size. Furthermore, it holds pointers to the scratch and image conversion buffers and the internal video session storage of the encoder. The video session object is internally represented by a general Mesa3D structure and our Lavapipe specific structure as shown in Listing 4.7.

```
1 // Mesa3D general video session structure - Lavapipe independent
2 struct vk_video_session {
3     struct vk_object_base base;
4     VkVideoSessionCreateFlagsKHR flags;
5     VkVideoCodecOperationFlagsKHR op;
6     VkExtent2D max_coded;
7     // ...
8
```

```
    union {
        struct {
            StdVideoH264ProfileIdc profile_idc;
        } h264;
        // ...
    };
};

// Lavapipe specific video session structure
struct lvp_video_session {
    // embed Mesa3D structure
    struct vk_video_session vk;

    // encoder persistent and scratch memory
    H264E_persist_t* enc;
    H264E_scratch_t* scratch;
    // buffers for 2->3 plane image conversion
    uint8_t* split_buffers;
    // store current rate control mode (from coding control cmd)
    VkVideoEncodeRateControlModeFlagBitsKHR rate_control_mode;
};
```

Listing 4.7: Vulkan Video Session - Internal Video Session Structure

Our implementation of `lvp_CreateVideoSessionKHR` is querying the encoder for the needed memory, allocating all structures and buffers, saving the parameters and initializing the encoder. At the end it is returning an handle of the newly created video session object to the application.

While the video session holds all parameters, which are immutable during a video coding session, a video session parameters object is holding all the coding format specific parameters, which can be changed for each picture or picture sequence. For H.264 these are the Sequence Parameter Sets (SPS) and the Picture Parameter Sets (PPS). A video session parameters object can hold multiple of each. The application can switch between them by specifying the PPS identifier as parameter to the encode command. This allows us to use different encoding parameters for each picture. Mesa3D contains already the infrastructure for handling video session parameters. In Lavapipe we only need to implement the Vulkan API functions, which then pass the parameters to the general Mesa3D code. As we do not support all possible parameters and combinations of parameters we check them before returning the handle of the parameters object to the application. If we detect a parameter which we do not support, we override the parameter as allowed by the Vulkan specification. We need to mark this condition, so that application can detect it by checking the override attributes in the parameters feedback structure. For example, we only support decode picture order count method 2 of the H.264 standard and therefore we have an override check in place as shown in Listing 4.8.

```
if (sps->pic_order_cnt_type != STD_VIDEO_H264_POC_TYPE_2) {
    sps->pic_order_cnt_type = STD_VIDEO_H264_POC_TYPE_2;
    params->has_sps_overides = true;
}
```

Listing 4.8: Vulkan Video Session - Parameter Override

Before a SPS or PPS can be used they need to be encoded into the bitstream. In contrast to many other Codec APIs, Vulkan Video does not add them automatically into the output. The application is responsible for doing this once at the beginning of the stream or for redundancy in regular intervals. To support the application, Vulkan Video provides the function `GetEncodedVideoSessionParametersKHR()` to generate the encoded data structures for a given parameters object. Encoding those data structures into bitstream data follows an exact process defined in the H.264 specification. We have to provide this API function, but can internally use the bitstream encoder which is already available in the Mesa3D code.

Video sessions and video session parameters objects need to be bound before they can be used by video encode commands. Binding those objects is done with commands which are recorded to a command buffer and executed on the video queue. Vulkan Video prohibits the binding of multiple video sessions or parameter objects at the same time to the same command buffer. Listing 4.9 shows how we store pointers to the video session and the parameter structure within the internal queue state.

```
struct rendering_state {
    // ... (non-video queue state)

    struct {
        struct lvp_query_pool *active_query;
        struct lvp_video_session *active_video_session;
        struct lvp_video_session_params *active_video_session_params;
        uint32_t active_query_num;
    } video_encode;
};
```

Listing 4.9: Vulkan Video Session - Lavapipe Queue State

During execution of video commands on that queue, we can retrieve the pointers from the queue state. We can then access the encoding parameters and the encoder buffers. When the application unbinds the video session using an end video coding command, we set the pointers to null. Our `video_encode` structure contained in the queue state, also holds the already discussed query pool pointer.

## 4.5. Vulkan Video Commands

To store commands, which should later get executed on a queue, Vulkan provides command buffers. From the application side a command buffer needs to be allocated from a command buffer pool. The application then starts the recording on the command buffer and records a number of commands into it by calling the respective `vkCmd*()` functions on the Vulkan API. For Vulkan Video, such commands can be used for controlling video sessions (e.g. `vkCmdBeginVideoCodingKHR()`), controlling related states (e.g. the query pool with `vkCmdBeginQuery()` or for the actual encoding of a picture or slice of a video `vkCmdEncodeVideoKHR()`. After all commands are recorded into the command buffer, the application ends the recording and can then submit the command buffer to a device queue. The command buffer is afterwards in pending state and waits for execution.

On the driver side a command buffer is a data structure which needs to hold a list of all recorded commands. When the command buffer gets submitted to a queue, the driver needs to transfer the internal data structure to the device and queue it for execution. In Lavapipe this is simplified as the recording and queue execution has access to the same memory and therefore the buffer does not need to be transferred. The command buffer does not only need to store a list of commands, but also needs to hold a copy of all parameters of these commands. The application is required to keep all resources which are referenced by the commands available until the execution is finished, but the parameters themselves need to be copied into the command buffer data structure. As the Vulkan API uses function parameters, which are pointers to structures, these whole structures need to be copied, not only the pointers. Those structures can recursively contain pointers to further structures. A special case is the `pNext` pointer, which is an un-typed pointer to any other structure. The type of the structure is determined by the content of the first member `sType`. Those structures contain again a `pNext` pointer building a chain of structures. Therefore, recording the commands and storing all parameters in the command buffer needs deep knowledge about the commands and their parameters itself.

To simplify this, Mesa3D contains a code generator, which can generate out of the Vulkan specification (in XML) the necessary C code to record all commands including their parameters. This generator `vk_cmd_queue_gen.py` is written in Python and runs during the build of the Lavapipe driver. The Vulkan specification `vk.xml` as released by the Khronos Group is read as input. The output are the files `vk_cmd_queue.c` and `vk_cmd_queue.h`. This code generator is limited to the functionality necessary for rendering commands and can not handle all details of the specification. Upon some exceptions these commands do not use nested data structures (directly nested or nested via `pNext` chain). For those rare exceptions, hand-crafted implementations were provided. The Vulkan Video Extensions introduced numerous commands with nested and very dynamic data structures. In our case nearly all commands would have needed hand-crafted command buffer handlers. To handle all specification details we decided that it makes sense to improve the code generator. We extended it so that it also reads `video.xml`, the specification of the Vulkan Video Extensions. To its outputted C code it now additionally adds functions to enqueue video commands copying all parameters (e.g. `vk_enqueue_cmd_end_video_coding_khr()`) and functions to clean up memory (e.g. `vk_free_cmd_encode_video_khr()`) used after executing the command. This allows us to use the automatically generated code to handle the recording and clean-up of video commands in Vulkan command buffers without using any hand-crafted functions.

`vkCmdBeginVideoCodingKHR()` is the first command which needs to be executed on a video queue. We implement this like all queue commands in `lvp_execute.c`. A large switch command in `lvp_execute_cmd_buffer()` selects, depending on the command type, the handler function. We extend the switch to call our handler function `handle_begin_video_coding()`. In this handler function we use the helper macro `VK_FROM_HANDLE` to get the video session and video session parameter pointers from the handles provided in the command. We store the pointers in our internal queue state for later use. This effectively binds the video session to the command buffer. Those

pointers in the queue state allow us to access the codec data structure, internal state and data buffers and the video parameters like SPS and PPS during execution of further video commands of this command buffer. Due to the infrastructure provided by Mesa3D, the handler function shown in Listing 4.10 is very compact.

```
static void
handle_begin_video_coding(struct vk_cmd_queue_entry *cmd, struct
    rendering_state *state)
{
   // get the command parameters
   struct vk_cmd_begin_video_coding_khr *begin_video_coding = &cmd->u.
    begin_video_coding_khr;

   // get the video session pointer from the provided handle
   VK_FROM_HANDLE(lvp_video_session, vid_session, begin_video_coding->
    begin_info->videoSession);
   // store it in the internal queue state
   state->video_encode.active_video_session = vid_session;

   // get the video session parameters pointer from the handle
   VK_FROM_HANDLE(lvp_video_session_params, vid_session_params,
    begin_video_coding->begin_info->videoSessionParameters);
   // store it in the internal queue state
   state->video_encode.active_video_session_params = vid_session_params;
}
```

Listing 4.10: Vulkan Video Commands - Bind Video Session

After using video coding commands and before the command buffer ends, the specification requires that the video session gets unbound with `vkCmdEndVideoCodingKHR()`. We handle this command in `handle_end_video_coding()`, which is called in the same way using the large switch. In our handler function we reset the internal state pointers to NULL to unbind the video session. This is not absolutely necessary, as executing video commands with invalid handles is not allowed per specification, but setting them to NULL makes it easier to track down bugs in the driver or the application.

After binding the video session, the application is required to reset the state of the video session using the command `vkCmdControlVideoCodingKHR()` with the flag `VK_VIDEO_CODING_CONTROL_RESET_BIT_KHR`. This must be done at least once before encoding the first frame, but can be done more often. This function is also used to set new bitrate or quality settings for the video encoding session depending on the structures provided in the `pNext` chain. The application can call it as often as necessary in a command buffer with a bound video session. On every reset our implementation in `handle_control_video_coding()` sets the rate control mode to default. The default rate control uses a fixed quantization parameter, which we take from the currently active PPS structure in the video session parameters. Alternatively, we also support disabling rate control using the flag `VK_VIDEO_ENCODE_RATE_CONTROL_MODE_DISABLED_BIT_KHR`. This allows (and forces) the application to specify the quantization parameter explicitly for each slice of the frame during the encode command by providing the information in the `VkVideoEncodeH264NaluSliceInfoKHR` data structure. As the selected rate control mode

is persistent for the video session and there is no need for the application to set it for each command buffer, we need to store the the rate control mode in the video session instead of the queue state. This allows us to retrieve the mode later in other command buffers binding the same video session.

After binding and initializing the video session and before executing any video coding commands, the application must start a query in the query pool using `vkCmdBeginQuery()` to be able to later retrieve the result of the coding operation. This requirement is relaxed with the extension `VK_KHR_video_maintenance1`, which allows to specify the query inline with the coding command. As we anyway need to support the original variant and the new variant is optional, we need to extend the existing handler of the begin query command `handle_begin_query()` and the corresponding `handle_end_query()` to support the new query type used for storing and retrieving the video encode status. If the begin command gets executed for a query pool of type `VK_QUERY_TYPE_VIDEO_ENCODE_FEEDBACK_KHR` we store both the pool pointer and the selected query number in our internal queue state. At the end of the query we reset the pool pointer to NULL, so that we do not write into inactive queries during later commands.

After setting up the complete infrastructure on both the Vulkan API and the encoder side, we are able to implement the execution of the Vulkan Video encode command. This command gets recorded in the command buffer using `vkCmdEncodeVideoKHR()` and is allowed to be recorded after binding a video session with the start command and resetting it using the control command. After the command buffer gets submitted to a video queue, Lavapipe assigns a worker thread to the execution of the command. Our implementation of the command is done in `handle_encode_video()`. The general workflow of the function is:

1. Getting the video session and parameters from the queue state.
2. Mapping input and reference picture memory.
3. Optionally converting the input image memory layout to 3-plane.
4. Getting the correct SPS and PPS from the parameters object.
5. Configuring frame type (I or P) and quality/rate control for encoder.
6. Calling the encoder via our encoder API.
7. Unmapping the image memory.
8. Mapping the output buffer memory and transferring the video bitstream.
9. Adding an entry to the query pool with the encoding result (status and bitstream length).

The frame encoding command first queries the current state for the video session and parameters, which we stored during the begin command. We need those data structures for all further encoding steps, as they store the pointer to the codec, internal buffers, the

rate control mode, the query pool pointer and the SPS/PPS encoding parameters. All other values necessary for encoding the frame must be provided through the command. These are source image, reference pictures, the picture type (IDR/I/P), the PPS number, the quantization parameter if rate control is disabled and optional reference picture list updates. The first step of the encode command handler is getting access to the input image and reference pictures. As already described we try to minimize copy operations, therefore we use our already described function `map_image_to_ioyuv()` which uses Gallium's `texture_map()` function to retrieve the pointer to the raw image data. As there is only CPU memory in use, no memory operations, like bus transfers, are necessary. The function only blocks Gallium from moving the location of the backing memory of the image as long as there is at least one mapping active. Additionally to mapping the input reference picture, we also need to map the output reference picture. We need to retrieve a pointer to its data block, which can then be filled with the new reference picture data by the encoder. While mapping for reading is done using the flag `PIPE_MAP_READ`, mapping for writing uses the flag `PIPE_MAP_WRITE`. Mapping images to pointers and immediately using them works for reference pictures as we ensure that they are always stored using a data layout which fits to our encoder requirements. For the input image we can not ensure this in all cases, therefore the encode function decides depending on the input image format, if mapping is possible. If the data layout is not compatible, it uses our function `convert_image_to_ioyuv()`. This function also maps the raw image data, but then copies it into a scratch buffer and reorganizing the data layout into 3 separate data planes. The encode function then uses the scratch buffer instead of the mapped input image as source for the encoder.

Before encoding can start, we need to retrieve the correct SPS and PPS from the parameters object. This object can contain multiple SPS and PPS and the application can choose for each slice a different one by specifying the identification number (ID) of the PPS, which in turn contains the ID of the SPS. As searching for these data structures is a very common operation in Vulkan Video, which is also needed in drivers for hardware accelerators, Mesa3D already contains helper functions `vk_video_find_h264_enc_std_sps()` and `vk_video_find_h264_enc_std_pps()` which retrieve the pointers to the SPS and PPS data structures. We can directly access the necessary values. After we have retrieved all necessary values and buffers, we can fill the encoder data structure `H264E_run_param_t` and call the encoder via the already described encoder API call `H264E_encode()`. This calls the encoder, which we modified to be stateless. Video encoders normally handle reference pictures internally storing them as encoding state. As the Vulkan Video specification requires that the application takes care about reference pictures, we had to adapt the encoder, so that it does not store the picture any more, but gets a pointer to the reference picture. Additionally, the currently encoded picture gets immediately decoded again to be available as another reference picture. This is normally done in an encoder internal buffer. In our case the encoder needs to write the decoded picture into the buffer provided by the application. The application can then decide to use this picture later as reference picture for another input image. The final call to the encoder is shown in Listing 4.11.

```
error = H264E_encode(video_session->enc, video_session->scratch, &
    run_param, &src_yuv, ref_yuv_ptr, &dec_yuv, &coded_data, &
    sizeof_coded_data);
```

Listing 4.11: Vulkan Video Commands - Encoder Call

We also changed the encoder to not encode SPS and PPS headers into the bitstream. According to the Vulkan specification this must be done by the application. Those headers must be embedded at least at the beginning of the video, so that the decoder can access it. Encoders often embed it in regular intervals to allow starting a video from random positions. In Vulkan Video the application has any freedom to decide about the content of stream and picture parameters and the positions where they get embedded into the bitstream. In contrast to the SPS and PPS headers, the encoder still needs to embed slice headers in to the bitstream. We had to adapt the slice header generation of minih264 in `encode_slice_header()`, so that it matches our reference picture handling controlled by the application.

After the encoder finished, we must unmap all pointers. This allows Gallium to move the memory again. The encoder returns an error code and a pointer to the encoded bitstream of the frame if the operation succeeded. In the latter case we map the provided output buffer using `buffer_map()` to get a pointer to the backing memory and copy the bitstream into the buffer as shown in Listing 4.12. After the copy operation finished we must unmap the buffer again. Mapping buffers in Gallium work in a similar way as mapping images, but mapping only 1 dimension.

```
uint32_t *dst;
struct pipe_transfer *dst_t;
struct pipe_box dst_box;

// get the 1-D box for the buffer
u_box_1d(encode_video->encode_info->dstBufferOffset, sizeof_coded_data, &
    dst_box);
// lock the buffer and get the memory pointer
dst = state->pctx->buffer_map(state->pctx,
                              dst_buffer->bo,
                              0,
                              PIPE_MAP_WRITE,
                              &dst_box,
                              &dst_t);
// copy the output bitstream of the frame
memcpy(dst, coded_data, sizeof_coded_data);
// unlock the buffer (backing memory is movable again)
state->pctx->buffer_unmap(state->pctx, dst_t);
```

Listing 4.12: Vulkan Video Commands - Transfer Output Bitstream

The final step after the encoder finished for both successful and failed results, is adding the result to the query pool as shown in Listing 4.13. The entry always contains the final status of the encode operation. Depending on the query pool configuration the entry can contain further data items, like buffer offset, bytes written and if overrides were applied.

```
// get the active query pool
```

```
struct lvp_query_pool *query = state->video_encode.active_query;
// each bit in video_encode_feedback enables one additional data item
// plus 1 for the always present status code
uint32_t query_size = util_bitcount(query->video_encode_feedback) + 1;
if (query) {
  // calculate offset of the currently active query entry
  uint64_t *data = (uint64_t *)query->data + state->video_encode.
    active_query_num * query_size;
  // we always fill the buffer from the beginning
  if (query->video_encode_feedback &
    VK_VIDEO_ENCODE_FEEDBACK_BITSTREAM_BUFFER_OFFSET_BIT_KHR)
    *data++ = 0;
  // store the size of the encoded data
  if (query->video_encode_feedback &
    VK_VIDEO_ENCODE_FEEDBACK_BITSTREAM_BYTES_WRITTEN_BIT_KHR)
    *data++ = sizeof_coded_data;
  // we do not override on slice/frame level
  if (query->video_encode_feedback &
    VK_VIDEO_ENCODE_FEEDBACK_BITSTREAM_HAS_OVERRIDES_BIT_KHR)
    *data++ = 0;
  // store the result code
  *data = status;
  // activate the next entry
  ++state->video_encode.active_query_num;
}
```

Listing 4.13: Vulkan Video Commands - Update Query Pool

After encoding one frame the application must end the query as the Vulkan specification does not allow more than one frame within one query. This restriction gets relaxed by the optional `VK_KHR_video_maintenance1` update. After the application ended the query, it can decide to encode a further frame by starting a new query and issuing another encode command or it can decide to unbind the video session and end the encoding for this command buffer. Further frames can then be encoded by submitting a new command buffer or the application can decide to end the encoding completely. As the resulting H.264 elementary stream does not need any end markers, no further command is necessary for ending the video session. The application only needs to free the allocated resources. Encapsulating the video stream into a container format needs to be done by the application.

Our implementation covers all parts necessary for providing an interface conforming to the Vulkan Video specification and encoding the provided video as H.264 elementary stream. The whole encoding process runs on the CPU and does not need a GPU or any other form of video accelerator hardware. Using our implementation the video encoder is wrapped by a Vulkan API. This is transparent for the application. It accesses our CPU based Vulkan Video implementation in the same way as it would one of a GPU.

# 5. Evaluation

In order to answer research question RQ3, confirming that our implementation conforms to the specifications, we had to evaluate the implementation. For this we defined an evaluation environment and cunducted functionality and conformance tests. Additionally we also evaluated the performance of our implementation compared with directly using the minih264 encoder. The evaluation environment is similar to the development environment. The description of the development environment contains already all necessary steps to install the Vulkan Conformance Test Suite (CTS) which we used for evaluation. The evaluation was done on a Windows 11 system with an Intel Core i9-10850K 10-core CPU with 32 GB of RAM. This CPU operates at 3.6 GHz, but can increase the speed to 5.2 GHz in case only a subset of the cores is used. The evaluation system also contained an Nvidia GPU. The GPU was only used for decoding the output of our software for comparison as described later in the conformance tests chapter. All evaluation was done with Release builds of the software. In addition to the main evaluation system, we tested the software functionality and performance also on two other systems. The second evaluation system was based on the same hardware as our main evaluation system, but was using Ubuntu 25.04 with the Linux kernel 6.14.0 as operating system. We also used this system for evaluating compiler and optimization differences as described later in the performance test chapter. As third evaluation system we used a Raspberry Pi 5 with a Broadcom BCM2712 CPU and 8 GB of RAM running Debian Linux 12 with kernel version 6.6.74. The CPU is based on a 4-core Arm Cortex-A76 design running at 2.4 GHz. We used this system to validate that the software can be ported to other CPU technologies.

## 5.1. Functionality Tests

For confirming the general functionality during both the evaluation and the development we used the Vulkan Video example `vulkan-video-encode-simple`. We have developed this software to provide an easy to use example for encoding videos using Vulkan Video. During the initial development of the example, we only had a Vulkan Video implementation from Nvidia as reference. Due to the platform independency of Vulkan, it could be expected that it works also with other implementations, but this was not the case. We found out that we had taken some short-cuts which we had to change to allow wider use. As we developed the example for Nvidia hardware only, we supported 2-plane images only. The Vulkan standard requires that we query the driver for the possible image formats and use one of the supported formats. As we only support 3-plane images as reference pictures in our CPU based implementation, we had to extend our example code to be more flexible. We now support all possible image formats for reference pictures (DPB).

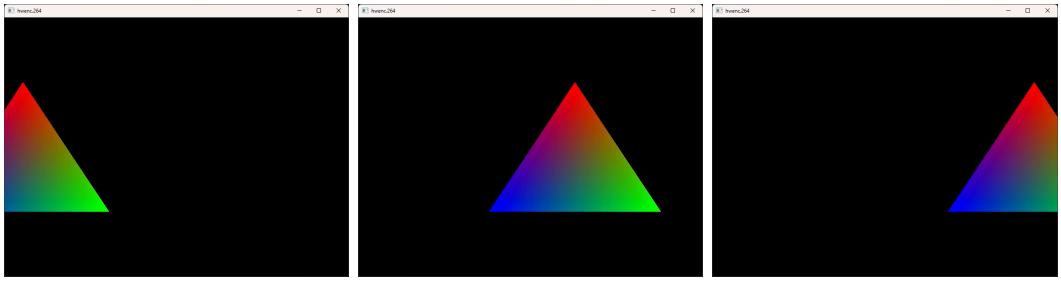

Figure 5.1.: Video Stills

Another shortcoming we found is that Nvidia does not check if the encoded bitstream fits into the maximum buffer size, but our implementation of Vulkan Video cares about buffer sizes and reports an encoding error in case of buffer overruns. Our example code did not specify the buffer size correctly as it was derived from an Nvidia sample, which was always providing a buffer size of zero. To get correct behavior according the specification we had to fix this in our example. The third shortcoming we found was that our example code was limited to using only variable bit rate control. The Vulkan Video specification states that an implementation only needs to support default rate control. Our CPU implementation supports default and disabled rate control, but not constant or variable bit rate. We extended our example code to support all possible rate control modes and choose the best available one. These three modifications to the example code were necessary to improve the standard conformance of our example and to be able to use it with our CPU based Vulkan Video implementation. We updated the public example code to contain those modifications.

The example simulates a cloud gaming scenario. It renders a simple scene (a moving triangle) into an offscreen image buffer. It then uses a compute shader to convert each rendered frame into the YCbCr color format. Finally, it uses Vulkan Video to encode the frame, transfers the video bitstream to the CPU and stores the H.264 elementary stream in a file. The example therefore requires a Vulkan implementation with support for graphics, compute and video queues. It does not require a presentation surface. This perfectly matches our Lavapipe based Vulkan Video implementation, which exactly supports those three queue types. Using the environment variable `VK_ICD_FILENAMES` we forced Vulkan to use our own Lavapipe driver instead of the system driver. We then started our Vulkan Video example application, which created the output file `hwenc.264` using the Lavapipe driver with our Vulkan Video extension.

We visually inspected the output using `ffplay` from the FFmpeg framework[Bel]. The result can be seen in Figure 5.1. It correctly shows the moving triangle. `ffplay` also validates the bitstream while decoding it. As it did not report any warnings and errors the bitstream from our encoder was correct. We did the same test on all 3 of our evaluation systems. Our Lavapipe driver with Vulkan Video support compiles not only on Windows but also on Linux systems and for ARM CPUs without any changes. The test results on all 3 systems were equivalent and were producing the same output video. The video

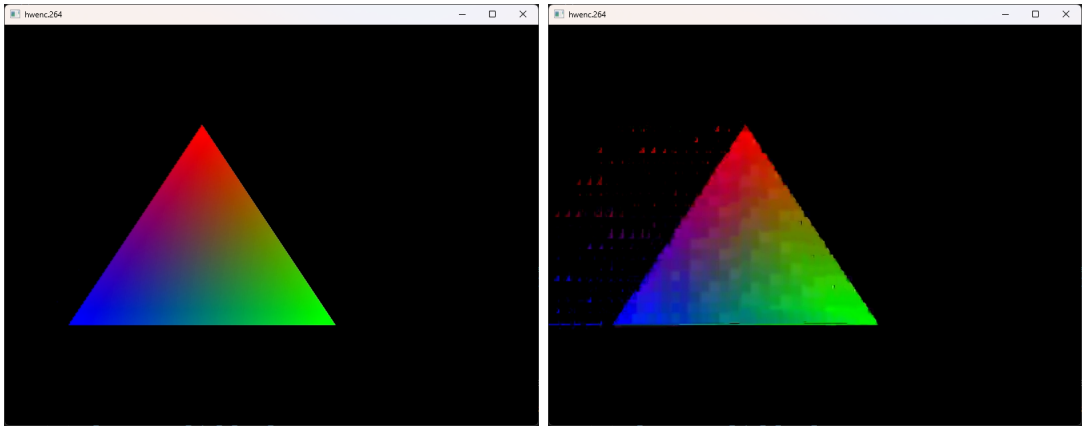


Figure 5.2.: Encoding Quality/Quantization Parameter QP 10 (left) vs. 51 (right)

shows the typical block artifacts of the H.264 encoding. The strength of those artifacts depend on the encoding quality. Figure 5.2 shows that the encoding parameters, here the quantization parameter controlling the video quality, is correctly handled. The left picture is taken from a video with quantization 10 and the right picture with quantization 51, which are the limits of the parameter. A lower quantization results in a better video quality. The test results proof the general functionality of our CPU based Vulkan Video implementation, showing that it is possible to implement Vulkan Video completely on CPU without the need for dedicated hardware accelerators. We still need to show that the implementation conforms to the Vulkan specification.

## 5.2. Conformance Tests

For validating the specification conformance, both regarding Vulkan Video specification and H.264 specification, we used the Vulkan Conformance Test Suite. The CTS organizes the tests in a hierarchy. The Vulkan Video category `dEQP-VK.video.*` contains 7261 different tests covering all possible coding formats, optional extensions and image formats. Only a small subset is important for us, especially those in the category `dEQP-VK.video.encode.h264.*` plus some outside of this category for testing video feature queries and synchronization. CTS automatically detects the supported features and does not execute test cases for unsupported optional features. 48 test cases are covering the features of our implementation of Vulkan Video. As the test cases were derived from the same Nvidia example code we used for our example application, they contain similar limitations. Therefore, we had to adapt the CTS code to be conforming to the standard. Similar to our video encode example application we had to improve the support for correct buffer size handling. Furthermore, we also had to add support for more reference picture formats and limited usage flags. To ensure the correctness and quality of our tests, we also provided our changes to the maintainers of the CTS. Those regarding detection of buffer size handling and usage flags of reference picture buffers were already merged into

the official CTS code, some other changes are still waiting for approval. Another set of modifications is specific to our situation with a Vulkan Video implementation supporting only video encoding. While this fully conforms to the specification, there is no other implementation supporting encoding, while not supporting decoding. Therefore, the CTS developers decided that they can use Vulkan Video decode for validating the output of the encode tests. To break this dependency we extended the CTS to allow to use a separate Vulkan device for output validation. In our test environment we used our extended Lavapipe driver as the device under test, but used an Nvidia device for decoding the output of the encode tests. As already mentioned some modifications are already part of the official CTS code, the changes which are still missing are contained in our public fork of the CTS. The 48 supported video tests of the CTS can be categorized into:

- **Feature Queries** with 4 passed tests,
- **Source Image Formats** with 2 passed tests (for 2 and 3-plane),
- **DPB Image Formats** with 1 passed test (for 3-plane),
- **Encoding** with 8 passed tests (from the encode.h264.* subcategory),
- **Synchronization** with 20 passed tests and
- **Synchronization2** with 13 passed tests.

The encoding category contains tests which are testing the complete encoding process. 8 of these tests are applicable for our implementation as they test supported features. The tests for optional features, which we do not support (bit rate control, B frames and quantization map) were automatically skipped. The executed encoding tests were:

- **I Frame Encoding** with 1 passed test,
- **Quality & Rate Control** with 3 passed tests,
- **I & P Frame Encoding** with 2 passed tests,
- **Query Pool** with 1 passed test and
- **Intermediate Resolution Change** with 1 passed test.

Validation of the encoding test output was done by decoding them (using the second Vulkan device) and comparing them with the original input data. As H.264 encoding is lossy, an exact comparison was not possible. The CTS is using a reconstruction quality metric using the peak signal-to-noise ratio (PSNR) for evaluating the correctness of the encoding process. It measures the similarity of two images as a ratio expressed in dB. A higher value means the images are more similar. Figure 5.3 shows the reconstruction quality over all frames of all executed encoding tests. The minimum necessary quality so that the CTS is not issuing a warning or counting the test as failed is 30dB. Nearly all frames had far better metrics then needed. Frame 14 and 15 were near the limit. This

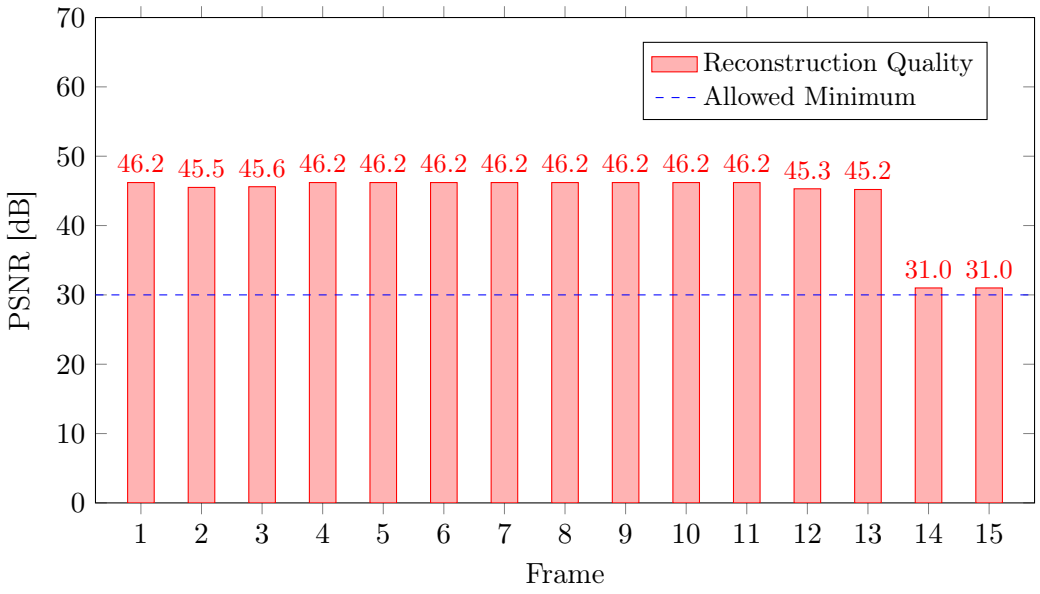


Figure 5.3.: Reconstruction Quality (higher is better)

was expected, as they were the last frames of a test case changing the encoding resolution during the encoding process. Those frames were encoded with only half of the resolution in both horizontal and vertical direction. Therefore, the decoded picture were containing far less details than the original picture, accounting in a worse reconstruction quality. Still all frames were within the necessary limits.

We also measured the impact of the the video quality setting, in our case the quantization parameter QP, on the reconstruction quality and compared it with the file size of the resulting video file as seen in Figure 5.4. A low quantization produces the best video quality, but also produces large files or, which is important for streaming applications, a high video bit-rate. Increasing the quantization by a small value already decreases the file size substantially. As the PSNR value is measured in dB, which is a logarithmic value, we scaled also the file size logarithmically in Figure 5.5. This shows that the file size decreases faster than the image quality over most of the available range of QP values. Therefore, it makes sense to evaluate the necessary video quality and choose the highest possible QP value. In streaming scenarios, the QP value can be adapted dynamically to utilize the available bandwidth. We used the mean of the PSNR values of all frames of each test video. The reconstruction quality is nearly the same for all frames of a video as it can be seen in Figure 5.6 showing the standard deviation of PSNR for different QP values. We found a significant increase of the standard deviation starting with QP values of 42. This was caused by a far less quality of P frames compared with I frames. A possible explanation could be that the low quality of the I frame used as reference picture caused even more quality problems in the dependent P frames. This effect should be analyzed in more depth before using QP values above 40, but is not a problem for our conformance tests, as these are done with the default QP value of 26.

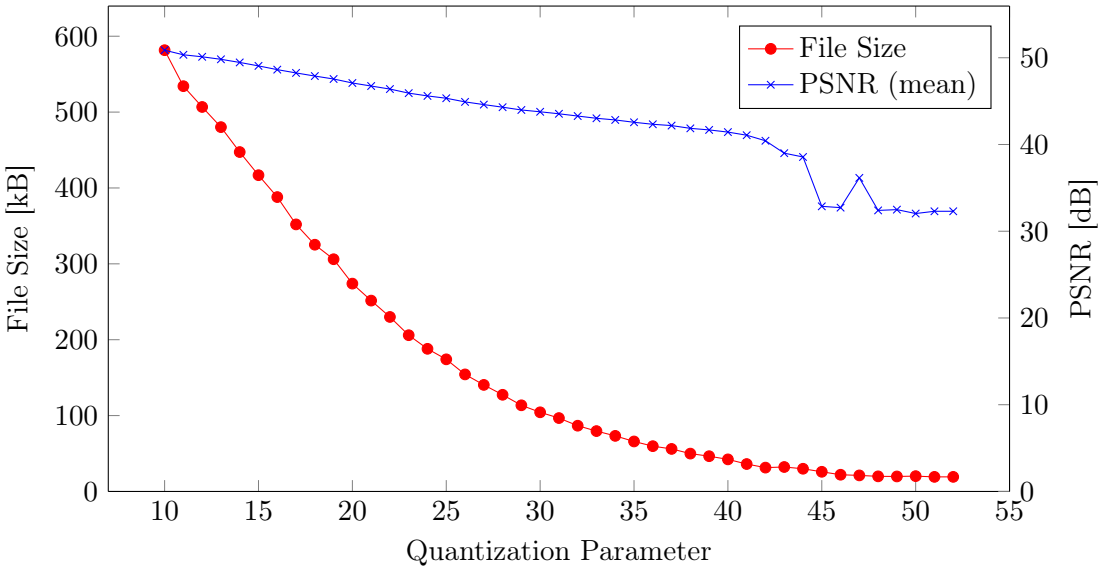


Figure 5.4.: Effect of Quantization Parameter on Reconstruction Quality and File Size

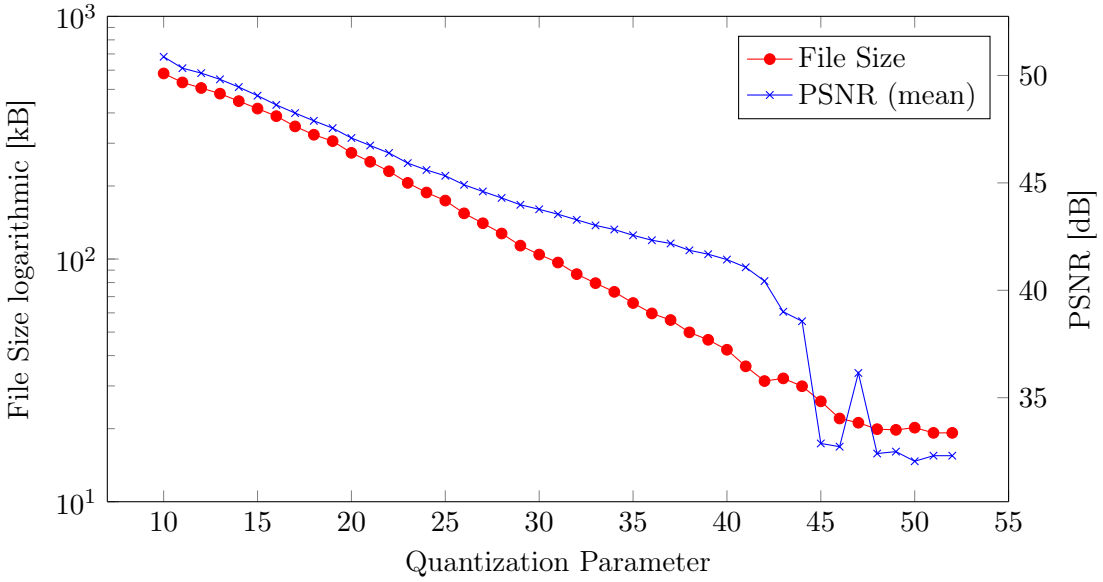


Figure 5.5.: Effect of Quantization Parameter on Reconstruction Quality and File Size with logarithmic scale (higher PSNR is better)

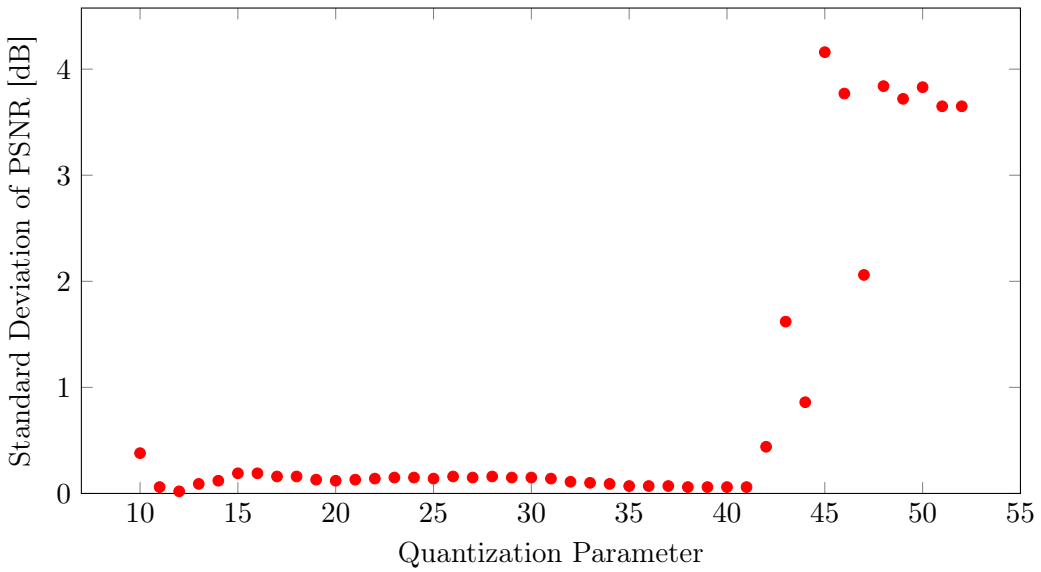


Figure 5.6.: Effect of Quantization Parameter on Reconstruction Quality Stability throughout a Video

The CTS executed all 48 test cases successfully and reported them as passed. There were no errors or warnings and the reconstruction quality was within the given limits. Therefore, we can say that our CPU based implementation of Vulkan Video conforms to the Vulkan specification and also to the industry standards for H.264 video encoding. To validate our implementation not only on Windows, but also on Linux, we built the CTS also on our Linux test system. We executed the same set of 48 test cases. All test cases passed successfully. Therefore, we can say, that our extended Lavapipe driver is conforming to Vulkan Video and H.264 standards both on Windows and on Linux.

## 5.3. Performance Tests

Additionally to the functionality and conformance tests we also wanted to evaluate if our implementation has an performance impact compared with using the codec directly without our Vulkan Video extension. For having a test scenario comparable to a real world situation of a cloud gaming server, we decided to use our simple Vulkan Video encoding example as a test application. This example renders a scene (a moving triangle) using Vulkan, converts it to YCbCr and encodes each frame using Vulkan Video into an H.264 elementary stream, which it then stores as file. To have an equivalent example without using Vulkan Video, we adapted the application, so that it grabs (copies) each frame after conversion from the Vulkan image buffers and feeds it directly into the minih264 video encoder. It then also stores the resulting H.264 elementary stream as file on the disk. We decided to render and encode a test sequence of 1000 frames accounting for about 33 seconds of video with a playback speed of 30 frames per second (FPS). We

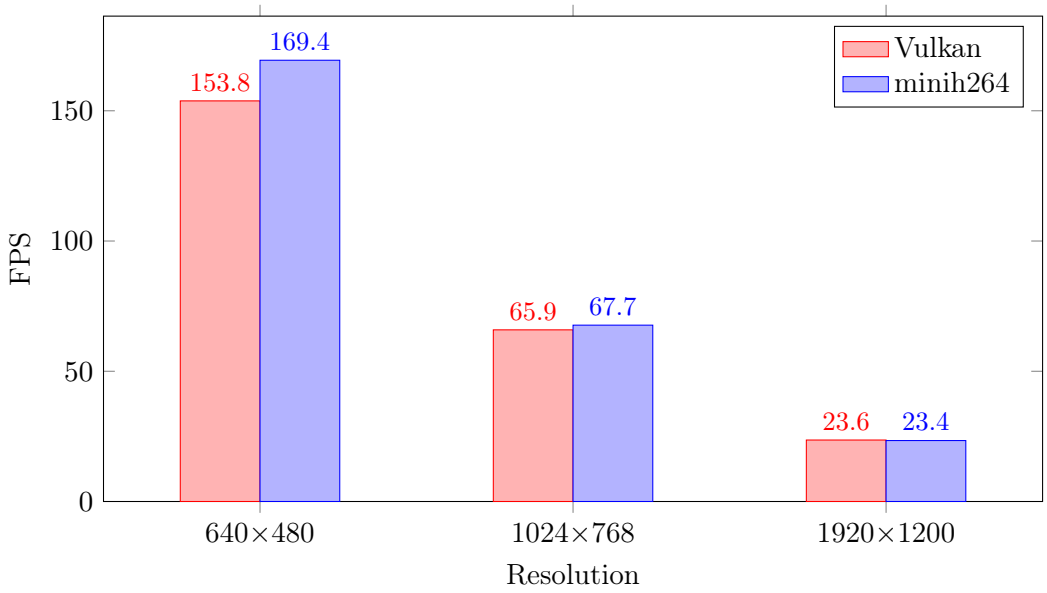


Figure 5.7.: Encoding speed comparison Vulkan Video vs. minih264 directly (higher is better)

measured the total execution time of the rendering and encoding process for each of the two applications to calculate the encoding speed (also in FPS). In both cases the rendering was done using the Lavapipe rasterizer, with and without using our Vulkan Video extensions. Every test run was done 5 times and the average of the runtimes were taken before the encoding speed was calculated. As we expected that the resolution has a non-linear impact we decided to run the tests with 3 standard resolutions. Figure 5.7 shows the comparison of the encoding speed in frames per second between our Vulkan Video implementation and our test application directly using minih264. Lower resolutions show the performance impact caused by the overhead of the additional API layer. For higher resolutions this impact gets neglectable. At full HD resolution our Vulkan Video implementation is even faster compared with the direct use of minih264. This can be explained with the zero-copy technique we use to provide the rendered frame to the video encoder. With an performance impact of 2.7% at 1024×768 and an performance gain at higher resolutions our Vulkan Video implementation is a viable alternative to directly using the video codec. We provide the complete measurement results in Appendix A.

Our implementation uses the pure C variant of minih264 to be platform independent, but minih264 also provides optimized variants using Intel SSE2 and ARM Neon instruction sets. Using these instruction sets reduces the portability, but as those two platforms are widely available, we also tested the performance of our Vulkan Video implementation using SSE2 optimization. Figure 5.8 shows that even for full HD resolution an encoding speed of over 72 frames per second is possible. This makes our CPU based solution a good alternative if hardware accelerators are not available. The encoder is using only one CPU core, so that all other cores are still available for rendering while the video encoding

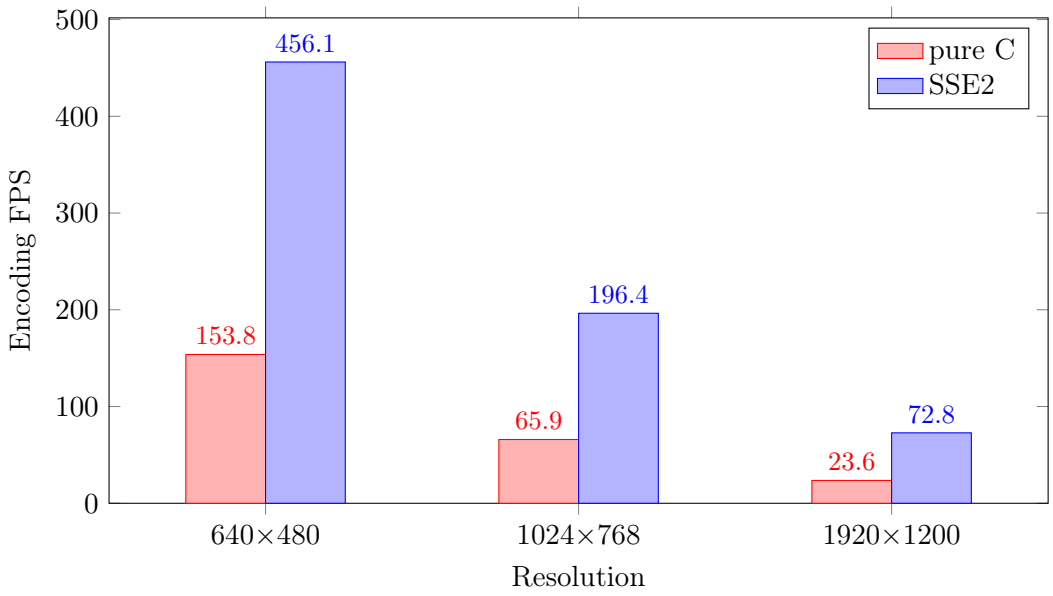


Figure 5.8.: Encoding speed of our Lavapipe driver comparing pure C vs. SSE2 implementations (higher is better)

is still running. The SSE2 optimized variant shows much better performance and is a competitive solution compared with hardware accelerated solutions. As it is available on all Intel compatible 64 bit platforms, we used the SSE2 variant for all further performance tests on our 2 Intel CPU based evaluation systems.

Our Lavapipe driver with Video Extensions is also available on Linux. Figure 5.9 shows the encoding speed comparison of Windows compared with Linux running on the same hardware. The Linux system showed a significantly higher encoding speed between 13 and 34%. Both systems ran natively on the hardware without virtualization, which could have impacted the performance. A possible explanation for these performance differences could be the operating system, but we also used a different compiler on the two evaluation systems. While we used Microsoft C/C++ 19.43.34810 on Windows, we used GCC 14.2.0 on Linux. We decided to evaluate the performance impact of using different compilers and optimization flags. For this test we compared the performance on our Linux test system with the 3 compiler (settings):

- **GCC default** using GNU GCC 14.2.0 with optimization: `-O2`,
- **Clang default** using LLVM clang 20.1.2 with optimization: `-O2` and
- **Clang native CPU** using LLVM clang 20.1.2 with: `-O2 -march=native`.

The last variant `-march=native` allows the compiler to use all instructions, which are available on the currently used CPU. Compared with the default settings, the compiled program can not run on older CPU generations, but is perfectly optimized for the current

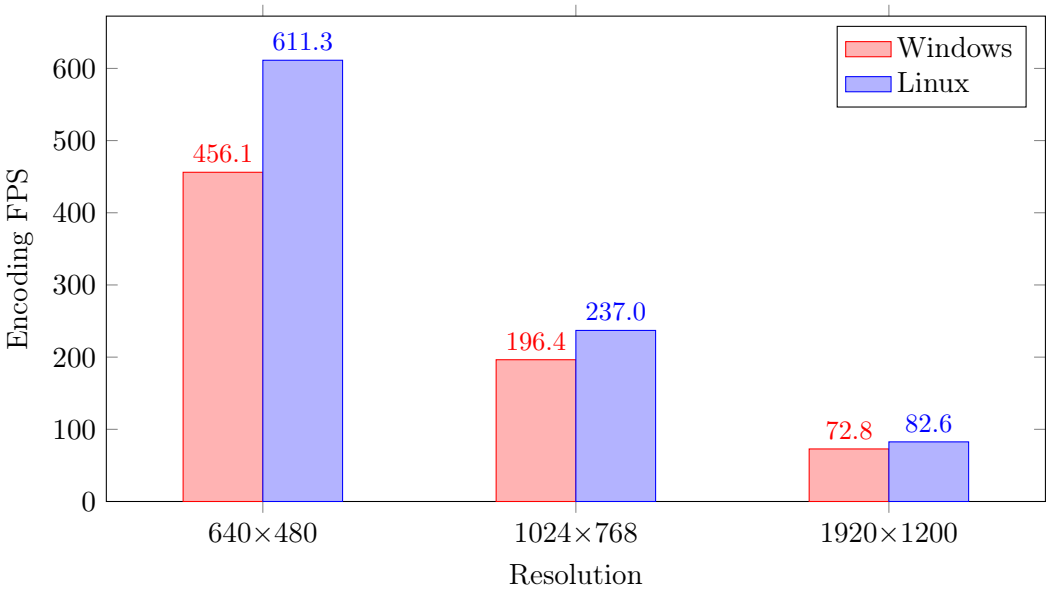


Figure 5.9.: Encoding speed comparison Windows vs. Linux (higher is better)

CPU. Figure 5.10 shows that the choice of the compiler can have a significant impact on the speed of our Lavapipe Vulkan Video driver. Switching to a better optimizing compiler and effectively using the instructions of the target CPU can bring a speedup between 10 and 13% without switching the hardware or the operating system. This shows that it definitely makes sense to optimize the Lavapipe driver for each hardware to reach the best results. Our third evaluation system based on a Raspberry Pi 5 is using an Arm based CPU. It has much less power and cooling requirements, therefore it can not achieve the performance of our Intel based evaluation hardware. Our Lavapipe driver implementation does not support ARM Neon SIMD optimization, therefore we compared in Figure 5.11 the driver without SIMD optimization. This shows that the Arm CPU can only reach about 40% of the performance of our reference system. As the Arm CPU is optimized for low power consumption instead of speed, a fair comparison would need to consider the power consumption and compare FPS/Watt to show which platform is more power and cost efficient.

One use case for our Lavapipe driver with Vulkan Video extensions is the rendering of a scene and directly encoding the result as video. Up to now we tested this functionality, but used the most simple scene possible, which is one triangle. We chose this to compare the performance of the video encoding part. Beside showing the performance of our implementation such a scenario is also useful, where a GPU is available and can be used for rendering, but not for video encoding. In situations without any GPU hardware at all, Lavapipe can also render more complex scenes on the CPU. For our comparison we chose the Crytek Sponza scene as it is included in the assets archive of the official Vulkan samples[KGc]. The scene shown in Figure 5.12 consists of 262267 triangles and 184330 vertices. For our test we used an animation, rendering a dolly shot moving along the scene.

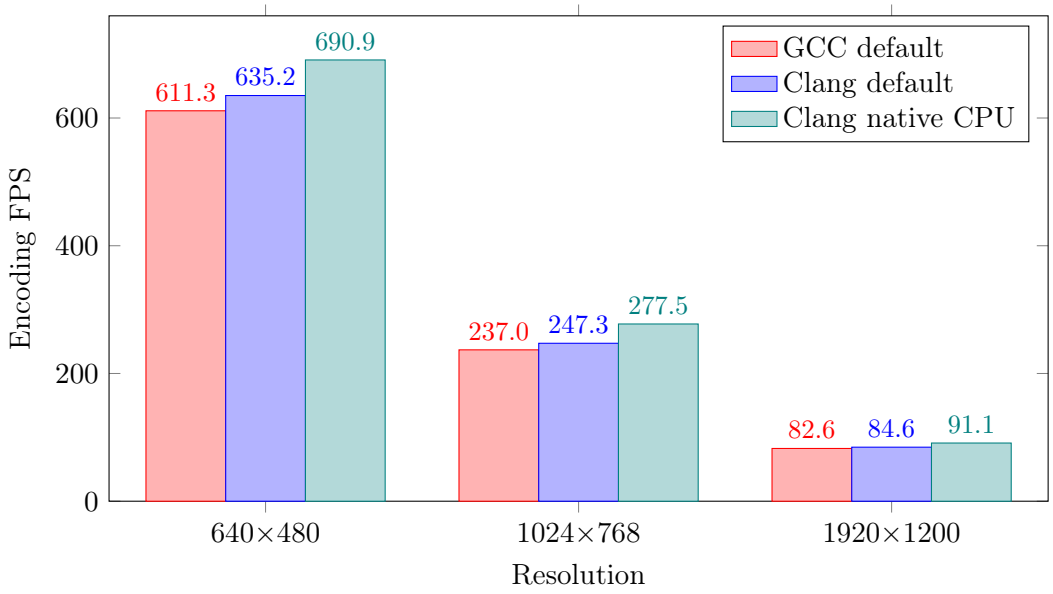


Figure 5.10.: Encoding speed comparison between different compiler and settings (higher is better)

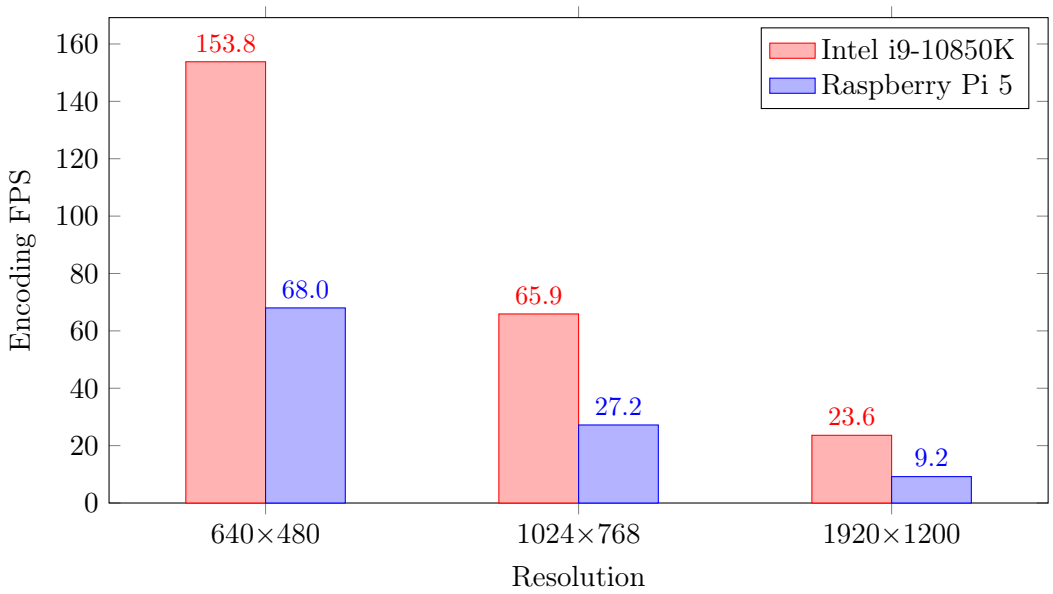


Figure 5.11.: Encoding speed comparison Intel CPU vs. Raspberry Pi (higher is better)

Figure 5.12.: Complex Scene Example

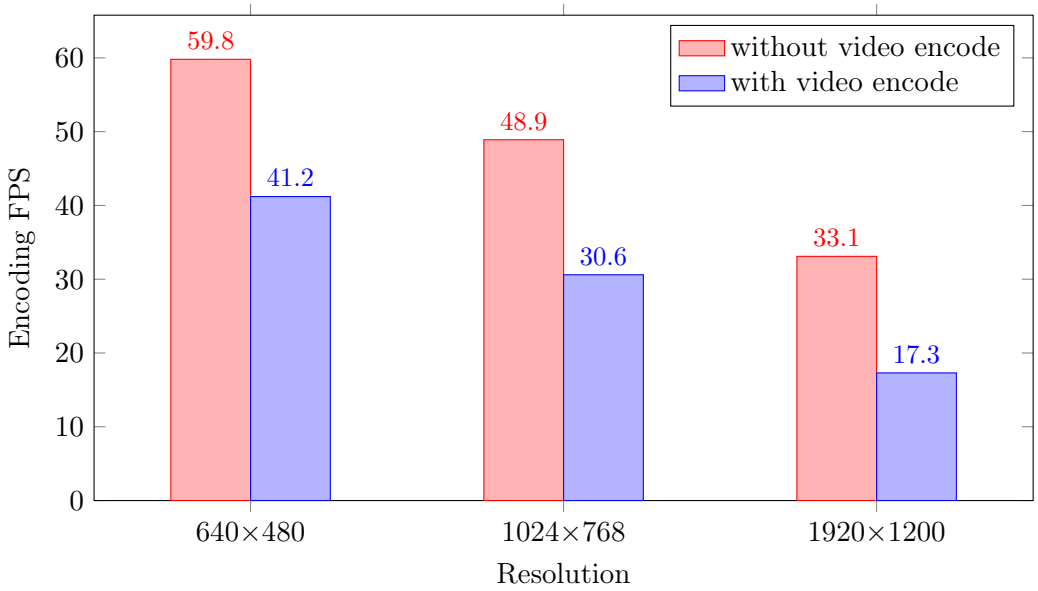


Figure 5.13.: Framerate comparison of complex scene rendering without and with video encoding (higher is better)

Figure 5.13 compares the frame rate of the rendering with disabled and with enabled video encoding. Due to the more complex scene, the overall framerate is lower compared with our previous tests. But we also see that video encoding causes a performance impact of 30 - 50%. Further studies should evaluate if this impact can be reduced by improving parallelization. Possible ways to do this are using multiple Vulkan queues within the application or improvements in our driver code using more threads.

Servers must be able to handle multiple users in parallel, therefore we analyzed how our driver scales to encode multiple video streams in parallel. In our test we rendered and encoded between 1 and 24 streams at the same time and measured the framerate. In the linear scaling case doubling the stream count would mean halving the framerate for each stream. As we used a 10-core CPU we expected better than linear scaling below 10 streams. Figure 5.14 shows the framerate over the number of parallel streams. We also added the expected curves for linear scaling based on the 1 stream and 10 stream values. The figure affirms our expectation that we scale better than linear below 10 streams. Over

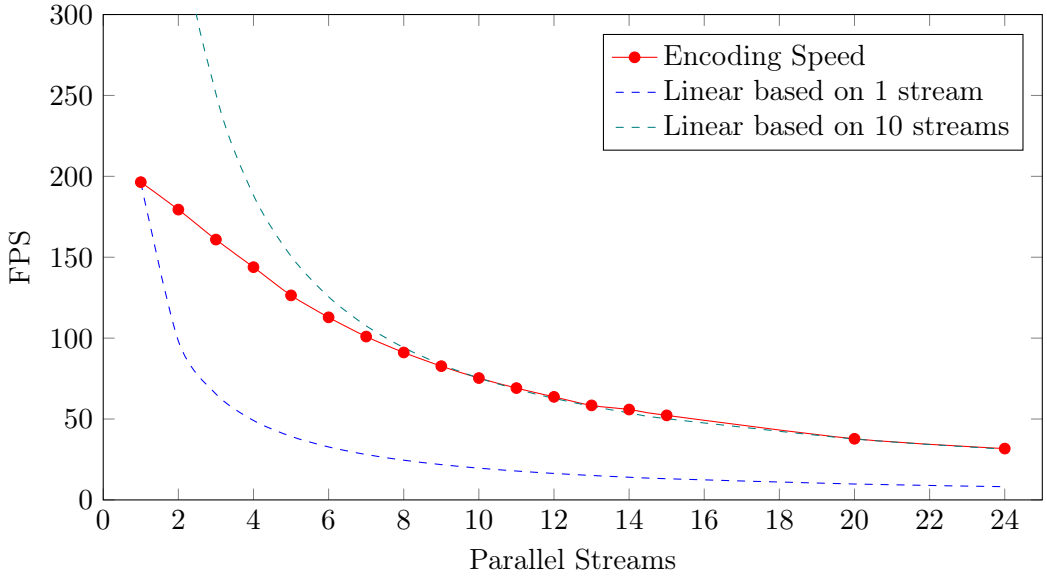


Figure 5.14.: Effect of scaling to multiple parallel streams including expectations in case of linear scaling

10 streams the scaling is nearly linear as no more cores can be utilized. The performance tests showed that our implementation can already be used for real-world use cases, but it also showed that there is even more room for improvement, especially in the area of parallelization of rendering and video encoding.

# 6. Conclusion and Outlook

We showed that it is feasible to implement Vulkan Video extensions completely on CPUs and therefore proofed our hypothesis H1. For this we implemented the Vulkan Video extension for video encoding on top of Mesa's Lavapipe driver, which answers our first research question RQ1. We designed a codec API and connected successfully a modified version of the H.264 encoder `minih264` to our extended Lavapipe driver, answering research question RQ2. Using the official Vulkan Conformance Test Suite (CTS) we could show that our CPU based Vulkan Video extension is both conforming to the Vulkan specification and the H.264 video encoding standard. Our implementation passes all test cases of the CTS and produces correct and standard conforming video bitstreams. We also showed that the decoded video is similar to the input data, which answers our last research question RQ3.

Apart from showing the functionality of our CPU based Vulkan Video implementation, we also measured the performance overhead. We showed that the overhead depends on the resolution and gets less for higher resolutions as we can use zero-copy techniques within the driver. Starting with full HD resolution (1920×1200) our solution is even faster than grabbing the frames and directly using `minih264`. With an optimized software codec our solution can easily support real-time encoding of full HD streams with over 70 frames per second. It is therefore a viable solution if an hardware accelerator is not available. Due to the standardized Vulkan Video API an application already supporting other Vulkan Video accelerators does not need to be changed to use our implementation as software fallback.

Our research focused on showing the feasibility of implementing Vulkan Video extensions on CPUs by supporting H.264 video encoding. For supporting further use cases we plan to research the possibilities and extend our solution to:

- support video decoding,
- research possible performance improvements, especially parallelization,
- handle more H.264 profiles and features, like B frames,
- implement support for further coding standard, like AV1 and
- get our changes merged into the official Mesa3D Lavapipe driver.

Our research laid the groundwork for using Vulkan Video Extensions on CPUs and demonstrated their viability for real-world applications. At the same time, it opens up numerous opportunities for further exploration and development in this field.

# Bibliography


[AfOM19] The Alliance for Open Media. AV1 Bitstream & Decoding Process Specification, 2019.

[AHSH14] Yong-Jo Ahn, Tae-Jin Hwang, Dong-Gyu Sim, and Woo-Jin Han. Implementation of fast hevc encoder based on simd and data-level parallelism. *EURASIP journal on image and video processing*, 2014(1):1–19, 2014.

[AMD] AMD. Advanced Media Framework. `https://gpuopen.com/advanced-media-framework/`. Last accessed 30 November 2024.

[App] Apple. Core Video. `https://developer.apple.com/documentation/corevideo`. Last accessed 30 November 2024.

[Bel] Fabrice Bellard. FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video. `https://www.ffmpeg.org/`. Last accessed 4 May 2025.

[Bit23] Bitmovin. The 7th Annual Bitmovin Video Developer Report. `https://bitmovin.com/downloads/assets/bitmovin-7th-video-developer-report-2023-2024.pdf`, 2023. Last accessed 19 November 2024.

[BS23] N Usha Bhanu and C Saravanakumar. Investigations of machine learning algorithms for high efficiency video coding (hevc). In *2023 International Conference on Signal Processing, Computation, Electronics, Power and Telecommunication (IConSCEPT)*, pages 1–5. IEEE, 2023.

[BYF+09] A. Bakhoda, G.L. Yuan, W.W.L. Fung, H. Wong, and T.M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[CGXJ14] Li Chen, Chen Gang, Tong Xin, and Li Jinyu. VGPU: a real time GPU emulator, 2014.

[CHW+21] Hao Chen, Bo He, Hanyu Wang, Yixuan Ren, Ser-Nam Lim, and Abhinav Shrivastava. Nerv: Neural representations for videos. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 21557–21568, 2021.

[Cis] Cisco. OpenH264. `https://www.openh264.org/`. Last accessed 8 June 2025.

[dBGR+06] V.M. del Barrio, C. Gonzalez, J. Roca, A. Fernandez, and Espasa E. Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, volume 2006, pages 231–241. IEEE, 2006.

[Fou] LLVM Foundation. The LLVM Compiler Infrastructure. `https://llvm.org/`. Last accessed 28 May 2025.

[fre] freedesktop.org. GStreamer. `https://gstreamer.freedesktop.org/`. Last accessed 8 June 2025.

[Fry25] Lucas Fryzek. Vulkanised 2025: Current state of Lavapipe: Mesa's software renderer for Vulkan. `https://www.vulkan.org/user/pages/09.events/vulkanised-2025/T5-Lucas-Fryzek-Igalia.pdf`, 2025. Last accessed 22 March 2025.

[FWW13] H. Fink, T. Weber, and M. Wimmer. Teaching a modern graphics pipeline using a shader-based software renderer. *Computers & graphics*, 37(1-2):12–20, 2013.

[Gooa] Google. Android MediaCodec. `https://developer.android.com/reference/android/media/MediaCodec`. Last accessed 30 November 2024.

[Goob] Google. Google Issue Tracker - Issue 293419320. `https://issuetracker.google.com/issues/293419320`. Last accessed 1 December 2024.

[Gooc] Google. SwiftShader. `https://github.com/google/swiftshader`. Last accessed 8 June 2025.

[GTC03] Steven Ge, Xinmin Tian, and Yen-Kuang Chen. Efficient multithreading implementation of h.264 encoder on intel hyper-threading architectures. In *Fourth International Conference on Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint*, volume 1, pages 469–473 Vol.1. IEEE, 2003.

[Her11] Per Hermansson. Optimizing an H.264 video encoder for real-time HD-video encoding. `https://www.diva-portal.org/smash/get/diva2%3A432684/FULLTEXT01.pdf`, 2011. Last accessed 9 June 2025.

[Inta] Intel. Intel® Video Processing Library. `https://www.intel.com/content/www/us/en/developer/tools/vpl/overview.html`. Last accessed 30 November 2024.

[Intb] Intel. VA-API: Video Acceleration (VA) API - libva. `https://intel.github.io/libva/`. Last accessed 30 November 2024.

[Intc] Intel. VAAPI (Video Acceleration API). `https://www.intel.com/content/www/us/en/developer/articles/technical/linuxmedia-vaapi.html`. Last accessed 30 November 2024.

[ISO93] ISO/IEC JTC 1/SC 29. ISO/IEC 11172-2:1993 : Information technology — Coding of moving pictures and associated audio for digital storage media at up to about 1,5 mbit/s part 2: Video, 1993.

[ITU03] ITU-T. H.264 : Advanced video coding for generic audiovisual services, 2003.

[ITU13] ITU-T. H.265 : High efficiency video coding, 2013.

[ITU20] ITU-T. H.266 : Versatile video coding, 2020.

[ITU23] ITU-T. P.910 : Subjective video quality assessment methods for multimedia applications, 2023.

[KGa] The Khronos® Group. Conformant Products - Vulkan. `https://www.khronos.org/conformance/adopters/conformant-products/vulkan`. Last accessed 6 October 2024.

[KGb] The Khronos® Group. Vulkan CTS :: Vulkan Documentation Project. `https://docs.vulkan.org/guide/latest/vulkan_cts.html`. Last accessed 22 March 2025.

[KGc] The Khronos® Group. Vulkan Samples Assets. `https://github.com/KhronosGroup/Vulkan-Samples-Assets`. Last accessed 8 June 2025.

[KPS07] Jin Ryong Kim, Il-Kyu Park, and Kwang-Hyun Shim. The effects of network loads and latency in multiplayer online games. In Lizhuang Ma, Matthias Rauterberg, and Ryohei Nakatsu, editors, *Entertainment Computing - ICEC 2007, 6th International Conference, Shanghai, China, September 15-17, 2007, Proceedings*. Springer, 2007.

[KVWG25] The Khronos® Vulkan Working Group. Vulkan® 1.4.311 - A Specification. `https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html`, 2025. Last accessed 22 March 2025.

[LCW+25] Matteo Leonelli, Addison Crump, Meng Wang, Florian Bauckholt, Keno Hassler, Ali Abbasi, and Thorsten Holz. TwinFuzz: Differential testing of video hardware acceleration stacks. 2025.

[LGK23] Björn Lundell, Jonas Gamalielsson, and Andrew Katz. Implementing the hevc standard in software: Challenges and recommendations for organisations planning development and deployment of software. *Journal of Standardisation*, 2, Feb. 2023.

[LH25] Jon Leech and Tobias Hector. Vulkan® Documentation and Extensions: Procedures and Conventions. `https://registry.khronos.org/vulkan/specs/latest/styleguide.html`, 2025. Last accessed 22 March 2025.

[lie] lieff. minih264 - Minimalistic H264/SVC encoder single header library. `https://github.com/lieff/minih264`. Last accessed 13 April 2025.

[Mica] Microsoft. Direct3D 12 Video Overview. `https://learn.microsoft.com/en-us/windows/win32/medfound/direct3d-12-video-overview`. Last accessed 30 November 2024.

[Micb] Microsoft. Windows Advanced Rasterization Platform (WARP) Guide. `https://learn.microsoft.com/en-us/windows/win32/direct3darticles/directx-warp`. Last accessed 30 November 2024.

[MID+23] William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. High-performance gpu-to-cpu transpilation and optimization via high-level parallel constructs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 119–134, New York, NY, USA, 2023. ACM.

[Mir13] Evgeny Miretsky. A new software based gpu framework, 2013.

[NBL+14] Ngoc-Mai Nguyen, Edith Beigné, Suzanne Lesecq, Duy-Hieu Bui, Nam-Khanh Dang, and Xuan-Tu Tran. H.264/AVC hardware encoders and low-power features. In *2014 IEEE Asia Pacific Conference on Circuits and Systems, APCCAS 2014, Ishigaki, Japan, November 17-20, 2014*, pages 77–80. IEEE, 2014.

[Nvia] Nvidia. Video Codec SDK. `https://developer.nvidia.com/video-codec-sdk`. Last accessed 30 November 2024.

[Nvib] Nvidia. Video Codec SDK - Get Started. `https://developer.nvidia.com/nvidia-video-codec-sdk/download`. Last accessed 1 December 2024.

[Pau] Brian Paul. The Mesa 3-D graphics library. `https://www.mesa3d.org/`. Last accessed 22 March 2025.

[PHO+15] Jason Power, Joel Hestness, Marc S. Orr, Mark D. Hill, and David A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE computer architecture letters*, 14(1):34–36, 2015.

[RDM+09] David J. Roberts, Toby Duckworth, Carl M. Moore, Robin Wolff, and John O'Hare. Comparing the end to end latency of an immersive collaborative environment and a video conference. In Stephen John Turner, David J. Roberts, Wentong Cai, and Abdulmotaleb El-Saddik, editors, *13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, Singapore, 25-28 October 2009*, pages 89–94. IEEE Computer Society, 2009.

[RGM06] A. Rodríguez, A. González, and Manuel P. Malumbres. Hierarchical parallelization of an H.264/AVC video encoder. In *Fifth International Conference on Parallel Computing in Electrical Engineering (PARELEC 2006), 13-17 September 2006, Bialystok, Poland*, pages 363–368. IEEE Computer Society, 2006.

[RKH24] Geetha Ramasubbu, Andrè Kaup, and Christian Herglotz. Modeling the energy consumption of the hevc software encoding process using processor events, 2024.

[RLCW00] Ann Marie Rohaly, John Libert, Philip Corriveau, and Arthur Webster. Final report from the video quality experts group on the validation of objective models of video quality assessment. Technical report, Video Quality Experts Group, March 2000.

[RVTS09] Sourabh Rungta, Kshitij Verma, Neeta Tripathi, and Anupam Shukla. Enhanced mode selection algorithm for h.264 encoder for application in low computational power devices, 2009.

[SCL+22] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. Vulkan-sim: A gpu architecture simulator for ray tracing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 263–281, Piscataway, NJ, USA, 2022. IEEE Press.

[Seu23] Konstantin Seurer. lavapipe: Implement VK_KHR_ray_query. `https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/25616`, 2023. Last accessed 9 June 2025.

[SFLB07] Klaus Schöffmann, Markus Fauster, Oliver Lampl, and László Böszörményi. An evaluation of parallelization concepts for baseline-profile compliant H.264/AVC decoders. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, volume 4641 of *Lecture Notes in Computer Science*, pages 782–791. Springer, 2007.

[SQ15] Tina Samajdar and Md. Iqbal Quraishi. Analysis and evaluation of image quality metrics. In J. K. Mandal, Suresh Chandra Satapathy, Manas Kumar Sanyal, Partha Pratim Sarkar, and Anirban Mukhopadhyay, editors, *Information Systems Design and Intelligent Applications*, pages 369–378, New Delhi, 2015. Springer India.

[TMM+] Praveen Kumar Tiwari, Vignesh V Menon, Jayashri Murugan, Jayashree Chandrasekaran, Gopi Satykrishna Akisetty, Pradeep Ramachandran, Sravanthi Kota Venkata, Christopher A Bird, and Kevin Cone. Accelerating x265 with Intel® Advanced Vector Extensions 512. `https://www.intel.com/co`

ntent/dam/develop/external/us/en/documents/mcw-intel-x265-avx512.pdf. Last accessed 9 June 2025.

[TNI+24] Razil Tahir, Jorji Nonaka, Ken Iwata, Taisei Matsushima, Naohisa Sakamoto, Chongke Bi, Masahiro Nakao, and Hitoshi Murai. Analysis towards energy-aware image-based in situ visualization on the fugaku. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPCAsia 2024, Nagoya, Japan, January 25-27, 2024*, pages 154–163. ACM, 2024.

[UJM+12] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2sim: a simulation framework for cpu-gpu computing. In *PACT'12 : proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, September 19-23, Minneapolis, Minnesota, USA*, pages 335–344, New York, NY, USA, 2012. ACM.

[WBSS04] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[Wil] Sascha Willems. Vulkan Hardware Database. https://vulkan.gpuinfo.org/. Last accessed 30 November 2024.

[WM08] S. Winkler and P. Mohandas. The evolution of video quality measurement: From psnr to hybrid metrics. *IEEE transactions on broadcasting*, 54(3):660–668, 2008.

[x26] x264, LLC. x264. https://x264.org/. Last accessed 8 June 2025.

# A. Appendix

## Development and Evaluation Environment Setup

For reference we provide here the development environment description file `README.md`:

```
# Mesa Development Environment

## Tested environment
* Windows 11 Pro 24H2 26100.3915
* Visual Studio 2022 Community 17.13.6 (with C++, git & cmake)
* Miniconda3 py312_25.1.1-2 (64-bit)
  (https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.
    exe)
* Vulkan SDK 1.4.304.1
* FFMPEG (or VLC) to view h264 files

## Get the development environment package
‘‘‘cmd
git clone git@github.com:clemy/mesa-dev.git
‘‘‘

For Windows Subsystem for Linux (WSL) see ‘mesa-dev\linux-wsl\README.md‘.

## Get dependencies
Get all dependencies and store them in sub directories of ‘mesa-dev‘.

### LLVM
Download LLVM 18.1.8 and extract into sub directory ‘mesa-dev\llvm\llvm‘:
https://github.com/llvm/llvm-project/releases/download/llvmorg-18.1.8/
    llvm-18.1.8.src.tar.xz

Download the LLVM cmake package and extract into sub directory ‘mesa-dev\
    llvm\cmake‘:
https://github.com/llvm/llvm-project/releases/download/llvmorg-18.1.8/
    cmake-18.1.8.src.tar.xz

### Flex & Bison
Download and extract the Windows Flex & Bison zip:
* https://sourceforge.net/projects/winflexbison/ -> win_flex_bison

### Mesa3D (with Lavapipe Vulkan Video Extensions)
‘‘‘cmd
git clone git@github.com:clemy/mesa.git -b lavapipe_video
‘‘‘

### Simple Vulkan Video Encode Example
```

```
```cmd
git clone git@github.com:clemy/vulkan-video-encode-simple.git
```

### Vulkan Conformance Test Suite (CTS) with compatibility patches
```cmd
git clone git@github.com:clemy/VK-GL-CTS.git -b lavapipe_test_2504
```

## Configure paths
Edit `config.cmd` and configure the path to Visual Studio and Conda.

## Setup environment
Execute `create-environment.cmd`. This will create a conda environment.

## Open a prompt with all path set
Execute `setpath.cmd`.
This will open a prompt with a pre-configured environment.

## Build LLVM
On the pre-configured command prompt call:
```cmd
llvm\build-llvm.cmd
```
This will compile and install LLVM into `llvm\llvm-installed`.

## Build Lavapipe
On the pre-configured command prompt call:
```cmd
build-lavapipe.cmd
```
This will compile and install Lavapipe in `mesa-installed`.

## Build and run the simple Vulkan Video Encode example
On the pre-configured command prompt call:
```cmd
rem Compile the example
cmake -B vulkan-video-encode-simple-build vulkan-video-encode-simple
cmake --build vulkan-video-encode-simple-build --config Release

rem Run the example (in subdirectory to find shader files)
cd vulkan-video-encode-simple-build
Release\headless.exe

# Show the created video file
ffplay hwenc.264
# alternatively (for other video players)
start hwenc.264

cd ..
```

## Build and run CTS
```

```
On the pre-configured command prompt call:
```cmd
rem Fetch dependencies
python VK-GL-CTS\external\fetch_sources.py
python VK-GL-CTS\external\fetch_video_encode_samples.py

rem Compile CTS
cmake -B VK-GL-CTS-build VK-GL-CTS -DCMAKE_BUILD_TYPE=Release ^
        -DSELECTED_BUILD_TARGETS="deqp-vk"
cmake --build VK-GL-CTS-build --config Release -j 10

rem Enable real GPU as first and Lavapipe as second Vulkan device
set VK_ICD_FILENAMES=
set VK_ADD_DRIVER_FILES=%PROJECT%\mesa-installed\share\vulkan\icd.d\
    lvp_icd.x86_64.json

rem Check GPU order
vulkaninfo --summary

rem Run CTS on Lavapipe (second device) with first device as reference
    decoder
cd VK-GL-CTS-build\external\vulkancts\modules\vulkan
Release\deqp-vk -n dEQP-VK.video.encode.h264.* --deqp-vk-device-id=2
```

## Visual Studio Development & Debug Project
For developing Lavapipe in Visual Studio follow these steps:

1. Use `setpath-debug.cmd` to open a command prompt.
2. Call `llvm\build-llvm-debug.cmd` to compile an LLVM version
   linked with the debug runtime library.
3. Call `build-lavapipe-debug-vs.cmd` to create VS projects.
4. Open the VS solution with `start mesa-build-vs-debug\mesa.sln`.
5. Create a VS solution for CTS:
   `cmake -B VK-GL-CTS-build-vs-debug VK-GL-CTS -G "Visual Studio 17
    2022"`
6. Open CTS solution: `start VK-GL-CTS-build-vs-debug\dEQP-Core-default.
    sln`
```

## Measurement Results

All time measurements were taken in milliseconds (ms). The encoding FPS are calculated based on 1000 frames, except for Raspberry Pi and complex scene tests, which are based on 300 frames.

**Windows 640×480**

| Run | minih264 | vulkan | sse2 |
|---|---|---|---|
| **1** | 5904.19 | 6488.97 | 2185.49 |
| **2** | 5891.96 | 6506.18 | 2209.85 |
| **3** | 5893.10 | 6507.05 | 2190.21 |
| **4** | 5913.19 | 6509.56 | 2183.71 |
| **5** | 5913.15 | 6498.37 | 2192.65 |
| **Mean** | 5903.12 | 6502.03 | 2192.38 |
| **FPS** | **169.40** | **153.80** | **456.12** |

**Windows 1024×768**

| Run | minih264 | vulkan | sse2 |
|---|---|---|---|
| **1** | 14763.0 | 15167.8 | 5088.29 |
| **2** | 14776.3 | 15187.2 | 5091.96 |
| **3** | 14775.8 | 15162.6 | 5087.67 |
| **4** | 14772.4 | 15226.2 | 5112.25 |
| **5** | 14790.8 | 15165.6 | 5077.34 |
| **Mean** | 14775.7 | 15181.9 | 5091.50 |
| **FPS** | **67.68** | **65.87** | **196.41** |

**Windows 1920×1200**

| Run | minih264 | vulkan | sse2 |
|---|---|---|---|
| **1** | 42627.7 | 42452.7 | 13733.2 |
| **2** | 42670.6 | 42416.1 | 13707.6 |
| **3** | 42724.4 | 42330.2 | 13754.3 |
| **4** | 42619.8 | 42374.2 | 13692.8 |
| **5** | 42669.5 | 42418.4 | 13776.0 |
| **Mean** | 42662.4 | 42398.3 | 13732.8 |
| **FPS** | **23.44** | **23.59** | **72.82** |

**Linux 640×480**

| Run | gcc | clang | clang-native |
|---|---|---|---|
| **1** | 1640.81 | 1569.49 | 1443.98 |
| **2** | 1627.51 | 1581.39 | 1452.43 |
| **3** | 1640.42 | 1570.18 | 1442.59 |
| **4** | 1629.36 | 1580.75 | 1451.67 |
| **5** | 1640.78 | 1569.31 | 1446.19 |
| **Mean** | 1635.78 | 1574.22 | 1447.37 |
| **FPS** | **611.33** | **635.23** | **690.91** |

**Linux 1024×768**

| Run | gcc | clang | clang-native |
|---|---|---|---|
| **1** | 4225.04 | 4091.31 | 3583.64 |
| **2** | 4205.14 | 4031.00 | 3600.96 |
| **3** | 4247.46 | 4052.77 | 3596.26 |
| **4** | 4218.40 | 4001.53 | 3637.99 |
| **5** | 4200.56 | 4045.40 | 3598.97 |
| **Mean** | 4219.32 | 4044.40 | 3603.56 |
| **FPS** | **237.01** | **247.26** | **277.50** |

**Linux 1920×1200**

| Run | gcc | clang | clang-native |
|---|---|---|---|
| **1** | 12133.9 | 11798.7 | 10981.7 |
| **2** | 12065.6 | 11832.9 | 10962.2 |
| **3** | 12104.5 | 11843.0 | 10997.9 |
| **4** | 12097.1 | 11811.8 | 10953.7 |
| **5** | 12109.4 | 11829.2 | 10968.5 |
| **Mean** | 12102.1 | 11823.1 | 10972.8 |
| **FPS** | **82.63** | **84.58** | **91.13** |

**Raspberry Pi 5**

| Run | 640×480 | 1024×768 | 1920×1200 |
|---|---|---|---|
| **1** | 4496.72 | 11032.9 | 32864.0 |
| **2** | 4372.85 | 11004.8 | 32500.9 |
| **3** | 4354.76 | 11013.8 | 32549.6 |
| **4** | 4465.41 | 11039.6 | 32559.1 |
| **5** | 4377.77 | 10973.3 | 32653.9 |
| **Mean** | 4413.50 | 11012.9 | 32625.5 |
| **FPS** | **67.97** | **27.24** | **9.20** |

**Complex Scene - rendering only**

| Run | 640×480 | 1024×768 | 1920×1200 |
|---|---|---|---|
| **1** | 5006 | 6099 | 9032 |
| **2** | 5035 | 6130 | 9047 |
| **3** | 4999 | 6154 | 9002 |
| **4** | 5037 | 6154 | 9172 |
| **5** | 5020 | 6164 | 9046 |
| **Mean** | 5019 | 6140 | 9060 |
| **FPS** | **59.8** | **48.9** | **33.1** |

**Complex Scene - rendering with video encoding**

| Run | 640×480 | 1024×768 | 1920×1200 |
|---|---|---|---|
| **1** | 7321 | 9800 | 17331 |
| **2** | 7272 | 9805 | 17277 |
| **3** | 7239 | 9818 | 17331 |
| **4** | 7271 | 9785 | 17297 |
| **5** | 7266 | 9817 | 17298 |
| **Mean** | 7274 | 9805 | 17307 |
| **FPS** | **41.2** | **30.6** | **17.3** |

**Parallel Streams**

| Streams | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Mean | FPS |
|---|---|---|---|---|---|---|---|
| **1** | 5088.29 | 5091.96 | 5087.67 | 5112.25 | 5077.34 | 5091.50 | **196.4** |
| **2** | 5561.19 | 5608.60 | 5580.14 | 5541.45 | 5573.36 | 5572.95 | **179.4** |
| **3** | 6228.38 | 6248.21 | 6166.96 | 6222.12 | 6208.29 | 6214.79 | **160.9** |
| **4** | 6979.75 | 6932.96 | 6957.11 | 6923.47 | 6963.09 | 6951.28 | **143.9** |
| **5** | 7932.14 | 7898.41 | 7972.70 | 7856.98 | 7888.79 | 7909.80 | **126.4** |
| **6** | 8835.28 | 8833.88 | 8891.91 | 8868.79 | 8873.53 | 8860.68 | **112.9** |
| **7** | 9925.17 | 9908.48 | 9946.86 | 9914.16 | 9828.40 | 9904.61 | **101.0** |
| **8** | 11081.9 | 10948.6 | 10923.1 | 11040.7 | 10878.8 | 10974.6 | **91.1** |
| **9** | 12171.2 | 11998.8 | 11993.4 | 12116.0 | 12200.3 | 12095.9 | **82.7** |
| **10** | 13316.5 | 13196.5 | 13292.0 | 13161.6 | 13417.5 | 13276.8 | **75.3** |
| **11** | 14509.5 | 14419.0 | 14446.5 | 14449.0 | 14508.1 | 14466.4 | **69.1** |
| **12** | 15580.7 | 15817.8 | 15650.2 | 15833.1 | 15618.7 | 15700.1 | **63.7** |
| **13** | 16979.7 | 17066.8 | 17147.5 | 17213.0 | 17174.4 | 17116.3 | **58.4** |
| **14** | 17978.0 | 17883.0 | 17792.2 | 18055.5 | 17790.0 | 17899.7 | **55.9** |
| **15** | 18887.7 | 19189.4 | 19331.6 | 18975.2 | 19290.3 | 19134.8 | **52.3** |
| **20** | 26100.4 | 26664.6 | 27309.6 | 26316.7 | 25905.5 | 26459.4 | **37.8** |
| **24** | 31378.5 | 31019.1 | 31461.7 | 32051.5 | 31653.4 | 31512.8 | **31.7** |

**Quantization Parameter Effect**

| QP | File Size (kB) | PSNR (mean) | PSNR (std.dev.) |
|---|---|---|---|
| **10** | 581.59 | 50.87 | 0.38 |
| **11** | 534.16 | 50.34 | 0.06 |
| **12** | 506.75 | 50.11 | 0.02 |
| **13** | 480.19 | 49.82 | 0.09 |
| **14** | 447.34 | 49.46 | 0.12 |
| **15** | 416.84 | 49.06 | 0.19 |
| **16** | 388.00 | 48.63 | 0.19 |
| **17** | 351.97 | 48.25 | 0.16 |
| **18** | 325.20 | 47.89 | 0.16 |
| **19** | 306.08 | 47.55 | 0.13 |
| **20** | 273.92 | 47.09 | 0.12 |
| **21** | 251.55 | 46.74 | 0.13 |
| **22** | 230.00 | 46.39 | 0.14 |
| **23** | 205.94 | 45.92 | 0.15 |
| **24** | 188.00 | 45.60 | 0.15 |
| **25** | 174.14 | 45.33 | 0.14 |
| **26** | 154.27 | 44.91 | 0.16 |
| **27** | 140.44 | 44.60 | 0.15 |
| **28** | 127.41 | 44.30 | 0.16 |
| **29** | 113.56 | 43.98 | 0.15 |
| **30** | 104.33 | 43.77 | 0.15 |
| **31** | 96.73 | 43.54 | 0.14 |
| **32** | 86.66 | 43.28 | 0.11 |
| **33** | 79.56 | 43.02 | 0.10 |
| **34** | 73.25 | 42.83 | 0.09 |
| **35** | 65.89 | 42.56 | 0.07 |
| **36** | 59.69 | 42.33 | 0.07 |
| **37** | 56.06 | 42.18 | 0.07 |
| **38** | 49.88 | 41.86 | 0.06 |
| **39** | 46.47 | 41.68 | 0.06 |
| **40** | 42.28 | 41.43 | 0.06 |
| **41** | 36.14 | 41.07 | 0.06 |
| **42** | 31.42 | 40.43 | 0.44 |
| **43** | 32.23 | 39.00 | 1.62 |
| **44** | 29.92 | 38.56 | 0.86 |
| **45** | 25.92 | 32.87 | 4.16 |
| **46** | 22.03 | 32.72 | 3.77 |
| **47** | 21.16 | 36.15 | 2.06 |
| **48** | 19.91 | 32.40 | 3.84 |
| **49** | 19.78 | 32.49 | 3.72 |
| **50** | 20.19 | 32.04 | 3.83 |
| **51** | 19.22 | 32.30 | 3.65 |
| **52** | 19.22 | 32.30 | 3.65 |